ChemTrains Design Study Supplement

John Rieman
Brigham Bell
Clayton Lewis

CU-CS-480-90 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

ABSTRACT

This document is a supplement to Problem-Centered Design for Expressiveness and Facility in a Graphical Programming Language (Lewis, C., Rieman, J., and Bell., B., [1990], Technical Report CUCS 479-90, Department of Computer Science, University of Colorado, Boulder CO.) That report 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.

In this document we include detailed descriptions of the problems, designs, and solutions developed during the problem-centered design process. For most sections, three groups of materials are presented: one for the OpsTrains design, one for ShowTrains, and one for ZeroTrains.

1. Designs

This section describes the three competing designs for ChemTrains: OpsTrains, ShowTrains, and ZeroTrains.

OPSTrains Design

- Brigham, 6/14/90

OPSTrains is a system derived from the ideas of the Chemtrains system, but is extended to work similarly to a standard production system language like OPS5. The graphical aspects of Chemtrains remain in OPSTrains, but the mechanism for reactions is more similar to OPS. Here are a list of aspects of OPSTrains.

- There are two main modes of the interface:

  assemble mode: allows the designer to create and edit pictures. All hidden things are shown in this mode.

  execute mode: things specified as hidden are hidden. reactions may occur. objects may be moved by the user.

- Pictures may be drawn using macdraw-like interface.

- Each individual graphical object is an object that may be matched, created, or deleted in a rule.

- An object contains another object if it's circumscribing rectangle encloses the object. Objects may be nested to a depth that is only limited by the spacial constraints.

- A path must connect to two objects. A unidirectional path allows objects to traverse in one direction and a bidirectional path allows objects to traverse in both directions. Paths may also be labelled with names.

- An object may only travel along a path if
  1) it is contained in an object that is connected by the path,
  2) either the path is bidirectional or the path is in the same direction as the object is travelling, and
  3) the object is placed on the path.
  When an object traverses a path all objects that it contains are moved with it.

- Any object, path, or name of path may be hidden.

- A rule has a name and two basic parts, a pattern picture and a result picture. Any number of objects and paths may be specified in the pattern picture. The result picture contains all objects from the pattern picture that are to remain. The result picture may also contain objects that are to be added. Any object or path that is left out of the result picture is deleted.

- a rule editor interface enables the user to edit the name, pattern, and result of a single rule.

- When the system is in "Execute Mode," if more than one rule can match the current picture, the rule that occurs earliest in the rule ordering is chosen. When a new rule is created it is added to the end of the rule list.

- a rule order interface enables the user to view and modify the order of importance of the rules using the name of a rule as its identifier.

- An object in a pattern matches an object in the picture, if:
  1) either the object is specified as a variable or the objects have identical displays, and
  2) all objects contained by the object in the pattern are also matched,

- Any object in a pattern may be specified as a variable. A variable object may match any object regardless of the display. If two objects in the pattern have an identical display and are both specified as variables, the objects that they match must have an identical display but not necessarily the same display as the objects in the pattern. The names of paths may also be specified as a variable.

- The absence of an object or a path may be tested in the pattern of a rule.

- A path in a pattern can match a path in the picture, if:
  1) the connecting objects have been matched,
  2) the directionality matches, and
  3) the name matches.
  The specific display of the path has no bearing on matching.

- A rule can not execute on the same unchanged set of objects more than once.

Showtrains
Manual and Specifications

John, June 1990

Part I. Specifications for ShowTrains

PLACES

Places are drawn on the screen by the user. Every place has a name, which need not be unique. Every Place has an associated graphic. The boundaries of the Place and the boundaries of its graphic need not coincide. Places cannot overlap, and they cannot be created, moved, or modified by Rule actions.

PATHS

Paths are drawn on the screen by the user. They can only be drawn between one Place with another (possibly the same Place). They may start and end anywhere within a Place or on its boundaries. They allow traversal in one direction only. Every Path has a name, which need not be unique. Every Path has an associated graphic, which describes the appearance of a single segment of the Path. Paths cannot be created, moved, or modified by system actions.

OBJECTS

Objects are created, moved, deleted, copied, and modified on the screen by either the user or the Rules. Objects are unstructured, except for an associated graphic and a name, which need not be unique. The location of an Object is approximately defined by its center of gravity.

RULES

Rules cause objects to be created, modified, deleted, and/or moved onto paths. Rules are specified by the following process (described in full detail later in this section):

1. Enter the name of the Rule.

2. Highlight (click on) the elements that form the condition of the Rule — these can include names, graphics, and outlines of Objects, Paths, and Places, anywhere on the screen.

3. Edit the screen to show the action of the Rule. The action can be any creation, deletion, or modification of any Object on the screen, as well as placing any existing or new Object onto a Path. The action cannot alter Places or Paths.

Conflict resolution, order, and fairness

Whenever a Rule fires, it completes all its actions before any other Rules are tested. If the conditions of more than one reaction are met, one of the possible Rules will be selected at random to fire. The order in which Rules are tested is not specified; however, once any Rule has been tested, all other Rules will be tested before the first Rule is tested again.

Once a Rule has placed an Object on a Path, the Path carries the Object to its destination, independently of whatever the Rules may be doing.

Predicate logic view of rules

The conditions can specify any combination of elements that could be written in predicate logic as:

"(objects A and B and ... in Place X) and (objects E and F and ... in Place Y) and ..."

where any object "N" may be replaced by "not N". To get the equivalent of "or," the user must write multiple rules.

Note that, since objects are unstructured and there are no variables, modifying an object is exactly the same as deleting it and creating a replacement. Similarly, copying an object is exactly the same as creating a second, identical object. Modify and copy routines are provided so that the user can apply the same editing techniques during reaction specification as are used during the initial creation and editing of Objects.

Part II. Detailed Instructions for ShowTrains

CREATING RULES BY DEMONSTRATION

ShowTrains learns how Objects interact by watching you demonstrate the interactions. This section describes exactly what you can demonstrate in a Rule and explains what effect the resulting Rule will have when the simulation runs.

ShowTrains gives you a lot of flexibility in creating Rules. For example, you might specify a very detailed rule, such as:

- If an object named "cat" with a cat graphic is in a place named "kitchen" with a kitchen graphic along with an object named "cat food" with a cat food graphic, then delete the "cat food" object and change the cat graphic to a happy-cat graphic and put the cat on the path to the bedroom.

This rule will only fire if a specific combination of Objects and Places is present. But you could also specify a very general rule, such as:

- If any object at all is in any place named "trash", then delete that object.

It's useful to think of the Rules as having two parts: the *condition*, which describes the state of the simulation that causes the rule to be applied, and the *action*, which is the effect of the Rule. When a Rule is applied, we often say it *fires*. To demonstrate a new Rule to ShowTrains, you set up the screen to show the conditions, then you perform exactly the actions that the rule will perform when it fires.

The generality or specificity of a Rule depends on how you specify its conditions. The conditions of a Rule can always be described in terms of *Objects in Places*. When you demonstrate the conditions of a new Rule, you should highlight only those things you want ShowTrains to check for before it fires the rule — everything else will be ignored. Here's what you can highlight:

- You can highlight a Place and one or more Objects within that Place. This means that the Objects must be found together in that Place in order for the Rule to fire.

- When highlighting an Object, you can highlight the name, the graphic, or both. If you highlight just the name, then any Object with that name will cause the Rule to fire (assuming any other conditions are satisfied). If you highlight just the graphic, then any Object with that graphic will cause the Rule to fire. If you highlight both, then only an Object with the same graphic and name will cause the Rule to fire.

- When highlighting an Object, you can highlight just the outline. This means that any Object at all will cause the rule to fire (if all other conditions are satisfied).

- For any Place, you can highlight the name, the graphic, or both.

- You can also specify a Place by highlighting Paths that are attached to it. For any Path, you can highlight the name, graphic, both, or just the outline (meaning: any Path will do, as long as it is attached to the Place in question.)

- For any Place, you can highlight just the outline. This means that any highlighted Objects within the Place must be found together within some Place, but not necessarily the Place highlighted.

- If you highlight one or more Objects, but don't highlight any part of the Place in which they are found, then matching Objects will satisfy the conditions of the Rule whenever there are found in any Place or Places on the screen, whether or not they are together.

- Anything you can highlight, you can "shimmer highlight" by holding down the cloverleaf key while you click. Shimmer-highlighted conditions are prohibited. If any of them are found, the Rule will not fire.

When you demonstrate to ShowTrains how a rule should modify the screen, you should use the same editing features that you've used to set up the simulation. However, only Objects can be affected by the firing of a rule — it can't modify Places or Paths. The actions of a Rule can have the following effects:

- An action can delete an existing Object.

- An action can modify an existing Object by changing its name or graphic.

- An action can create one or more new Objects, in any Place or Places.

- An action can duplicate an existing Object, and put the new Object in the same Place or in any other Place.

- An action can move any Object from one Place to another — this movement will appear to be instantaneous when the simulation is run.

- An action can take an Object that is in a Place and drag that Object onto the beginning of a Path leading out of that Place. Whenever an Object is placed on a Path, it will travel down that Path, and its motion will be visible on the screen.

- An action can create an Object in a Place, or instantaneously move an Object to a different Place, and then drag that Object onto the beginning of a Path leading out of the Place. (Note that an action can't drag an Object directly from a Place to a Path that leads out of some other Place.)

As you demonstrate the actions of a Rule, ShowTrains notices whether the actions affect Objects that were highlighted as part of the condition of the Rule. Actions can affect things that weren't highlighted in the conditions, but you must highlight all or part of those things first, to specify what parts are important. Here is how this affects the Rule's actions:

- If you modify or delete any Object that is all or partially highlighted as part of the Rule's condition, then ShowTrains will create a rule that always modifies the same Object that caused the Rule to fire.

- If you modify or delete any Object that is not highlighted as part of the Rule's condition, the ShowTrains will create a Rule that looks for an Object that matches the one you highlighted during the action, and modifies or deletes that Object. (The affected Object is guaranteed not to be one of the Objects highlighted in the condition.) If no such Object is found, the Rule will simply ignore this action, but it will do any other actions you've specified.

- You can create a new Object in any Place that is all or partially highlighted. This will create a Rule that always creates a new Object in the Place that caused the Rule to fire.

- You can create a new Object in any Place that was not highlighted in the condition. This will create a Rule that tries to create a new Object in a Place that matches the one you have highlighted in the action. If no such Place exists, no Object will be created, but any other actions specified by the Rule will be performed. (The new Object is guaranteed not to be created in any of the Places highlighted in the condition.)

- You can drag an Object from a Place onto any Path out of that Place that was highlighted in the condition. This will create a Rule that always puts the Object onto the Path that caused the Rule to fire.

- You can drag an Object from a Place onto any Path out of that Place that is not highlighted in the condition. This will create a Rule that tries to drag the Object onto a Path that matches the one you highlighted while specifying the action. If no such Path exists, the Object will remain in its original Place, but any other

actions specified by the Rule will be performed. (The new Object is guaranteed not to be dragged onto any of the Paths highlighted in the condition.)

- Since things that were shimmer-highlighted in the condition of a Rule must not exist for the Rule to fire, you can't take any action on shimmer-highlighted items.

ChemTrains Zero Design

Clayton, 5.31.90

Design Assumptions

There are objects, places, paths , and reactions

Objects are just icons.

Places are regions which can overlap and be nested.

Paths connect one place to another. Ends of paths have names.

Reactions specify a group of objects (before), and a group of objects optionally labelled with a path name (after). The after group is divided into one or more subgroups associated with objects in the before group.

A reaction can occur whenever the before objects exist together anywhere inside any place that has paths whose names match any named in the reaction provided at least one of the before objects is immediately contained in that place (Clayton's membrane rule).

When a reaction occurs any object appearing before but not after is deleted; any object occuring after but not before is created in the same location as the corresponding before object, and any object with a path specified after is placed on that path.

Objects placed on a path move along it and enter the place on the other end.

2. Doctrines

This section presents the doctrines for the three designs, as used in the programming walkthroughs.

OPSTrains Doctrine

- Brigham, 6/14/90

Building an interface in OPSTrains is not a simple task that can simply be stated as a set of simple steps to follow. Backtracking through work, repeating steps, and incremental implementation may be commonly practiced. OPSTrains Doctrine contains two parts, definitions and rules. The following is a rule based process program for building simulation interfaces in OPSTrains. The designer may decide to follow any rule at any time if the condition matches the current state of design.

OPSTrains Doctrine Definitions:

An *object* is a graphical object of the display.

An object *contains* another object if the graphical display of that object circumscribes the other object.

A *place object* is an object that typically contains other objects.

A *path* is a continuous line connecting one object to another object.

A *replacement rule* is a mechanism for defining general changes in a picture. A replacement rule has two parts: a pattern picture and a result picture.

OPSTrains Doctrine Rules:

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 five 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."

RO3: IF an object can move or be placed on top of an object that is too small to contain the object,
THEN draw an object that is big enough to contain the object, and specify that the object is to be "Hidden in Simulation."

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.

RO5: IF a place object is to be used to hold different types of objects 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 creating paths.

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.

RP2: IF two or more paths are connected to the same object, and
the paths are to be used for different purposes,
THEN label each path with a different name.

RP3: IF a set of paths are all used for a common purpose,
THEN label each of the paths with the same name.

RP4: IF making a path on which objects may travel in both directions,
THEN specify that the path is bidirectional.

RP5: IF making a path on which objects may travel in only one direction,
THEN specify that the path is unidirectional.

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.

RR4: IF an object is to traverse a path when a rule is executed,
THEN place that object on the path to be traversed.

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)

RR6: IF a set of objects in the pattern of a rule is to match objects with an identical but unknown display,
THEN make sure that each of these objects have an identical display themselves, and then specify that all of these objects are variable.

RR7: IF an object in the result picture of a rule should be identical to a variable object existing in the pattern picture,
THEN create the identical object in the result picture, and
specify that this object is a variable.

RR8: IF a rule should only be executed when a specific object does not exist,
THEN copy the object onto the pattern, and
specify that this object is to be negated.
(a big X will be placed over the negated object in the pattern)

The following rules are for executing and debugging the simulation.

RD1: IF all objects, paths, and rules have been specified,
THEN enter "Execute Mode."

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.

RD3: IF a particular rule is not behaving as expected,
THEN enter the Rule Editor for this rule and make corrections.

Showtrains
Overview and Doctrine as the Programmer Would Receive Them

John, June 1990

OVERVIEW

[O1, O==Overview] ShowTrains allows you to create animated graphical simulations on the Macintosh. They're "animated" because things move; they're "graphical" because the end result is a moving picture; and they're "simulations" because the activities on the screen depend on the properties of the world you have created — they're not just a series of predefined pictures displayed in rapid succession.

[O2] Here's a simple example that will introduce the important things you need to know to use ShowTrains. The task is to simulate a cat and a bird in a cartoon garden. As the simulation starts, the cat is asleep on a lawn chair. The bird flies down from its nest, hops around the yard, and finds a worm. It begins to eat the worm. The cat wakes up and sneaks up on the bird. But just as the cat arrives, the bird flies back to its nest. Figure 1 shows how the simulation will look when the bird first reaches the worm.

[O3] In ShowTrains, the bird, the cat, and the worm are *Objects*. These are the things that a ShowTrains simulation can display as moving, changing, being created or deleted, and interacting with each other.

[O4] The nest, the chair where the cat sleeps, and the spot where the bird eats the worm are ShowTrains *Places*. Whenever an object isn't moving, it has to be in a place. Whenever two objects interact, they have to be in a place. Notice that it's convenient to make the chair a Place in this ShowTrains simulation, even though it's an object in the real world.

[O5] The route that the bird travels as it flies down from the nest and wanders around the garden is a ShowTrains *Path*. The cat's route as it sneaks up on the bird is also a path, as is the bird's return route to its nest. Whenever an object is shown moving on the screen in ShowTrains, there is a path that defines where it moves.

[O6] Figure 2 shows the cat and bird simulation again, with the Places, Paths, and Objects marked. Notice the two additional items — a tree and a birdbath — that aren't important to the simulation. Those are part of the ShowTrains *Background* for this simulation. The Background is just a MacPaint picture that stays on the screen during the simulation, but doesn't interact with any of the Objects. It's sometimes convenient to use a background that looks the way the simulation should look, with invisible places and paths that control the movement and interaction of objects.

[07] There's one more important part of a ShowTrains simulation: the *Rules* that determine how objects will act. In the cat-and-bird simulation, there are the following rules:

- [O8] If the bird is in its nest and there's a worm in the garden, the bird should get on the path that leads to the worm. (Whenever a rule puts an Object on a Path, the Object will automatically follow that Path to its end, and the motion will be visible on the screen.)

- [O9] If the bird is in the same place as the worm, it should "eat" the worm — this rule really says that the worm should disappear and the bird should change to a bird with a worm hanging out of its beak.

- [O10] If the bird is eating the worm, then the cat should get on the path that leads to the bird.

- [O11] If the bird and the cat are together, then the bird should get on the path that leads to the nest.

[O12] To create each of these Rules in ShowTrains, we set up the objects on screen exactly the way they will be when the Rule should take effect, then move or modify the objects to show how they should change. In other words, we tell ShowTrains, "If the screen looks like *this*, then *make these changes*."

GUIDELINES FOR CREATING SIMULATIONS IN SHOWTRAINS

[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. [D9] **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.

ADDITIONAL GUIDELINES FOR DIFFICULT SITUATIONS

[D03] Sometimes you'll go through all of the above guidelines and find that they just aren't enough. Maybe the simulation doesn't work at all. Maybe there are so many cases that you don't think you'll ever be able to demonstrate the Rules for them all. Maybe you can understand exactly what is happening, but it isn't what you want and you can't think of any way to prevent it. This section describes some additional guidelines that may help in those situations.

Simulating Complex Movements

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.

12. [D12] **String Places Together Like Beads.** For many simulations, a long route is best represented by a single, one-way Path, perhaps interrupted occasionally by a small place. This is true even if the real world has cross-connected paths and return routes. For example, to show the bus route to the airport, a simple simulation might use just a single Path, starting at the bus station and ending at the airport. A slightly more complicated simulation could add places wherever the bus stopped for passengers. Either version would be much simpler than drawing Paths for every street, places for every intersection, and creating Rules to ensure that the bus took the right turns and travelled in the right direction.

13. [D13] **Put Path Ends Where Objects Should Stop.** The end of a Path can be located anywhere in a Place. Objects travelling down the Path will stop where the

Path ends. (If there is already an Object at that point, it will be pushed aside in a random direction.)

14. [D14] **Use Instantaneous Movement.** Don't forget that Rules can show instantaneous "movement" of an object from one place to another. If you want to simulate the action of the Star Trek transporter, you don't need a path.

Creating Simpler and More Powerful Rules

15. [D15] **Put Interacting Objects in the Same Place.** Rules can show Objects interacting no matter where the Objects are, but it's often simpler to allow interactions only when the Objects are in the same Place. For example, a Rule that says the bird eats the worm if they are in the same Place can apply anywhere in the simulation — but there's no simple way to write a single Rule saying a bird eats a worm if they are in any two adjacent Places.

16. [D16] **Add Things to the Condition for Specificity.** If a rule is firing when it shouldn't, then add more items to its condition. For example, on your first attempt to create the bird-eats-worm Rule, you might have highlighted just the bird and the worm as conditions. This rule would fire even if the bird and worm weren't in the same place. To fix the rule, edit it so that the bird and worm are in the same place, they are both highlighted, and the outline of the place is also highlighted.

17. [D17] **Remove Things from the Condition for Generality.** If a rule doesn't always fire when it should, it probably has too many things highlighted in its condition. For example, in the bird-eats-worm Rule, if the name of the place is highlighted, then the bird can only eat the worm in a place having that name.

18. [D18] **Use Marker Objects to Identify Places.** Sometimes it's useful to create Rules that identify a Place by the names of invisible "marker" objects that it contains, instead of using the Place's actual name. For example, you might want your simulated people (Objects) to eat in the kitchen (a Place) or the dining room (another Place). Of course, you could have two Rules, one that made them eat in a place named "kitchen" and one that made them eat in a place named "dining room." But a more versatile solution is to put an invisible object named "eating-place" in both places, and have a single Rule that fires for any place containing that object. This technique makes it easy to extend or modify the simulation. For example, you might move the "eating-place" object from the dining room to the patio in the summer.

19. [D19] **Use the Name or Graphic to Note an Object's State.** It's fairly common to have a moving Object that should behave one way at one time, another way at another. Often you can change the Object slightly so a different set of Rules will apply. For example, you may have one set of Rules that apply to objects named ANGRY DOG and another that apply to objects named SLEEPY DOG. You could get the same effect by using just the name DOG, but changing the Object's appearance.

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.

Repeating Patterns of Places/Paths/Objects/Rules

21. [D21] **Use Rule Copy.** The Rule Copy menu item will allow you duplicate a Rule, then change what its conditions refer to. Anything item changed in the conditions will also be changed in the actions. For example, say you have a Rule that says, "If there's a bird and a worm, delete the worm and change the bird to a fat bird." You can copy this rule, then select the Change Condition menu option, unhighlight "worm" and highlight "bird seed." The new Rule will show the bird eating bird seed — the action of deleting the worm will automatically be replaced by an action of deleting the bird seed.

22. [D22] **Use Block Copy with Rename.** The Block Copy menu item will allow you select a pattern of Places, Paths, and Objects and make a copy of it. When it makes the copy, ShowTrains will ask whether you want the items renamed and whether you want any associated Rules to be duplicated for the new names. You might use this, for example, if you were showing how a network of many identical

transistors operates. First create one transistor — using places for each of its three connections — along with the rules that define how it operates. Then duplicate it one or more times, using new names and duplicating the Rules. (You need to use new names and Rules so that input to the connections of the original transistor won't affect the behavior of the copies.) Now "wire" together the duplicates with additional paths, and run the simulation.

23. [D23] **Take Advantage of Symmetry By Moving Markers.** Sometimes you will have a symmetrical pattern in the simulation. You'll write a rule for one orientation, and you'll want it to apply to any orientation. For example, the simulation might show serving an apple pie, and the Rule might be to always take the piece counterclockwise from the last piece taken. To utilize the symmetry, place an invisible Object named MARKER with an "E" graphic into a wedge-shaped section of the pie pan, and put an invisible Object named MARKER with an "F" graphic into a section immediately counterclockwise of "E". Then create the Rule that fires when the segment marked E doesn't have pie in it and the segment marked F does. The action of the Rule is to serve the piece of pie from the F section. But this takes care of just one of the many possible configurations the pie may be in. To make this single Rule apply to the entire pie, create a Rule that rotates the marker Objects in the pan. Then the rule for the E-F pair of adjacent segments will have a chance to fire no matter where in the pie tin the pattern appears. (Rotating the markers full circle on each Rule cycle is tricky — see Appendix X for a detailed example.)

APPENDIX— SAMPLE SOLUTIONS

This appendix contains sample ShowTrains solutions to several problems. You may find tricks in these solutions that you can apply to your own problems.

X. Pie Serving — A Pattern-Matching Example

Some problems require Rules to match a pattern of Objects in Places. For example, you might be looking for any place on a chess board where a Knight could effect a capture. If you had to write a different set of Rules for every square on the board, the task would be hopeless. What you need is a way to write a single set of Rules in terms of one square, then apply that set to other squares — sort of like creating a cut-out template and moving it around to see where it fits.

The following problem demonstrates how pattern matching can be done in a very simple situation. The solution is really too complex for this problem, but the technique it demonstrates can be used effectively in more difficult simulations.

The problem is to simulate the serving of six precut wedges of pie. The simulation should show a pie tin, initially filled with pie; and a desert plate, initially empty. Whenever the desert plate is empty, a piece of pie should travel along a path from the pie to the plate. The first piece removed from the pie tin should be selected at random. After that, the piece removed from the pie tin should always be the piece counterclockwise from the previous one removed. Whenever the desert plate has a piece of pie on it, the piece should disappear. In this problem, the pattern to be matched is "empty wedge in some position, full wedge one space counterclockwise." Here's the solution.

1. Sketch the problem — a pie tin, a desert plate, and a trajectory over which the pie wedges will travel.

2. Create six Objects named PIECE-OF-PIE, with graphics in appropriate orientations.

3. Create a Place for each PIECE-OF-PIE — these Places together will represent the pie tin. Make them invisible, and name them each TIN-SECTION. Create two objects, both named MARKER, with graphics that look like the E and F (for "empty" and "full"). Put the E marker in one TIN-SECTION Place, and put the F marker in the TIN-SECTION Place counterclockwise from it. Put one PIECE-OF-PIE in every TIN-SECTION.

4. Create a Place named DESERT-PLATE, with a desert-plate graphic.

5. Create a single, invisible Place named ALL-TESTED, and create two invisible Objects named TESTED and place them in the ALL-TESTED Place.

6. Create a path from each TIN-SECTION place to the DESERT-PLATE. Name them all TO-DESERT-PLATE. Create a path from each TIN-SECTION place to the one counterclockwise from it. Name them all C-CLOCKWISE.

7. Now we're ready to demonstrate the Rules. First we need a Rule to get the first piece of pie. Remember that unspecified conditions match at random, so it's easy to get a random piece of pie:

- If there are six Objects named PIECE-OF-PIE in six Places named TIN-SECTION, then put one of the PIECE-OF-PIE Objects onto the Path named TO-DESERT-PLATE. (Be sure not to highlight the marker objects in this Rule.)

Now we need a Rule to consume the pie from the desert plate.

- If there's an Object named PIECE-OF-PIE in the place named DESERT-PLATE then delete the PIECE-OF-PIE.

The third Rule gets the correct piece of pie from the pie tin whenever the desert-plate is empty and the F marker is in a TIN-SECTION place with a PIECE-OF-PIE and the E marker is in an empty TIN-SECTION place.

- If there's an Object named PIECE-OF-PIE and an Object named MARKER with the graphic F together in any place, and there's an Object named MARKER with the graphic E in a different place together with no Object named PIECE-OF-PIE, and there's no Object named PIECE-OF-PIE in the place named DESERT-PLATE, then put the Object named PIECE-OF-PIE onto the path named TO-DESERT-PLATE.

The fourth Rule will let us know when all the Rules have fired once. Since all ShowTrains Rules fire (in some unspecified order) before any Rule can fire again, we create a Rule that will fire under any conditions. When that Rule has fired twice, we know that all other Rules must have fired at least once in the same period.

- If (no condition specified), then create a TESTED object in the Place named ALL-TESTED.

The fifth Rule will move the marker objects around the pie tin, so the second Rule can be tried once for every possible position. We don't want to move the markers until we've given the second Rule a chance to fire — keeping track of that is the function of the ALL-TESTED Place. (Note that the markers may be moving while the pie is being eaten, and we can't say where they'll be at the start of each serving — but this doesn't matter.)

- If there are two TESTED Objects in the Place named ALL-TESTED, then delete the Objects named TESTED, and put the Objects named MARKER on the Paths named C-CLOCKWISE (i.e., you should demonstrate how the E marker is placed on the path named C-CLOCKWISE that leads out of a TIN place, and the

F marker is placed on the path named C-CLOCKWISE that leads out of a TIN place).

Summary: Here's what this solution demonstrates. To match a pattern of Objects in Places, use marker Objects to name the Places of interest, write the Rules in terms of the markers, and then move the markers so the same Rules can apply to many configurations of Places.

ZeroTrains Doctrine

Clayton, 6.23.90

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

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.

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

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.

If ends of different paths play the same role in different parts of the model give them the same name.

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.

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.

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.

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.

If a trigger refers to some predicate P, create an object that will be present just when P is true.

If P is of the form A and B, for other predicates A and B, create objects for A and B and test for the presence of both.

If P is of the form A or B write two rules, one triggered by an object associated with A and one triggered by an object associated with B.

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.

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.

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.

You may want to trigger on an object arriving on some particular path. unfortunately this cannot be done directly. You must provide distinctive variants of the objects you wish to do this for, one for each arrival path, and make sure this variant is used when the object is put on the path. Then the variant that arrives indicates which path it came on.

Would like test for what path mouse came in on. Doctrine says must have object be different according to path. So have back and ahead, with variants to indicate arrival through L, R, and Out, giving six mouse

variants: ahead-fromR, ahead-fromL, ahead-fromOut, back-fromR, back-fromL, back-fromOut. String variants possible are LR, LOut, RL, ROut, OutR, OutL, named as from-to. Also need marker for deadend.

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.

If events should occur in a particular order, make an object that is created when the first event occurs and test for it in the trigger for the second event.

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.

In the *overlap* approach you create places which enclose each Pi but also enclose a place in which the general state information is kept. These new places thus all overlap on this latter place. Put a catalyst object in each new place, and add a test for this to the condition to be tested, along with a test for the relevant general state information.

If the problem requires processsing just one case in which the condition is found, make sure that processing this case modifies the general state information in such a way that no further cases will be processed.

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.

This latter point is one key to choosing between the overlap and messenger approaches. In the overlap condition it is complicated to determine when the condition has not been satisfied anywhere, whereas this is easy in the messenger approach. Another advantage of the messenger approach is that it is easy to order the places being examined so that the first Pi in this order is the one processed, while this cannot be done in the overlap approach. On the other hand, if these issues do not arise, and the required pattern of overlaps is simpler to set up than the paths needed to connect the Pi, the overlap approach should be used.

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.

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.

Residual notes

change X in P to just X if no other place could contain X

Idea: require absence of objects to be created.(LeChatelier's principle) Could this be used to test absence? What useful behaviors would be excluded?

have I dealt adequately with distinctions among O occurs anywhere, O occurs in some specified place, O occurs in the same place as O'?

3. Raw Problems

The six raw problems described in this section were used to develop and test the sufficiency of the doctrine for each design.

Chemtrains – Raw Problems

June 4, 1990

Here's a description of each problem, expressed as a task that a ChemTrains programmer must complete. (The "user" is the person who will interact with a completed simulation.)

BUNSEN BURNER

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, medium, 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 medium position, a medium 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.

OFFICE

Purpose: Demonstrate several aspects of document flow in an office.

Task Description: Show four rooms on the screen: the mail room and the offices of Joe, Pat, and Bill. Each office should have the appropriate person in it, and the mail room should have a copier in it. Allow the user to create a memo in any office. If the memo is addressed "TO X," where X is one of the office holders, then the memo should automatically travel to the mail room and from there to office of its addressee. If a memo is addressed "TO X" and it is in X's office, then it should be destroyed. If the memo is addressed "TO OFFICE," then it should travel to the mail room, copies addressed "TO JOE," etc., should be made for all of the office holders (including the sender of the memo), and the copies should travel to the appropriate offices. The original should be destroyed in the copy process.

MAZE

Purpose: Show how a mouse with intelligence (but a poor sense of smell) explores a maze to find cheese.

Task Description: Show a simple maze, with a mouse at its entrance and cheese at some distant point. (See graphic of maze.) The mouse should move through the maze. If it reaches a dead end, it should backtrack and try a different route. It should do this intelligently – i.e., it should not randomly retry routes that it has already found to be dead ends. If it reaches the cheese (it must, eventually), it should stop and eat the cheese. The mouse should be able to find the cheese in any dead end or doorway.

Desiderata: The technique used by the mouse should work without change if a different maze is used.

PETRI NET

Purpose: Model a given Petri net, using techniques that would allow any arbitrary Petri net to be modelled.

Task Description: Model the Petri net shown in the attached graphic. Allow the user to create and destroy tokens at any input place in the net. The Petri net should react to tokens in the following manner: if both input places to a transition are occupied by tokens, then the transistion should "fire." Whenever a transition fires, it places a token on its output place, and the tokens that caused it to fire disappear. Note, however, that a given token can only cause the creation of one other token. That is, if a token occupies the input places for *two* transitions, then only one of the transistions can use that input token to create a new output token – which of the two junctures gets to use it should be randomly determined.

Desiderata: The Petri net should be created in a way that will allow it to be easily modified.

TURING MACHINE

Purpose: Model a given Turing machine, using techniques that would allow any arbitrary Turing machine to be modelled.

Task Description: A Turing machine consists of the following:

- A Tape. The tape is divided into distinct squares, into which may be written arbitrary symbols. The symbol # indicates a blank square.

- A Head. The head is situated, at any one moment, on a single square of the tape. The head can read and write symbols on the square where it is currently located. It can also move the machine, one square at a time, to other squares on the tape.

- A finite set of internal States. The machine must be in one of these states at all times.

- A set of Rules. The rules are of the form:

  IF: the square where the Head is situated contains the symbol "T-1"
  and the machine is in State "S-1"
  THEN: Change the state to "S-2"
  Write a "T-3" into the current square (replacing the previous contents)
  Move the machine to the left one square.

(The rule can move the machine one cell to the right, one cell to the left, or not move it at all. The tape is theoretically endless, so the machine never runs out of tape.)

The task is to model a Turing machine with a tape in which five adjacent squares initially contain either a one or a zero, and the rest of the tape is blank. (See graphic.) The head is initially located to the left of the first cell in which something is written. The rules should cause the machine to modify the tape as follows: if there are an even number of ones on the tape, then the machine should write a zero in the cell just to the right of the string of ones and zeros; if there are an odd number of ones, the machine should write a one in the far right cell. Don't worry about modelling an endless tape – just show as many cells as you need for this task.

Desiderata: The rules and the tape should be easy for the user to modify, so different sets of rules and configurations of tapes can be tested.

TIC TAC TOE

Purpose: Create a Tic Tac Toe board and an intelligent opponent against which the user may play.

Task Description: On the screen, show a Tic Tac Toe grid. Allow the user to (1) start a new game at any time, and (2) place an "X" marker at any position on the board when it is his/her turn. After the user places an "X" the machine should respond by placing an "O" in the strategically best position on the board. Assume the user is honest – don't worry about preventing illegal moves.

Desiderata: The strategy should be specified as compactly as possible.

4. Walkthroughs of Raw Problems

These walkthroughs demonstrate how each competing design, with its doctrine, can yield a solution to the raw problems.

Micro Walkthroughs for OPSTrains.

- Brigham, 6/14/90

Bunsen Burner Problem in OPSTrains.

RO1: draw a bunsen burner, beaker, and control panel, made up of three separate rectangles. (shown in figure 1a.)
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. (shown in figure 1b.)
RR1: add a rule named "Turn Flame to medium"
copy the "medium" 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 medium 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.
(the resulting rule is shown in figure 1c.)
RR1: add a rule named "Turn Flame to high"
follow the same steps as in making "Turn Flame to medium." ...
(the resulting rule is shown in figure 1d.)
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.
(the resulting rule is shown in figure 1e.)
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.
(the resulting rule is shown in figure 1f.)
RR1: add a rule named "Change to gas"
follow the same steps as in making "Change to liquid" ...
(the resulting rule is shown in figure 1g.)
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.
(the resulting rule is shown in figure 1h.)

Office Problem in OPSTrains.

RO1: draw the mail room and three offices. In each office create an object showing the occupant's name. Each office is connected to the mail room with bidirectional paths. Also an example memo may be drawn with some contents and an object specifying the address.
(the resulting picture is shown in figure 2a.)
RR1: add a rule named "Send Mail"
copy to the pattern and result pictures the mail room, an office, a path connecting these two place objects, and a memo in the office.
RR4: place the memo on the path, so that any memo in an office will be sent to the mail room.
(the resulting rule is shown in figure 2b.)
RR1: add a rule named "Distribute Mail"
copy to the pattern and result pictures the mail room, an office, a path connecting them, and a memo in the mail room.
RR6: make the address of the memo and the address of the connected office identical, and specify that these objects are variable.
RR4: place the memo on the path connecting the mail room to the office.
(the resulting rule is shown in figure 2c.)
RR1: add a rule named "Read mail"
copy to the pattern and result pictures an office that contains a memo.
RR6: make the address of the memo and the address of the office identical, and specify that these objects are variable.
RR2: remove the memo from the office.
(the resulting rule is shown in figure 2d.)
RR1: add a rule named "Make copies of office mail"
copy to the pattern and result pictures the mail room, an office, a path connecting them, and a memo to "office."
RR3: copy the memo again into the mail room, change the address of the copied memo to the address of the connected office.
RR5: make the contents of the memo a variable.
RR7: make the contents in both memos in the result picture identical variables to the contents shown in the pattern.
RR5: make the address of the office a variable.
RR7: make the address of the copy of the memo the same variable object as the address of the office.
(the resulting rule is shown in figure 2e.)
RD1: run the simulation, notice that mail entering an office is immediately sent back to the mail room.
RD2: the rule named "Send mail" is executing instead of the rule named "Read mail," when a memo is sent to its final destination. Reorder the rules so that "Read mail" has priority over "Send mail."
RR3: create an object that represents that a memo has been sent once, and place it on the memo in the result picture.
RR8: make a copy of the "sent once" object on the pattern of the picture and negate it.

Maze Problem in OPSTrains.

RO1: draw a maze, mouse, & cheese.
(the resulting picture is shown in figure 3a.)
RO2: draw an empty rectangle for every decision point, dead end and opening in the maze.
RP1: draw a path for every two maze places that can connect through a corridor of the maze.
RP4: make each path bidirectional.
(the added hidden objects and paths are shown in figure 3b.)
RR1: add a rule named "Mouse goes forward."
copy the mouse, the place object containing the mouse, a path connecting from this object, and the other place object connected to this path.
RR3: create an object that represents a piece of string and place it on the originating place of the mouse in the result picture.
RR4: place the mouse on the path in the result picture, so that it will move forward.
RR8: place a copy of the string object onto the destination place of the mouse in the pattern and specify this object to be negated, so that the mouse will not go forward over the string.
RR1: add a rule named "Mouse goes backward."
copy the mouse, the place object containing the mouse, a path connecting from this object, and the other place object connected to this path onto the pattern and result picture. Add a string object to the destination place in the pattern of the rule.
RR2: remove the string from the destination on the result picture.
RR3: create an object that represents whether a place has been seen completely and reinvestigation is unnecessary. Place this object on the originating place of the mouse on the result picture.
RR4: place the mouse on the path in the result picture, so that it will move backward. (the resulting rule for going backward is shown in figure 3d.)
RR8: re-edit the rule named "Mouse goes forward."
place a copy of the "seen marker" object onto the destination place of the mouse in the pattern and specify this object to be negated, so that the mouse will not go forward over a seen place. (the resulting rule for going forward is shown in figure 3c.)
RR1: add a rule named "Mouse eats cheese."
copy a maze place object onto the pattern of the rule. Copy the mouse and the cheese into this place. Copy these also into the result picture.
RR2: Remove the cheese from the result picture.
(the resulting rule for eating the cheese is shown in figure 3e.)
RD1: Execute the simulation. Realize that mouse runs the maze but doesn't eat the cheese.
RD2: Enter the Rule Order Editor, reorder the importance of the rules so that the "Mouse eats cheese" is before "Mouse goes backward."

Petri Net Problem in OPSTrains.

RO1: draw a picture of a petri net, including all transitions, their inputs and outputs. draw a storage area for tokens that can be used on the petri net. (the resulting picture is shown in figure 4a.)
RO3: create a place object that is big enough to contain input tokens and copy it onto all nodes.
RO3: create a place object that is big enough to contain input tokens and copy it onto all the transitions.
RP1: draw a path from every node place to the appropriate transition place.
RP1: draw a path from every transition place to the appropriate output node place.
RP5: Specify that all the paths are unidirectional.
(the resulting picture is shown in figure 4b and 4c.)
RR1: add a rule named "Consume tokens."
place two tokens on the inputs two a single transition.
copy that transition, its two input places, the paths that connect them onto the pattern picture, and the two tokens onto the pattern picture. copy these objects onto the result picture.
RR4: place the tokens on the paths connecting to the transition.
(the resulting rule is shown in figure 4d.)
RR1: add a rule named "Continue token."
place two tokens on a transition. copy that transition, its output place, the path that connects to the output place, the two tokens, and the rectangle that holds input tokens onto the pattern picture. copy these objects onto the result picture.
RR4: place one token on the path that leads to the output place.
RR2: remove the other token from the transition.
RR3: add a token to the rectangle that holds tokens.
(the resulting rule is shown in figure 4e.)

Turing Machine Problem in OPSTrains.

RO1: create a picture of a Turing Machine, including the tape, the contents of the tape, the tape head a place for the state, and the current state. (the resulting picture is shown in figure 5a.)
RO2: draw rectangle objects that can contain just the contents of a square and a tape head. hide them.
RP1: make paths connecting tape rectangles to adjacent tape rectangles.
RP2: since two paths are connected to a single tape rectangle with different purposes, label the two paths "r" for right and "l" for left.
RP5: make the "r" & "l" paths unidirectional.
RP1: make another set of paths connecting the tape rectangles so that the tape head can move either right or left from any object.
RP3: label one set of paths "r."
RP3: label the other set of paths "l."
RP5: make all paths unidirectional.
(the resulting picture is shown in figure 5b.)
RR1: add a rule named "Move on even over 0."
copy onto the pattern and result pictures the state container, an "even", the tape head, a tape contents rectangle, and an "r" path connecting to the contents rectangle.
RR4: place the tape head on the "r" path in the result picture.
(the resulting rule is shown in figure 5c.)
RR1: add a rule named "Move on even over 1."
copy onto the pattern and result pictures the state container, an "even," the tape head, a tape contents rectangle, and an "r" path connecting to the contents rectangle.
RR4: place the tape head on the "r" path in the result picture.
RR2: remove the "even" object from the state container.
RR3: create an object "odd," and add it to the state container.
(the resulting rule is shown in figure 5d.)
RR1: add a rule named "Move on odd over 0."
follow analogous steps from "Move on even over 0." ...
(the resulting rule is shown in figure 5e.)
RR1: add a rule named "Move on odd over 1."
follow analogous steps from "Move on even over 1." ...
(the resulting rule is shown in figure 5f.)
RR1: add a rule named "Report even."
copy onto the pattern and result pictures the state container, the state, the tape head, an "even", and a tape contents rectangle.
(the resulting rule is shown in figure 5g.)
RR2: remove the "#" from the tape contents rectangle in the result.
RR3: add a "0" to the tape contents rectangle in the result.
RR1: add a rule named "Report odd."
follow analogous steps from "Report even." ...
(the resulting rule is shown in figure 5h.)

Tic Tac Toe Problem in OPSTrains.

RO1: draw a picture of the tic tac toe board, a bin to hold X tokens that the user may draw from, and places to denote the outcome of the game, and the status of the current game. (the resulting picture is shown in figure 6a.)
RO2: no place objects exist for the individual squares of the board, so 9 squares are created on the board and hidden.
RO4: one significant difference between board squares is that each square may be in one of three different rows: create three objects "r1," "r2," and "r3" for each row. place the appropriate one of these objects in each square.
RO4: create three objects "c1," "c2," and "c3" to denote the difference in columns. place the appropriate one of these objects in each square.
RO4: create two objects "d1" and "d2" to denote what diagonal that a square is on. place the appropriate one of these objects in each square.
(the resulting picture with hidden objects is shown in figure 6b.)
RR1: add a rule named "Detect X win."
copy onto the pattern and result pictures the object places that describe the status of the game with "O" to move, and three board squares with an "X" and a an "r1" in each square.
RR2: remove the "X" and "O" from the "Turn" and "Wait" boxes in the pattern.
RR3: add the "X" to the "Won" box, and the "O" to the "Lost" box.
RR6: specify that the row name "r1" is a variable for the three squares.
RR7: specify that the row names in the result are also variable.
The resulting rule looks like:
(square brackets denote a rectangle;
angle brackets denote a variablized object;
and a minus sign denotes a negated object)

```
Rule: Detect X win.
      [Turn O] [Wait X] [Won] [Done]
      [<r1> X] [<r1> X] [<r1> X]
     =>
      [Turn] [Wait] [Won X] [Lost O]
      [<r1> X] [<r1> X] [<r1> X]
```

RR1: add a rule named "Detect O win."
follow analogous steps as in building "Detect X win."
The resulting rule looks like:

```
Rule: Detect O win.
      [Turn X] [Wait O] [Won] [Done]
      [<r1> O] [<r1> O] [<r1> O]
     =>
      [Turn] [Wait] [Won O] [Lost X]
      [<r1> O] [<r1> O] [<r1> O]
```

RR1: add a rule named "Detect draw."
copy onto the pattern and result pictures the object places that describe the status of the game with "O" to move, and all 9 squares filled in with 5 "X"s and 4 "O"s.
RR2: remove the "X" and "O" from the "Turn" and "Wait" boxes in the pattern.
RR3: add the "X" and the "O" tokens to the "Draw" box.
The resulting rule looks like:

```
Rule: Detect draw.
      [Turn O] [Wait X] [Draw]
      [X] [X] [X] [X] [X] [O] [O] [O] [O]
    =>
      [Turn] [Wait] [Draw O X]
      [X] [X] [X] [X] [X] [O] [O] [O] [O]
```

RR1: add a rule named "Clear X tokens"
copy onto the pattern and result pictures the "Turn" box, the "X" holding bin, and a single board square containing an "X."
RR8: copy an "X" onto the turn box and negate it.
RR8: copy an "O" onto the turn box and negate it.
RR2: remove the "X" from the board square.
RR3: add an "X" to the bin.
The resulting rule looks like:

```
Rule: Clear X tokens.
      [Turn -X -O] [Bin] [X]
    =>
      [Turn] [Bin X] []
```

RR1: add a rule named "Clear O tokens"
analogous to "Clear X tokens."
The resulting rule looks like:

```
Rule: Clear O tokens.
      [Turn -X -O] [O]
    =>
      [Turn] []
```

RR1: add a rule named "Detect user's move"
copy onto the pattern and result pictures a square with an "X" and the "Turn" and "Wait" boxes.
RR2: remove the "X" and "O" from "Turn" and "Wait"
RR3: add "X" to the "Wait" box and "O" to the "Turn" box.
The resulting rule looks like:

```
Rule: Detect user's X move.
      [X] [Turn X] [Wait O]
    =>
      [X] [Turn O] [Wait X]
```

RR1: add a rule named "Play a win move"
copy onto the pattern and result pictures the "Turn" box with an "O," the "Wait" box with an "X," and three board squares with an "r1." Two of the board squares must have an "O."
RR6: specify that the row name "r1" is a variable for the three squares.
RR7: specify that the row names in the result are also variable.
RR8: copy an "X" onto the empty square and negate it.
RR8: copy an "O" onto the empty square and negate it.
RR2: remove the "O" and "X" from "Turn" and "Wait"
RR3: add "O" to the "Wait" box and "X" to the "Turn" box.
RR3: add "O" to the empty square for the winning move.
The resulting rule looks like:

```
Rule: Play a win move.
      [Turn O] [Wait X] [<r1> O] [<r1> O] [<r1> -O -X]
     =>
      [Turn X] [Wait O] [<r1> O] [<r1> O] [<r1> O]
```

RR1: add a rule named "Play a block move"
follow analogous steps taken in creating "Play a win move"
The resulting rule looks like:

```
Rule: Play a block move.
      [Turn O] [Wait X] [<r1> X] [<r1> X] [<r1> -O -X]
     =>
      [Turn X] [Wait O] [<r1> X] [<r1> X] [<r1> O]
```

RR1: add a rule named "Play a force win"
follow analogous steps taken in creating "Play a win move"
except that two rows that have a common square are detected.
The resulting rule looks like:

```
Rule: Play a force win move.
      [Turn O] [Wait X]
      [<r1> O] [<r1> -X -O] [<r1> <c1> -X -O]
      [<c1> -X -O] [<c1> O]
     =>
      [Turn X] [Wait O]
      [<r> O] [<r>] [<r> <c> O]
      [<c>] [<c> O]
```

RR1: add a rule named "Play center" shown here:

```
Rule: Play center.
      [Turn O] [Wait X] [r2 c2 -X -O]
     =>
      [Turn X] [Wait O] [r2 c2 O]
```

RR1: add a rule named "Play anything" shown here:

```
Rule: Play anything.
      [Turn O] [Wait X] [-X -O]
     =>
      [Turn X] [Wait O] [O]
```

RD1: the tic tac toe player will work when an X token is dragged into the "Turn" box and an O token is dragged into the "Wait" box. This simulation will only allow the user to play "X" and the computer to play "O." The simulation could be extended to allow the computer and user to play either by creating a "Computer plays" box and a "User plays" box and by variablizing the player tokens in many of the rules.

This simulation is deficient in playing perfect Tic Tac Toe. There are a couple of cases where the user can win.

Also the order of the rules are created in this walkthrough in a way that will work. There are a couple of important aspects about the rule ordering. The rules "detect X win" and "detect O win" must appear before "detect draw," before "detect user's X move," and before any of the play making rules. The rules "clear X tokens" and "clear O tokens" may appear anywhere in the rule ordering. The playing rules must appear in the exact order shown: "play a win move," "play a block move," "play a force win move," "play center," and "play anything." These rules are ordered in the importance of the play making.

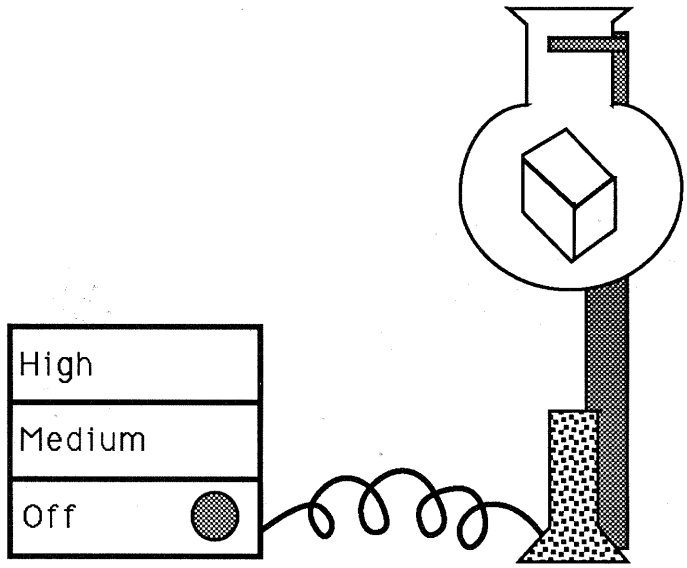


Figure 1a: Initial Beaker Drawing

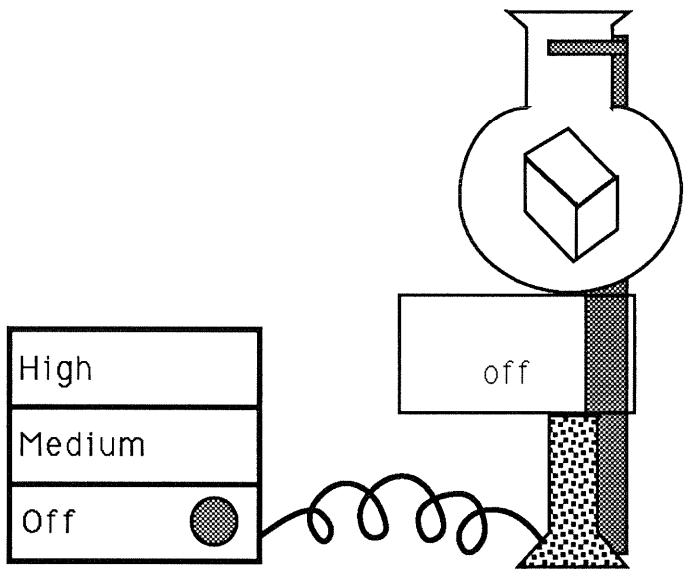


Figure 1b: Beaker with additional objects.

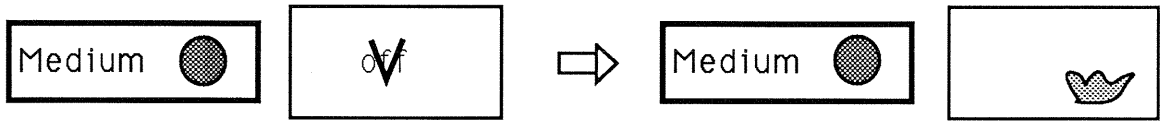


Figure 1c: Bunsen Burner Rule - Turn Flame to Medium

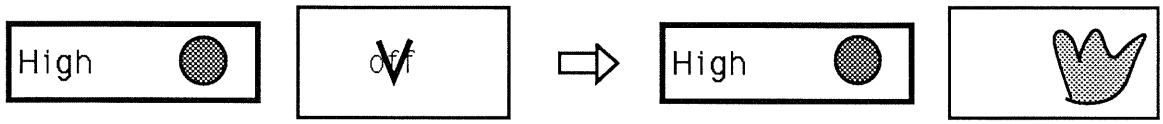


Figure 1d: Bunsen Burner Rule - Turn flame to high

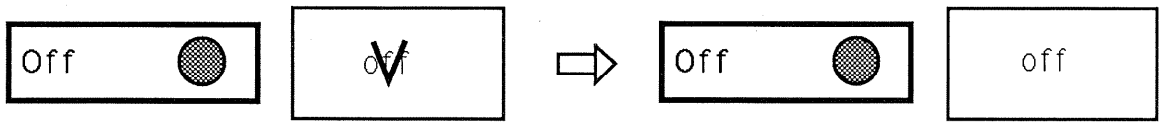


Figure 1e: Bunsen Burner Rule - Turn flame off

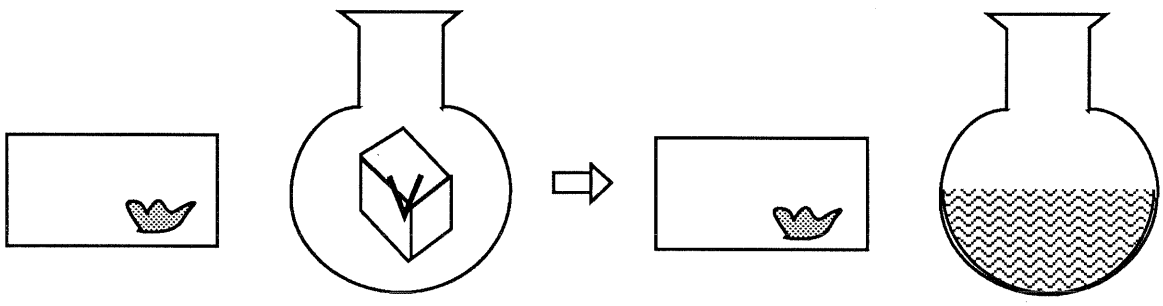


Figure 1f: Bunsen Burner Rule - Change to Liquid

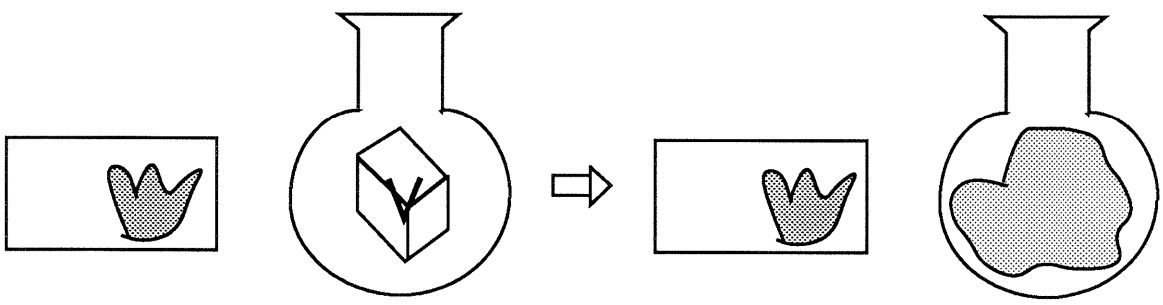


Figure 1g: Bunsen Burner Rule - Change to Gas

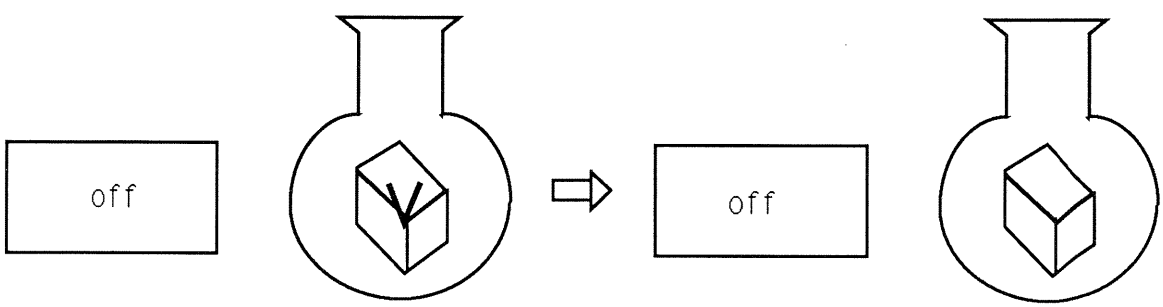


Figure 1h: Bunsen Burner Rule - Change to Solid

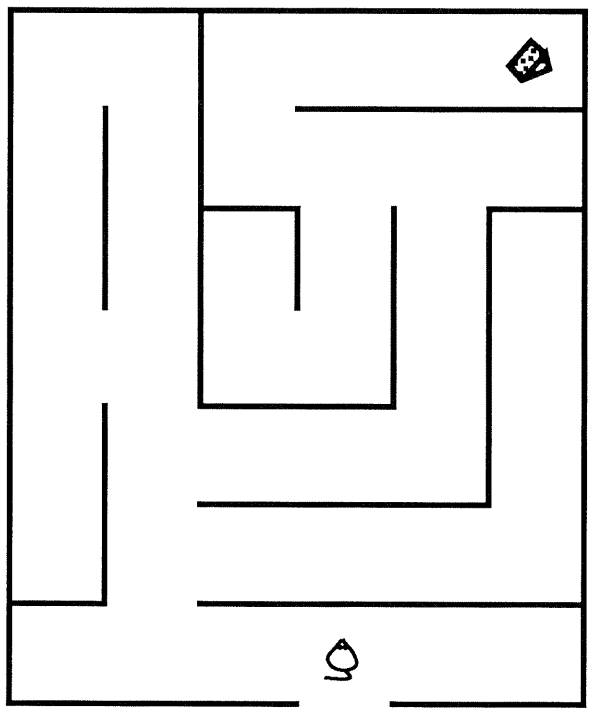


Figure 3a: Maze Picture

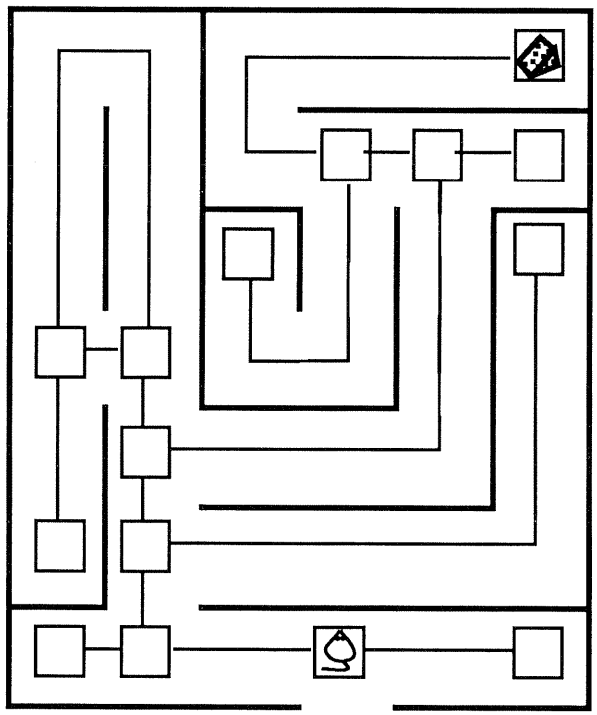


Figure 3b: Maze picture with hidden objects & paths.

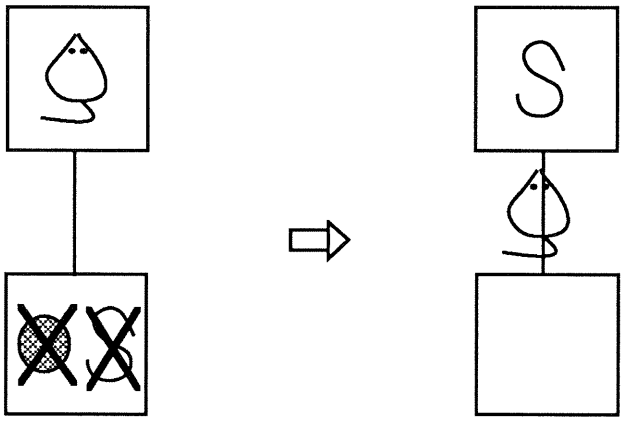


Figure 3c: Maze Rule - Mouse goes forward

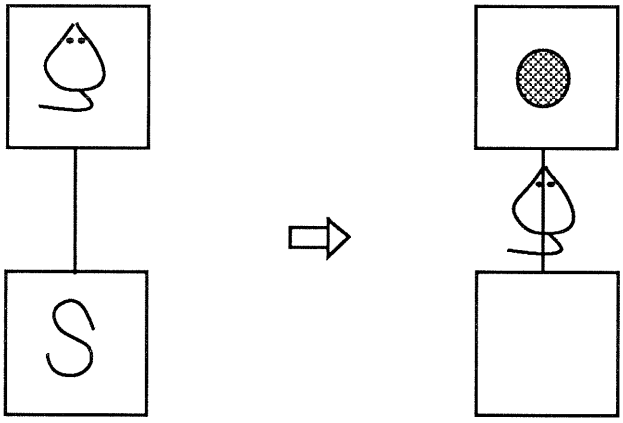


Figure 3d: Maze Rule - Mouse goes backward

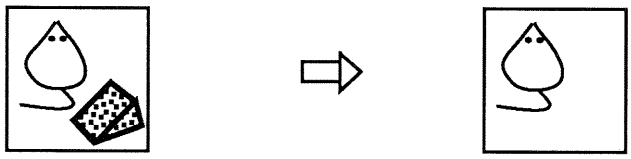


Figure 3e: Maze Rule - Mouse eats cheese

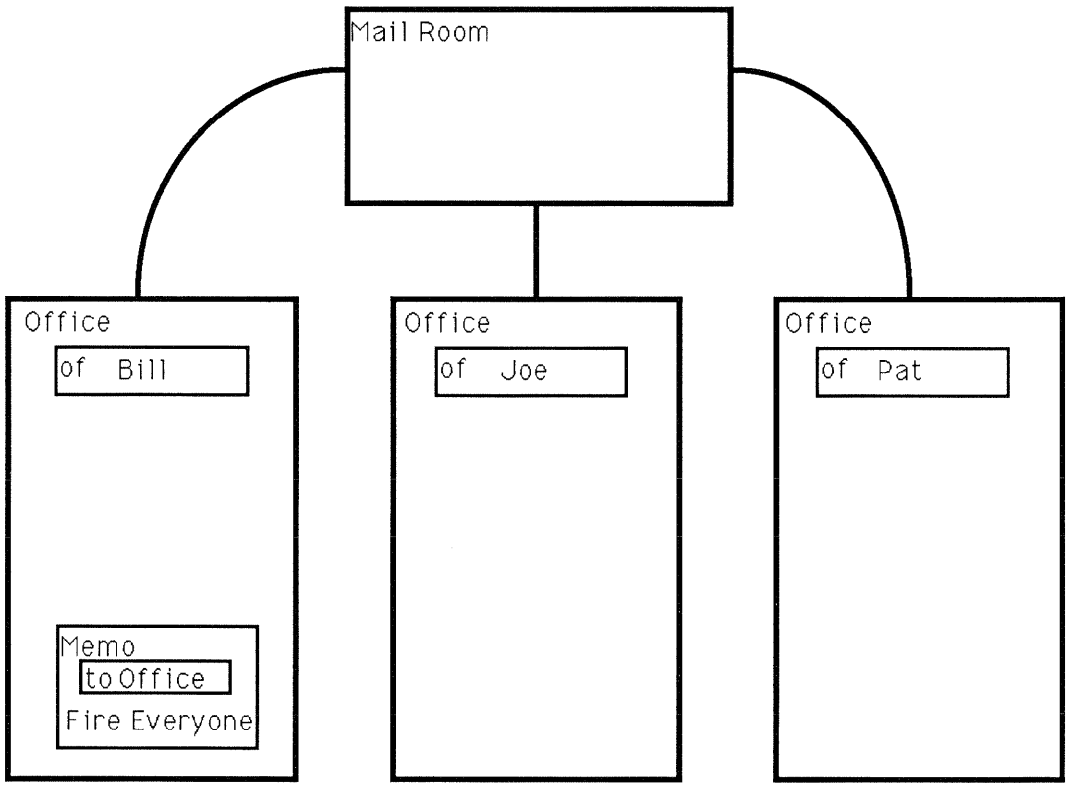


Figure 2a: Office Picture

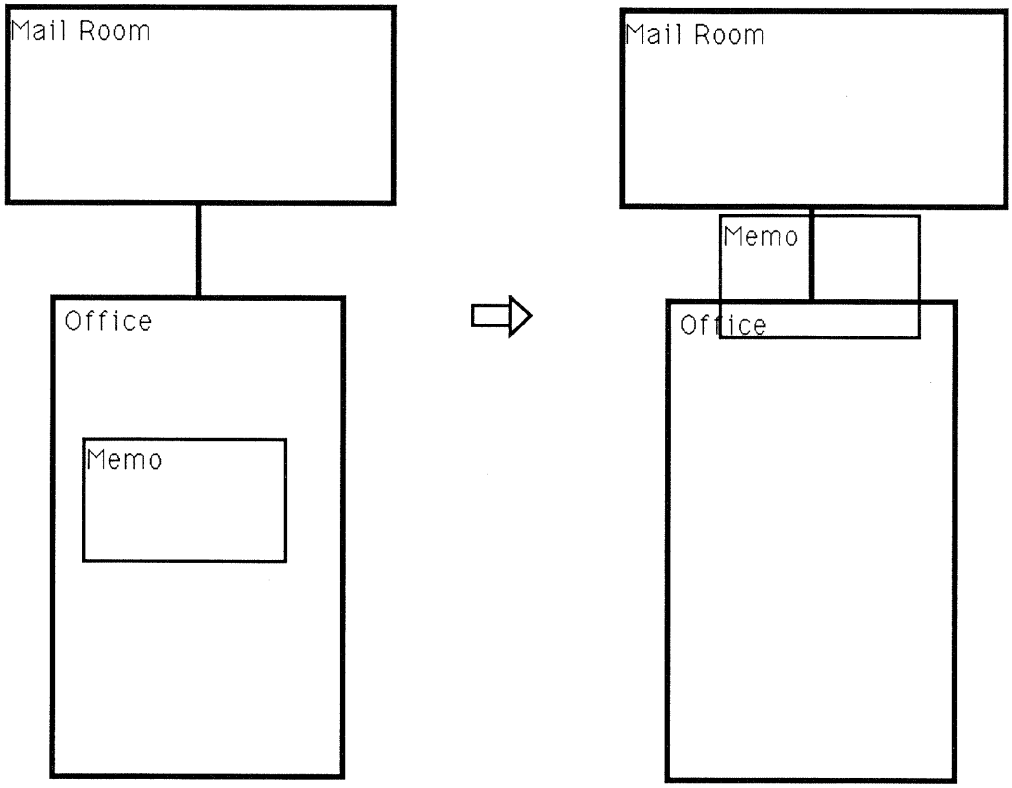


Figure 2b: Office Rule - Send Mail

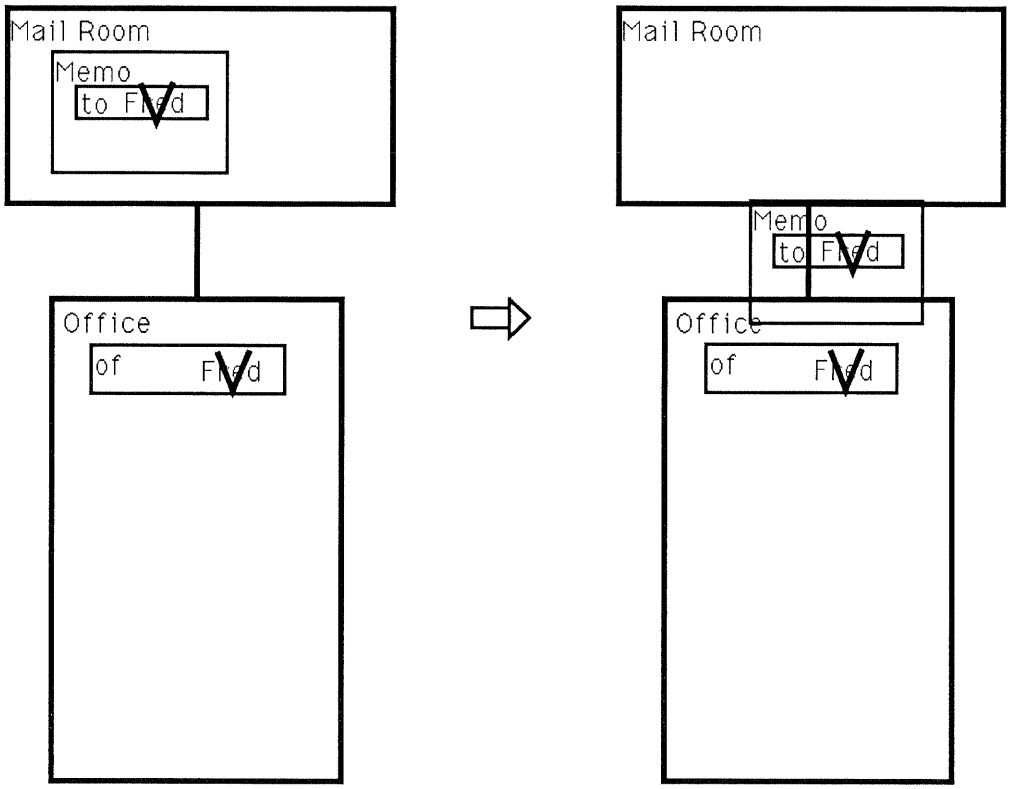


Figure 2c: Office Rule - Distribute Mail

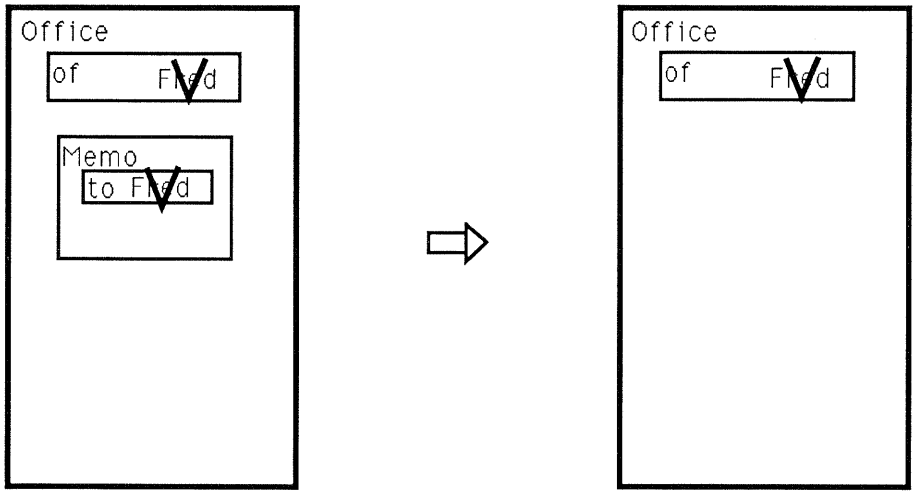


Figure 2d: Office Rule - Read Mail

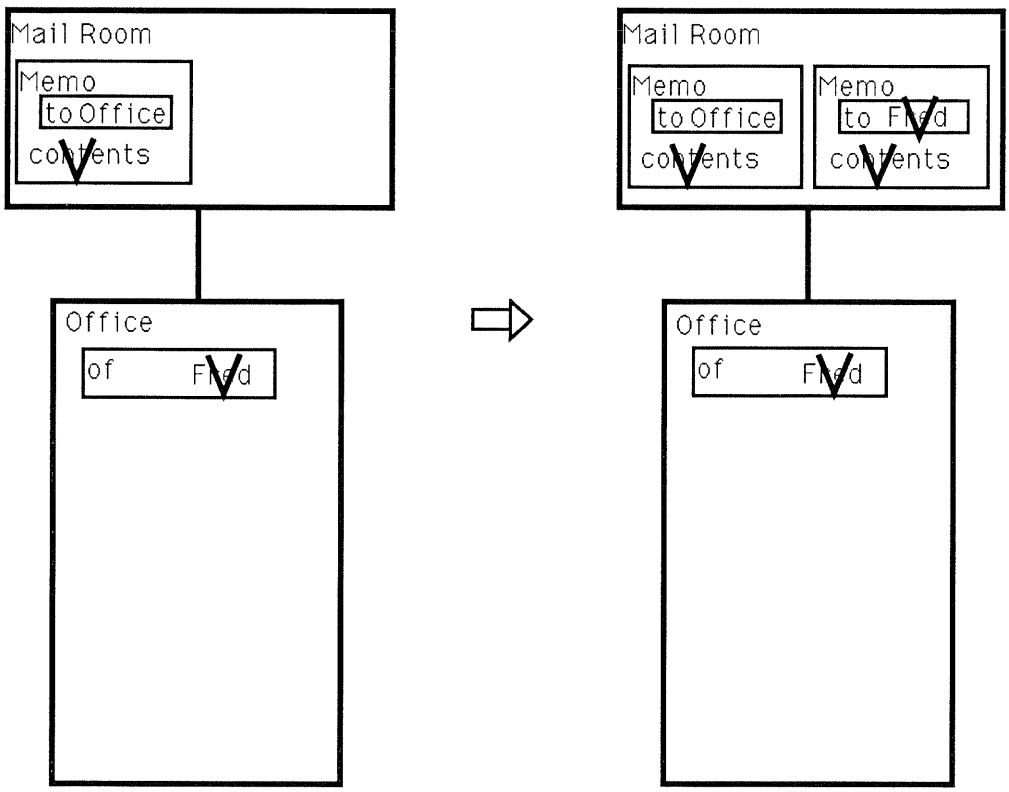


Figure 2e: Office Rule - Make copies of office mail.

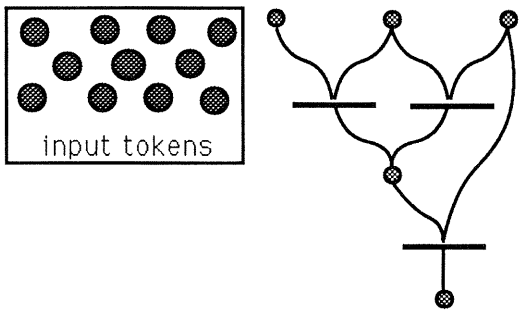


Figure 4a: Petri Net Picture

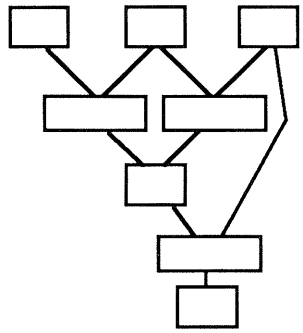


Figure 4b: Hidden objects & paths.

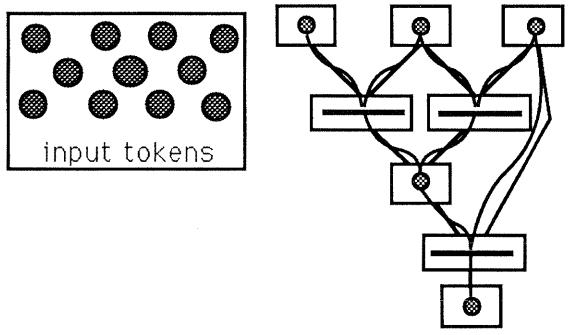


Figure 4c: Picture + hidden objects & paths.

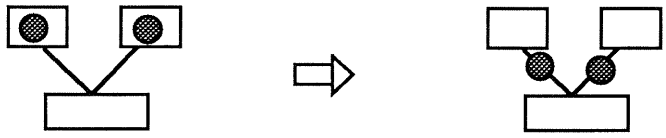


Figure 4d: Petri Net Rule - Consume tokens

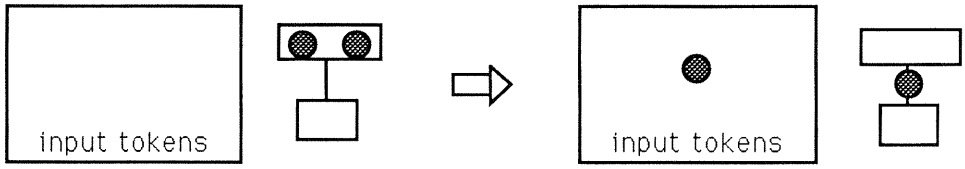


Figure 4e: Petri Net Rule - Continue token

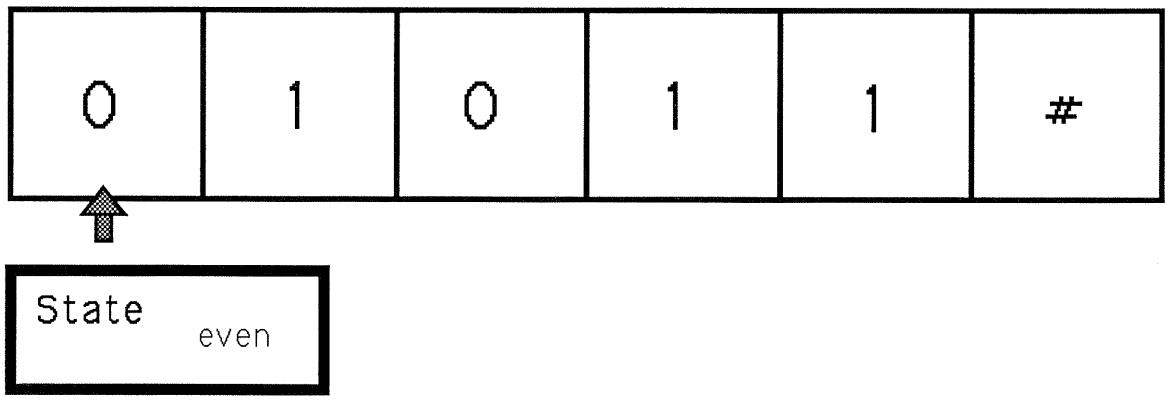


Figure 5a: Picture of Turing Machine

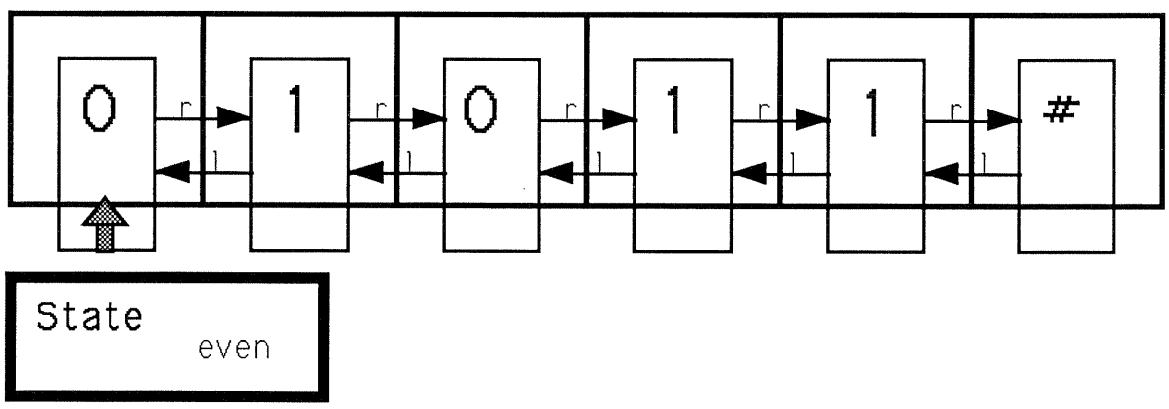


Figure 5b: Picture of Turing Macine with hidden objects & links

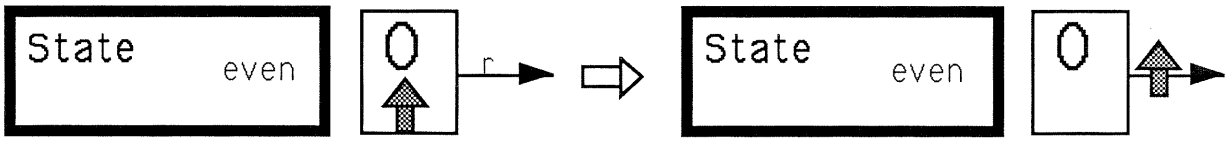


Figure 5c: TM rule - Move on even over 0

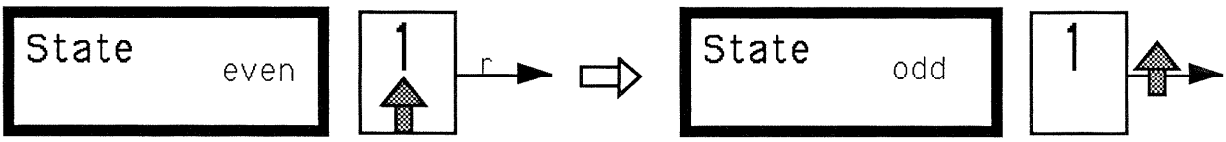


Figure 5d: TM rule - Move on even over 1

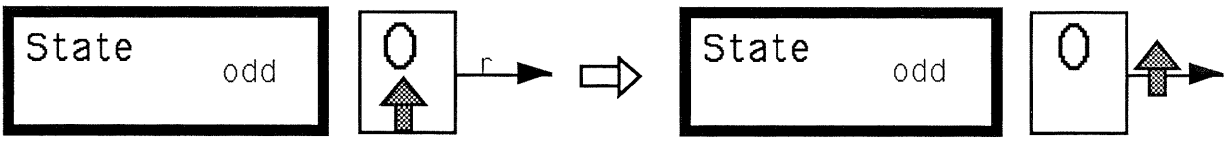


Figure 5e: TM rule - Move on odd over 0

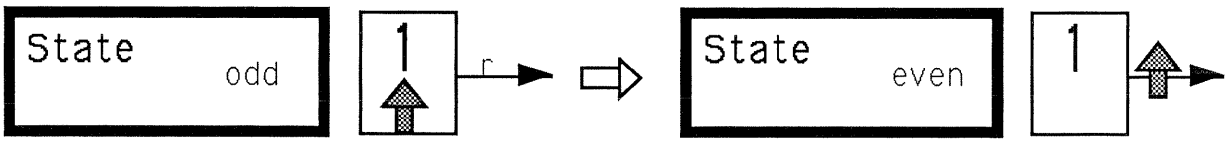


Figure 5f: TM rule - Move on odd over 1

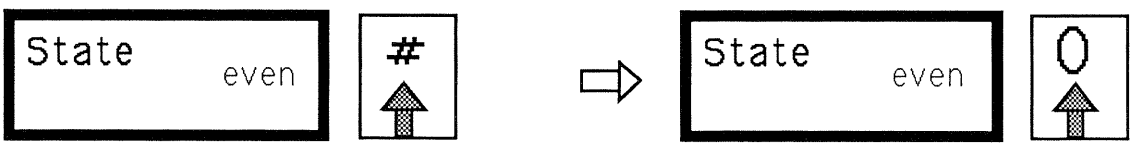


Figure 5g: TM rule - Report even

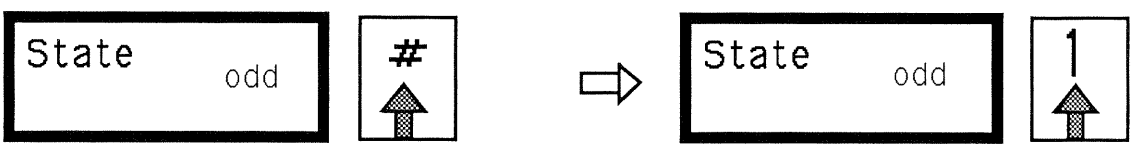


Figure 5h: TM rule - Report odd

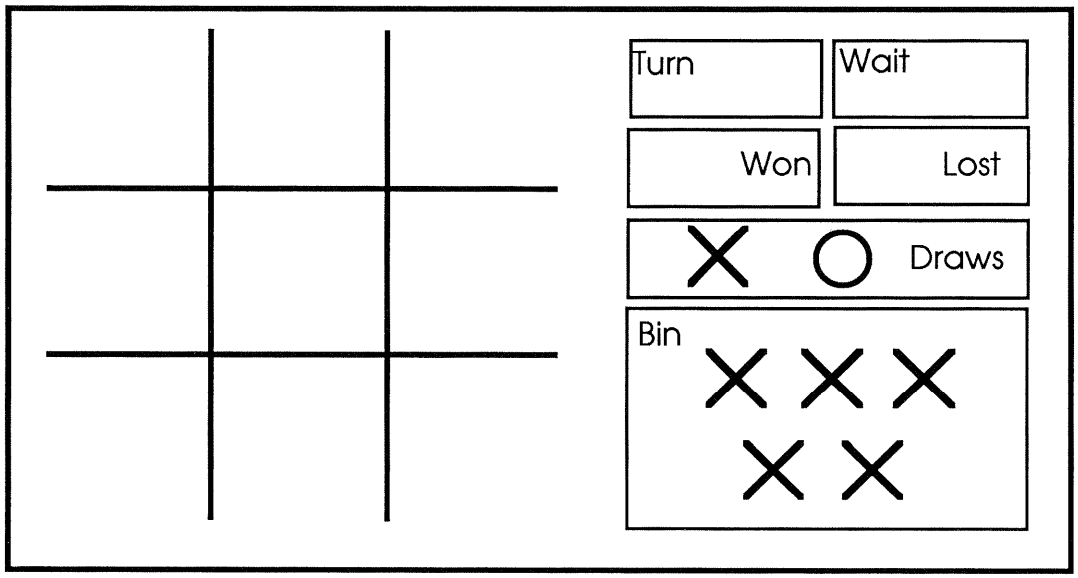


Figure 6a: Tic Tac Toe Picture

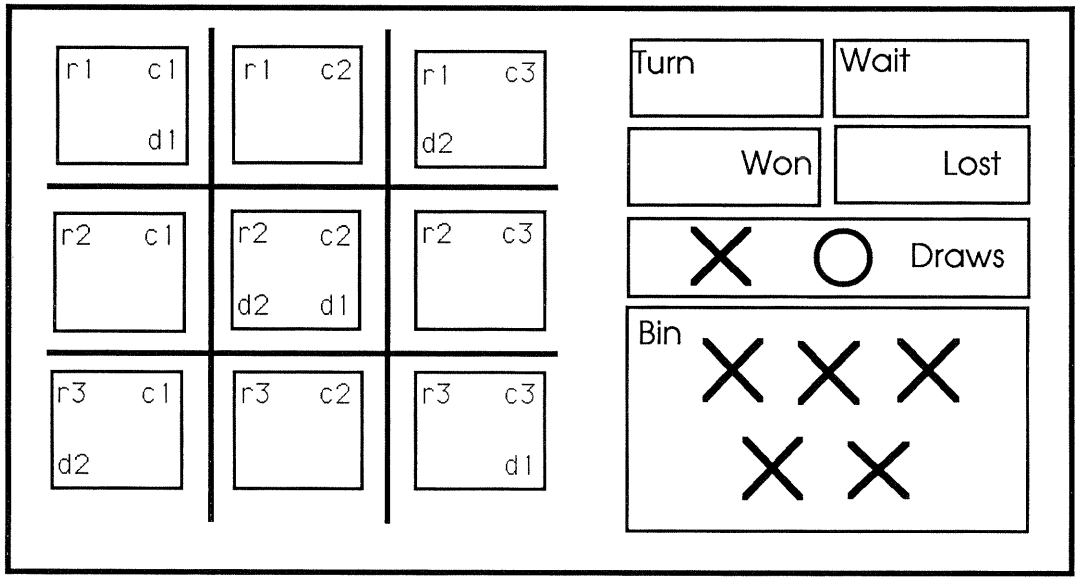


Figure 6b: Tic Tac Toe Picture with hidden objects

ShowTrains
Walkthroughs of Six Raw Problems

John, June 1990

Bracketed identifiers (e.g., [D1]) refer to Doctrine and Overview. One Rule creation (under the Office Problem) is described in detail; all other Rules are just described as if-then pairs, unless there is something difficult about creating them. Throughout, P refers to the Programmer.

BUNSEN BURNER

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, MEDIUM, 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 OFF place, set the graphic of the water to ice and set the graphic of the flame to invisible. Per guideline [D6a], P demonstrates the conditions for this Rule by highlighting the names of the knob, OFF place, ice, 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 MEDIUM place, set the graphic of the water to water and set the graphic of the flame to medium-flame. Highlighting is similar to the previous Rule [D6a].

    - If the knob is in the HIGH place, set the graphic of the water to steam and set the graphic of the flame to high-flame. 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.

Comments

This should be the easiest of the raw problems under ShowTrains. The basic guidelines seem to give the Programmer all the guidance that is necessary. There is a possibility (as always in ShowTrains) of confusion over what to highlight. The simulation is simple enough, however, that any errors should be easy to correct, especially with the help of the Additional Guidelines [D16] and [D17], which tell how to increase/decrease Rule specificity.

OFFICE

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

2. [D1] P sketches a rough layout of the office.

3. [D2] Memos will move, so P creates a Memo with a memo graphic and the name TO OFFICE.

4. [D3] P decides that memos will stop in offices, so P creates and names Places for each office and a Place for the mailroom, with simple (rectangle) graphics.

5. [D4] Memo copying occurs in the mailroom, which is already a Place. Destruction of memos occurs in the offices, which are already Places. So P makes no changes.

6. [D5] Memos travel between the mailroom and the offices, so P decides these are Paths. P decides the Paths represent pneumatic tubes, so P makes them visible with a pneumatic-tube graphic. P names the Paths TO MAILROOM, TO JOE, etc.

7. [D6] P identifies each Place where changes will occur and creates Rules:

    - In the mailroom, the activity is making copies for the office. P drags the memo named TO OFFICE into the mailroom Place. P selects menu item New Rule and the Rule Control Panel appears, with the Condition button highlighted. P enters Rule name in the Panel. P highlights the TO OFFICE memo Object entirely (full highlight is the default). P clicks Action button in Rule Control Panel. P duplicates the old memo twice and changes the names on the three memos (original and two copies) to TO JOE, TO BILL, and TO PAT. P clicks Rule Done button.

    - In each office, the activity is the reading (deletion) of memos to the office occupant. P selects New Rule from menu, and enters Rule name in the Rule Control Panel. P decides that reading should occur only if there is a person in the office, so P creates a JOE, PAT, and BILL object for the respective offices. P creates a TO JOE memo in Joe's office and highlights the memo and Joe entirely. The outline of the surrounding Place automatically highlights. P clicks the Action button and deletes the TO JOE memo.

- P repeats the above Rule for Pat and Bill, so they will read and destroy their memos.

8. [D7] P identifies what Objects will move on Paths and creates Rules:

   - P notes that TO OFFICE Objects will travel from any office to the mailroom, and creates a Rule that looks for Objects with the memo graphic and TO OFFICE name, and places those Objects on the TO MAILROOM path attached to the Place where the Object is found.

   - P notes that TO JOE/PAT/BILL Objects will travel from the mailroom to appropriate offices, and creates three more Rules.

   - P notes that TO PAT/BILL Objects will travel from the Joe's office to the mailroom, and creates two more Rules. P creates two similar Rules for TO JOE/PAT memos in Bill's office, and two more for TO JOE/BILL memos in Pat's office.

8. [D8] P creates Rules stating:

   - If a *click* object appears in Joe's office, then create a memo Object named TO BILL.

   - If a *double-click* object appears in Joe's office, then create a memo Object named TO OFFICE.

9. [D9] P selects Run Simulation from the menu and clicks in Joe's office. The simulation runs as planned. No iteration is necessary.

10. [D7] P draws some furniture in the background of the offices and a copier in the mailroom.

Comments

A potential stumbling block in this problem is preventing a memo TO JOE from traveling from Joe's office to the mailroom. If P writes a single Rule that that looks for an Object with the memo graphic in any room and puts it onto a TO MAILROOM path, the problem will occur. By using very simplistic reasoning — one Rule for every type of memo — P avoids the problem. Another way to avoid it would be to add in/out box places to each office.

MAZE

Presolution comment

Given that P knows and intends to use the string-and-crumb solution, the solution requires at least two critical modifications to P's conception of the problem:

- There's no way to lay down a continuous string, so the mouse will have to drop pieces of string in each place.
- There's no simple way for the mouse to mark a specific path leading from a place, so the paths must either be premarked (my programmer's initial solution) or individual places must be created for each path that leaves an intersection (Clayton's solution and my programmer's ultimate solution).

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

2. [D1] P sketches a rough graphic of the maze, with the mouse and the cheese.

3. [D2] P decides that the mouse will move, so P creates a MOUSE object. Since the cheese will be eaten, it is something that changes, so P creates a CHEESE object.

4. [D3] The problem says the cheese can be placed in any doorway or dead-end, so P creates Places at each intersection and dead-end.

5. [D4] There is no user interaction or obvious change of objects that would occur except when the cheese was found, so P makes no more places.

6. [D5] Since the mouse has to travel forward and back (when backtracking) along the corridors of the maze, P connects all pairs of adjacent Places with Paths in both directions. P knows that when the mouse is in a Place it will have to distinguish one Path from another, so P names the Paths leading from any given place "Path-1," Path-2", etc. [D5a] (Since Paths are unidirectional, it's no problem that the name of the Path *to*, e.g., Place-X may be the same as the name of the Path *from* Place-Y... not yet, anyway.)

7. [D6] P notes that the cheese can be eaten wherever it and the mouse co-occur, so P demonstrates the following Rule:

   - Rule-1: If the mouse (name highlighted) and the cheese (name highlighted) are together in any Place (Places all have the same name and no visible graphic, so

exactly what's highlighted doesn't matter), then delete the cheese and change the mouse's graphic to a fat mouse. Highlighting is per [D6a].

8. [D7] P considers the Rules needed to move the mouse from each Place onto an appropriate Path. This is a recursive problem, and hard to think about: should P consider what the mouse does when it enters a place or when it leaves? Since [D7] is asking for a Rule to make the mouse move, P decides to consider the mouse's action as it leaves a Place on its first exploratory path. P wants a Rule that says something like, "if the mouse hasn't tried any of the Paths, try the first one, and trail a piece of string behind." P decides to have the mouse drop one crumb for every Path it tries, so the number of crumbs (plus one) will indicate which path to try next. P's initial cut at a Rule is:

   - Rule-2: If the mouse (name highlighted) is in any Place where there are no crumbs, put a single crumb in the Place and put the mouse onto Path-1 leading out of that Place.

   P notes two problems with this Rule: it doesn't leave any string, and it doesn't handle the case where Path-1 leads to the Place the mouse just came from. P realizes that these are related, and considers how the mouse that has just entered a Place can identify the return Path. The crumb-and-string solution calls for a string leading back down the corridor the mouse just traversed.

   [potential gap] At this point P has to realize that, since ShowTrains can't display a continuous string, the mouse must drop bits of string. [D11] in Guidelines for Difficult Situations might suggest this idea, but not very strongly.

9. But even dropping bits of string doesn't solve the problem. The string must somehow identify a Path. P realizes that the "Path-n" naming scheme is going to make this difficult, since the Paths to and from a given Place have different names.

   P considers a solution in which the Paths are named "To-Place-n" and "From-Place-n," and every Place has a unique name. But this seems to put too much information into the maze, as well as requiring specialized Rules for every Place.

   P then considers a solution in which the bits of string are dropped in the Place before the mouse leaves it. Then to return from Place-B to Place-A, the mouse must run back down every Path from Place-B until it finds a bit of string. P rejects this as also being untrue to the string-and-crumb solution, and as having difficulties in the case where more than one Path from Place-B leads to Place-A.

After further consideration, P identifies the problem as an inability to mark individual Path entrances and exits. P reviews again the Guidelines for Difficult Situations, and from [D11] and [D12] gets the idea of using a small place for the portal to each Path. P worries briefly about an additional Place to represent the chamber in which the Paths intersect, but decides that this can be done by shaping the portals so they cover the entire chamber.

P redefines the Paths and Places so that:

- there is a Place for each portal;
- portals within a chamber are arranged in a ring, connected unidirectionally with Paths named "To-Next-Portal";
- each portal has a path leading out of it into its corridor, labelled "Corridor"; thus, each portal also has a Path leading into it from the portal at the other end of the corridor;
- dead-end chambers (including the entrance) have a single place.

10. P now conceptualizes the solution as one in which the mouse drops a bit of string when it enters a chamber, thus identifying the return path from that chamber. The mouse will explore each corridor accessed through the ring of portals. When it gets all the way around the ring, it will see the string and know that exploration of that chamber is complete. Of course, the mouse will be dropping crumbs all along, so that it won't be tempted to reexplore areas that it has already seen. P considers the situation where the mouse is already in a chamber and demonstrates these Rules:

    - Rule-2: If the mouse (name) is in any place where there is a path labelled "To-Next-Portal" and there is a crumb, then take the path labelled "To-Next-Portal."
    - Rule-3: If the mouse (name) is in any place where there is a path labelled "Corridor" and there is no crumb, then drop a crumb and get on the path labelled "Corridor." (P considers dropping the crumb after the corridor has been explored, but decides that this would be a trailing subgoal which the mouse would probably forget.)

11. P now wants to say, "have the mouse drop a bit of string when it arrives." Guideline [21] suggests a way to make the mouse behave differently in the special situation of arriving in a new place.

[potential gap] P is thinking of the chamber as having a special characteristic (just-arrived-in), but it is the mouse that must carry the information (arriving-in-a-chamber).

P demonstrates the following Rule:

- Rule-4: If an object named mouse with a mouse-exploring graphic (the mouse is wearing a pith helmet) arrives in a place, change it to an ordinary mouse graphic and create a bit of string in the place.

P also modifies Rule-3, the Rule that puts the mouse on the "Corridor" path, so that it changes the mouse to a mouse-exploring. Finally, P modifies both Rule-2 and Rule-3 so that both the mouse name and the mouse graphic (ordinary mouse) are highlighted in the condition. This means that Rules 2 and 3 won't apply to a mouse-exploring. [D6a]

12. P now considers two other situations: (1) the mouse has checked every corridor on the ring (it has arrived back at the string), and (2) the mouse has travelled back down a corridor to a ring that it is in the process of exploring. P demonstrates the following Rules [D19, D6a]:

    - Rule-5: If an ordinary mouse (name & graphic) is in a place with a string, change it to a returning-mouse graphic (it's carrying a suitcase with labels pasted all over it), delete the string (so it won't cause trouble in the future), and get on the path labelled "Corridor."

    - Rule-6: If a returning-mouse is in a place with a crumb, change it to an ordinary mouse.

13. [D8] There is no user interaction specified in the problem, so P doesn't add any Rules.

14. [D9] P tries the simulation. Under the Test-Run option, conflicting Rules are identified before one is chosen to fire. This shows P that Rules 2, 3 and 5 can all fire as soon as the exploring-mouse transforms into an ordinary mouse in an unexplored chamber. After a couple of false starts, P realizes there are several related problems: The mouse can't just return without exploring the ring; it shouldn't explore at all if there is already a crumb in the place where it arrives; and dead-ends need special treatment. P modifies and adds Rules accordingly:

- Rule-2 (modified to prevent firing when Rule 5 should): the mouse gets on the "To-Next-Portal" Path only if there is no string.

- Rule-3 (modified to prevent firing when Rule 5 should): the mouse gets on the "Corridor" Path only if there is no string.

- Rule-4 (modified to also do what Rule-2 did in the first portal): the exploring-to-ordinary mouse transition takes place only if there is no crumb, and it drops a crumb (as well as a string) and places the mouse on the Path labelled "To-Next-Portal."

- Rule-7 (new, prevents exploring areas with crumbs): If an object named mouse with a mouse-exploring graphic arrives in a place with a crumb, change it to a returning-mouse graphic and put it on the path labelled "Corridor."

Finally, to handle dead-ends, P considers specially marking them in the maze and adding Rules accordingly, but realizes that the problem can be solved by adding a single Path labelled "To-Next-Portal" that loops from each dead-end into itself.

15. P tries the simulation again. [D9] It works correctly until the mouse reaches the cheese, then another Rule conflict is flagged under the Test Run: the cheese-eating Rule conflicts with whatever mouse-moving Rule is about to fire. P modifies the Rules again:

    - Rules 2-7 (modified to prevent any movement out of a place before the cheese can be eaten): the condition of each Rule is augmented to require the absence of cheese.

    [potential gap] P may not realize that the fat mouse graphic won't trigger any of the mouse-moving rules, so the mouse will stop as soon as it eats the cheese. This is probably the desired solution (the problem statement is ambiguous). But it seems to have been achieved only by accident.

16. [D10] P draws a background of maze corridors and hides the Paths and Places. [It's clear that drawing the background first would have been a better choice in this problem, but in general I think it's more efficient to put time into the graphics only after the simulation is running.]

Comments

Without the "Test Run" feature, which notifies the programmer whenever several Rules might fire, the burden of ensuring that the Rules are reliably nonconflicting is fairly high. Once the conflicts are identified, however, absence testing, makes fairly easy to modify the Rules to correct the problem.

PETRI NET

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

2. [D1] P sketches a rough graphic of the Petri net. Glancing over the Additional Guidelines, P notes that the Block Copy item should be useful. P decides to initially create one transition, then copy it.

3. [D2] P decides that tokens are created and deleted, so they are objects. P creates a couple of tokens.

4. [D3] P decides that tokens stop at input and output places, so those must be Places. P creates the input and output Places for a single transition. P names the Places INPUT-A, INPUT-B, and OUTPUT.

5. [D4] Changes to tokens occur at the input and output Places, and no special user-interaction is specified, so no additional places are needed.

6. [D5] P suspects that no ShowTrains Paths are needed, but P wants to make the simulation look right. Also, P wants to copy a complete piece of the network. For appearance more than anything else, P creates a Place for the transition itself and Paths leading from the input Places to the transition and from the transition to the output Places. The Paths are named TO-TRANSITION and TO-OUTPUT.

7. [D6] P demonstrates an object-changing Rule for the Transition:

   - If there is a token in INPUT-A and a token in INPUT-B, then delete both those tokens and create a new token in OUTPUT.

8. [D7] There is no movement, so P doesn't create any movement Rules.

9. [D8] No user interaction is specified in the problem, but P decides to create a Rule that will let the user create new tokens on any Place:

   - If there is a *click* Object in any Place, then create a token Object in that Place and delete the *click* Object.

10. [D9] P tries the single transition. It works, but P thinks it would be nice if the transition itself did something when it fired. P modifies the existing firing Rule to add the action of placing a FLASH Object in the transition place, then adds another Rule:

- If there is a FLASH Object in any Place, then delete the FLASH object.

The FLASH Object has a graphic that is the reverse of the graphic used for the transition place.

11. [D22] P now uses Block Copy with Rename to make several copies of the original transition and its input/output places. The new items are named INPUT-A.1, INPUT-B.1, OUTPUT-B.1, and INPUT-A.2, INPUT-B.2, etc. A new Rule is automatically created that fires each new transition. P links some of the transitions together, in a configuration similar to the one required by the problem.

12. [D9] P tries the network. It works, but P can't think of any way to get the exact configuration described by the problem, since the problem shows places that serve the dual function of output for one transition and input for another. P's Places are identified by their names, so they can only be an input or an output. P returns to the Advanced Guidelines and discovers [D18], the guideline that suggests using marker Objects.

13. P starts over. This time, instead of naming the input/output Places INPUT-A, etc., P puts a marker Object named INPUT-A, etc., in each Place. P demonstrates three Rules (only the first Rule is different):

    - If there is a token Object and an Object named INPUT-A in any Place, and there is a token Object and an Object named INPUT-B in some other Place, then delete both token Objects, put a new token Object in the Place containing the Object named OUTPUT, and put a FLASH Object in the Place named containing the Object named TRANSITION.

    - If there is a FLASH Object in any Place, then delete the FLASH object.

    - If there is a *click* Object in any Place, then create a token Object in that Place and delete the *click* Object.

    P again uses Block Copy, but now the resulting transitions can be connected by placing (for example) the OUTPUT.1 marker in the INPUT-A.3 Place, and deleting the Place where the OUTPUT.1 marker was originally found.

14. [D9], [D10] P tries the simulation. It works. There is no Background drawing necessary, since the transitions, places, and the paths between them are all represented by active ShowTrains items.

Comments

This approach would be cumbersome without the Block Copy item, but for the simple network in the problem it would be feasible. With the block copy, the goal of making an easily modifiable network is satisfied. A potential problem is that the appearance of the net doesn't directly define its operation — that depends on the location of the INPUT-A/B and OUTPUT markers. A misplaced marker would cause a net to perform incorrectly, even though it appeared correct.

TURING MACHINE

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

2. [D1] P sketches a rough graphic of the Turing machine. It isn't clear from the graphic whether the tape or the head moves.

3. [D2] P decides that the head will move, so P creates an object HEAD and gives it a pointer-shaped graphic. P decides that the state of the machine will change, so P creates a STATE object. P decides that each square of the tape will change, so P creates a number of objects named SQUARE-1, SQUARE-2, etc., with graphics of #, 0, or 1. P arranges these squares in a row, with the head above them.

4. [D3] P realizes that the head must stop above each square, so P creates a series of places above the tape, naming them HEAD-LOC-1, etc. P also realizes from the Guidelines that the tape squares can't be free floating, so P creates a series of Places in which the tape squares are placed, as well as a single place named MACHINE in which the state object is placed.

5. [D4] Changes occur to the tape squares and the state, and these are already in Places, so P creates no more Places.

6. [D5] The only moving Object is the head, which travels back and forth among the HEAD-LOC-N places. P creates Paths between each pair of adjacent head-loc places, naming them "LEFT" and "RIGHT" under the influence of the problem description.

7. [D6], [D7] P immediately identifies ShowTrains Rules with the rules mentioned in the problem. P starts to create Object-changing Rules, but realizes while rereading the problem that changes and movement must happen simultaneously, so P decides to create Rules that do both.

   The first activity P tries to simulate is "Head over a 1 and In State 1, then Write a 1, Change to State 2, Move Right." There is an immediate problem, since there is no easy way to indicate "Head over a 1." P turns to the Guidelines for Difficult Situations, and from [D15] decides that the head and the squares should be in the same Places. P deletes the Places where the squares were and puts the squares into the head-loc places. Now P can write all the Rules, of the form:

- If the HEAD object is in the same place as an object with the "1" graphic, and there is a STATE object with the graphic "1" in the MACHINE place, then change the graphic of the STATE object to a "2" and put the HEAD object on the path named RIGHT.

8. [D8] There is no user interaction specified in the problem, so P adds no more Rules.

9. [D9] P tries the simulation. It works, but the head overlaps the tape-square Objects, making them difficult to read. P looks under the Guidelines for Difficult Situations (movement), and from [D13] decides to edit the LEFT and RIGHT paths so they place the Head at the top of the head-loc places. P runs the simulation again and it looks good.

10. [D10] P can't think of anything that the Background needs, so does nothing.

Comments

The basic guidelines (1–10) are actually a little misleading in this situation. Do we need a guideline that suggests making real-world objects into places if they don't move? (The chair in the Overview, [O4], is such a place, so P has had exposure to the idea.)

TIC TAC TOE

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

2. [D1] P sketches a rough graphic of the Tic Tac Toe board. The graphic also has a spot to display "You Win" or "I Win" and a button to start a new game.

3. [D2] Nothing moves, but X's and O's get created or deleted, so P identifies X's and O's as objects. P creates one of each, just to get started. P also creates an Object named WINNER, which will have the graphic "You Win" or "I Win."

4. [D3] X's and O's are placed in squares, so P decides that squares must be places. P creates a Place for each square, naming the "R1-C1" etc. P also makes a place to hold the winner Object.

5. [D4] P makes a place called NEW GAME with a button graphic, for the user to click on.

6. [D5] Nothing moves, so P doesn't create any paths.

7. [D6], [D8] P now has to demonstrate to ShowTrains how to play Tic Tac Toe. P successfully demonstrates a few Rules, such as:

    - If a *click* object appears in the NEW GAME place, then delete an X and an O object from every place, if such objects exist.

    - If a *click* object appears in any of the square Places, then put an X in that place.

    At this point P suspects there will be a need to distinguish whose turn it is. P creates a creates a TURN Place, creates a TURN Object, and has the game-initialization rule give the TURN Object the graphic "Your Turn." P also has the *click*-in-any-square Rule give the TURN Object the graphic "My Turn."

    P now starts writing the substantive Tic-Tac-Toe rules.

    - Top-Row Block Rule: If there are X's in R1-C1 and R1-C2, and nothing in R1-C3, and the TURN graphic is "My Turn," then put an O in R1-C3.

    - First Diagonal Win Rule: If there are O's in R1-C1 and R2-C2, then put an O in R3-C3 and change the graphic of the WINNER Object to "I WIN."

8. P realizes that it will take a long time to demonstrate every possible situation, so P looks at the Guidelines for Difficult Situations for help. P finds two applicable Guidelines. From [D23] P gets the idea of using moving markers to represent the symmetry of the board. P revises the board so the Places are identified by tokens that have the same names as the Places originally did (R1-C1, etc.), and Places are named SQUARE. P then uses the example in Appendix X as a basis for a set of Rules that rotate the markers into the 8 possible orientations of the board (four plus four mirrored), testing all other Rules on each rotation. (P complains that this is a lot like programming.)

   P can now write a smaller set of Rules that cover all situations — for example, twelve Rules cover all possible winning plays for the machine. P reconsiders and decides this is still not feasible. P reads the Guidelines for Difficult Situations again and from [D18] gets the idea of naming the squares with more than one marker, e.g., R1 and C1. P now starts on a fresh set of Rules, such as:

   - Top Horizontal Win Rule: If there are O's in two places named R1, and no O in a third Place named R1, then put an O in the third Place and change the graphic of the WINNER Object to "I WIN."

   These Rules are fairly general. The above Rule, plus a Rule for Middle Horizontal Win and a Rule for Diagonal Win will handle all win situations. P creates several more Rules, realizing that the job is not yet done, then continues to the next step.

9. [D7] There are no movements, so P creates no movement Rules.

10. [D9] P runs the simulation. It is immediately obvious that the Rules aren't interacting correctly. P realizes that what needs to happen is that all Rules of a given priority are tested, (is there a win?), then Rules of the next priority (block?), etc. P again turns to the Advanced Guidelines, [D20], and creates a hidden Object called RULE-SET. P then revised all the Rules to test the state of the RULE-SET Object. P is able to coordinate this with the marker rotation, but only after a false start in which Rules of all priorities are tested, then the markers are rotated (so a low priority Rule in the initial marker configuration will fire instead of a high priority Rule in a rotated configuration).

    If P is a programmer, he or she will realize at this point that the simulation is emulating a set of nested loops. The outermost loop switches from User to Machine play. The next inner loop cycles through sets of Rules, and the innermost loop rotates the markers.

11. P runs the simulation again. Whether or not the simulation plays non-losing, or even winning, Tic-Tac-Toe depends on how many situations P has thought to program into the Rules. If the game should lose, P should be able to identify the juncture at which the loss could have been prevented and add an appropriate Rule without difficulty.

12. [D10] P draws lines to form the Tic-Tac-Toe grid on the background, and makes the Places that the X's and O's appear in invisible.

Comments

At first glance, multiple places in a condition and action really pays off in the Tic-Tac-Toe problem. It allows the Programmer to write some Rules with very little overhead. However, there are 3 to the 9th, or about 20,000, possible configurations of X, O, and blank. Not all are reachable, of course, but this is clearly too many to handle without generalization. Generalizing the initial solution requires some backtracking and renaming of Places using markers. The marker rotation gives an 8-fold reduction in the Rule count, but this solution is still untenable. P's final solution, generalizing by using multiple markers per square, still doesn't yield the kind of logic we'd like to see, e.g., "If there are two X's in a row, play an O to block." It seems that the seductively easy initial simulation may actually discourage thinking the problem through before programming begins.

ZeroTrains Walkthroughs

Clayton, 6.23.90

--don't forget to print out and keep earlier versions of this and doctrine!

Bunsen Burner Problem

sketch...

objects

control
flamelo, flamehi
ice, water, steam

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

paths: none

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

.

To deal with O in P tests add objects off, lo, and med 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

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

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.

Office Problem

Draw picture of offices with a place for each office and paths connecting. Objects are memos and people and copier (since they are mentioned in problem).

Events:

what	when
memo to office becomes memos to persons	memo to office, copier in same pl
memo to person deleted	memo to person, person in same pl
memo to mailroom	memo in office not of recip
memo to office of person	memo in mailroom

Only issue here is getting memo to Joe destroyed in Joe's office rather than being sent on. Rewrite "in office not of recip" as "in office of other person" and expand for each person.

Petri Net Problem

Draw picture of Petri net. Make places for places and for transition, because want to create tokens at transition and send to output place. Objects are tokens.

Events:

what	when
tokens removed from input places token created at transition and sent to output place	there is a token in each input place for transition

To implement O in P tests need to put a marker in each input place. To avoid having to handle each transition separately mark them so each transition has an input place labelled B and one labelled W. Add events for O in P processing:

what	when
change token to token-B	token, B in same place
change token to token-W	token, W in same place

Create new places enclosing transition and its places. Add a catalyst object to each, and test for it in the rule that looks across places. --need to point out in doctrine when to make a bunch of catalysts the same.-- Having done this rest is easy.

Turing Machine Problem

Draw picture of tape with symbols in it. Places are squares; objects are symbols including #. Include object indicating where head is, and a row of places for this object below squares of tape. Connect these places by paths labelled L and R.

Events:

what	when
move R change state to seen odd	state is seen even, current sq 1
move R	state is seen even, current sq 0
move R change state to seen even	state is seen odd, current sq 1
move R	state is seen odd, current sq 0
change # to 0 change state to halt	state is seen even, current sq #
change # to 1 change state to halt	state is seen odd, current sq #

Make objects for states halt, seen even, seen odd. Make a place to hold them. Make places that enclose each tape square, corresponding head position, and this place. Add a catalyst to each. --Note: if we agree to think of putting the head on the tape can use one big place--

Tic Tac Toe Problem

Draw board with a place for each cell.

Objects are X and O markers

Events:

what	when
place an O in space	my move
change move	space is blank
	other two spaces in line have O
place an O in space	my move
change move	space is blank
	other two spaces in line have X
	no win
place an O in center	my move
change move	center is blank
	no win
	no block
place an O in upper left	my move
change move	center is filled
	upper left is blank
	no win
	no block
place an O in space	my move
change move	center is filled
	upper left is filled
	space is blank
	no win
	no block

Use messenger approach. Create places for lines and link together. Route mymove through line places. Change to yourmovewin if win found. Change to mymovenowin if emerges from end and route thru again. Change to yourmove if block found. Change to mymovenowinnoblock if emerges from end. Create paths connecting individual cells, ordered so that center is first and upper left is second. Route through this. Change to yourmove if move found.

Maze Problem

picture: maze

places: points at which mouse faces a choice

Concept is that of laying down and retracing string. Objects are mouse, cheese, segments of string. Segment of string indicates path entered and left.

Events:

what	when
mouse eats cheese mouse stops	mouse and cheese in same place
mouse lays string and turns right	mouse is in new place, going ahead
mouse turns back	mouse hits dead end, going ahead
mouse turns back	mouse hits string, going ahead
mouse picks up and re-lays string to right turns ahead	mouse is going back, branch to rt exists
mouse retraces string and picks it up	mouse is going back, no branch to rt exists

Make mouse variants for ahead, back, and stop.

Would like test for what path mouse came in on. Doctrine says must have object be different according to path. So have back and ahead, with variants to indicate arrival through L, R, and Out, giving six mouse

variants: ahead-fromR, ahead-fromL, ahead-fromOut, back-fromR, back-fromL, back-fromOut. String variants possible are LR, LOut, RL, ROut, OutR, OutL, named as from-to. Also need marker for deadend.

Rewrite events:

what	when
delete cheese	ahead-fromR or ahead-fromL or ahead-fromOut
mouse to stop	cheese, in same place
string to OutR mouse to ahead-fromL moves along R	no string, ahead-fromOut in same pl
string to LR mouse to ahead-fromOut moves along Out	no string, ahead-fromL in same pl
mouse to back-fromOut moves along Out	deadend, ahead-fromOut in same pl
mouse to back-fromR moves along L	ahead-fromL, any string in same pl
mouse to back-fromOut moves alongOut	ahead-fromOut, any str in same pl
string to LR mouse to ahead-fromL moves along R	back-fromOut in same pl
string to no string moves along L	back-fromR in same pl

Note only no, Lout and LR strings are needed, and only stop, ahead-fromL,

ahead-fromOut, back-fromOut, back-fromR for mouse.

Problem: need to handle cheese so preempts moving. Probably put test for no cheese in? Seems pretty ghastly since would have to put "no cheese" in every "no cheese" place. Only graceful way would be to put cheese at a dead end and not mark it as dead end.

Reflections

lack of absence testing is a big nuisance

restriction of groups of events to one place also appears to lose, but should get a clearer look at this in other designs

most direct solution is not the best in some other senses

want doctrine to produce moving state rather than moving pointer in TM?

need to explore messenger doctrine for ttt

may want "simplification" doctrine... have used a little in PN to keep from having diff rules for each transition. Also used some in TM to keep path names in hand

would it be better to start w wt's rather than sol'ns? Unclear, because need some feel for the doctrine.

5. Reserve Problems

The four reserve problems were intended to test the designs and doctrine. They were not considered during the original design process and doctrine development.

Chemtrains – Reserve Problems

June 1990

DOORBELL

Purpose: Show how an electromechanical doorbell operates.

Task Description: Show a model of an electric doorbell. The parts include

- doorbell button
- a battery
- a coil
- an iron core that can move in and out of the coil
- a spring holding the core partly in the coil
- a set of contacts attached so that they are closed (allow current to pass) when the core is at the point where the spring holds it, and they are open when the core is fully in the coil
- an arm with the following characteristics:
  - it pivots at its lower end
  - it has a clapper on its upper end
  - it is attached somewhere near its midpoint to the moving iron core
- a bell
- a complete electrical circuit, including the button, the battery, the coil, and the contacts.

The model should demonstrate the following behavior: When the button is pressed, current flows through the circuit. This causes a magnetic field to be created in the coil, which pulls the iron core further into the coil. The arm attached to the core moves with it, causing the clapper to strike the bell, which rings once. The moving core opens the contacts, breaking the electrical circuit, and allowing the magnetic field to collapse. The spring pulls the core out of the coil, the contacts close again, and the cycle repeats as long as the button is down.

Desiderata: It should be easy to manipulate the simulated parameters, and the model should respond correctly. Parameters include the battery voltage, the strength of the spring, the point in the core's travel at which the contacts open and close (not necessarily the same point), the inertia of the arm, and the point at which the arm attaches to the core. Good graphic display of the rising and collapsing magnetic field and current flow scores points, as does accurate representation of the effect of the magnetic field on the core (stronger when the core is further into the coil) and the behavior of the spring (force exerted increases as it is stretched further).

SUNSPOTS: FORMATION AND FLARING

Purpose: Show how a pair of sunspots forms and produces a solar flare.

Task Description: Unlike the earth, the sun is not a solid globe. It is composed of hot gasses, and as the sun rotates, the material at the equator takes longer to complete a full rotation than the material near the poles. This differential motion creates an ever-changing pattern of currents, not just on the sun's surface but deep within. As the gases move, they generate electric currents, which in turn generate magnetic fields. One configuration of magnetic fields that may be formed is a tube-like structure, which twists about below the sun's visible surface under the influence of the changing currents (Figure S-1).

Sometimes the currents may form a bend or a loop in a tubular magnetic field, and part of that loop may be forced above the sun's surface (Figures S-2 and S-3). It is only the magnetic field itself that extends outward — the moving material that generates the field remains within the sun, maintaining the loop by maintaining the subsurface fields to which it is connected.

The circles formed where such a loop leaves and reenters the sun's surface are visible from earth as sunspots, areas that are cooler than the surrounding surface because the magnetic field inhibits thermal activity. As the loop is initially forced above the surface, only a single spot is visible (Figure S-2). As it rises higher, two spots become visible, of opposing polarities (N and S, Figure S-3).

The currents may push the loop still higher, until some of the outermost magnetic field lines break loose from the loop and trail out into space (Figure S-4).

Now, instead of all adjacent field lines being in the same direction, there are adjacent fields at the top of the loop that have opposing directions. This is an unstable situation. If some subsurface activity causes material to erupt from the neutral area between the sunspots, this material may shoot upward through the loop, forcing it even higher, and inducing the adjacent but opposing fields to reconnect into a new magnetic field above the original loop (Figure S-5).

At the moment the fields reconnect, energy is released in the form of a solar flare. The flare may be visible, as some of the ejected material is flipped out into space in the form of two ribbons, one on either side of the new magnetic field. The new field itself forms a linear accelerator, which may shoot particles into space at up to 99.5 percent of the velocity of light. At the same time, electrons within the original loop may be driven down the tube into the sun, where they will slow abruptly, creating high-energy X-rays known as Bremstrahlung radiation.

Model a partial cross-section of the sun. Show a tubular magnetic field being forced above the surface. Show it rising until the field breaks at the top and extends out into space. Conclude the simulation with the ejection of material from the neutral area and the production of a solar flare. Your simulation should allow the user to adjust some of the parameters that make flare production more likely. These include the size of the sunspots (i.e., the cross-sectional area of the magnetic tube, which may differ for the entrance and exit spots), how near the spots are together, and whether the spots are staying in a single orientation relative to the rest of the sun or rotating about some common center. See Figure S-6 for combinations that are more or less likely to produce flares.

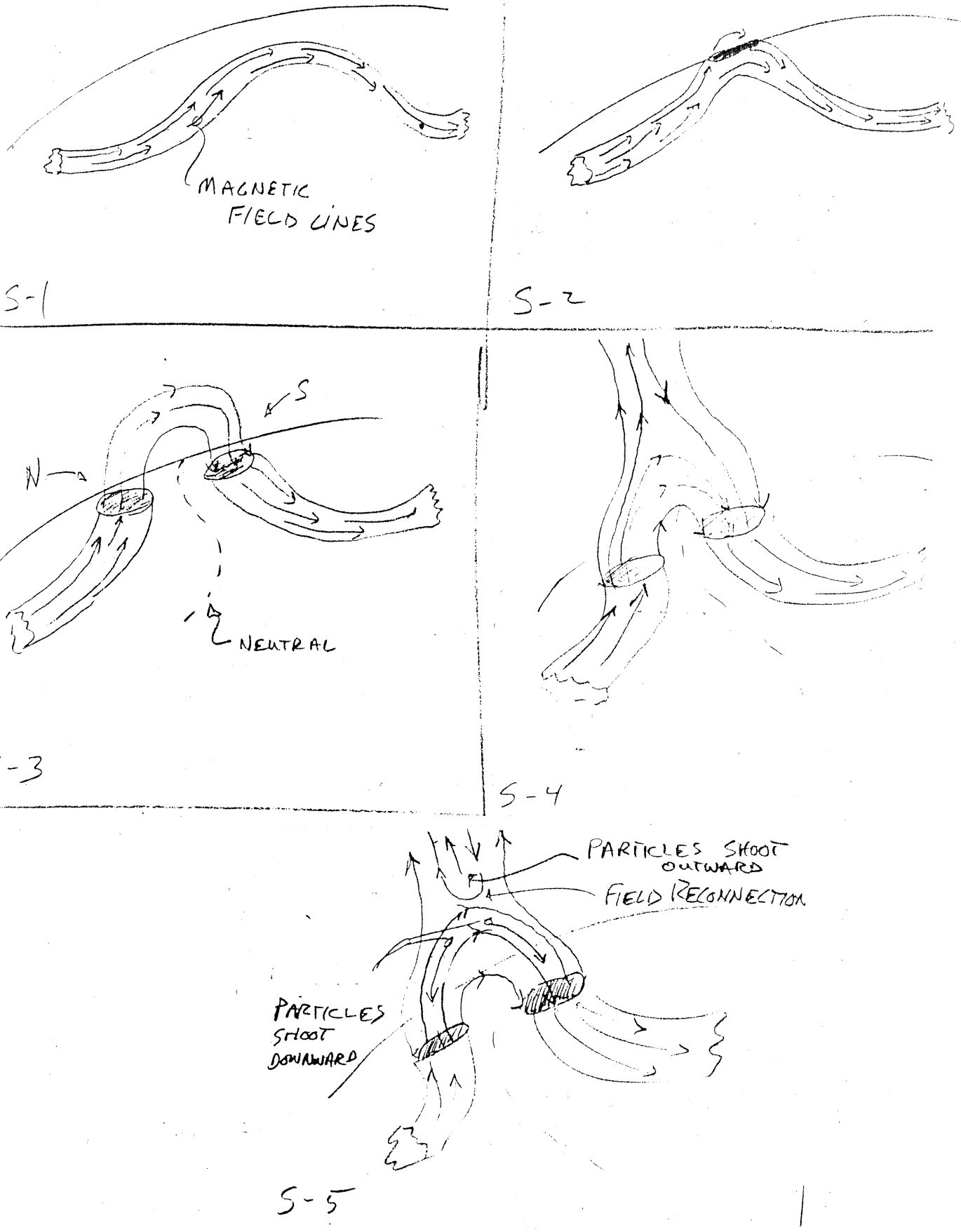
Desiderata: To the extent possible, the model should be more than a simple animation. For example, the visible sunspots should actually be caused by the magnetic loop; the likelihood of the upper lines of force breaking loose should depend on how sharply the magnetic tube is bent, which relates to the cross-section of the tube and the proximity of its entrance and exit points; the energy of the flare should depend on the size of the loop's cross-section (i.e., the size of the sunspots); etc.

References:

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

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

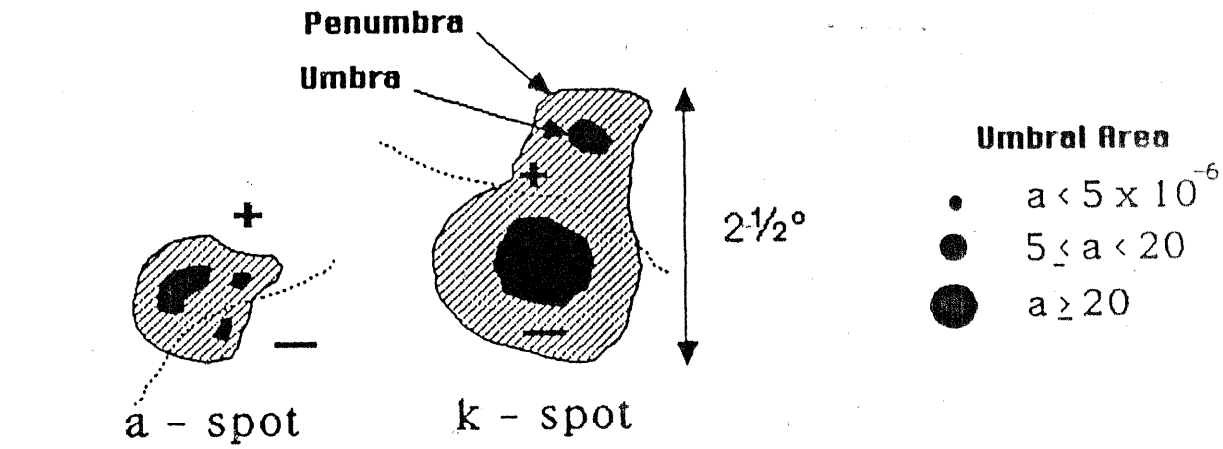
Tandberg-Hanssen, E. and Emslie, A. (1988) *The physics of solar flares*. New York: Cambridge University Press.

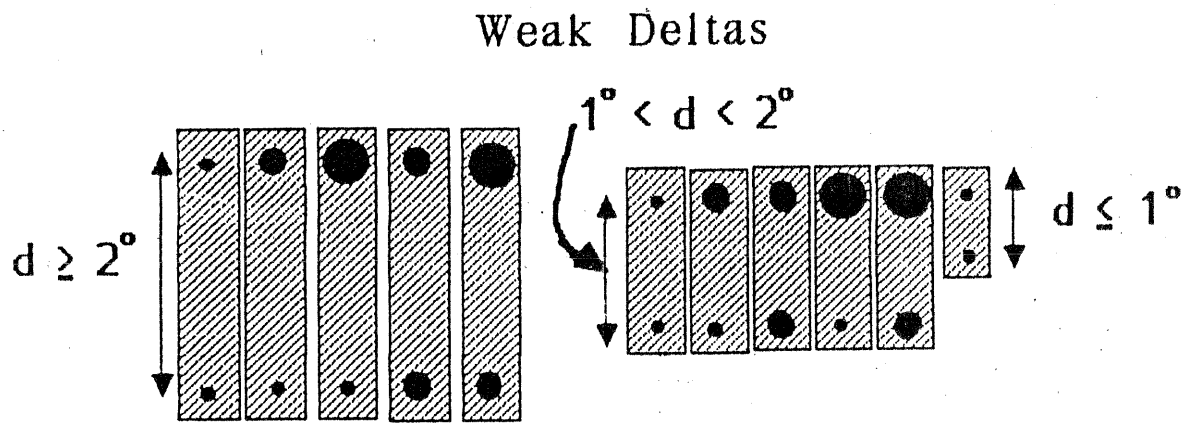


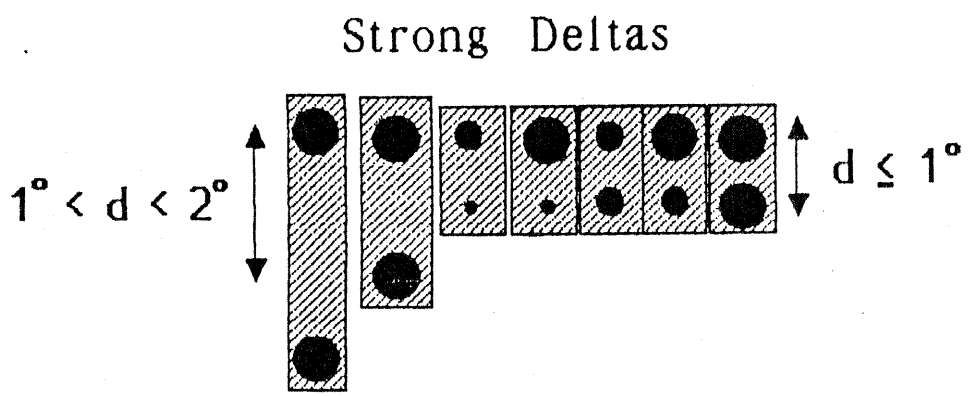
(after McIntosh, 1990, Noyes, 1982, and Tandber-Hanssen and Emslie, 1988)

Figures S-1 through S-6. Development of a solar flares.

Delta Configurations







(courtesy of Pat McIntosh)

Figure S-6. Conditions under which solar flares are more likely to occur (strong deltas).

GRASSHOPPERS

Purpose: Produce a simulation which, by adjusting appropriate parameters, can model competition for resources in a grasshopper population. Individual grasshoppers should be units within the simulation.

Task Description: In a grasshopper population, the life cycle of a grasshopper (simplified) is as follows:

> An egg hatches, producing a juvenile grasshopper. The juvenile competes with other juveniles for the resources it needs to stay alive. The juvenile turns into an adult, which competes with other adults for resources it needs to stay alive and produce eggs (assume these are different resources than those needed by juveniles). The adult lays a number of eggs (assume it is between zero and 100). The eggs "compete" with other eggs for the resources they need to stay alive and eventually hatch (i.e., they compete to see who doesn't get eaten).

During each stage of competition, the grasshopper may live or die, depending on the competition and the resource. Some resources, such as ready-made burrows in which to lay eggs, can be competed for in a scramble situation, something like an Oklahoma land rush: an individual grasshopper either gets all of the required resource, or it gets none. Other resources, such as food, can be competed for in a contest situation, where an individual may get only part of its required amount, or it may get all. In contest, it's possible that no grasshoppers get enough of the resource to survive, and the entire population dies out. During each stage of the grasshoppers life there will be several critical resources, some of which may be scramble type and some contest.

The model should produce a summary graphic display of the adult grasshopper population over several life cycles, perhaps in the form shown in Figure G-1.

Desiderata: It should be easy to manipulate the simulated parameters.

Addendum to grasshoppers -- I assumed this for both ShowTrains and OpsTrains.

- There will be a grid of places, highly interconnected by paths. The places will represent the field in which the grasshoppers live, and the paths will represent jumps of varying distance and direction between places.

- Individual grasshoppers — juvenile and adult — will jump at random (with some constraints) from place to place.

- All grasshoppers act as parallel, independent units.

- Any place may contain any number of the resources that the grasshoppers need, and a grasshopper may consume several units of various resources within a single place.

- When a grasshopper is out of resources in a place, or when it has been in a single place for a "long enough" time (varied randomly), it will jump.

- Each jump increases the grasshopper's need for its energy resource.

- If a grasshopper's energy drops too low, it dies.

- Stage changes (egg to juvenile to adult) are synchronized for all grasshoppers. When the stage-change "bell" rings, all living grasshoppers that currently have the necessary resources will change stage. Other grasshoppers will die.

- For each generation, the simulation will graph the number of transitions from juvenile to adult.

Reference:

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

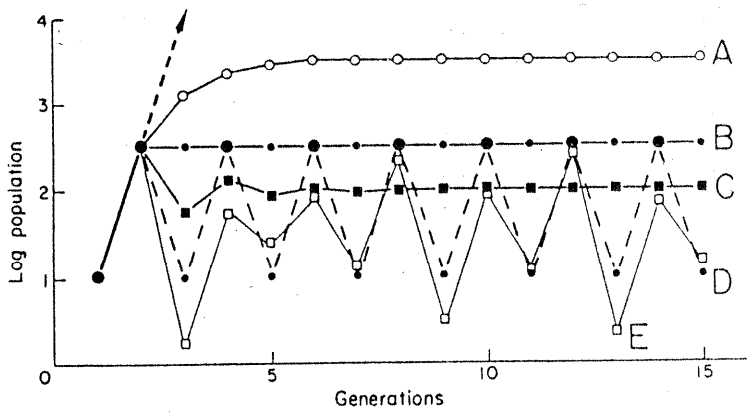


Fig. 2.10 The consequences of the curves A–E in Fig. 2.8 (and of their equivalent logarithmic reproduction curves A–E shown in Fig. 2.9) on a population which has the initial size of 10 and in which log F = 1·5.
A gradual attainment of stability,
B attainment of stability in one generation,
C dampened alternations leading to stability,
D alternations with alternate generations of equal size, and
E irregular sized alternations.

(taken from Varley, Gradwell, and Hassell, copyright 1973, University of California Press.)

Figure G-1. Graphs of adult grasshopper populations.

XEROX™

Purpose: Accurately model a Xerox™ model 1075 office photocopier, showing normal behavior and appropriate faulty behavior when various parameters are incorrectly adjusted.

Task Description: A Xerox™ 1075 photocopier has the parts shown in Figure X-1, which operate as follows:

> Charging: Dicorotron-1 places a static electric charge (negative) on the segment of the belt directly opposite it.
>
> Imaging and Exposure: The belt rotates into position under the lens, where it is exposed to the projected image of the original document. Where white light strikes the belt, the belt becomes electrically conductive, and the negative charge in these areas is conducted to ground. This leaves the belt with a pattern of negative charge that reproduces the dark areas in the original.
>
> Development: The belt rotates past the toner, tiny particles of black, positively charged material that are held on negatively charged brushes. The strongly charged negative areas of the belt are more negative than the brushes, so those areas attract the toner particles, which stick to to the belt.
>
> Transfer: The belt rotates further, until the paper and the belt are moving together. Dicorotron-2 creates a negative charge behind the paper which is higher than the negative charge on the belt, causing the toner particles to be transferred from the belt to the paper.
>
> Fusing: The paper is transported past a heated roller, which seals the toner onto its surface, and is ejected from the machine.
>
> Cleaning: The belt moves past a set of brushes that remove any excess toner.

The task is to model the machine as described, using graphics similar to Figure X-1. The model should simulate both the machine's normal operation and its operation in at least the following fault conditions:

> If dicorotron-1 places too high a charge on the belt, the copy will be too dark. It may even show some graying in areas where the original was pure white.

If the developer brushes are too strongly charged, they will not allow enough toner particles to transfer from them to the belt, and the copy will be gray where it should be black.

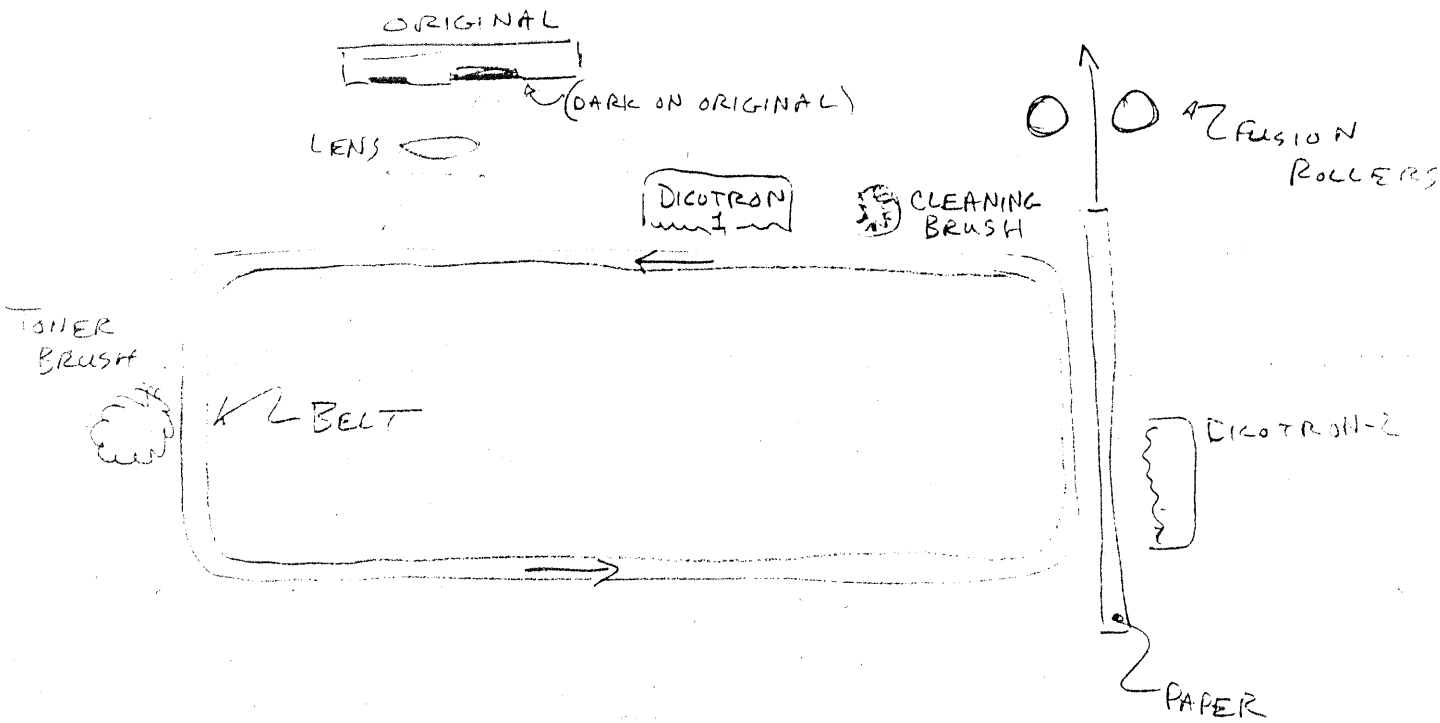
The image being copied in the model can be very simple, with fairly large segments of solid black or white.

Desiderata: The model should be easily adaptable to show the behavior of a different (but similar) type of photocopier.

References:

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

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



(after Schrager et al, 1987)

Figure X-1. Rough sketch of the office photocopier to be modeled.

6. Walkthroughs of Reserve Problems

The walkthroughs of the reserve problems show strengths and weaknesses for each design.

OpsTrains
Walkthroughs of Four Reserve Problems

JR
June 1990

[General comments

- I did Doorbell and Grasshopper first in ShowTrains, then in OpsTrains. I did Sunspots and Xerox first in OpsTrains.

- I've been a little sloppy in using "place" or "object" — they're the same thing in OpsTrains.]

DOORBELL

R01: Draw a picture of the bell, clapper, arm, pivot, coil, core, moving and fixed contacts, spring, battery, button, electrons, and connecting wires.

[big problem: We're about to draw invisible objects to hold the arm in its resting and ringing positions, when we realize that it has different orientations, so it's not clear where those positions are. There's no doctrine to help us with this, but assume the programmer arrives at the following solution: let the clapper move, and represent the arm as a path between the clapper and a fixed pivot. Move the core and contacts separately (but in coordination with the arm). This still leaves us with the spring, which we'll have to approximate using different graphics.

R02: Draw objects to contain the clapper, moving contacts, and core in their rest and ring (when the clapper strikes the bell) positions.

R03: Draw an object, we'll call it "coil-surround," to hold the magnetic field.

R05: Draw a hidden object in the coil-surround to indicate that it's empty **[Win:** I didn't think I'd need this, but did it because the doctrine told me to. Turns out I did need it! Order of doctrine is important, though, contrary to Brig's claim.]

RP1: Draw paths to describe the motion of the clapper, core, and moving contact. Make them bidirectional [RP4].

RP1: Draw paths for the electrons. **[small problem:** the wires are already drawn as objects. assume programmer deletes them.] Make the wires unidirectional (we only want the electrons to go one way, no matter how they might move in real wires) [RP5]. Name them all WIRE [RP3]. **[small problem:** no doctrinal guidance on how to handle the opening and closing contacts] Draw a path from the fixed to the moving contact, also naming it WIRE.

RR1: Write a rule to start the arm moving when there is a magnetic field.
- copy the clapper, moving contact, core, and their resting places and swing paths into the pattern.
- create a magnetic field object in the coil-surround, and copy the coil-surround and field into the pattern.
- copy the spring into the pattern.
- copy the pattern into the result.
- [RR4] in the result, put the clapper, moving contact, and core onto their swing paths.
- [RR2/3] in the result, approximate the change in spring length by deleting the spring and creating a half-extended spring.

RR1: Write a rule to ring the bell and disconnect the contacts when the arm reaches the end of its swing.
- copy the clapper and moving contact and their ringing places into the pattern (they should be in their ringing places). Also copy the spring and the path from the moving contact to the fixed contact.
- copy the bell into the pattern.
- copy the pattern into the result.
- [RR2/3] in the result, delete the bell and create a highlighted bell with the word "DING" written on it.
- [RR2/3] in the result, approximate the change in spring length by deleting the half-extended spring and creating a fully-extended spring.
- [RR2] in the result, delete the path (wire) from the fixed to the moving contact, to simulate opening the contacts. **[big win here!]**

RR1: Write a rule to start the arm back on the swing path when the magnetic field is released.
- copy the clapper, moving contact, core, and their ringing places and swing paths into the pattern.
- create a magnetic field object in the coil-surround, and copy the coil-surround and

field into the pattern.
- copy the fully-extended spring into the pattern.
- in the pattern, negate (i.e., prohibit) the magnetic field
- copy the pattern into the result.
- [RR4] in the result, put the clapper, moving contact, and core onto their swing paths.
- [RR2/3] in the result, approximate the change in spring length by deleting the spring and creating a half-extended spring.
- [RR2/3] in the result, delete the bell-with-ding and create a simple bell.

RR1: Write a rule to reconnect the contacts and retract the spring when the arm is at rest.
- copy the clapper and moving contact and their resting places into the pattern (they should be in their resting places).
- copy the half-extended spring into the pattern.
- copy the pattern into the result.
- [RR2/3] in the result, approximate the change in spring length by deleting the half-extended spring and creating an unextended spring (copy it in from a previous rule, to be sure it matches).
- [RR3] in the result, create the path (wire) from the fixed to the moving contact, to simulate closing the contacts.

RR1: Write a rule (called "current-in-wire") to move electrons through the wire when there is an electron in a place and none in the downstream place:
- put an electron in a place and in the place to which the outgoing wire leads.
- copy both places, the electrons, and the wires into the pattern.
- [RR8] negate the downstream electron in the pattern.
- [RR5/7] make the places variables in the pattern.
- copy the pattern into the result.
- [RR2/3] in the result, delete the electron in the upstream place and copy an electron into the downstream place.

RR1: Start to write a similar rule to move electrons through the battery:
- **[problem:** this seems to be the natural thing to do, but it's an error. The current-in-wire rule will take care of the battery. Assume the programmer realizes this, perhaps by testing [RD1, sort of] the current-in-wire rule (the circuit is currently complete, with wires between places throughout, so current will flow.)]

RD3: Modify the current-in-wire rule to prohibit the motion of electrons through the button unless it has a button-down graphic **[problem:** as above, it seems more natural to write a new rule for the button, not modify the wire rule]:
- enter the rule editor and edit the current-flow rule as follows:.

- put a button-up graphic in the downstream place of the pattern.
- [RR8] prohibit the button-up graphic

[small problem: we'll need to do something similar for the coil. It might have been better to create a NO-FLOW marker that could have been placed in any object that shouldn't allow the wire rule to produce current flow, but doctrine doesn't suggest this idea. Conceptually, it's a little odd that the conductivity of the connected objects isn't already a factor in the current-in-wire rule.]

RR1: Create a rule to let the user control the push-button with the mouse. **[small problem:** don't know how to do this. assume it's trivial.]

Now we need to create rules that create and collapse the magnetic field in the coil surround. **[big problem:]** We would really like the field to exist when there are flowing electrons, and to collapse in the absence of flowing electrons. But we can't sense flow directly, and (as implemented) the electrons don't go away when the circuit is opened — they just stop moving.

We will do the job with several rules, which (1) create a marker in the coil when an electron flows through, (2) raise/maintain a magnetic field if the marker exists, (3) collapse the magnetic field if the marker is absent, and (4) delete the marker on every cycle. There's not much doctrinal support for this kind of sequence. Note that the rules have to be ordered exactly as described [RD2]:

RD3: Modify the current-in-wire rule so it doesn't push electrons out of the coil.
- First place an invisible object in the coil with the text-graphic "coil". **[small problem:** RO4 almost suggests this, but not quite. Having variablized the places in the rule, there doesn't seem to be any way to use the coil's existing graphic to identify it.]
- In the rule editor, modify the current-in-wire rule so its pattern includes a negated "coil" object. [RR8]

RR1: Create a rule that pushes electrons out of the coil and creates a current marker:
- put an electron in the coil and in the place to which the coil's output wire leads.
- copy the coil, the outgoing wire and the place to which it leads, and the electrons into the pattern.
- [RR8] negate the electron in the outgoing place.
- [RR6] make the outgoing place variable.
- copy the pattern into the result.
- [RR3] in the result, create an invisible current object (an arrow graphic) in the coil.
- [RR2/3] in the result, delete the electron in the coil and create an electron in the downstream place.

RR1: Create a rule that creates/maintains a magnetic field if there is a current marker.
- note that there is already an "empty" object in the coil-surround.
- copy the coil, the current object, the coil-surround, and the empty object into the pattern.
- [RR6] mark the empty object as Variable.
- copy the pattern into the result.
- [RR2] in the result, delete the empty object.
- [RR3] in the result, create a magnetic field object.

RR1: Create a rule that deletes (collapses) the field if there is no current object in the coil.
- copy the coil, the current object, the coil-surround, and the magnetic field into the pattern.
- [RR8] mark the current object as negated
- copy the pattern into the result.
- [RR2] in the result, delete the magnetic field object.
- [RR3] in the result, create an empty object in the coil surround. **[small problem:** this is a trailing subgoal]

RR1: Create a rule that deletes the current object at the end of each rule cycle.
- copy the coil and the current object into the pattern.
- copy the pattern into the result.
- [RR2] in the result, delete the current object.

RD1: Try it out. It works. Note that there's no problem with the magnetic field pulling the arm back too soon, since the arm doesn't stop midway between the rest and ring positions. Thus, the user doesn't notice the fact that contact open/close time is critical.

Summary Comments

Pluses (mostly expressiveness):
- moving paths make it possible to show the moving arm (a little kludgy).
- contacts opening/closing are easy because we can delete/create paths.
- we can look at the destination of the path and move electrons accordingly.

Minuses:
- doctrine works pretty well for the basics, but the beginner has some pretty big leaps of intuition to make in the more difficult situations.
- it would be nice if the paths could have user-specified graphic appearance.
- the field creation/collapse sequence is too complex, although rule-ordering helps.

Big Hit on Expressiveness: TIME (same as ShowTrains):

- there's no easy way to control timing and synchronization of reactions and movement. This makes it hard to coordinate movement and hard to implement the adjustable parameters (voltage, spring constant, inertia) described in the desiderata. Timing could also be used to solve the field collapse problem — we'd like it to simply collapse as time passed, unless it was renewed by electrons flowing thorugh the coil.

GRASSHOPPERS

R01: We start to sketch the graphic, and realizes that the problem statement seriously underspecifies the appearance of the simulation. We decide that the general operation of the simulation will be as follows [identical to ShowTrains]:

- There will be a grid of places, highly interconnected by paths. The places will represent the field in which the grasshoppers live, and the paths will represent jumps of varying distance and direction between places.
- Individual grasshoppers — juvenile and adult — will jump at random (with some constraints) from place to place.
- All grasshoppers act as parallel, independent units.
- Any place may contain any number of the resources that the grasshoppers need, and a grasshopper may consume several units of various resources within a single place.
- When a grasshopper is out of resources in a place, or when it has been in a single place for a "long enough" time (varied randomly), it will jump.
- Each jump increases the grasshopper's need for its energy resource.
- If a grasshopper's energy drops too low, it dies.
- Stage changes (egg to juvenile to adult) are synchronized for all grasshoppers. When the stage-change "bell" rings, all living grasshoppers that currently have the necessary resources will change stage. Other grasshoppers will die.
- For each generation, the simulation will graph the number of transitions from juvenile to adult.

P sketches the simulation, including a grid of places (patterned after coins tightly packed on a flat surface), a bar graph, a clock-like object that will advance from stage to stage, several juvenile-grasshoppers, and several types of resource, distributed randomly about the grid. [**problem:** We envision a grid of at least 1000 interconnected places. Drawing all these places and paths without system help is almost inconceivable. Since OpsTrains can create places and paths, we should be

able to start with a core set and write rules that will generate the rest. Doctrine, and perhaps examples, suggesting this approach would help.]

R04: Each grasshopper may be significantly different than every other, in that its state reflects what resources it has consumed. Create state objects to be contained inside the grasshopper, including a stomach, heart, and egg-sack. **[small problem:** the state objects won't actually distinguish the grasshoppers. Instead, we'll put objects into the state objects, so they can be distinguished. This is a level of hierarchy beyond what the doctrine suggests.]

RP1: We've already drawn paths under RO1; **[small problem:]** assume we knew to draw them as paths and not as objects. From RP4, make them bidirectional. RP3 suggests giving them all the same name (they're all for jumping), but this wouldn't leave any way to distinguish between long and short jumps. So name them ONE, TWO, or THREE, depending on how long the jump is.

Now we're ready to write some rules. Start with a resource consuming rule:

RR1: Write a rule showing how a juvenile-grasshopper consumes a bit of food:
- copy the juvenile-grasshopper, its place, and a bit-of-food (in the same place) into the pattern
- mark all of the contents of the grasshopper as variables
- copy the pattern into the result
- [RR2] in the result, delete the bit-of-food
- [RR3] in the result, put five small bits-of-food in the grasshopper's stomach place (food in the grasshopper represents energy, and we want to parcel it out in small bits).

RR1: Write a rule showing how a juvenile-grasshopper dies if it has no food:
- copy the juvenile-grasshopper into the pattern. make sure the grasshopper has a small bit-of-food in its stomach.
- [RR8] negate the bit-of-food.
- copy the result into the result.
- delete the grasshopper in the result.

[small problem: this rule may conflict with eat-a-bit-of-food, so an empty juvenile has a chance to eat before it dies; this is not a problem, but the programmer might think it is and try to avoid it.]

Other consumption rules are similar. Now create rules showing how juvenile and adult grasshoppers jump when there are no resources in their place:

RR1: Write a rule showing that a juvenile-grasshopper can take a length-two jump.
- copy the juvenile-grasshopper (which must have two bits of food in its stomach), its place, and each of the resources it can consume into the pattern
- negate each of the resources
- mark all of the grasshopper's contents as variables, except two bits-of-food.
- copy the pattern into the result
- [RR3] delete two bits-of-food from the juvenile's stomach
- place the juvenile-grasshopper onto a path marked TWO.

RR1: Write a rule showing that a juvenile-grasshopper can take a length-one jump. (As above, except one bit of food and path ONE.)

The two rules conflict, so sometimes the grasshopper will jump a distance of one, sometimes two. **[problem:** this use of random conflict resolution isn't spelled out well in doctrine.]

Now we need a rule to make the grasshopper jump when it's been in a place more than three cycles. We'll start by setting up an internal state object in the grasshopper, which will contain one tick for every rule cycle since the grasshopper last jumped. [Note again that there's no doctrinal support for timing.]

RR1: Write a rule that makes the grasshopper jump when it's been in a place more than three rule cycles:
- copy the grasshopper, which must include an enclosed sitting-time object, into the pattern
- make all parts of the grasshopper variable.
- copy the pattern into the result
- create a "tick" object in the sitting-time object.

RR1: Write a rule that forces the hopper to jump when it has been sitting for three cycles:
- copy a grasshopper, its place, its enclosed sitting-time-object (with three ticks), and its stomach (with a bit-of-food) into the pattern.
- make all parts of the grasshopper variable except the three ticks and the bit-of-food
- copy the pattern into the result
- [RR2/3] in the result, delete the ticks and the bit of food, and put the grasshopper on a path marked ONE.

RR1: As above, but for jump length two and two bits of food.

Now let's write a transition rule, to change a juvenile into an adult. But first, we have to set up the clock that synchronizes the changes.

RR1: Write a rule that ticks the clock every rule cycle:

- copy the clock object, which currently contains a marker shaped like a juvenile grasshopper, into the pattern.
- copy the pattern into the result.
- create an invisible "tick" object in the clock.

RR1: Write a rule that advances the clock to the next grasshopper stage after 100 cycles.

- put 100 tick objects in the clock.
- copy the clock, the tick objects, and the current clock marker (juvenile grasshopper) into the pattern.
- copy the pattern into the result.
- [RR2/3] in the result, delete the tick objects and the juvenile marker, and create an adult marker.

RR1: Write similar rules that advance the clock from adult to egg and egg to juvenile.

Now we can write the actual stage-change rules:

RR1: Write a rule to change juveniles to adults.

- Copy the clock (with an adult marker) into the pattern.
- copy a juvenile and its place into the pattern. The juvenile must contain sufficient resources (food, fluid, whatever) to transform into an adult
- variablize any aspects of the juvenile that aren't needed for the transformation
- copy the pattern into the result
- [RR2/3] in the result, delete the juvenile and create an adult, containing the state-objects of the juvenile [**small problem:** not clear that we can do this directly -- do the contained objects get deleted with the containing object? We could do it as a sequence of two rules, though.]
- [RR2/3] delete anything from the state objects that was used in the transformation.
- [note: need to mark the bar graph]

RR1: Write a rule to change eggs to juveniles: Similar to above.

RR1: Write a rule to change adults to eggs:

- copy the clock (with the egg marker) into the pattern.
- copy an adult and its place into the pattern. The adult must contain the resources needed to lay eggs, and the place may need to contain a resource relating to where the eggs can be laid.
- mark noncritical aspects of the place and grasshopper as variable.
- copy the pattern into the result.
- [RR2/3] in the result, create a number of eggs, appropriate to the grasshopper's

current state [may need separate rules for strong, medium, and weak adults, who lay different numbers of eggs]
- [RR2/3] delete the adult grasshopper.

Now lets modify the juvenile-to-adult rule so it updates the bar graph. I won't show the exact rules, but here's the scheme: We'll build each bar of the bar graph into a stack of many places, connected by one-way paths named MOVE-UP, which lead from each place to the place above it. The bottom-most places are connected from left to right by paths marked NEXT-GENERATION. When the clock changes from adults to eggs, a marker, CURRENT-GENERATION, is placed on the NEXT-GENERATION path. Whenever a juvenile turns into an adult, a new object, BAR-MARKER, is placed into the bottom of the bar marked CURRENT-GENERATION. Whenever there are two BAR-MARKER objects in a place, one of them is placed on the path named MOVE-UP. In short, each transformation from juvenile to adult pushes the bar up one division. (With a little more work, we could build a line graph using the same technique: just make the lower bar-marker objects invisible.)

One last detail: rules to restock the simulation during the egg stage. All are of the form:

RR1: Write a rule to put 300 bits of food in random places out of the 1000 possible.
- copy 300 grasshopper-field places into the pattern. **[small problem:** this is pretty tedious. Doctrine might give an example of how to make a single rule execute 300 times... but even that would be difficult without support for arithmetic]
- copy the pattern into the result.
- put a bit-of-food into each of the places.

RD1: Try it. It works.

Summary Comments

Pluses (mostly expressiveness):
- objects-within-objects allow us to give the grasshopper several state variables and fill/empty them as it lives.
- random conflict resolution (for multiply instantiated rules) allows some nondeterminism to be built into the behavior.

Minuses:
- again, basic doctrine is OK, but more difficult situations seem to require fairly sophisticated unsupported problem solving.

- i'm not sure i understand the difference between not showing something in a pattern and making it a variable.
- synchronizing the cycles and setting the grasshopper's time-to-jump are a bit kludgy -- another hit on the lack of a clock.
- can't specify truly random rule ordering, so may get some unexpected/unwanted regularities due to fact the (e.g.) eat rule always fires before drink rule.

SUNSPOTS

[problem]: The appearance of the simulation changes quite a bit during its run, so it's not clear where to start on the drawing. In any case, as with grasshoppers, some decisions have to be made up front, before drawing anything. The big leap of intuition is that the magnetic flux lines can be represented by paths, even though nothing travels directly along them. We assume this is not really such a big gap, since the lines in the problem *look* like OpsTrains paths.

Given that the flux lines will be paths, we have to figure out a way to change their shape. OpsTrains lets rules create new paths, but it doesn't seem to let us control their placement. We assume they just come into being as straight lines between two objects. What we'll have to do is represent a curved flux line as a series of short paths between invisible objects, which we'll call "points." Then we'll need invisible paths, roughly normal to the flux lines, along which the points will travel, dragging the flux lines with them.

We start the walkthrough with a sketch of the invisible paths, including invisible stopping places for the points at several stages of the flux tube's travel [R02, RP1]. We label all the paths OUT and make them unidirectional, away from the sun. As an afterthought, we also create IN paths that almost overlap.

[backtrack]: This seems to be the best way to display the action, but the movement is pretty constrained. What if we want to show the tube bending in a different manner, based on some as-yet-unspecified rules? A possible answer is to make a grid of invisible paths, and have rules determine what route through the grid a point should take. This approach still makes use of the paths-between-points to display the flux lines, but now the lines can be displayed wherever we want.

[attempt1:]

For now though, let's put the complex idea of a grid on the back burner, and see if we can animate a simple sequence. [note: other problems in ST have gone pretty easily, so we have hopes that the simple approach will work here too -- maybe we'll see some way to turn the single, simple sequence into a more general system.

We have the invisible paths that the points will follow, and its time to create the points [R01]. We make three points (top, middle, and bottom of the flux tube) for each invisible path, and connect all paths of the same type (e.g., all top's) with visible paths. The screen now shows three somewhat ragged lines, which we can imagine to be a flux tube.

Next, we need rules to make the points move, dragging the flux lines with them. **[problem]:** The problem doesn't really specify what causes the tube to twist upwards, so it's hard to write rules that are anything more than a sequence of directions for animation. Still, this may be a good way to start: at least we can test the animation itself. We implement a clock object **[problem** again; use the same technique as in grasshoppers], then write a series of rules of the form:

RR1: Write a rule to move all points outward every time the clock reaches time 0:
- set the clock to 0 and copy it into the pattern
- copy a point into the pattern
- copy the pattern into the result
- [RR4] in the result, place the point on the path marked OUT.

We run this and the tube moves outward along the defined paths. We fiddle with the paths and places a bit, and the simulation takes on the appearance we want, up to figure S-3.

Now we'd like to have a couple of the flux lines break loose and stream into space. This turns out to be fairly difficult, because we haven't named the invisible places where the points rest, nor have we named the points. However, we are able to make it work by naming the far-from-sun invisible places and writing a rule in terms of those places and a pattern of paths.

RR1: Write a rule to break loose a magnetic flux line.
- name the places where the line should break loose as E-Break and W-Break.
- copy E-Break and W-Break, each containing invisible points connected by a flux line, into the pattern. Also copy the OUT path from E-Break and W-Break into the pattern, along with the invisible places to which the OUT paths lead.
- copy the pattern into the result
- [RR2] in the result, delete the path between the points
- [RR3] in the result, create new points in the places that the two OUT paths lead to.
- [RR3] in the result, create new visible paths from each point to the new point further out.

We note that this doesn't show the flux lines streaming very far into space, but decide it's a good start. We create two more E-Break and W-Break points, so additional (inner) flux lines can also break loose.

Now we'd like to show material breaking loose from the sun's surface and pushing out through the flux loop. This requires all new paths, as well as new objects, but they are easy to create. We decide (for no principled reason) to start this material on its path

outward at two clock periods after the high flux lines break apart. This requires modifying the clock, which did consist of discrete places. The new clock includes paths between its places. Now we can modify the flux-breaking rule so it also has the effect of putting markers (MATERIAL-FLING-TIME) into the current clock segment plus 2, 3, and 4, and we can add the following rule:

RR1: A rule to fling material from the surface after the flux breaks (material flings for three clock periods; this rule describes one).
- copy the clock marker, the clock segment it is in, the previous two segments, and the paths connecting them into the pattern. Clock segment current-minus-two should have the MATERIAL-FLING-TIME marker in it.
- copy the material-to-fling and the path-for-flinging into the pattern.
- copy the pattern into the result
- [RR4] in the result, put the material-to-fling onto the path-for-flinging
- [RR2] in the result, delete the flux-break-time marker.

We really would like to show some interaction between things, not just a dumb animation. We set the path-for-flinging to feed into the OUT path used by the invisible point that defines the center of the loop, and we write a rule for the flung-material that places it on any available OUT path. We want the flung material to catch up to the flux line, so it has to move faster than the points:

RR1: Rule (one of two) to move flung material at two times the velocity the tube is moving (assume the clock has 10 periods)
- copy the flung material, the place it is in, and the OUT path from that place into the pattern
- copy the clock into the pattern. Make sure it shows time = 0.
- [RR6] variablize the name of the place, if any.
- copy the pattern into the result
- [RR4] in the result, move the material onto the OUT path.

RR2: Rule (two of two) to move flung material onto OUT path:
- as above, except clock should show time = 5.

[**big problem:** This last rule set shows a place where we're really trying to set up a dependent relationship. We'd like this to be easy to modify, but clearly it isn't. Another tactic would be to fire the rule whenever the clock's hand was in a clock segment with a "fling" marker, and to put fling markers in segments 0 and 5. This is easier to modify, but still fairly far removed from our original goal: move the flung material at twice the velocity of the tube.]

The flung material is on the same path of invisible places as the centermost flux point, so we can watch for them to collide and take some action, e.g., recoupling the field above the tube. (Actually, we want to let the material go a bit further, so the recoupled field will contain some of it.) We write this rule:

RR1: Rule to prepare for recoupling.
- put the flung material and a point in the same place, and copy them into the pattern.
- copy the clock and the clock-hand into the pattern.
- copy the pattern into the result
- [RR3] in the result, put a RECOUPLE marker in the current clock segment minus one.

RR1: write a rule to recouple the flux when the clock hand reaches the recouple marker.
- put the clock hand in the segment with the recouple marker, and copy the clock into the pattern
- copy the innermost outward-streaming flux lines into the pattern.
[problem: we can't uniquely identify these flux lines. Backtrack and put marker objects (NEW-E and NEW-W) into the points we created when the flux lines broke loose, as well as the points on the same flux line but one closer to the sun (OLD-E and OLD-W). Now we'll recouple using the same points that broke loose.]
- copy the pattern into the result
- [RR2] in the result, delete the recouple marker
- [RR3] in the result, put a "flare" marker in the current clock segment plus 2.
- [RR2] in the result, delete the flux lines between NEW-E and OLD-E, and the flux lines between NEW-W and OLD-W.
- [RR3] in the result, add new flux lines between OLD-W and OLD-E, and between NEW-E and NEW-W.

Last, we'll write a rule to show some of the effects of the flare.

RR1: Partial Flare.
- copy the clock and the clock-hand into the pattern, along with a FLARE marker in the same segment as the hand.
- copy the flung material into the pattern
[problem: we want to do two things with the flung material: if it's in the high, recoupled magnetic circuit, we want to shoot it outward as cosmic radiation. If it's in the inner circuit, we want to drive it down the tube so it can create X-radiation. But we have to know which magetic circuit it's in. Let's just handle the outward flung material in this rule. Backtrack and modify the prepare-for-recoupling rule so it modifies the flung material by adding a marker to it, OUTER-MATERIAL. Make

sure this doesn't mess up the material-moving rule.]
- copy the OUTER-MATERIAL marker into the pattern, along with the flung material. Also copy flung material into the pattern that is not marked OUTER-MATERIAL.
- copy the NEW-E and NEW-W points into the pattern, along with the points further out to which they connect
- copy the pattern into the result.
- [RR2] in the result, delete the flare marker from the clock
[now we have to create a new path, using the NEW-E and NEW-W markers as guides, and place the flung material on it. [**small problem:** can we create a path and put something on it in a single rule?]
- [RR3] in the result, create a path-objectX-path between NEW-E and NEW-W
- [RR3] in the result, create a path-objectY-path between the points to which NEW-E and NEW-W connect
- [RR3] in the result, create a path between objectX and objectY
- [RR4] in the result, put the flung material marked OUTER-MATERIAL into objectX
- [RR3] put a CONTINUE-FLARE in the next clock segment

RR1: Continue the flare
- copy objectX and its path into the pattern
- copy the clock, the clock hand, and the CONTINUE-FLARE marker into the pattern (hand and marker in same segment)
- copy the pattern into the result
- [RR2] in the result, delete the CONTINUE-FLARE marker
- [RR4] in the result, put the flung material onto the path out of objectX.

RD1: Time to try it out. First the programmer has to set everything back to the start state. This is a hassle, which the system should help with. Now run the program. The tube bends, breaks the surface, rises, and breaks off a couple of flux lines. Then material breaks loose from the surface, and rises through the tube. Flux lines recouple, and material is ejected as cosmic radiation. Since there's no stop on the simulation, the unbroken parts of the flux tube continue to rise until the invisible paths end.

[attempt2:]

Brief consideration of the more general simulation, where we use a grid of places: We can keep lots of state information hidden in each object, which should give us a lot of options in controlling the action. Also, we can look for patterns of objects and

paths, so we don't need to use the ShowTrains method of sending messages to adjacent flux lines whenever a flux line moves. These two features really help with the solution. However, we're still working in a different editor for the rules, which makes it harder to keep track of what the graphic changes should/will do.

Summary Comments

Pluses:

- the paths work fairly well as flux lines, even though their curvature is a little choppy.

Minuses:

- using paths as graphic elements isn't supported by doctrine.
- we had to build our own clock again -- it's getting easier with practice.

Big Minus, either for expressiveness or my solution:

- In attempt1, I've just kludged together an animation, without really simulating the physical interactions. It still seemed like a lot of work. Maybe paths-as-flux-lines was the wrong way to go, even though the graphic similarity is really powerful. Should I have used moving places to represent sections of the tube? It seems that I'd still be constrained by fixed paths. Maybe I need to be able to specify the trajectory of a new path when I create it with a rule.
- Even in attempt2, the physical interactions are minimal -- the flux lines hang together. Might be that we need to bring more physics in, but that calls for velocity fields, induction, etc. Is this an area that ChemTrains just isn't suited for? Am I totally on the wrong path?

XEROX

[disclaimer: this is a micro-walkthrough]: Running short of time, so I'll just hit the high points.

R01: Draw these objects: both dicotrons, the original, the paper, a lens, both brushes, the fusion rollers, the belt.

[small problem: the belt can't move in a circuit. draw the belt as many small sections. also draw the paper as many small sections.]

R02: draw many places for the belt sections and the paper sections to stop in.

RP1: Draw the following paths: Light paths from the original through the lens to the belt place below it, charge paths from the charging dicotron to the belt place, toner paths from the toner brush to the belt place, toner paths from the belt place to the cleaning brush, charge paths from the transfer dicotron to the paper place, toner paths from the belt place to the paper place.

RP1: Also draw paths for the belt and paper sections to move along.

[problem: need to time things. create a clock as in sunspots, grasshopper, with about 50 divisions (assume a copy covers four belt sections, and we want to advance the belt one segment per tick). Note that this might get tricky. Does the belt have to stop while it's being exposed?]

RR1: Write rules to activate the charge dicotron:
If the clock hand is in division 1,2,3, or 4, then put MINUS objects onto the ten paths from the charge dicotron to the belt place.

RR1: Write rules to pick up the charge on the belt:
If any belt segment is in the same place as free MINUS objects, put the MINUS objects into the segment.

RR1: Write rules to discharge the belt when exposed to light:
If any belt segment is in the same place as free PHOTON objects, delete one MINUS object (and one PHOTON object) for every PHOTON object.

[**Note:** The entire simulation depends on this sort of one-for-one conversion and transfer, so it's probably a good idea to write out this one rule as an example of how OpsTrains handles the problem. Here goes:

RR1: Rule: discharge the belt, one MINUS for each PHOTON, when exposed to light.
- put a single MINUS object in the belt-segment and a single PHOTON object in the belt place.
- copy the belt-segment, its place, the PHOTON object, and the MINUS object into the pattern.
- copy the pattern into the result.
- in the result, delete the MINUS object and the PHOTON object.

If I understand OpsTrains correctly, this rule will instantiate itself for every possible set of objects to which it can apply. So if there are five toner particles, the rule should instantiate and fire five times. I'm not sure what order they fire in, though -- one per cycle? or all per cycle, intermixed with other rules? or all per cycle, all at once?

RR1: Write rules to flash the light on the paper.
- (light areas) If the clock hand is in division 10 (whatever), and if there is no DARK object in a ORIGINAL segment, then put five PHOTONs on the path leading from the ORIGINAL to the belt place.
- (dark areas) If the clock hand is in division 10 (whatever), and if there is a DARK object in a ORIGINAL segment, then put one PHOTON on the path leading from the ORIGINAL to the belt place.

RR1: Write rules to pick up toner:
If any belt segment is in the same place as free TONER objects, put one TONER object into the belt segment for every MINUS object.

RR1: Write rules to move toner from the brush to the belt place, and to endlessly resupply the toner brush:
If there are less than 20 toner particles in the belt place, put five toner particles from the brush onto the paths to the belt place, and put five new toner particles in the toner brush.

RR1: Write rules to activate the transfer dicotron:
If the clock hand is in divisions 20, 21, 22, ... whatever, then put MINUS objects onto the paths from the transfer dicotron to the paper.

RR1: Write rules to charge the paper:

If any paper segment is in a place with free MINUS objects, then put the MINUS objects into the paper segment.

RR1: Write rules to differentially transfer toner from the belt segment to the paper place.

If any paper segment is in a place with a path leading to a place with a belt segment, then for every MINUS particle that the paper has beyond what the belt has, put one TONER particle onto the path from belt to paper.

RR1: Write rules to pick up the toner on the paper.

If any paper segment is in a place with free toner particles, then for every MINUS particle in the paper segment, put one TONER particle into the paper segment.

RR1: Write rules to clean the belt.

If any belt segment is in a place with POSITIVE particles, then delete (if it exists) one minus particle and (if it exists) one TONER particle for every POSITIVE particle.

RR1: Write rules to charge the cleaning area.

If there are less than 20 POSITIVE particles in the belt area connected to the cleaning brush, then put five POSITIVE particles from the cleaning brush onto the paths to the belt place, and add five new positive particles to the cleaning brush.

RR1: Write rules to fuse the paper.

If there are TONER particles in paper that is in a place with a path leading to a FUSION roller, then delete the TONER particles and replace them with FUSED-INK particles.

RR1: Write rules to move the belt and paper, and to advance the clock every five ticks (this gives time for all the toner particles etc. to transfer before the belt segment moves on).

- If <no condition> put one *tick* object in the division of the clock containing the hand.
- If the division of the clock containing the hand contains five *tick* objects, and if the division is number 35, 36, ..., <whatever>, then put each segment of the paper onto the path named PAPER-FORWARD.
- If the division of the clock containing the hand contains five *tick* objects, then put every segment of the belt onto the path named BELT-FORWARD.
- If the division of the clock containing the hand contains five *tick* objects, then put the hand onto the path named NEXT-DIVISION. **[note:** this has to be the last rule in the clock-dependent sequence]

RD1: Try it. It works, except the segments don't change orientation when they go around corners. Fix this by putting markers for TOP, LEFT, RIGHT, and BOTTOM in each belt segment place, and changing the belt segment accordingly. **[problem]:** This requires that the new segment pick up exactly the contents of the old one, something we're not sure how to do in OpsTrains without using two rules and transferring the contained objects to an intermediate place.

RD1: Try to simulate faults:

- Fault 1: Dicotron-1 charges too strongly so copies are too dark. We modify the rule that puts MINUS objects onto the paths to the belt place so it puts a total of 15 objects. Now the belt is more negatively charged when it picks up toner, so it picks up more. If dicotron-2 is strong enough, this will make a darker copy. We may need to fiddle with dicotron-2 as well, before this fault will show up (but the correct operation will work with the new dicotron-2 setting). A bigger problem is that we can't exactly compare the original to the copy. More fiddling, but certainly within OpsTrains' capabilities.
- Fault 2: Toner brushes are too negatively charged. In our hasty simulation, we neglected to model this aspect of the machine. We revise the simulation so the toner transfer from the brush is handled similarly to toner transfer onto the paper (i.e., by considering the balance-of-power), and this fault can now also be modelled.

Summary Comments

Pluses:

- the simulation is, in some difficult-to-define sense, a *real simulation*, not just an animation.
- putting charged particles and toner "into" the segments of the moving belt seems especially natural.
- this walkthrough went very quickly, with no major problems.

Minuses:

- representing the belt and paper as segments was an easy decision for me, but that's what Schrager does in his simulation. It might be a bigger leap for someone new to the problem.
- we'd really like to just rotate the existing belt segments as they round the corners, not completely recreate them -- same problem we had in the doorbell with the arm.
- from Schrager's work, we know that balancing the parameters in a full-scale simulation is almost impossible without an underlying quantitative model. We're skirting the edge of the need for arithmetic in ChemTrains.

ShowTrains
Walkthroughs of Four Reserve Problems

JR
June 1990

DOORBELL

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

2. [D1] P sketches a rough graphic of the doorbell as described.

3. [D2] P identifies the following as objects because they move: the button, the iron core, one of the contacts, the arm.

   [D2] P identifies the following as objects because they change: the spring, the magnetic field.

   [small problem] P decides that electric current will be represented as moving electrons, so electrons must be objects.

   [small problem] P decides that the magnetic field will be a single object that changes.

4. [D3] P identifies the following as places where things stop or rest: The coil (core stops, field changes), closed and open positions for the moving contact, the rest position for the iron core, the location of the spring, and the location of the arm.

[big problems]: Where are the arm and the spring located? The spring changes shape as it moves; in fact, one end doesn't move at all. Similarly, one end of the arm is fixed at the pivot. P decides, without any doctrinal prompting, that the arm and the spring will have to be identifed by a series of graphics, so they look like they're moving even though they're really staying in one place. Having already conceived of the spring as a separate object, P makes a place for it and a place for the arm.

5. [D4] P identifies the following additional places where things interact: the location of the button (where the user will click) and the location of the bell (where the clapper will cause a graphic "DING" to appear)

6. [D5] P considers each object in turn and traws the following paths, named according to [D5a]:

   - the iron core moves from its resting place into the core and back, so two paths are needed, named INTO-COIL and OUT-OF-COIL, and almost overlapping.

   - the spring and the arm don't move, so no paths are required.

   - the moving contacts travel along with the arm from its rest to its ringing position and back, so P draws two paths, almost overlapping, parallel to the paths for the iron core. **[problem:** P can't be sure that the items will travel simultaneously, even if they are placed simultaneously on parallel paths of the same length. We haven't specified that detail in the spec.]

   - the real button travels up and down, but P decides to display button position with two different graphics, to be toggled between by clicking on the button place. So no path is needed here.

   - P identifies the electric circuit as the route over which electrons will move, and creates unidirectional paths, all labelled FROM-BAT, that lead from the negative pole of the battery to the button place, from there to the unmoving contact place, and from the coil place to the positive pole of the battery.

   [problem]: The circuit isn't complete P must connect the moving contact to the coil. In ShowTrains, there's no way to connect a path to a moving object. P reconceptualizes the moving contact as a bridge across two fixed contacts. The fixed contacts are simply graphic elements on either side of a single place, which will be the place where the bridge-contact must be for the circuit to be complete. P connects a path from the button to the fixed contact place and a path from that place to the coil. Later, P will write a rule to produce the bridging effect.

7. [D6] P is now ready to create the Rules that change the objects. P decides to use five steps to display the movement of the arm and spring.

[problem, backtrack]: At this point, P realizes that the multiple arm and spring graphics must coordinate with the movement of the arm and contacts over a path. There doesn't seem to be any way to do this in ShowTrains. P decides that what's

good for the goose is good for the gander, and redefines the moving core and the bridging contact as unmoving objects, whose graphics will change to simulate movement. P demonstrates a set of four rules that make the arm, core, and spring "move" in the presence of the magnetic field. All rules have the form:

- If the arm, spring, and contacts have their current appearance (everything is highlighted, [D6a]), and if there is a magnetic field object in the coil place, then change the graphics of the arm, spring, and contacts so they appear one step closer to the coil.

Next, P demonstrates a rule to "ring" the bell:

- If the arm has the closest-to-the-bell appearance, change the bell's graphic to a highlighted bell with the word "ding."

P demonstrates a set of rules to swing the arm back, of the form:

- If the arm, spring, and contacts have their current appearance (everything is highlighted, [D6a]), and if there is no magnetic field object in the coil place, then change the graphics of the arm, spring, and contacts so they appear one step further from to the coil.

P starts to write a rule to create and collapse the magnetic field, but this depends on electron flow, which P hasn't thought through yet, so P postpones the problem.

8. [D7] P is now ready to write the rules that show electron flow. **[problem]:** P would like to have each electron "sense" that there is already an electron at the end of the path it's on, or perhaps that there's one behind it. But ShowTrains doesn't allow this. P decides to have a rule that has electrons "push" extra electrons out of a place:

   - If there are two electrons in a place, then put one of them on the path named FROM-BAT (which leads to the next place).

   P worries that single electrons moving along long wires won't be very interesting graphically. From [D11], P gets the idea of stringing many small places along each wire. Thinking about how these objects will work, P realizes that the battery, coil, and button are functionally identical — so current will flow through the circuit even when it is not a closed circuti. From [D18], P gets the idea of using marker objects to indicate whether a place is conductive or not. P creates many small places along the wire, placing a CONDUCTOR object in each. P then revises the conduction rule:

- If there are two electrons in a place, and there is also a CONDUCTOR object in the place, then put one of the electrons on the path named FROM-BAT.

P can now write rules for the button and contacts:

- If there is a *click* object and no CONDUCTOR object in the place called BUTTON, then put a CONDUCTOR object in that place, delete the *click* object, and change the graphic of the button object to button-down.

- If there is a *click* object and a CONDUCTOR object in the place called BUTTON, then delete the CONDUCTOR object and change the graphic of the button object to the button-up.

- If the contacts object has the closed appearance (remember, nothing actually moves here), and there is not CONDUCTOR object in the contacts-place, then put a CONDUCTOR object in the contacts-place.

- If the contacts object has the open appearance, then delete the CONDUCTOR object (if any) from the contacts-place.

9. {D6] P is now ready to return to the problem of creating and deleting the magnetic field, which should depend on the flow of electrons through the coil. **[problem]:** P can't just write a rule saying that two electrons in the coil cause the field to form, because the rule changing two electrons to one might fire first. However, this rule only fires if there's a conductor object present. So P removes the conductor object from the coil and writes the following rule, which moves the electrons through the coil and creates a magnetic field:

- If there are two electrons and no magnetic field in a place named COIL, then put one electron on the FROM-BAT path out of the coil, and create a magnetic field object in the coil.

P now writes a similar problem that will collapse the field when electron flow stops:

- If there is one electron in the place named COIL, then delete the magnetic field object (if any) in the coil.

[really big problem] P can't get this to work without rule ordering, since the field may be deleted as soon as it's formed. P checks the Advanced Guidelines and discovers [D20]. P decides this is too complex, but eventually develops something

that is very similar and only slightly simpler. (These replace the last two rules above):

- (Rule to create/maintain field): If there are two electrons in a place named COIL, then put one electron on the FROM-BAT path out of the coil, and delete the magnetic field object in the coil (if there is one), and create a magnetic field object in the coil, and delete the FIELD-DECAYING object in the coil (if there is one).

- (Rule to flag each cycle of no current) If there is one electron and a magnetic field in a place named coil, then put a FIELD-DECAYING object in the coil.

- (Rule to collapse the field) If there is one electron and a magnetic field and two FIELD-DECAYING objects in the coil, then delete the magnetic field and the FIELD-DECAYING objects.

10. [D10] P tries the simulation. It works, but the arm cycles back and forth between full ring and one-away from full ring. P fixes this by changing the graphic of the contacts on the arm-release cycle, so they don't appear to close again until the arm is fully at rest.

Summary Comments

Pluses:

- the doctrine got everything started well, but it fell apart because we couldn't swing the arm or stretch the spring.
- advanced doctrine helped with the CONDUCTOR marker.

Minuses:

- we can't show the moving arm, and since everything is physically linked to it, we end up doing mostly bit-map animation instead of object movement.
- even when it looked like we'd use paths, we didn't know how to synchronize them.
- the field collapse solution is awful, even with doctrine.

Big Hit on Expressiveness: TIME (same as OpsTrains):
- there's no easy way to control timing and synchronization of reactions and movement. This makes it hard to coordinate movement and hard to implement the adjustable parameters (voltage, spring constant, inertia) described in the desiderata. Timing could also be used to solve the field collapse problem — we'd like it to simply collapse as time passed, unless it was renewed by electrons flowing thorugh the coil.

GRASSHOPPERS

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

2. [D1] Before P can sketch the simulation, he must conceive of some general way in which it will work. Doctrine gives no help with this, nor do any of the examples, since none of them deal with large numbers of identically behaving objects.

   P decides that the general operation of the simulation will be as follows [identical to OpsTrains]:

   - There will be a grid of places, highly interconnected by paths. The places will represent the field in which the grasshoppers live, and the paths will represent jumps of varying distance and direction between places.

   - Individual grasshoppers — juvenile and adult — will jump at random (with some constraints) from place to place.

   - All grasshoppers act as parallel, independent units.

   - Any place may contain any number of the resources that the grasshoppers need, and a grasshopper may consume several units of various resources within a single place.

   - When a grasshopper is out of resources in a place, or when it has been in a single place for a "long enough" time (varied randomly), it will jump.

   - Each jump increases the grasshopper's need for its energy resource.

   - If a grasshopper's energy drops too low, it dies.

   - Stage changes (egg to juvenile to adult) are synchronized for all grasshoppers. When the stage-change "bell" rings, all living grasshoppers that currently have the necessary resources will change stage. Other grasshoppers will die.

   - For each generation, the simulation will graph the number of transitions from juvenile to adult.

   P sketches the simulation, including a grid of places (patterned after coins tightly packed on a flat surface), a bar graph, and a clock-like object that advances from stage to stage.

3. [D2] P identifies the following as objects because they move: adult and juvenile grasshoppers. P creates one of each. P also identifes the pointer of the clock as an object. P creates a round dot to act as the pointer.

   [D2] P identifies the following as objects because they change: eggs, resources, and bars on the graph. P creates an egg, several different resources (R1, R2, R3, ...) , and a single bar (for one generation).

4. [D3] P identifies the following as places: each place in the field and the location of each bar in the bar graph. P decides that a good simulation will need about 1000 places, but he only creates 25 to start with. **[problem:** creating a grid of 1000 interconnected places is tedious. Block copy will help some, but the interconnections between the blocks will still have to be done by hand.] P also identifies the locations where the clock pointer will rest, and creates a series of three pie-wedges, forming the clock face.

5. [D4] P isn't sure what user interaction will occur, so he creates no places for it. Object interaction (grasshoppers with resources) takes place in the field places.

6. [D5] P interconnects the places so that from any place a grasshopper may jump one, two, or three places away, in any of the six directions leading from the place. P labels all the one-jump paths ONE, all the two-jump paths TWO, and all the three-jump paths THREE [D5a]. **[problem, design note:** P must be thinking ahead to using random conflict resolution of ShowTrains, which will put a grasshopper on a randomly selected path if several have the same name. Neither doctrine nor examples really discuss this, since the raw problems all had to avoid random behavior.]

   P also creates paths from one segment of the clock to the next, named NEXT-SEGMENT.

7. [D6] P creates "object-changing" rules, which have to do two types of things: cause the grasshoppers to consume resources (thus affecting their state), and transform the grasshoppers to the next stage.

7a. **[big, big expressiveness problem]:** P now considers how to record the state within a grasshopper. [D19] suggests, use a name or graphic to denote an object's state. This would work if the grasshopper only had one attribute that varied, such as energy level. But P wants to consider several resources: energy, contact (with an opposite-sex grasshopper), space, etc. There doesn't seem to be any easy way to do this.

- P considers using several objects for each grasshopper, with state information carried in the objects' graphics. But moving these objects around together seems difficult. If there are two grasshoppers in a single place, one grasshopper's stomach may hop off with another grasshopper's feet.

- P considers using a "table" of values for each grasshopper, pointed to by each grasshopper's unique name. **[doctrine/design failure:** there's nothing in the doctrine giving the idea of pointers.] Each grasshopper object will have a grasshopper graphic that triggers rules. For each grasshopper, there will be several additional objects, on the side of the screen (not moving), in places named STATE-TABLE. Each STATE-TABLE place will also hold a marker object with the same name as the grasshopper. When the grasshopper object consumes a resource of a given type, the effect will be recorded by changing the graphic of the appropriate side-screen object. However, this won't work without writing a separate set of rules for each grasshopper name, since matching variablized objects from condition to action isn't supported in ShowTrains. P decides to take this approach, which is supported by rule copy. **[design not specified:]** But the specs don't make it clear whether the rules will act in parallel or not.

P writes a rule to simulate the grasshopper consuming a unit of a resource, in this case food:

- (attempt:) If there is a grasshopper named #1 in a place with an object named food, and there is a grasshopper-marker named #1 in a place named STATE-TABLE, then delete food and change the graphic of the object named STOMACH in the STATE-TABLE to full.

P realizes that he really wants to just increment the contents of the stomach, so he revises the pointer scheme to point to several off-screen places fodr each grasshopper, in which:

- each place-name indicates the state variable (energy, water...)

- each place contains a marker with the same name as an individual grasshopper.

- units of the state can be added or subtracted by placing objects in the place.

P can now write a more useful rule for consuming a resource:

- If there is a grasshopper named #1 in a place with a food object, and there is a grasshopper-marker object named #1 in a place named STOMACH-STATE, then instantaneously move the R1 object into the place named STOMACH-STATE.

P creates and writes similar rules for other resources, all for the grasshopper named #1. P can now use block-copy to duplicate the grasshopper and it's state-places, and new rules will be created which apply to the new objects and places. **[problems:** it's not clear from the specs if the right things will get renumbered in the block-copy. it's also a big problem if P decides to change any rule after the block copy is done.]

7b. P is now ready to create rules that change the state of a grasshopper. State change depends on two things: First, the state-places for the given grasshopper must be filled with the necessary contents. This is easy to test in ShowTrains. Second, the clock must indicate that it's time for the transition to occur.

P first writes rules to advance the clock [D20 gives some help]:

- If [nothing specified], then put a "tick" object in the segment of the clock that holds the pointer. (This fires once on every cycle).

- If there are 100 tick objects in the segment of the clock with the pointer, then put the pointer on the NEXT-SEGMENT path and delete all the tick objects.

Now P can write transition rules, which are of the form:

- If there is a juvenile-grasshopper object in any place with the name #1, and there are 10 food-objects and a grasshopper-marker named #1 in a place named STOMACH-STATE, and there are 5 water-objects and a grasshopper-marker named # 1 in a place named FLUID-STATE, then delete the 10 food objects and the 5 water objects, and change the juvenile-grasshopper (graphic) to an adult-grasshopper.

This is the point where P would like to increment the bar graph. P decides to do this as follows **[small problem:** there's not much doctrine supporting this]:

- Each bar of the graph is represented by a tall, narrow place, into which leads a path named TO-BAR from a place named GENERATION. The places named GENERATION are connected together by paths named NEXT-GENERATION, and one of the places contains a marker, CURRENT-GENERATION, which is advanced to the next place when the clock arrives in the adult-generation segment.

- When a grasshopper becomes an adult, the transformation rule (bullet above) places a small dot onto the TO-BAR path leading out of the place with the CURRENT-GENERATION marker. This dot travels up to the BAR, and pushes the existing dots out of the way, raising the height of the bar.

P now needs to write rules that will kill off grasshoppers that have consumed insufficient resources. This fairly easy, although once again it requires writing a single rule, then copying it for all the grasshoppers. Rules are of the form:

- If there is a grasshopper named #1 in any place, and if there are no FOOD objects in the STOMACH-CONTENTS place containing the grasshopper-marker named #1, then delete the grasshopper named #1.

8. [D7] P now creates the rules to move the grasshoppers. This requires P to use the random functionality of ShowTrains, which isn't represented in any of the examples. P writes rules of the following form:

   - If an adult grasshopper named is in a place and there are no other objects in the place (i.e., no resources -- P doesn't want to specify each possible resource that must be absent), then put the grasshopper on the TWO path.

   - If an adult grasshopper is in a place and there are no other objects in the place, then put the grasshopper on the THREE path.

   These two rules will cause the adult to be put on one of the many TWO and THREE jump-length paths leading out of a place, if there are no resources. P must also modify the above rules so they decrease the grasshopper's energy state. This requires that the rules refer to a specific grasshopper, which means that copies of the rule have to be made for all the grasshoppers.

9. [D8] P writes several user-interaction rules that stock the field with resources. The fact that all field places have the same name makes this fairly easy. Rules are of the form:

   - If there is a *click* object in the place called STOCK-FOOD, then put food in 30 places called FIELD-PLACE.

10. [D9] P tries the simulation. It works, so far as specified above, but the egg-laying behavior of the grasshoppers hasn't yet been defined. **[problem]:** The problem is this: since grasshoppers have unique numbers, and there are only a fixed number of rules, how can the eggs be numbered when they are layed?

P solves this problem, again without help from doctrine, by creating places in which to maintain a stock of adults, juveniles, and eggs. When a grasshopper lays eggs, a number of eggs are selected from the stock and placed where the adult was; the adult is returned to stock. When the eggs turn into juveniles, they are replaced with juveniles having the same number, and the eggs are returned to stock. The juvenile-to-adult transformation is similar.

P also adds a rule to make a grasshopper jump when it's been in a place long enough. This is done by having the grasshopper drop grasshopper-droppings with its number on them into the current place, one dropping per rule cycle. When three dropping accumulate, another rule forces the grasshopper to jump and deletes the droppings.

P still needs to write rules to restock the environment, probably while grasshoppers are in the egg stage. These can simply put n resource items into n different (unspecified) field places.

Summary Comments

Pluses:

- random conflict resolution makes it easy to simulate underspecified activity.

Minuses:

- Since objects don't have a complex structure (beyond name and graphic), we have to keep data about each grasshopper in a separate table. This requires numbering the grasshoppers individually:

- Numbering individual grasshoppers wouldn't be so bad if we could use variables to say "if grasshopper <X> is in a place with food, then put the food in stomach <X>." But ShowTrains variables don't transfer that information from condition to action, so we have to write individual rules for each grasshopper. Without something like a spreadsheet, this is impossible to track.

- There's not much doctrinal support for using the random features. In fact, it's not even clear from the spec exactly how they work.

- Once again we've run into a need for timing, even in an essentially random simulation. Forcing the user to build clocks and leave grasshopper droppings seems a little kludgy.

SUNSPOTS

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

2. [D1] P sketches a rough graphic of the sunspots as described, just before the flare is about to occur.

3. [D2] **[big problems:]** P tries to identify "objects" that move or change. The particles ejected when a flare is produced seem to be objects, but it's not clear whether the flux tube itself is an object. It certainly moves and changes, so doctrine suggests it should be an object, but P can't decide what shape or location it should have.

   Advanced doctrine suggests that complex movement may require many small places, so P extends this idea and decides to construct the tube out of many small tubelets, which will move along outward-extending paths through many small, invisible places. P constructs many tubelets and arranges them in the initial configuration.

4. [D3] P decides that tubelets could stop most anywhere, and decides that rules should determine where the stop and how they move. To allow this, P constructs a radially symmetric grid of places (i.e., lying on radii of the sun), interconnected with paths labelled UP, LEFT, UP-LEFT, DOWN, DOWN-LEFT, etc. [D5a] Block copy helps somewhat with this task. [D22] P gives each place a unique name, using a double-marker scheme (e.g., a place might have a V-2 marker and an H-5 marker). [D18]

5. [D4] P decides that there are plenty of places for interaction and doesn't add any more. For now, P decides to create a single simulation, with no options for user interaction.

6. [D5] P decides that the movement of the tubelets is sufficiently enabled by the directional paths already drawn. P considers drawing additional paths to simulate the movement of the radiation, but decides this is an unnecessary complication.

7. [D6,7] P is now ready to create the Rules that move the tubelets. Some of the constraints P would like to express are:

   - The tubelets hang together -- if one moves, the adjacent tubelets space themselves out to fill the intervening space.

   - If the intervening space becomes too large, another tubelet is formed.

- The graphics of the tubelets change to reflect their current orientation.

[problem:] All these rules require that a given tubelet have some information about its neighboring tubelets. P decides to incorporate the information into an object's graphic. [D19] This promises to work fairly well, since a tubelet that is connected (e.g.) to tubelets on the LEFT and RIGHT paths needs to have a horizontally oriented graphic, while a tubelet connected on the UP and DOWN paths needs a vertical graphic.

[big problem:] Even though the graphic now contains information concerning its neighbor, writing rules to use this information is not simple. We really want rules that say something like, "If a horizontal tubelet moves up, then change it to an inverted V graphic, and change the DOWN-LEFT and DOWN-RIGHT graphics so they point to the inverted V." But this is difficult in ShowTrains, since we can't refer to patterns of places and paths. P decides [no doctrinal support] to use a messaging system, as follows:

- Rule: If a horizontal tubelet is in a place with a MOVE-UP marker, then change the tubelet to an inverted V graphic, and send an "CHANGE-TO-UP-LEFT" message down the DOWN-RIGHT path, and a "CHANGE-TO-UP-RIGHT" message down the DOWN-LEFT path, and put the tubelet on the UP path. (This sounds a lot more complicated than it is with ShowTrains visual interface.)

- Rule: If a horizontal tubelet receives a "CHANGE-TO-UP-LEFT" message, then change its shape to a horizontal tubelet with the left end angled upward.

- Rule: (need rules like above that allow each graphic shape to send the appropriate messages down the appropriate paths when they are moved, and for each graphic shape to change for each of the possible messages) **[problem:** lot of rules!]

The above rules effectively tie together the tublets, so that when one moves, the adjacent tubelets bend to point to its new position. P can now write rules that actually put the MOVE markers into the places with the tubelets. The rules essentially specify that sharp angles, like the inverted V, cause stress, causing adjacent tubelets to move in order to reduce the stress. Less sharp angles, like the horizontal line with a single side angled up or down, are not stress causing.

- Rule: If a tubelet has an inverted V graphic, then send a MOVE-UP message down the DOWN-LEFT path.

- Rule: If a tubelet has an inverted V graphic, then send a MOVE-UP message down the DOWN-RIGHT path.

Since rule ordering is random, either rule could fire, but that will cause an adjacent tubelet to move up, with will change the inverted V shape, and then neither rule will apply. **[problem again:** lot of rules!]

P now tries this much of the simulation. He drags a single tubelet out of line, and the adjacent tubelets deform, and then move into a less-stressful configuration.

[key decision, possible win for ShowTrains: P decides that this is a neat way to control the simulation. He will let the user drag the magnetic flux tube into new configurations, and have the simulation respond accordingly. This seems to be a direct fall-out of ShowTrains' visual approach. Note, however, that we can't directly sense the user's movement of the object. We might give the user a flux-drag tool, which can be attached to a section of flux and driven around by clicking on compass points -- kind of kludgy.]

P would like to create rules that break the flux lines apart if they are too sharply bent. This is a **small problem,** since P conceives of a sharply bent line in terms of several neighboring points. P modifies the graphic interactions among points to give finer grained display of how sharply lines are bent at their apex. **[Lot more rules].**

P also decides that time should be a factor: a sharply bent flux line that stays sharply bent for a long enough time (because the other tubelets can't move fast enough to relax the stress) should cause a break in the tube. **[problem]:** P needs to time what's going on. He uses the scheme of dropping *tick* markers into the place with the sharply bent graphic on each rule cycle. Now he can watch for, say, 5 *tick* markers in a place that contains a sharp-bend graphic, and fire a flux-breaking rule on that condition:

- Rule: Break a flux line that's been sharply bent for 5 ticks. If there are five *tick* objects and a sharp-bend tubelet in a place, then put a BREAK message on the DOWN-LEFT and DOWN-RIGHT paths, and delete the ticks, and change the tubelet from a three-flux-line tubelet to a two-flux-line tubelet.

- Rule: If a tubelet and a BREAK message are in the same place, then change the shape of the tubelet to a tublet with one flux line pointing upward, and put an EXTEND message on the UP path. (This isn't exact -- there have to be several rules, for the several tubelet shapes that might receive the BREAK message.)

- Rule: If an EXTEND message is in a place with an UP path, then create a vertical flux line in that place, and put the EXTEND message on the UP path.

The simulation will now show flux lines breaking loose when the user pulls the tubelets into too sharp a bend. **[small problem]:** However, at this point P realizes he should have been working with flux lines instead of tubelets. He continues with tubelets, intending to redo the simulation later.

P now creates a set of rules, similar to the tubelet rules, to drag a blob of material out of the sun's surface. The user drags a bloblet, and other blobs follow. Their ability to hang together is determined by their graphic appearance.

P is ready to model a flare. The rules must look for bloblets in the same place as, or between, two close but oppositely directed flux lines, and then cause those lines to form a new circuit. **[big problem]** The problem is, again, that rules can't recognize configurations of places, paths, and objects. P again lets the graphics carry the burden. Whenever a new flux line is formed, it sends out messages to all its neighbors, indicating its presence and its direction. Any neighboring flux line that receives such a message returns it. The new flux line and the neighbor both change appearance to indicate their adjacency, as well as the direction of the adjacent line (same or opposing). Now the flare can form whenever flung material enters a place in which a flux line has the graphic that indicates an adjacent, but opposing, flux line.

- Rule: Start Reconnect. If a flux line with one of the adjacent-opposing graphics is in the same place as a piece of flung material, then change the graphic to point in the direction of the adjacent opposing line, and send a CONNECT message to the adjacent opposing line, and change the flung material to a cosmic-ray object, and put the cosmic-ray object on the up path.

- Rule: Complete Reconnect: If a CONNECT message and a flux line are in the same place, then change the graphic of flux line to point to the adjacent opposing line.

(P can add similar rules to create high-energy particles below the reconnection, which follow the old tubes down to the sun's surface, where they produce X-rays.)

8. [D9,10] P tries the simulation, then adds a line in background for the sun's surface.

Summary Comments

Pluses:

- We are actually simulating the behavior of units of physical "stuff," not just doing a preprogrammed animation. Giving the user the ability to drag the flux tube is a nice touch. This approach seems to be a natural result of ShowTrains' graphic approach to specifying what happens.

Minuses:

- We have to write a huge number of rules. This may be inevitable when there are a huge number of different graphic appearances that small objects can take on, but the task seems pretty formidable. Note that the problem of trying to maintain multiple attributes for an object (ala grasshoppers) didn't show up, but it would if we wanted to keep track of things like energy content and velocity, along with position and orientation.

- There's a fair amount of kludging required whenever we want to recognize adjacent flux lines, etc -- even more when a more complex pattern is important. Being able to specify a pattern of paths, places, and objects would be a big plus.

- Timing is a minor problem again.

- The paths aren't really used as paths, since there's a solid grid of places. The paths just specify which adjacent place to move into... agent sheets?

- There's no easy way to tell when the user has moved something, or to constrain its movement. A "constrained movement" option, that forced things to be dragged along paths and left *source-pathname* objects in the places they were dragged to (or *target-pathname* in the from places) might be useful. Need to think more about user interaction in general.

XEROX

[disclaimer: micro-walkthrough] Running short of time, so I'll just hit the high points..

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

2. [D1] P sketches a rough graphic of the machine as described.

3. [D2] P identifies the following as objects because they move: the belt, the paper, the toner particles.

   [D2] P identifies the following as objects because they change: the original, the dicotrons (turn on and off), the brushes (charge state changes).

   [small problem] P decides that charge will be represented as moving objects, so electrons are objects.

   [small problem] P decides that the light will be represented as moving objects, so photons are objects.

4. [D3] P identifies the following as places where things stop or rest: The place for the original, the places for the brushes, the places for the dicotrons, several places where things happen to the paper, the place where the belt sits. P creates these places and names them all <whatever>-place.

   [small problem] P has probably done extra work by making the dicotrons and brushes objects instead of places, but we'll see.

5. [D4] Object interaction takes place in the places already created. P creates a COPY place for the user to click in to make a copy.

6. [D5] P starts the define paths step by attempting to define a path for the belt. This exposes the problem: the belt doesn't "move," it rotates, a motion ShowTrains doesn't support. P rethinks the problem and redefines the belt in terms of belt segments, moving from one small place to another. **[problem:** no doctrine or example supports this, although the problem mentions "segments" of black and white]. Now P can draw paths, labelled BELT-FORWARD, between each of the belt places. P also draws paths for photons, charge (electrons from the dicotrons), and the paper (also redefined in terms of segments), labelling them TO-BELT, etc. P is a little confused about where things will travel when the dicotron behind the paper is

active, but decides on two sets of paths, one from dicotron to paper-place, another from paper-place to belt-place.

7. [D6] P begins to create object changing rules. From the problem statement, P realizes that dicotron-1 may have different states of charge. P decides to represent this by different graphics. The first rule handles a medium charge.

   - Rule: fire the charging dicotron-1: If dicotron-1 has a "medium-charge" graphic [D19], then create 5 electron objects in the dicotron place and put them on the path TO-BELT.

   - Rule: pick up the charge on the belt: If a belt-place has a belt-segment and an electron object in it, then change the belt-segment graphic to show an electron on the belt, and delete the electron object.

   [big problem] P realizes that this approach will require separate rules for every combination of precharge and free charge in the belt place (i.e., if the belt already has two electrons in its graphic, and there are three floating, then change it to show five). This seems intolerable. P considers using the approach used with grasshoppers, i.e., each belt segment is numbered, and the charges are kept in a separate place. This also seems intolerable, since the simulation should show the charge moving with the belt. Finally, P redefines the belt again, as a series of places with paths between them, but with no object actually representing a belt segment. Moving the belt will now involve moving everything in one belt-place to the next, something ShowTrains can do with its limited variablization. P deletes the last rule above, since charges in the belt-place are now "on" the belt.

   P is about to write a send-photons-down-paths rule, when he realizes that, like the belt, the original needs to *contain* state information, not *be* state information. P extends this principle and makes the dicotrons, the brushes, the original, and the paper segments all places, with appropriate graphics. (Of course, this means that "moving" the paper and belt won't be as nice graphically.)

   P revises the "fire-charging-dicotron-rule" to fire according to a marker object within the dicotron, then writes the following rules to expose the belt to the original and transfer toner:

   - If the original has a "light" marker in it, put one photon onto the TO-BELT path. (P will represent normal white paper with 10 "light" markers, and full black with 1 "light" marker).

- If the belt-place has a photon in it, and the belt-place has an electron in it, then delete the electron and the photon.

- If the toner-brush has a "medium-charge" marker in it, then create 5 toner objects in the toner-brush place and put them on the path to the belt-place.

- If the belt-place has a toner object in it, and the belt-place has an electron object on it, then delete them both and create a toner-electron object. (P will not let the belt-movement rule move free toner).

P now creates rules to transfer toner to the paper. This is a more difficult problem:

- If the dicotron-2 place has a "medium-charge" marker in it, create five electron objects and put them on the path to the paper place.

- If the paper place has an electron object in it, create a "charge-pull" object and put it on the path to the belt place, and delete the electron object.

- If the belt-place has a toner-electron object and a charge-pull object in it, delete the toner-electron object, delete the charge-pull object, and put a toner object on the path to the paper place.

P creates a rule to clean the belt:

- If the cleaner-brush place has a "medium-charge" object in it, then put five "positive-charge" objects onto the path to the belt place.

- If the belt place has a positive charge and there is an electron in the belt place, delete the electron and the positive charge.

P creates a rule to fuse the paper:

- If the fusion roller has the "hot" graphic, then put five "heat" objects onto the path to the paper-place.

- If the paper place has a toner object and a heat object, then put a sealed-ink object in the paper place and delete the toner and heat objects.

8. [D7] P creates rules to move the belt and paper. **[problem:** need a timer, so the belt doesn't move before all reactions occur in a place.]

P creates a rule to move the belt with every fifth rule cycle:

- If <no condition> then put a *tick* in the timer place.

- If there are five *tick*s in the timer place, then delete the ticks and put all electron objects and toner-electron objects in BELT places onto the BELT-FORWARD paths, and put all toner objects in PAPER places onto the PAPER-FORWARD paths.

9. [D8] P creates a rule that resets everything to its default state when the user *clicks* the copy button -- actually, the copier should maintain its own state, but this will help debug the system.

10. [D9] P tries the system. It works, but modelling faults is difficult, since the charge transfer mechanism is somewhat kludged. Also, the random rule ordering makes it difficult to debug the simulation.

11. [D10] P draws a background graphic.

Summary Comments

Pluses:

- We can do the simulation, and it's almost a correct simulation. with a little more thought about what a "charge" should be, things might work out OK.

Minuses:

- We want to move a charged belt segment, not move the charges themselves and pretend the segment is moving. But since ShowTrains objects don't have attributes, we have to use places where objects would be appropriate.

- We really want to look across space to see what the charge is in the adjacent place. We can look at more than one place, if they're named, but we can't recognize adjacency by the presence of a common path. The doctrine seems to have discouraged using different names, so we end up with a kludge that sometimes has to transfer "anti-particles" to give the effect of a nearby charge.

- Timing again becomes a problem, more so because we have random rule ordering. The movement of the machine is going to be a bit erratic, even if it is ultimately correct.

ZeroTrains Walkthroughs
JR, 6.25.90 (based on CL's 6.23.90 doctrine)

Doorbell Problem

Sketch as described...

object list:
- button (must show switch open or closed)
- bell (show ringing or still)
- arm (including clapper and pivot)
- spring (must show stretched in various positions)
- core (moves)
- movingcontact (moves)
- electrons (move)
- magfield (show full or collapsed, maybe partial)
- battery (show with more or fewer electrons?)

place list:
- field-place
- button-up-place, button-down-place
- bell-place
- arm-ringing-place (position of arm when it hits bell)
- arm-resting-place
- spring-place
- battery-place
- fixed-contact-place

path list:
- for circuit (route): from button-place
  - to moving-contact-place
  - to fixed-contact-place
  - to mag-field-place, where it spirals (== coil)
  - to battery-place
  - to button-place.
- name all the downstream wire path ends FLOW
- for arm: from arm-resting to ringing place
- name one end RING, one REST

event list:

what	when
electrons flow on all wires	the button is down and and the contacts are closed

[problem solving]: Want to show electrons -- many of them -- flowing along the wire. Can we just keep putting them onto the ends of the wire paths, and assume they'll space out nicely? Assume they will. Now we have a two-predicate situation (button and contacts) combined with a multiple-place situation (several path-ends of the wire). But the doctrine only talks about multiple places in a *condition* -- we have multiple *action* places as well...

Solution? (can't quite get all of this out of the doctrine): Create one big place that encloses the entire circuit. Put a catalyst object in it, but not in any other place, called "doesn't-conduct." Put a marker called "bdn" in the button-down place, a marker called "bup" in the button-up place, a marker called "mco" in the moving-contacts open place, and a marker called "mcc" in the moving-contacts closed place. Specify these events:

what	when
change doesn't-conduct to does-conduct	button and bdn and moving-contact and doesn't-conduct and mcc are in same pl.
change does-conduct to doesn't-conduct	button and bup and does-conduct are in same pl.
change does-conduct to doesn't-conduct	moving-contact and mco are in same pl.

Now we want to put the electron on the flow path when there's a does-conduct object in the enclosing place. But Clayton's rule (as I read it) works against us, since it seems to say the paths have to be attached to the enclosing place, not the inner places. So we do the job in two steps:

what	when
change electron to flowing-electron	does-conduct and electron are in the same place
change flowing-electron to electron	doesn't-conduct and flowing-electron in same place

put flowing-electron on flow path	flowing-electron in a place (implicit: with a flow path)

OK, the current flows. Now build the magnetic field. Turns out that the current is just flowing through the coil (a looping path), and we can't sense it there. So we create a coil place and run the circuit through there. Note that the coil place is contained in the magfield place. To satisfy Clayton's rule, add a no-field object directly in the magfield place.

what	when
create magfield	flowing-electron and no-field in pl.
delete magfield	electron and magfield in pl.

[bonus: didn't have to sense current "flow," since we already had the electron marked -- compare show- and ops-trains.]

Now let's get the iron core, moving contact, and arm moving when the magfield exists. Looks like another job for overlapping places! Build an overlapping place that contains the magfield place, and the rest- and ring- places for the contacts, core, and arm. Put a catalyst object in it, "end-ring." Here's the events:

what	when
change end-ring to start-ring	magfield in pl w/ end-ring
change start-ring to end-ring	no-field in pl w/ start-ring
[put arm, contact, and core on to-bell paths, and change appearance of spring]	[start-ring in same pl. w/ those objs]
[put arm, contact, and core on return paths, and change appearance of spring]	[end-ring in same pl. w/ those objs]

Again, Clayton's rule seems to say the last two rules have to be built in stages, either by modifying the objects themselves (as with

flowing-electron), or by creating a new objects (e.g., do-arm-move) and then specifying an event that consumes the do-arm-move while putting the arm on its path. Anyway, we can do the job, it's just tedious.

OK, finally we write rules that ring the bell. We do this with two more overlapping places, one to connect the arm's ring-place with the bell-place, and one to connect the arm's rest-place with the bell-place. Of course, each contains a catalyst.

what	when
change bell to ringing bell	arm, bell, and bell-ring in same place
change bell to silent bell	arm, bell, and bell-quite in same place

Now, we think through the simulation to see if it will run. User drags button into button-down place (specs don't cover user action, but we can get the button down somehow). Contacts should already be closed. Only rule that can apply is to make the circuit conducting. Now electrons turn into flowing electrons throughout the circuit. Two rules can now apply: move the electrons, or create the magfield. Assume field is created. Now arm can change to moving-arm, and be placed on move path. Still a conflict with electron move, but no matter -- eventually arm will move. When arm reaches "ring" place, three rules could apply: circuit could go nonconducting, or electrons could flow, or bell could ring. If circuit goes nonconducting, then electrons could turn to nonflowing, or bell could ring. If electrons turn, then field could collapse, arm could return, so bell might never ring.

Conclusion: need to sequence things a little better. Doctrine gives help for this, but it's not real specific. Looks like we'll need one huge overlapping place, that contains a sequencing object. Will the overlap mess up any of the existing rules? Can't see how, but it's a worry.

Summary

Pluses:

++ Got the job done, with only one major problem-solving episode.

++ Overlapping places apply in a lot of situations -- once they're understood we have considerable power.

Minuses:

-- Graphics are kind of rough:
- catalyst objects show
- arm doesn't "swing," it just moves
- not sure arm, contacts, and core are moving together
- spring can't be tied to arm.

-- Didn't meet desiderata of having adjustable parameters. Maybe we could, but not without a lot of work, since spring constant, arm inertia, and battery voltage aren't really modelled in the simulation.

-- Clayton's rule, which evidently requires any specified paths to be on the outermost of nested objects, seems to work against us in every case where paths are an issue... of course, the rule is also what keeps the large overlapping places for causing all kinds of things to interact that shouldn't, something we don't even consciously consider. So in that respect, the rule is "natural."

Grasshopper Problem

Sketch as described...

object list:
- grasshoppers (lots of 'em) -- they jump on the paths,
  and they change in various ways:
- -- adults (some male, some female) change:
  - from hungry to less-hungry to full
  - similar for other resources
  - into nothing (if they die)
- -- juveniles change:
  - from hungry to less-hungry to full
  - similar for other resources
  - into adults
  - into nothing (if they die)
- -- eggs change:
  - from eggs to juveniles
  - from eggs to nothing (they get eaten)
- resources (food-bits, water-bits, parasites, whatever)

place list:
- field-places (lots of 'em, in a grid)
- bar-graphs: stack of places for each grasshopper cycle

path list:
- from each place to adjacent places one, two,
  and three jumps away, named ONE, TWO, THREE
  [problem: this is directly contra doctrine, which
  says can't have same names on two paths out of a place.]
- from each bar-graph place to the place above it, named UP.

events (in general terms):
- adult grasshopper eats food, gets fatter
- adult grasshopper drinks water, gets less thirsty
- fat, unthirsty adult grasshopper lays 10 eggs and dies
- eggs live or get eaten
- eggs hatch into juveniles
- juveniles eat and get fatter
- juveniles eat and get less thirsty
- fat, unthirsty juveniles turns into adult

[problem-solving: The first issue is how to maintain the state information about each grasshopper -- whether it's thin, medium, fat; whether it's adult or juvenile or egg; whether it's thirsty or unthirsty; etc. for all resources and attributes. In ZeroTrains, we don't have much choice: we need a different icon for every possible combination of attributes, and a different rule for the relationship between each icon and each resource. Here's some examples:]

what	when
thin, thirsty adult becomes thin, unthirsty adult	thin, thirsty adult and water in pl.
med, thirsty adult becomes med, unthirsty adult	med, thirsty adult and water in pl.
fat, thirsty adult becomes fat, unthirsty adult	fat, thirsty adult and water in pl.
etc.	

No conceptual problem, just tedious.

Problem re Desiderata: If we add another resource to the simulation, we have to throw out all the old rules and start over with new icons. Guess we might use generic attribute markers on the grasshopper icons, and key those to attributes off-line; but this reduces graphic fidelity.

[Problem solving]: Second issue is how to synchronize the cycles, so we change all the juveniles at roughly the same time. Doctrine suggests creating a sequence of trigger objects. This doesn't quite solve the problem. We don't want juveniles to start changing to adults as soon as some specific thing happens; we want them to start changing as soon as they've all had "long enough" to consume resources.

Let's surround everything with a "time" place, in which we'll put an object indicating the current stage. We'll have events to advance time:

what	when

put a "tick" into the time	there is an "adult-stage" object in the place.
put a "tick" into the place	there is an "egg-stage" object in the place.
put a "tick" into a place	there is a "juvenile-stage object in the place
remove all ticks, change adult-stage to egg-stage	there are 1000 ticks and adult-stage in a place
remove all ticks, change juvenile-stage to adult-stage	there are 1000 ticks and juvenile-stage in aplace
remove all ticks, change egg-stage to juvenile-stage	there are 1000 ticks and egg-stage in a place.

[small problem: putting 1000 tick objects in a rule is pretty tedious.]

[potential problem: we don't know that ZeroTrains rules fire fairly, so we aren't sure that other events will have a chance to occur while the clock advances.]

Now we can tie stage changes to the stage-marker:

what	when
change an egg to a juvenile	there is an egg and a juvenile stage object in a pl.
change a juvenile to a just-turned adult	there is a fat, unthirsty juvenile and an adult-stage object in a place
change an adult to 10 eggs	there is a fat, unthirsty adult and an egg-stage object in a place.
change an adult to nothing	there is a thin, thirsty adult and an egg-stage object in a place

Note that we need individual rules to kill off or promote every icon. The "just-turned" adult is a special graphic that we'll use in updating the bar graph:

Now let's make the grasshoppers jump. Remember we have paths leading out of every place, marked ONE, TWO, and THREE. Jump distance should depend on the grasshopper's condition and stage. This is just another long list of rules, of the form:

what	when
put small, thirsty adult on jump path TWO	large, thirsty adult in pl.
put small, thirsty adult on jump path ONE	medium, thirsty adult in pl.

note: we've changed hopper from large/medium to small as it jumps, to indicate energy loss. if we use this scheme, we'll need a finer division than "small, medium, large" -- more rules and icons!

[Rule-conflict **problem...]** The jump rules may conflict with the resource-consumption rules. We can just let the conflict be part of the indeterminism of the simulation. What we'd like to do is have the grasshopper stay in each place for a fixed (or random within limits) period, to give the resource-consumption rules a chance to fire. A version of the solution used by ShowTrains should work, as shown in these modified movement rules:

what	when
put small, thirsty adult on jump path TWO and delete three dropping objs.	large, thirsty adult in pl., along with three dropping objs.
put medium, thirsty adult on jump path ONE and delete three dropping objs.	medium, thirsty adult in pl., along with three dropping objs.
put dropping obj. into place	any kind of adult or juvenile grasshopper in a place.

[Rule explosion again]: Note that this is conceptually just one rule, but we'll have to instantiate it for every different juvenile and adult icon.

[Related problem solving] One more thing we'd like to do is kill off grasshoppers that have dropped below a certain level of strength, after they've been given a chance to consume local resources that might revive them. We can do this with rules such as:

what	when
delete small, thirsty adult	small, thirsty adult in pl. with four dropping objs.

[Plus for Clayton's Rule] The above rule would fire for droppings and grasshoppers across different places in the grasshopper field, if it werent' for Clayton's rule, since the time and bar-graph fields connect all the small places.

[Problem solving]: Another issue is how to update the graph when a juvenile turns into an adult. Use essentially the same solution as ShowTrains: each bar of the bar graph into a stack of many places, connected by one-way paths named UP, which lead from each place to the place above it. The bottom-most places are connected from left to right by paths marked OVER. One of the bottom-most paths has the "current-generation" marker in it. A huge place overlaps all the grasshopper-field places and the bottom-most bar-graph places.

what	when
(modify the adult-to-egg clock rule so it says (underlined):)	
remove all ticks, change adult-stage to egg-stage, change current-gen to move-current-gen	there are 1000 ticks and adult-stage in a place
change move-current-gen to current-gen and put it on OVER path	there is a move-current-gen
change just-turned-adult	just-turned and current-

into adult and create bar-increment in current-generation place	generation in "same" pl.
put bar-increment on UP path	bar-increment and current-generation in same pl.
one bar-increment-graphic goes onto UP path	two bar-increment-graphics in same place.

Summary

Pluses:

++ Got the job done, conceptually fairly simple, except for the graph and the clock.

++ Again, overlapping places apply in a lot of situations.

Minuses:

-- Rule and icon explosion is horrendous. We might put together this simulation for two or three attributes and resources, with three or four steprs per attribute -- but modifying it to add or delete an attribute would be really time-consuming and error-prone.

-- Timing things and sequencing the stages is a big issue. There's a little support in doctrine for sequencing, but nothing related to timing random blocks of behavior.

-- Doctrine tells us use different names for all paths out of a given place. This leans toward a deterministic simulation. Need additional doctrine for nondeterministic cases.

Sunspots Problem

Use sketch provided with problem...

First attempt... Following the doctrine yields complete flux lines (or tubes) as objects, with overlapping tube-like places to hold them as they move/change. This leads to a "dumb" simulation, which just animates a single flare sequence, with no real interaction of the parts. A sequence of bit-maps (frames) would be about as good, although the Z-trains setup lets us drag around the places a bit, so fiddling with the animation might be a bit easier.

Second attempt... We reconsider the problem, rereading the doctrine for hints. No hints. From somewhere ("out of the blue of the western sky") comes the idea of representing segments of lines as individual objects, and combining these into long lines and flux tubes. Can we do it?

object list:

- flux lines, in various orientations...
  - make 'em short
- sunspots (don't worry too much about these just yet... if we do this right, they may appear as a result of the flux tubes!)
- electrons (particles, shoot inward)
- bremstrahlung radiation (shoots outward)
- cosmic-rays (particles, shoot outward)

place list:

- seems like any of these things can be anywhere, so cover the simulation with places. Lay 'em out radially, which is suggested by the physical situation.

path list:

- from every place to each of its neighbors, ends labelled with directions out of the place (U, D, L, R, UL, UR, DL, DR)
- maybe add some special paths for radiation and fast particles, so they don't have to slog it through all those little places? can do this as needed.

[potential problem? -- doctrine too specific. doctrine suggests breaking these grid places into many places, connected in a

ring. doctrine doesn't say why, just says "consider this." can't imagine what it would buy us in this situation.]

OK, now the real work. We have to conceive of flux-tube movement and flare formations as "events." The problem describes both these things in terms of what we might call "macro-events." Seems like the place to start is in moving the tube.

Here's a tube movement event: A segment of the tube moves "up" and the segments next to it either follow, or bend (change graphic), or stretch (create new segments). Doctrine tells us that we'll need an object to flag the move, and that the object will have to be in the place where the move occurred. More doctrine seems to suggest we're an a Pi (pattern of many places) situation.

Reject overlapping places because can't imagine what the big place would look like... every possible tube shape? How would that help?

Will the messenger approach work? Send a message down the tube to... the place where the flux-line is now missing? But how to find where it went? Say we don't move it without leaving a forwarding address. Now, can we send a message "down the tube"? The tube isn't a fixed path, it's a route through the grid. But we might leave markers along the route, pointing to the next spot.

[Attempt 2a: messengers running down the tube]: So here's the plan. A flux tube is a series of flux linelets, contained in contiguous places. In each of those places is a marker, giving the name of the path where the next flux linelet is located (it can't really give the name, but it can have the same effect). We move a flux linelet, leaving a new marker "saying" where it went.

Now a messenger travels down the tube, triggered by a big overlapping object with a catalyst that changes whenever a linelet is moved. The messenger does this:

what	when
put linelet on path U, leave "gone up" in place	<whatever causes up moves... not yet specified>

[now the events that move the messenger on unbroken path]

put messenger on path marked U	messenger and U marker and flux linelet in a place
put messanger on path marked UR	messenger and UR marker and flux linelet in a place

etc.

[Note that we'll have to make versions of these rules for all possible flux linelet graphics. Now do events that let messenger notify other linelets to change when a linelet has moved...]

change messenger to go-back-aim-up	messenger and gone-up marker in a place
put aim-up-go-forward on path left (assume msgr moves left to right, so back = left)	go-back-aim-up in a place
change linelet to angled-up-right-linelet, and put aim-up-continue-forward on path right	aim-up-go-forward messenger in place with horizontal linelet
delete gone-up marker and put aim-up-become-msgr on path right	aim-up-continue-forward and gone-up marker in a place
change linelet to angled-up-left-linelet, and change aim-up-become-msgr to messenger	aim-up-become-msgr and horizontal linelet in place

[**Problem:** rule explosion. This may be a solution, but it ain't much of one.]

[Attempt 2b: overlapping sensor places] Here's a different approach. Around each place, create another place, overlapping the eight places adjacent. Now we can (in principle) sense where a linelet has moved, and act on that information in the appropriate place. Let's try the above rule again. A horizontal linelet will be

moved up (by some unspecified condition) and we want to point the ends of the adjacent linelets toward it...

what	when
put linelet on path U; put "R-gone-up" on L; put "L-gone-up" on R.	<whatever causes up moves... not yet specified>

[Eureka! Send "my new address is" messeges to all neighbors! Now I don't need to overlap anything! I'll get rid of the overlapping places, then...]

change linelet to angled-up-right-linelet, and delete R-gone-up	R-gone-up and horizontal linelet in place
change linelet to angled-up-left-linelet, and delete L-gone-up	L-gone-up and horizontal linelet in place
etc.	

[Still a problem: Rule explosion. The rules shown just handle one linelet of the three making up a flux tube. We'll need to send messages to the other linelets as well. This probably means distinguishing between the upper and lower linelets, so the messages will know when to stop propogating. Or, we can revise the solution to show tubelets, but that will cause problems when it's time to break up the tube.]

[Attempt 2c: overlapping sensor places, really] One last try. Reestablish the place around each grid place, overlapping all adjacent grid places. Now write rules as follows:

what	when
change linelet's graphic to inverted U; put it on path U	<whatever causes up moves... not yet specified>
change inverted U to inverted V, change linelets	if inverted V and two horizontal linelets in a place

so one points up-left, one up-right. (i.e., the big place)

[Oops. Won't work -- can't specify which linelet changes which way. End up sending messages again.]

[Conclusion on tube movement: Can do by having each linelet send out messages when it moves. Rule explosion is a major problem, especially since graphic interactions have to be rewritten in terms of messages -- not real intuitive.]

Write additional events to fling material, move it along, hold it together, in similar manner to flux tubes.

Specify an event to break a flux line when there's material in the tube. Now maybe we can finally use overlapping places. Draw lots of overlapping places, one for every radially aligned set of three places -- i.e., one for every possible flux-tube position. Event we want is something like...

what	when
put "break" marker in upper flux line place	if flung material in same place with three flux lines

But of course, we can't tell which is the "upper" flux line. Doesn't seem to be any way to track this, except by even further rule expansion.

Summary

Pluses:

++ Looks like it would be easy to do the "dumb" simulation, but only if we can edit full-screen-sized objects in the reactions.

++ Doctrine supports... sort of... the sequencing we'd need for the dumb simulation.

Minuses:

-- Doesn't seem to be any reasonable way to solve the "real" simulation problem -- rule explosion is terrible. Maybe I'm missing

something? Checked doctrine again, and only matches seem to be "many paths leaving a place --> use a ring" and "patterns that may occur in many places must be dealt with --> overlap or messenger."

-- Doctrine was pretty confusing when it came to distinguishing between messenger and overlap approaches in this problem.

-- Switching back and forth between a rule-editor and the screen to specify graphic interactions in terms of messages seems hopelessly error-prone.

Xerox Problem

[micro-walkthrough]

Use sketch supplied...

object list:

- original
- dicotron 1 and 2 (charge state changes to simulate faults)
- cleaning brush (ditto, and gets dirty)
- toner brush (ditto, and has more or less toner on it, maybe)
- belt (moves around, sections get charged, collect toner)
  - -- make segments of the belt into objects
- paper (charge changes, toner collects)
  - -- make segments of the paper into objects
- [backtrack: make segments of the original into objects]
- fusion rollers -- get hot or cold.

place list:

- resting place for each of the above, except belt and paper
- segment-resting places for belt and paper segments

[problem solving: About here we have to think about the charge and the toner. Are they objects? Then they move from place to place. We could do this, moving (e.g.) neg-charges from the dicotron place to the belt place, where they would charge the belt. This seems right -- let's do it.]

path list:

- dicotron-1 to belt-segment-places below it
- segment-places of original to belt-segment-places below it.
- toner-pl to belt-segment-pl below it.
- dicotron-2 to paper-segment-pl next to it.
- paper-segment-pl (next to dic.2) to belt-segment-pl next to it
- fusion rollers to paper-segment-place next to them (2 paths)
- cleaning-brush to belt-segment-place below it.
- belt-segment-place to next belt-segment-place
- paper-segment-place to next paper-segment-place

Don't know yet exactly how paths will be used, so we'll name 'em on the fly.

event list (following the problem description):

what	when
put high-charge on dic-to-belt path	high-charge-dic-1

[problem-solving: Looks like this is happening all the time. We need to make sure it happens only as the first few steps of the simulation, interleaved with belt advances. Doctrine seems to say change a sequencing object. But where should this object be? Entire simulation has to have access to it... this seems like an exact description of the overlapping places doctrine!

So build an overlapping place, covering the dicotron and with a bulb in which lives the machine-state object. Rules must now be:

what	when
put high-charge on dic-to-belt path, and change state to charge-belt-1-state.	high-charge-dic-1 and start-state
change belt-seg to high-chrg-belt-seg and put it on next-belt-place-path	high-charge and belt-seg and charge-belt-1-state together

[problem: Now we'll have to write similar rules for all combinations of advance, dicotron charge, and belt charge.]

[big problem, lots of thrashing: How can we be sure that putting a single segment on a path causes all the other segments to move? Could move all when the state-object indicates, but then who changes to the next state?

Seems that I don't quite understand how the rules work. If I have five rules, with a common condition but different objects, can I be sure that they'll all fire, even if one of them negates the condition?

How about this solution: put a pusher object on the path ahead of a seg whenever it moves, and have rules that move any seg forward when it's in a space with a pusher. Oops: now we'll have two belt-segs and a pusher in a place, and either could go forward. All right,

set up intermediate places between all the belt places. Try these rules:

what	when
put belt-seg and pusher on forward path	belt-seg, pusher, seg-place-marker
put pusher on forward path and change intermed-mark to intermed-mark-x	pusher and intermed-mark
put belt-seg on forward path and change interm-med-mark-x to intermed-mark.	belt-seg and intermed-mark-x

Still not sure this will work. May need to set up some sort of marker that changes the state of the pusher to indicate that it's grabbed the next segment and moved on, then change the state of intermed-mark-x based on the state, then move belt-seg conditional on that. This all hangs on how path and rule activities coordinate. Table further attention.

[next problem-solving topic:] OK, let's try one of the rules for discharging the belt when photons hit it. (We're assuming the dicotron doesn't operate again until the entire copy process is finished -- i.e., no parallelism in the copier.)

[Let's just number the states -- semantic name are a hassle]

what	when
put high-intens-photon on to-belt-place path and change state to state-6	state-5 and white-original in pl.
change belt to low-charge and create pusher and change to state-7	high-charge-belt-seg and high-intens-photon and discharge-belt-6-state

Seems to work -- just need rules for every original density and for every combination of belt-seg charge and photon-intens.

[next problem-solving topic:] Now we have to transfer the toner, based on the charge on the belt and the charge on the toner brush. Can't sense both at once directly, but could do it with an overlapping place. Build such a place, with a catalyst. Rules are:

what	when
change med-charge-belt to med-toner-belt	med-charge-belt and med-charge toner brushes in pl. w/ transfer-toner catalyst.

This is pretty easy, although we now have to link it into the state object. Probably want to create the transfer-toner catalyst based on the state object, then change it back to a no-transfer object as part of the transfer rule, then change the state-object's state when the no-transfer object appears. **[small eureka:** Bet this would be a good general scheme for the machine. Have each subunit activated by a catalyst object, which interacts with the central machine-state object.]

Now the toner-to-paper transfer. With overlapping places, we can do this in one step, altering the paper-seg and the belt-seg based on the dicotron-charge, and creating pusher objects for the paper and the belt in the process.

Fusion and cleaning are (relatively) trivial.

Summary

Pluses:

++ Overlapping places help when we need to compare charge states of the dicotron and the toner, then alter the paper accordingly.

++ The idea of a central state object is suggested by doctrine, and slight extension gives us what looks like a powerful scheme: a hierarchy of interacting state objects, with machine subactivities depending on the states at the lowest levels.

Minuses:

-- I spent a *lot* of time thrashing over ways to make the belt segments move together. Maybe this isn't really so difficult, but without more specific info on path/reaction sequencing, I can't tell.

-- Rule explosion is a problem again, although not so much as in other simulations. Consider, e.g., the belt-segment-object. It can be charged (several levels), tonered (several levels), or a mix. Whatever scheme I use to move the belt has to recognize each of those different icons. And whatever scheme I use to transfer toner from belt to paper, also has to recognize all those icons, crossed with the number of possible dicotron icons.

7. Design Space

Before the three competing designs coalesced, the design space was recorded in the HyperCard stack presented in this section Each large block of text is an issue; buttons (rectangles) above the block link upward, and buttons below link downward. Circular links are possible.

Topic: ChemTrains Design Issues

What issues, answers, and scenarios influenced the design of ChemTrains?

- Goal: Easy-to-program simulations
- Influence: Forbus, qualitative physics
- Scenarios: OFF, DIST, etc.
- Problem: Specifying objects
- Problem: Specifying trajectories
- Problem: Specifying changes

Delete Card | UnLink | Up Link | Down Link | New Subissue | 1

- Problem: Specifying changes
- Answer: Filter rules on ends

Problem: Managing rule sequence

How should the application of rules be managed, and the application of rules and filters?

- Scene/Problem: PASS-BY
- Answer?: Require all rules to fire
- Answer?: Have rule change object
- Answer?: Hope nondeterminism works
- Answer?: Rule ordering
- Scene/Problem: LOOP
- Scene/Problem: COPIER-EATS-NICKEL

Delete Card | UnLink | Up Link | Down Link | New Subissue | 2

- Problem: Managing rule sequence

Scene/Problem: LOOP

How can an action like copying be controlled so that it happens only once (or as many times as required)?

- Answer?: Modify one of its inputs
- Answer?: Consume an input
- Answer?: Process path filters, quick

Delete Card | UnLink | Up Link | Down Link | New Subissue | 3

- Scene/Problem: LOOP

Answer?: Process path filters, quick

Process the path filters before the rules get a chance to reapply.

- Scene/Problem: PASS-BY

Delete Card | UnLink | Up Link | Down Link | New Subissue | 4

- Scene/Problem: LOOP

Answer?: Consume an input

Require a catalyst or token that is a required condition and that is consumed by the rule action.

- Scenario: NICKEL
- Scene/Problem: COPIER-EATS-NICKEL

Delete Card | UnLink | Up Link | Down Link | New Subissue | 5

- Answer?: Consume an input
- Problem: Managing rule sequence

Scene/Problem: COPIER-EATS-NICKEL

If the actions of copying and destroying the nickel are in separate rules, it could happen that the nickel is destroyed before the copy is made, or that more than one copy is made before the nickel is destroyed.

- Answer?: Related actions in one rule
- Answer?: Coordinated rules
- Observation: Multiple rules unnatural

Delete Card | UnLink | Up Link | Down Link | New Subissue | 6

- Scene/Problem: COPIER-EATS-NICKEL

Answer?: Coordinated rules

Coordinate the application of separate rules.

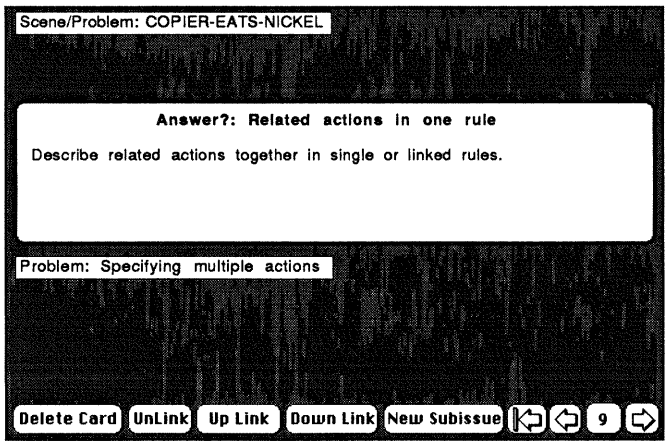
Delete Card | UnLink | Up Link | Down Link | New Subissue | 7

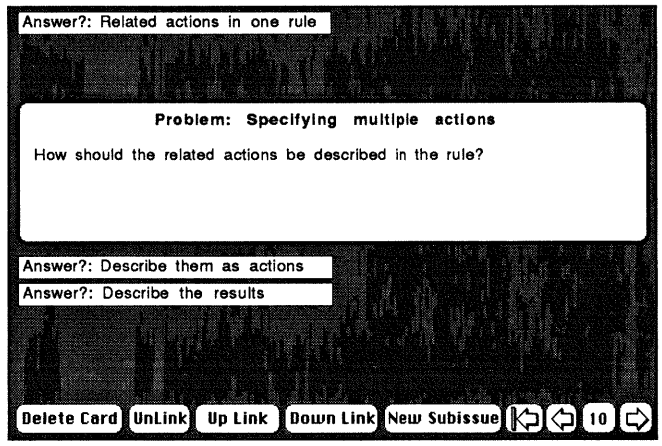
- Scene/Problem: COPIER-EATS-NICKEL

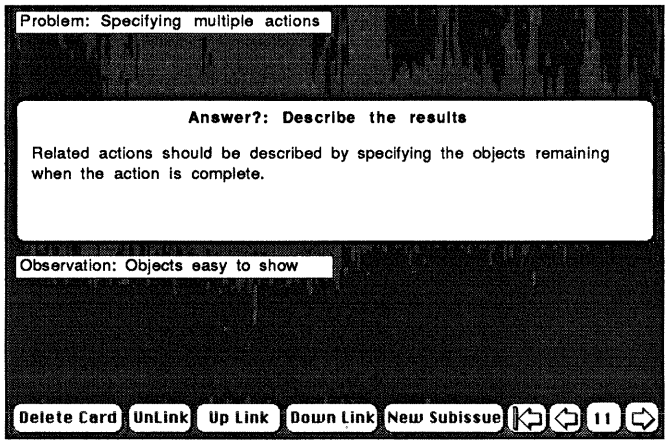
Observation: Multiple rules unnatural

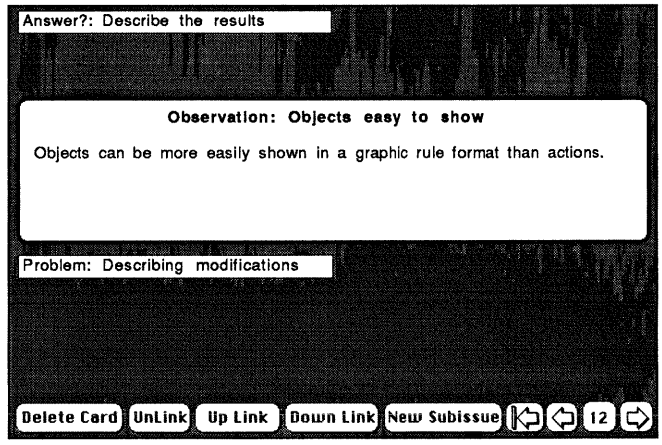
Contrary to our general principle of using "natural" categories of create, delete, etc. for rules, it seems unnatural in this case to specify the copy operation and the nickel deletion separately.

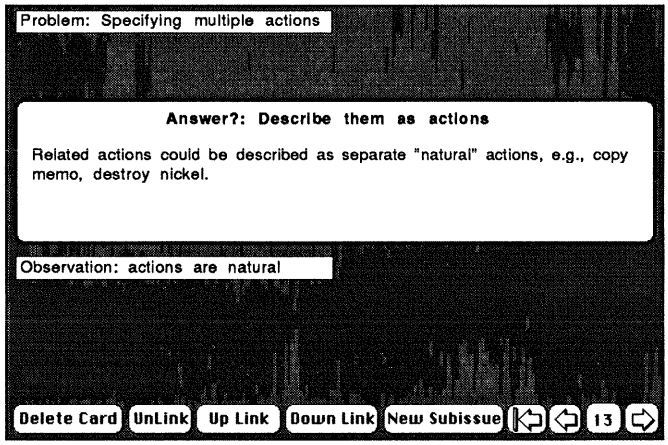
Delete Card | UnLink | Up Link | Down Link | New Subissue | 8

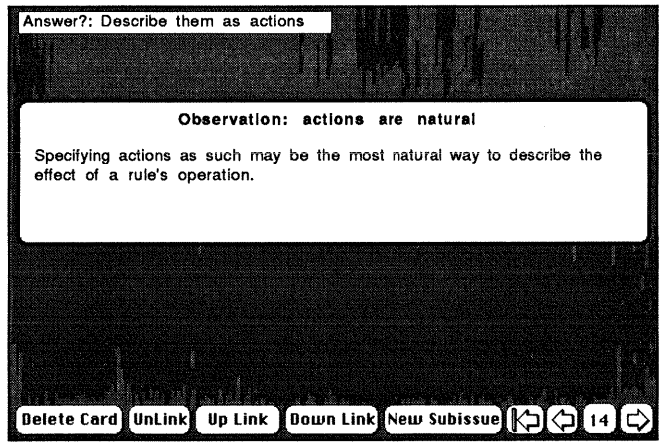

Scene/Problem: COPIER-EATS-NICKEL
Answer?: Related actions in one rule
Describe related actions together in single or linked rules.
Problem: Specifying multiple actions
Delete Card
UnLink
Up Link
Down Link
New Subissue
9

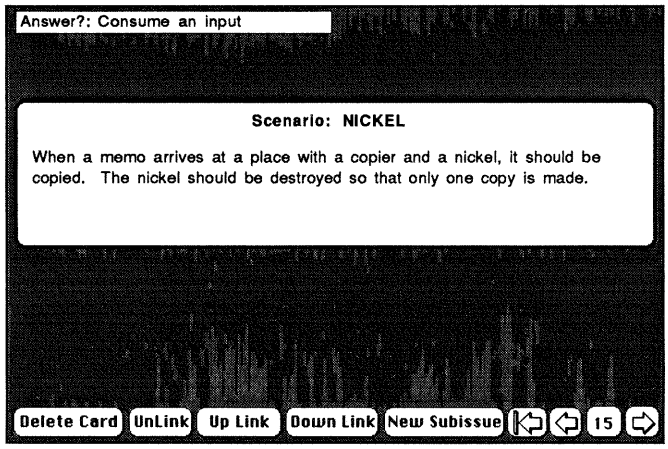

Answer?: Related actions in one rule
Problem: Specifying multiple actions
How should the related actions be described in the rule?
Answer?: Describe them as actions
Answer?: Describe the results
Delete Card
UnLink
Up Link
Down Link
New Subissue
10

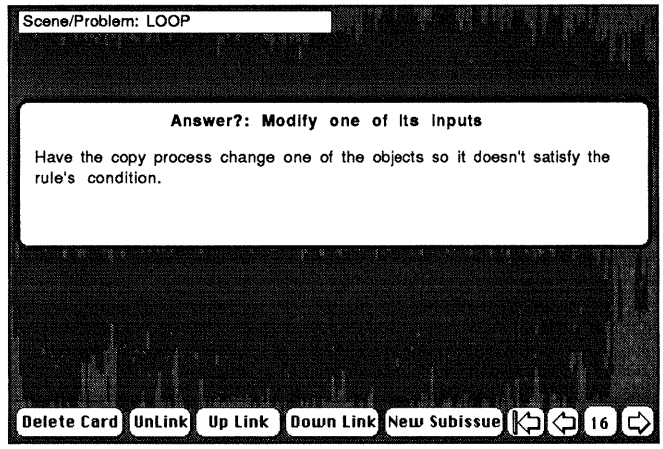

Problem: Specifying multiple actions
Answer?: Describe the results
Related actions should be described by specifying the objects remaining when the action is complete.
Observation: Objects easy to show
Delete Card
UnLink
Up Link
Down Link
New Subissue
11

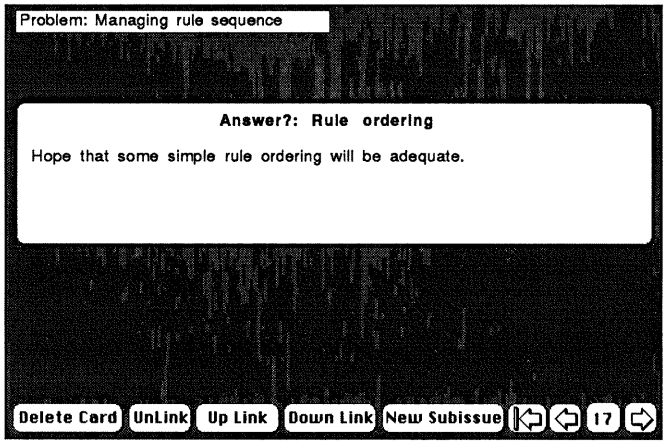

Answer?: Describe the results
Observation: Objects easy to show
Objects can be more easily shown in a graphic rule format than actions.
Problem: Describing modifications
Delete Card
UnLink
Up Link
Down Link
New Subissue
12

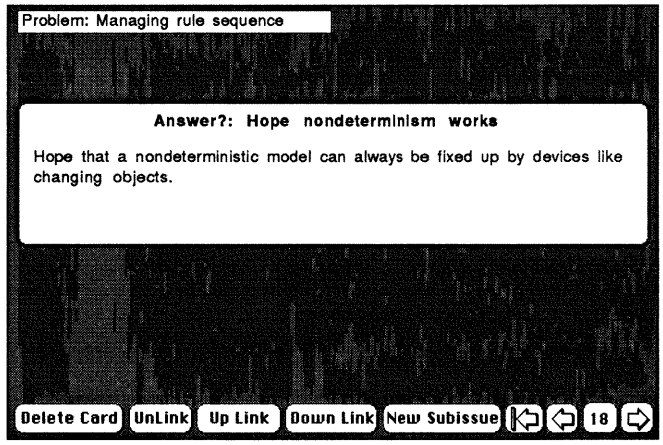

Problem: Specifying multiple actions
Answer?: Describe them as actions
Related actions could be described as separate "natural" actions, e.g., copy memo, destroy nickel.
Observation: actions are natural
Delete Card
UnLink
Up Link
Down Link
New Subissue
13

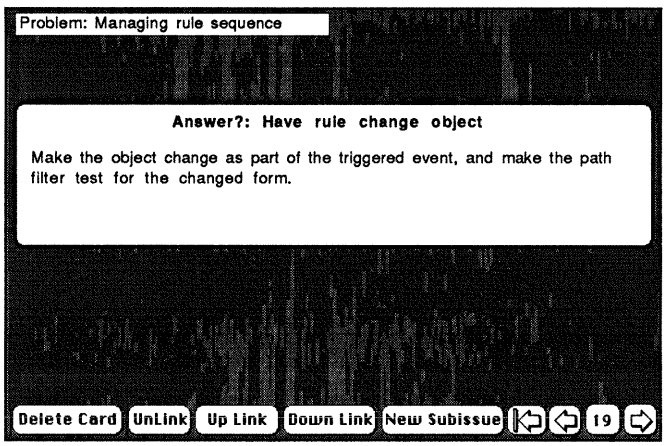

Answer?: Describe them as actions
Observation: actions are natural
Specifying actions as such may be the most natural way to describe the effect of a rule's operation.
Delete Card
UnLink
Up Link
Down Link
New Subissue
14

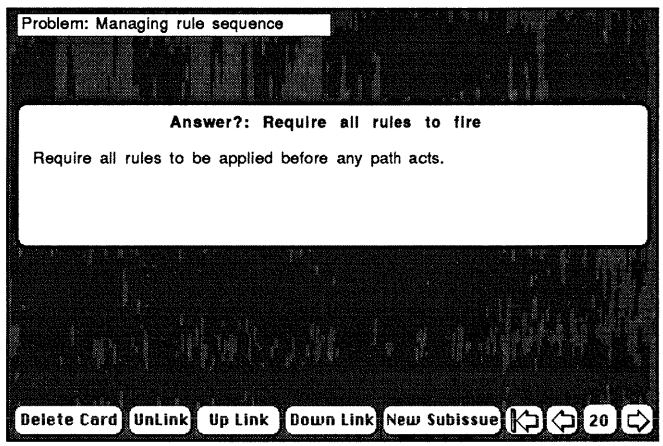

Answer?: Consume an input
Scenario: NICKEL
When a memo arrives at a place with a copier and a nickel, it should be copied. The nickel should be destroyed so that only one copy is made.
Delete Card
UnLink
Up Link
Down Link
New Subissue
15

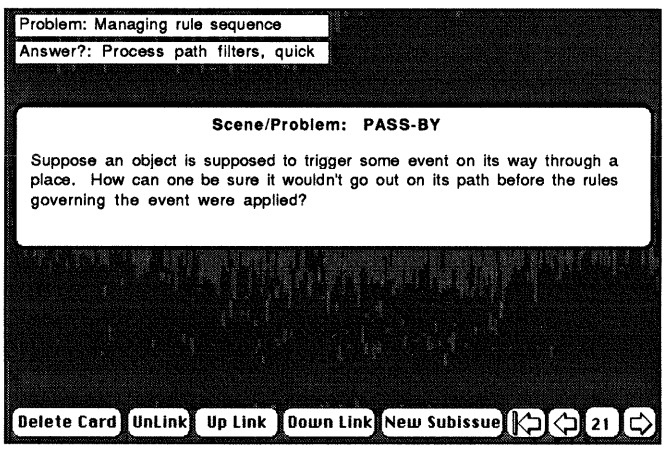

Scene/Problem: LOOP
Answer?: Modify one of its inputs
Have the copy process change one of the objects so it doesn't satisfy the rule's condition.
Delete Card
UnLink
Up Link
Down Link
New Subissue
16

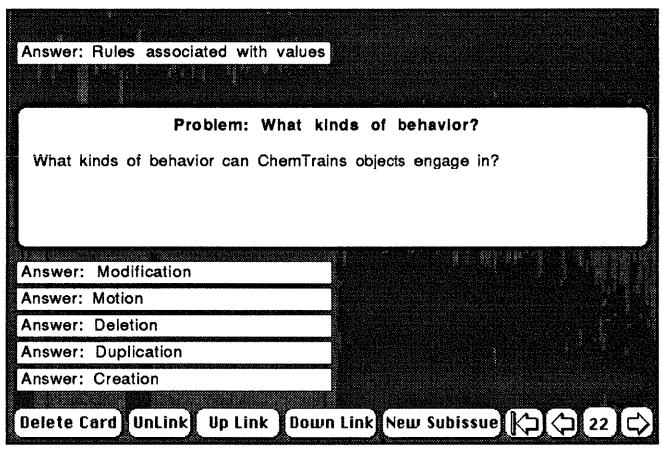

Problem: Managing rule sequence
Answer?: Rule ordering
Hope that some simple rule ordering will be adequate.
Delete Card
UnLink
Up Link
Down Link
New Subissue
17

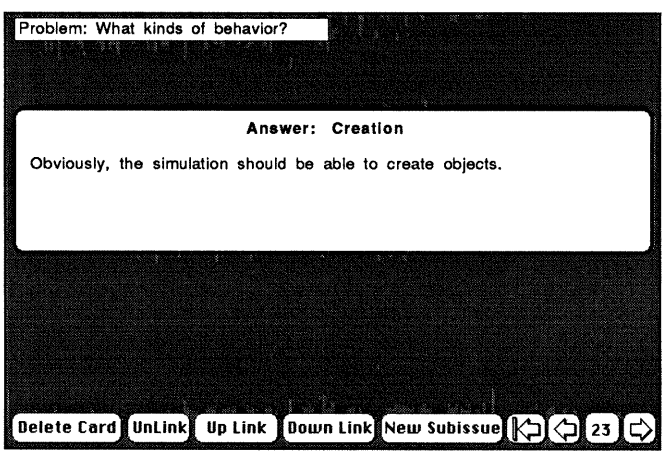

Problem: Managing rule sequence
Answer?: Hope nondeterminism works
Hope that a nondeterministic model can always be fixed up by devices like changing objects.
Delete Card
UnLink
Up Link
Down Link
New Subissue
18

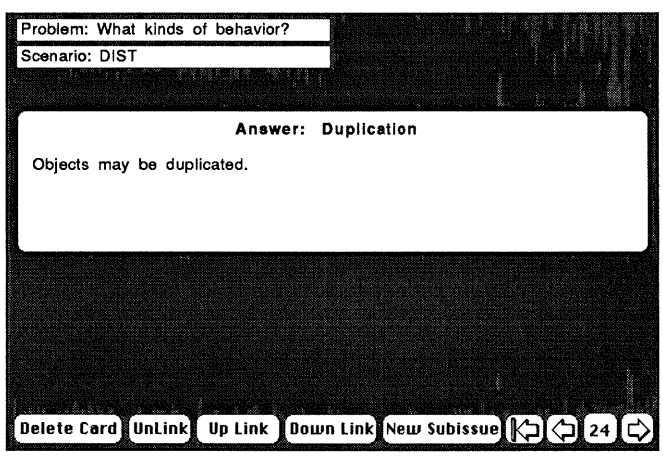

Problem: Managing rule sequence
Answer?: Have rule change object
Make the object change as part of the triggered event, and make the path filter test for the changed form.
Delete Card
UnLink
Up Link
Down Link
New Subissue
19

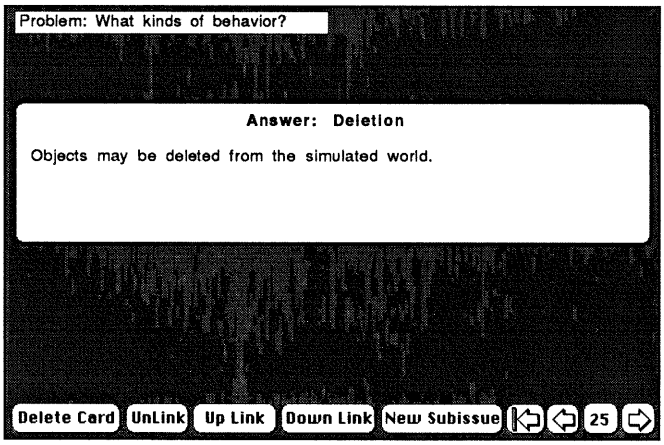

Problem: Managing rule sequence
Answer?: Require all rules to fire
Require all rules to be applied before any path acts.
Delete Card
UnLink
Up Link
Down Link
New Subissue
20

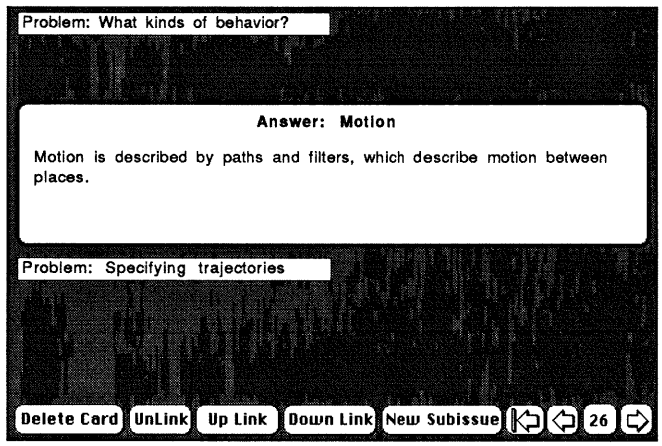

Problem: Managing rule sequence
Answer?: Process path filters, quick
Scene/Problem: PASS-BY
Suppose an object is supposed to trigger some event on its way through a place. How can one be sure it wouldn't go out on its path before the rules governing the event were applied?
Delete Card
UnLink
Up Link
Down Link
New Subissue
21

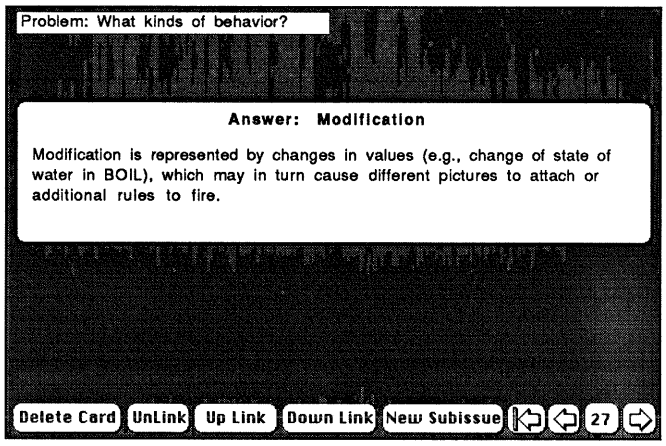

Answer: Rules associated with values
Problem: What kinds of behavior?
What kinds of behavior can ChemTrains objects engage in?
Answer: Modification
Answer: Motion
Answer: Deletion
Answer: Duplication
Answer: Creation
Delete Card
UnLink
Up Link
Down Link
New Subissue
22

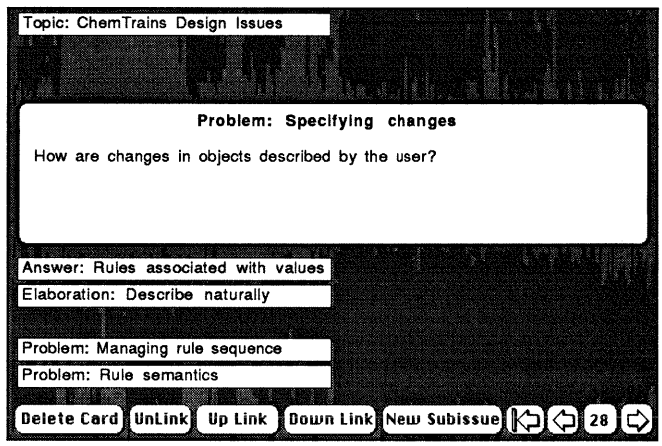

Problem: What kinds of behavior?
Answer: Creation
Obviously, the simulation should be able to create objects.
Delete Card
UnLink
Up Link
Down Link
New Subissue
23

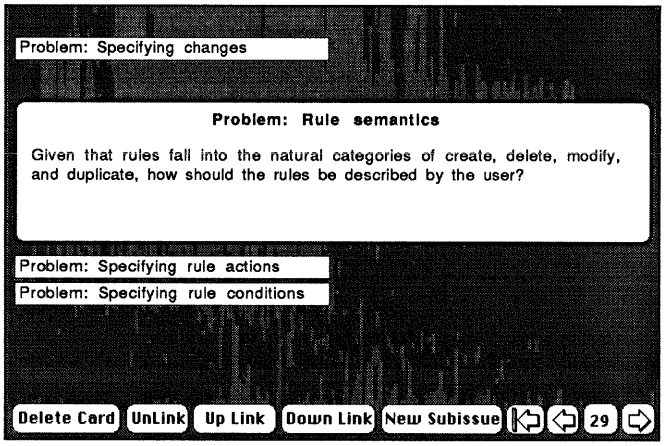

Problem: What kinds of behavior?
Scenario: DIST
Answer: Duplication
Objects may be duplicated.
Delete Card
UnLink
Up Link
Down Link
New Subissue
24

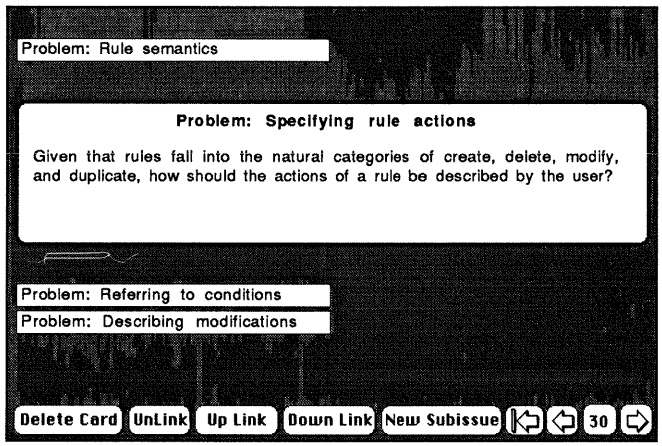

Problem: What kinds of behavior?
Answer: Deletion
Objects may be deleted from the simulated world.
Delete Card
UnLink
Up Link
Down Link
New Subissue
25

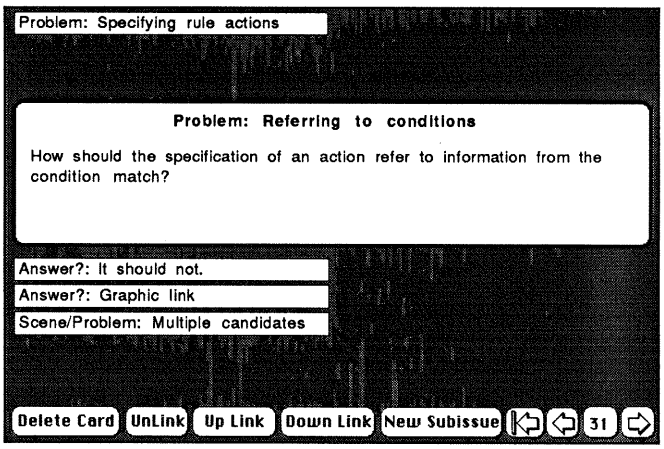

Problem: What kinds of behavior?
Answer: Motion
Motion is described by paths and filters, which describe motion between places.
Problem: Specifying trajectories
Delete Card
UnLink
Up Link
Down Link
New Subissue
26

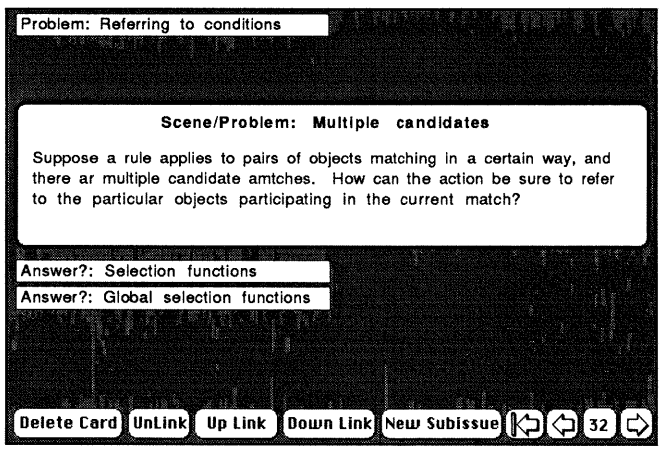

Problem: What kinds of behavior?
Answer: Modification
Modification is represented by changes in values (e.g., change of state of water in BOIL), which may in turn cause different pictures to attach or additional rules to fire.
Delete Card
UnLink
Up Link
Down Link
New Subissue
27

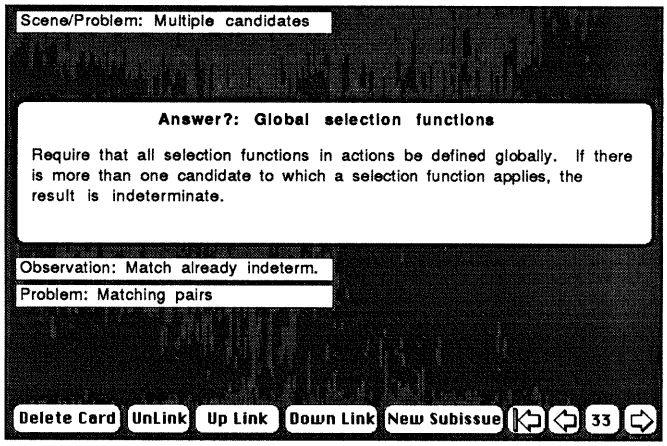

Topic: ChemTrains Design Issues
Problem: Specifying changes
How are changes in objects described by the user?
Answer: Rules associated with values
Elaboration: Describe naturally
Problem: Managing rule sequence
Problem: Rule semantics
Delete Card
UnLink
Up Link
Down Link
New Subissue
28

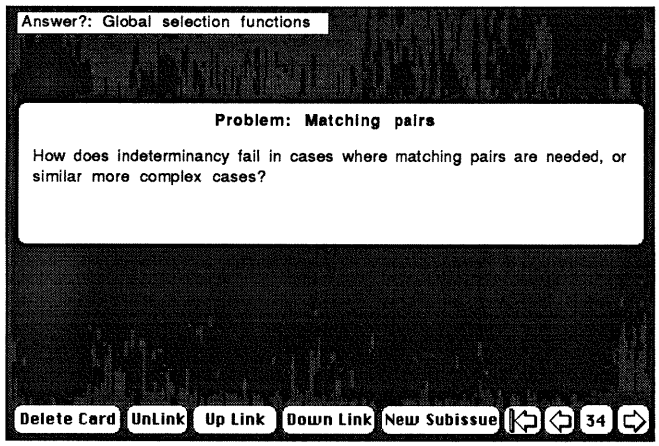

Problem: Specifying changes
Problem: Rule semantics
Given that rules fall into the natural categories of create, delete, modify, and duplicate, how should the rules be described by the user?
Problem: Specifying rule actions
Problem: Specifying rule conditions
Delete Card
UnLink
Up Link
Down Link
New Subissue
29

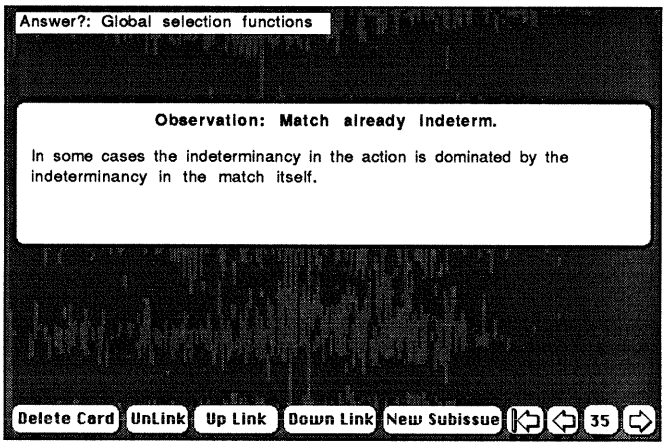

Problem: Rule semantics
Problem: Specifying rule actions
Given that rules fall into the natural categories of create, delete, modify, and duplicate, how should the actions of a rule be described by the user?
Problem: Referring to conditions
Problem: Describing modifications
Delete Card
UnLink
Up Link
Down Link
New Subissue
30

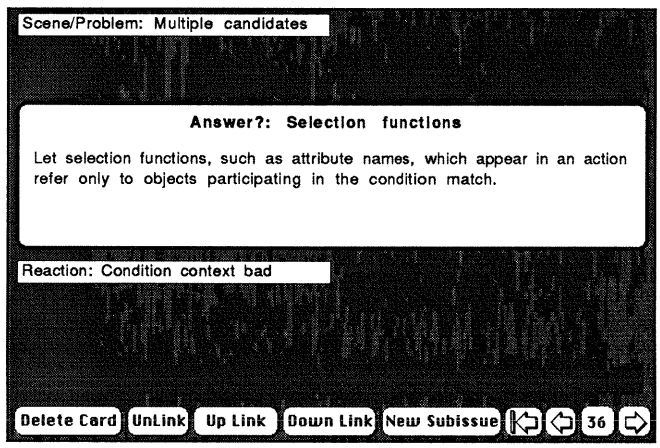

Problem: Specifying rule actions
Problem: Referring to conditions
How should the specification of an action refer to information from the condition match?
Answer?: It should not.
Answer?: Graphic link
Scene/Problem: Multiple candidates
Delete Card
UnLink
Up Link
Down Link
New Subissue
31

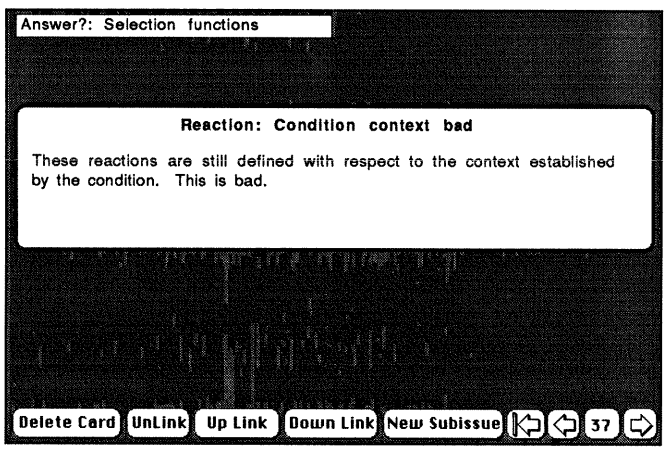

Problem: Referring to conditions
Scene/Problem: Multiple candidates
Suppose a rule applies to pairs of objects matching in a certain way, and there ar multiple candidate amtches. How can the action be sure to refer to the particular objects participating in the current match?
Answer?: Selection functions
Answer?: Global selection functions
Delete Card
UnLink
Up Link
Down Link
New Subissue
32

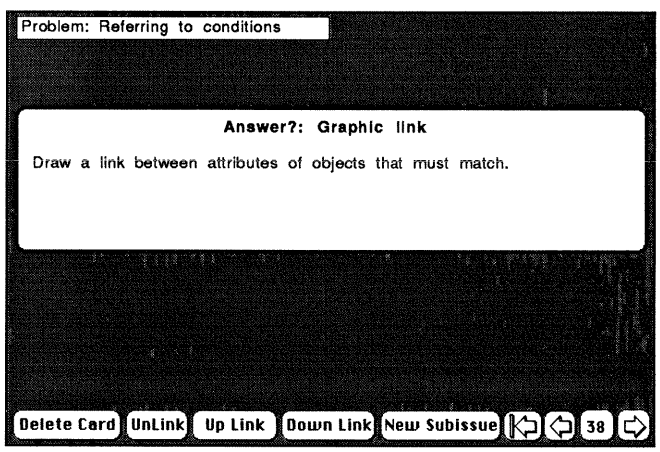

Scene/Problem: Multiple candidates
Answer?: Global selection functions
Require that all selection functions in actions be defined globally. If there is more than one candidate to which a selection function applies, the result is indeterminate.
Observation: Match already indeterm.
Problem: Matching pairs
Delete Card
UnLink
Up Link
Down Link
New Subissue
33

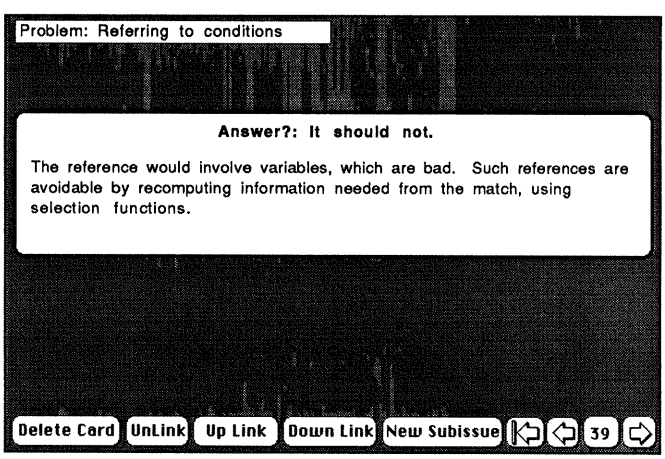

Answer?: Global selection functions
Problem: Matching pairs
How does indeterminancy fail in cases where matching pairs are needed, or similar more complex cases?
Delete Card
UnLink
Up Link
Down Link
New Subissue
34

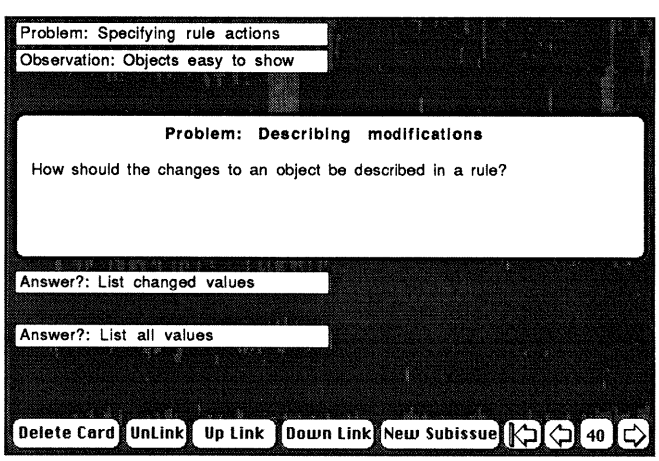

Answer?: Global selection functions
Observation: Match already indeterm.
In some cases the indeterminancy in the action is dominated by the indeterminancy in the match itself.
Delete Card
UnLink
Up Link
Down Link
New Subissue
35

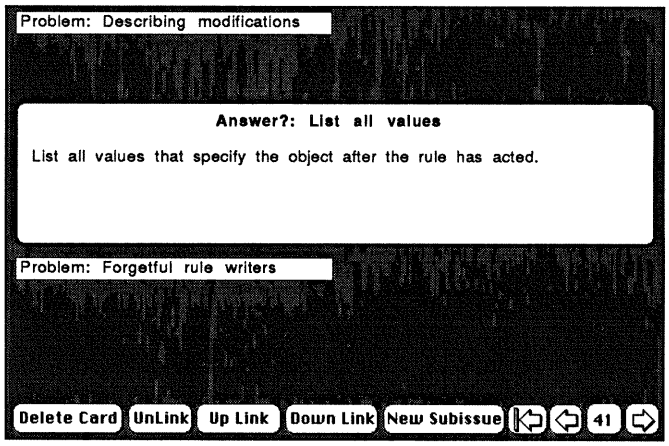

Scene/Problem: Multiple candidates
Answer?: Selection functions
Let selection functions, such as attribute names, which appear in an action refer only to objects participating in the condition match.
Reaction: Condition context bad
Delete Card
UnLink
Up Link
Down Link
New Subissue
36

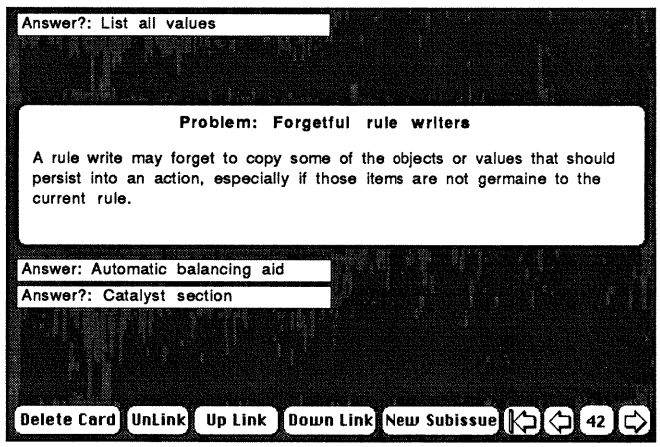

Answer?: Selection functions
Reaction: Condition context bad
These reactions are still defined with respect to the context established by the condition. This is bad.
Delete Card
UnLink
Up Link
Down Link
New Subissue
37

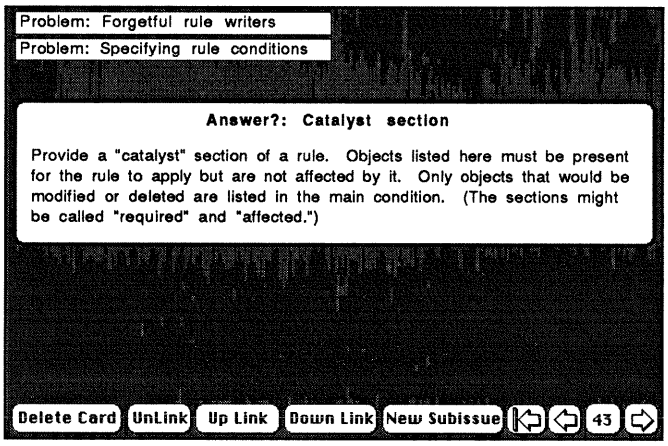

Problem: Referring to conditions
Answer?: Graphic link
Draw a link between attributes of objects that must match.
Delete Card
UnLink
Up Link
Down Link
New Subissue
38

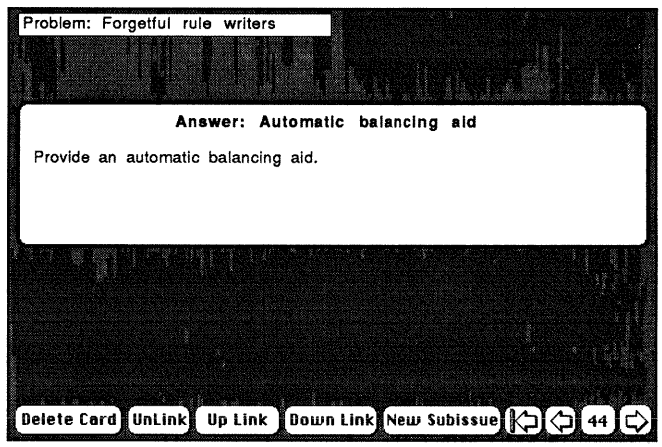

Problem: Referring to conditions
Answer?: It should not.
The reference would involve variables, which are bad. Such references are avoidable by recomputing information needed from the match, using selection functions.
Delete Card
UnLink
Up Link
Down Link
New Subissue
39

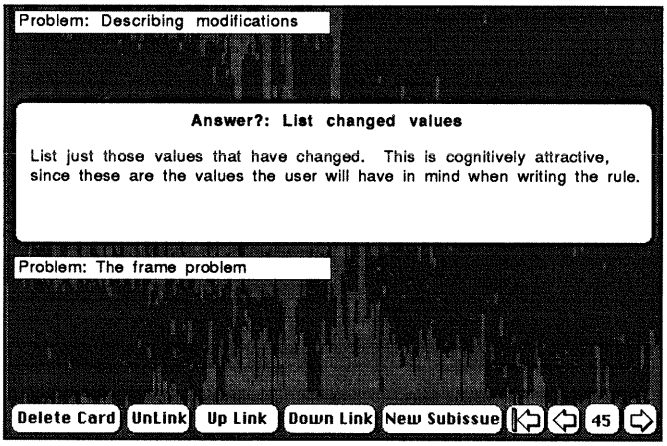

Problem: Specifying rule actions
Observation: Objects easy to show
Problem: Describing modifications
How should the changes to an object be described in a rule?
Answer?: List changed values
Answer?: List all values
Delete Card
UnLink
Up Link
Down Link
New Subissue
40

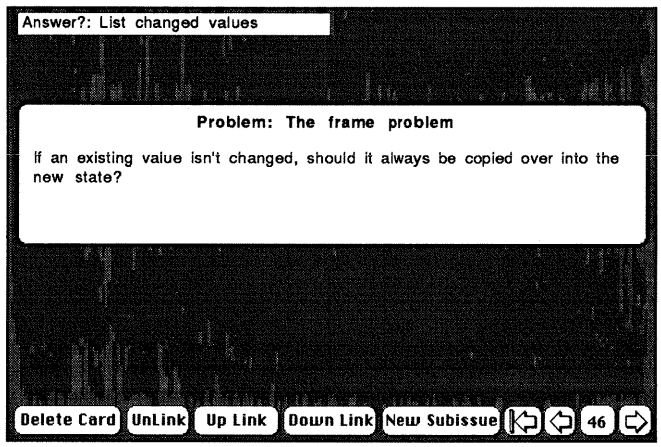

Problem: Describing modifications
Answer?: List all values
List all values that specify the object after the rule has acted.
Problem: Forgetful rule writers
Delete Card
UnLink
Up Link
Down Link
New Subissue
41

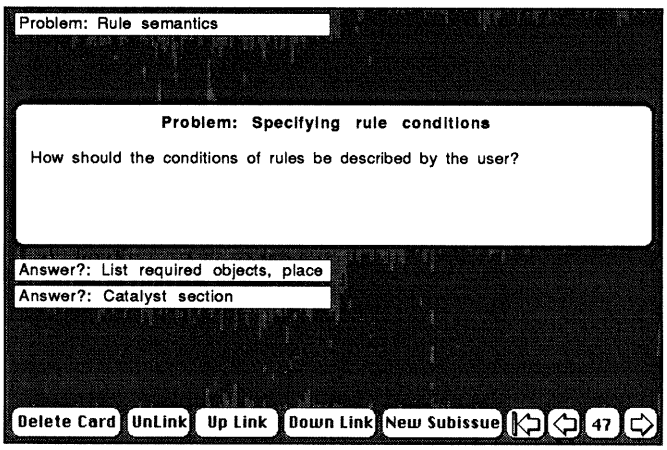

Answer?: List all values
Problem: Forgetful rule writers
A rule write may forget to copy some of the objects or values that should persist into an action, especially if those items are not germaine to the current rule.
Answer: Automatic balancing aid
Answer?: Catalyst section
Delete Card
UnLink
Up Link
Down Link
New Subissue
42

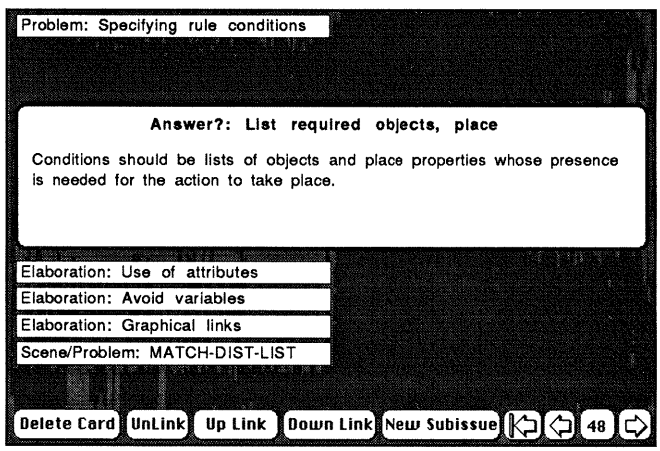

Problem: Forgetful rule writers
Problem: Specifying rule conditions
Answer?: Catalyst section
Provide a "catalyst" section of a rule. Objects listed here must be present for the rule to apply but are not affected by it. Only objects that would be modified or deleted are listed in the main condition. (The sections might be called "required" and "affected.")
Delete Card
UnLink
Up Link
Down Link
New Subissue
43

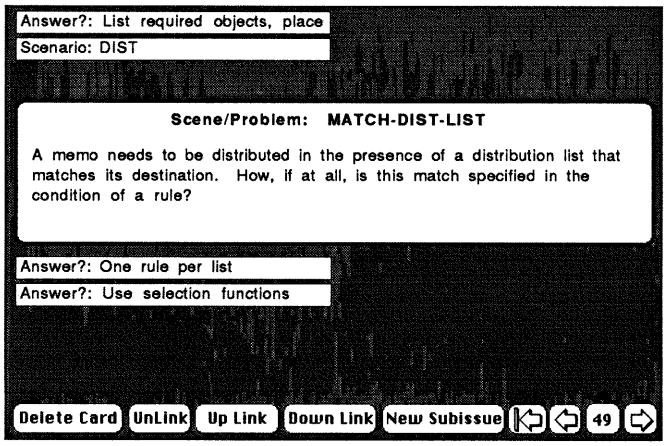

Problem: Forgetful rule writers
Answer: Automatic balancing aid
Provide an automatic balancing aid.
Delete Card
UnLink
Up Link
Down Link
New Subissue
44

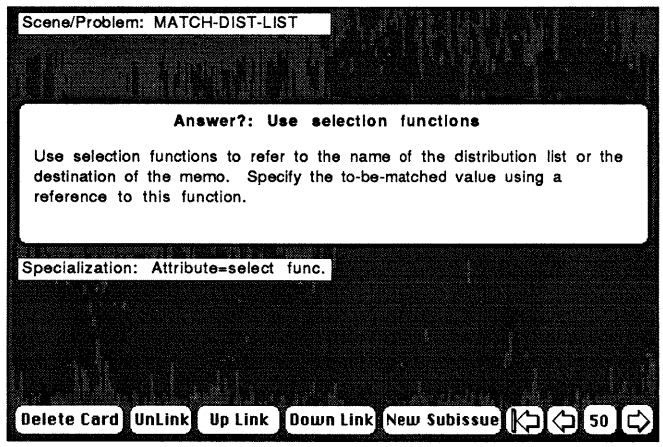

Problem: Describing modifications
Answer?: List changed values
List just those values that have changed. This is cognitively attractive, since these are the values the user will have in mind when writing the rule.
Problem: The frame problem
Delete Card
UnLink
Up Link
Down Link
New Subissue
45

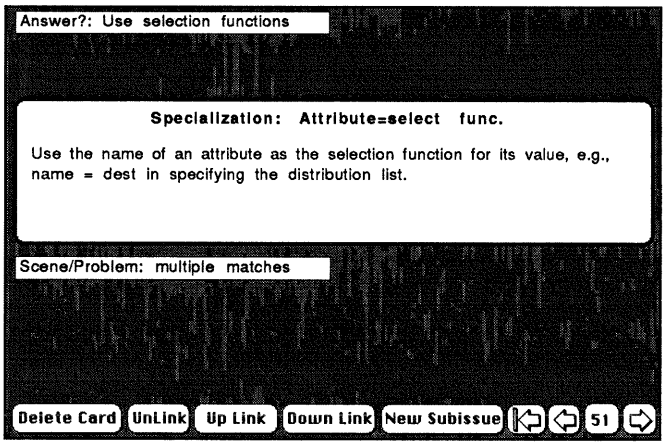

Answer?: List changed values
Problem: The frame problem
If an existing value isn't changed, should it always be copied over into the new state?
Delete Card
UnLink
Up Link
Down Link
New Subissue
46

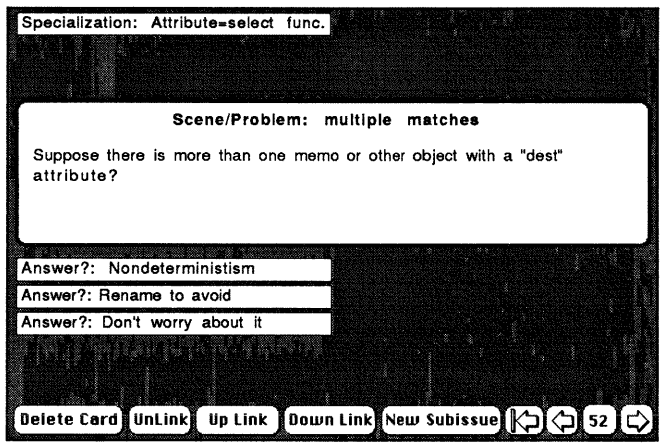

Problem: Rule semantics
Problem: Specifying rule conditions
How should the conditions of rules be described by the user?
Answer?: List required objects, place
Answer?: Catalyst section
Delete Card
UnLink
Up Link
Down Link
New Subissue
47

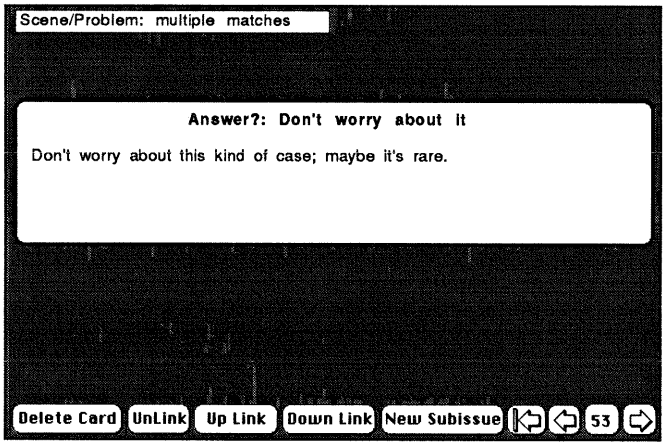

Problem: Specifying rule conditions
Answer?: List required objects, place
Conditions should be lists of objects and place properties whose presence is needed for the action to take place.
Elaboration: Use of attributes
Elaboration: Avoid variables
Elaboration: Graphical links
Scene/Problem: MATCH-DIST-LIST
Delete Card
UnLink
Up Link
Down Link
New Subissue
48

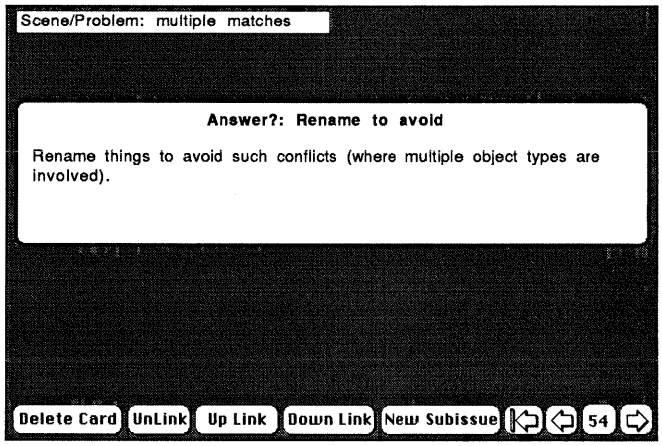

Answer?: List required objects, place
Scenario: DIST
Scene/Problem: MATCH-DIST-LIST
A memo needs to be distributed in the presence of a distribution list that matches its destination. How, if at all, is this match specified in the condition of a rule?
Answer?: One rule per list
Answer?: Use selection functions
Delete Card
UnLink
Up Link
Down Link
New Subissue
49

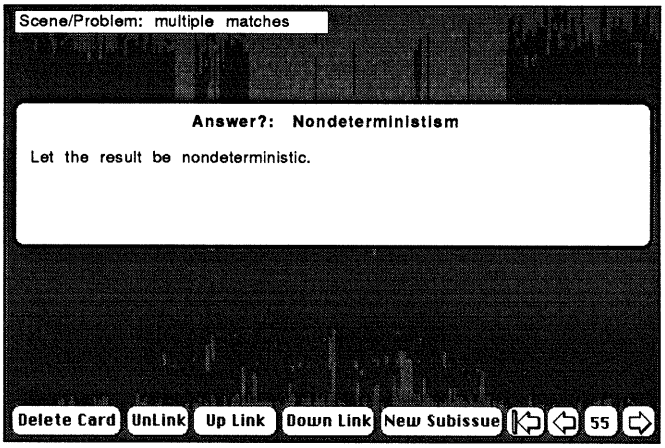

Scene/Problem: MATCH-DIST-LIST
Answer?: Use selection functions
Use selection functions to refer to the name of the distribution list or the destination of the memo. Specify the to-be-matched value using a reference to this function.
Specialization: Attribute=select func.
Delete Card
UnLink
Up Link
Down Link
New Subissue
50

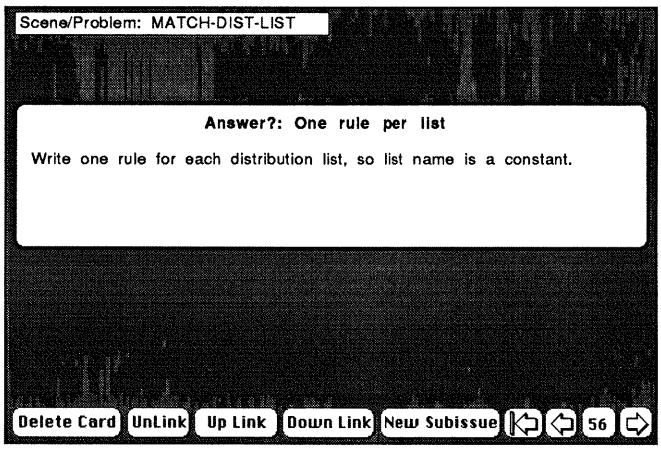

Answer?: Use selection functions
Specialization: Attribute=select func.
Use the name of an attribute as the selection function for its value, e.g., name = dest in specifying the distribution list.
Scene/Problem: multiple matches
Delete Card
UnLink
Up Link
Down Link
New Subissue
51

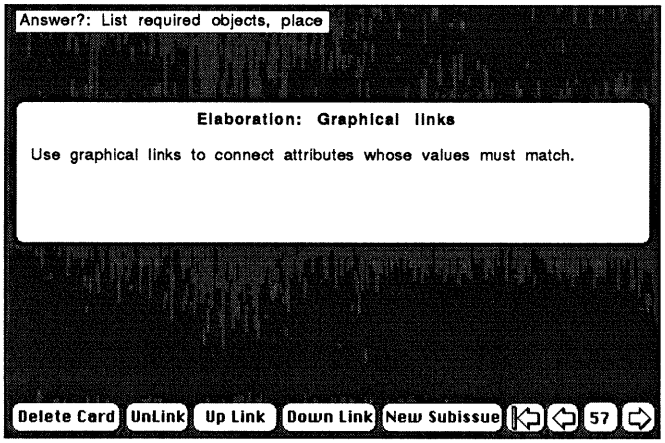

Specialization: Attribute=select func.
Scene/Problem: multiple matches
Suppose there is more than one memo or other object with a "dest" attribute?
Answer?: Nondeterministism
Answer?: Rename to avoid
Answer?: Don't worry about it
Delete Card
UnLink
Up Link
Down Link
New Subissue
52

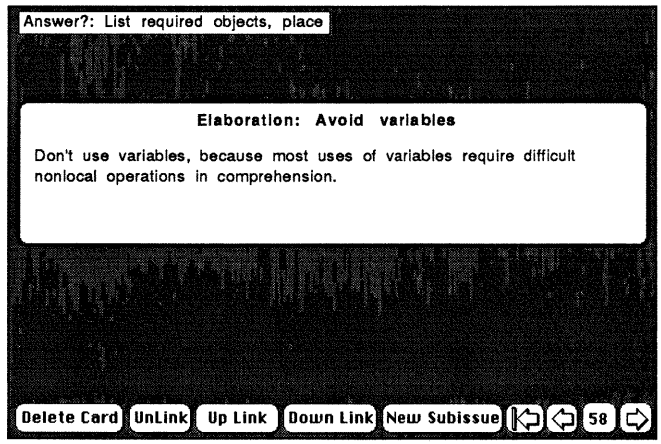

Scene/Problem: multiple matches
Answer?: Don't worry about it
Don't worry about this kind of case; maybe it's rare.
Delete Card
UnLink
Up Link
Down Link
New Subissue
53

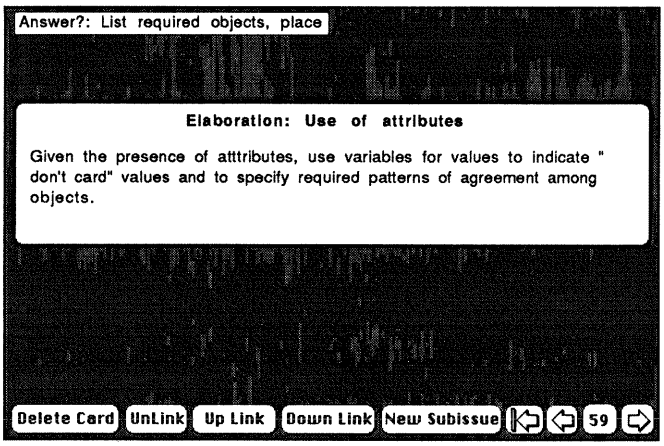

Scene/Problem: multiple matches
Answer?: Rename to avoid
Rename things to avoid such conflicts (where multiple object types are involved).
Delete Card
UnLink
Up Link
Down Link
New Subissue
54

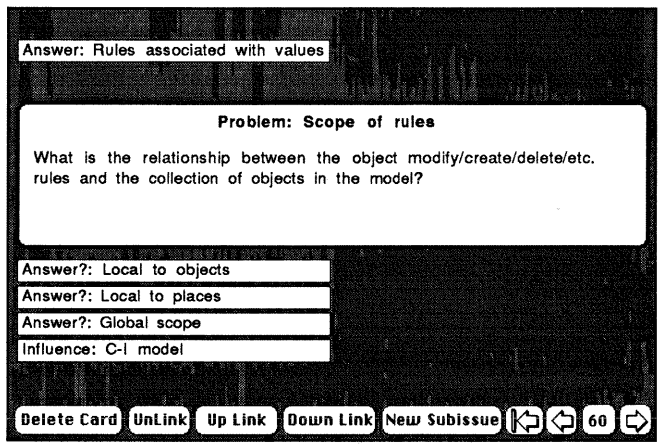

Scene/Problem: multiple matches
Answer?: Nondeterministism
Let the result be nondeterministic.
Delete Card
UnLink
Up Link
Down Link
New Subissue
55

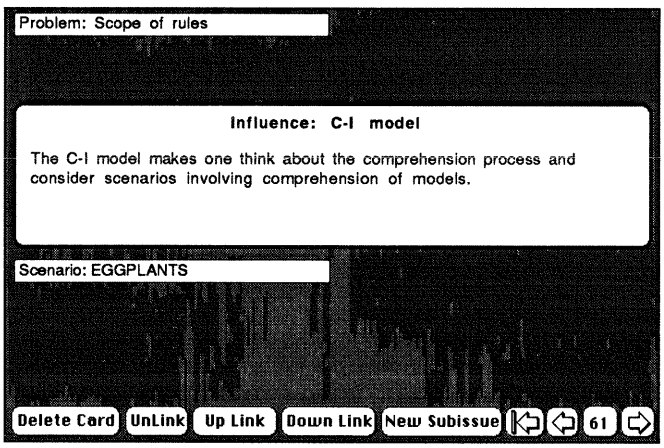

Scene/Problem: MATCH-DIST-LIST
Answer?: One rule per list
Write one rule for each distribution list, so list name is a constant.
Delete Card
UnLink
Up Link
Down Link
New Subissue
56

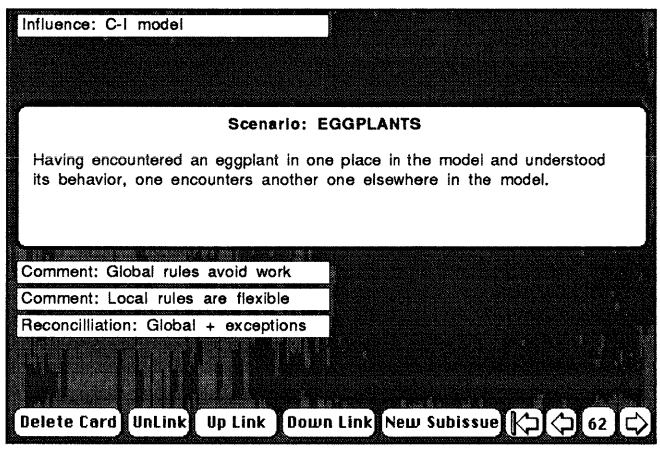

Answer?: List required objects, place
Elaboration: Graphical links
Use graphical links to connect attributes whose values must match.
Delete Card
UnLink
Up Link
Down Link
New Subissue
57

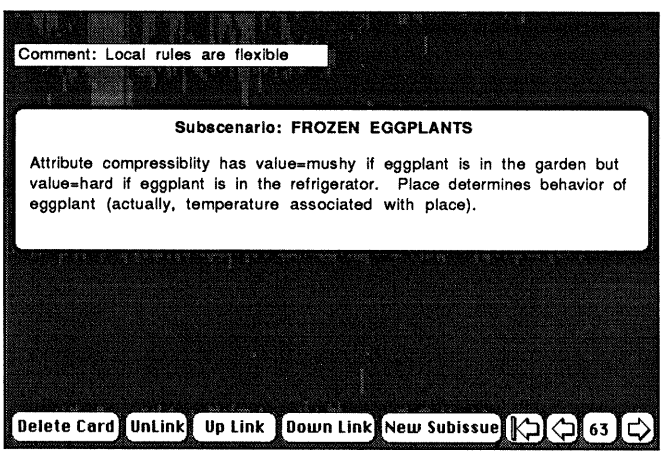

Answer?: List required objects, place
Elaboration: Avoid variables
Don't use variables, because most uses of variables require difficult nonlocal operations in comprehension.
Delete Card
UnLink
Up Link
Down Link
New Subissue
58

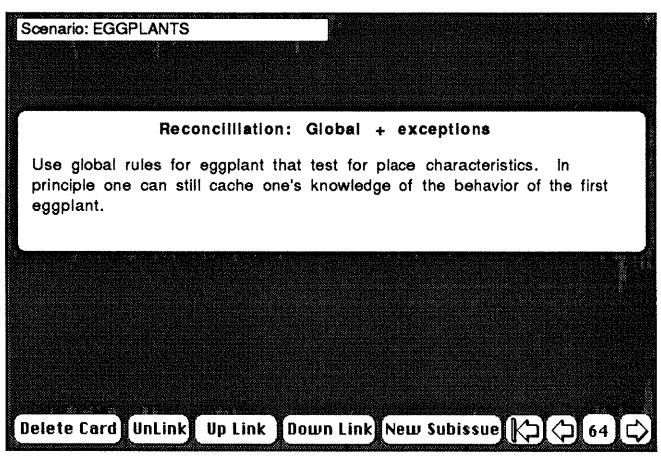

Answer?: List required objects, place
Elaboration: Use of attributes
Given the presence of atttributes, use variables for values to indicate " don't card" values and to specify required patterns of agreement among objects.
Delete Card
UnLink
Up Link
Down Link
New Subissue
59

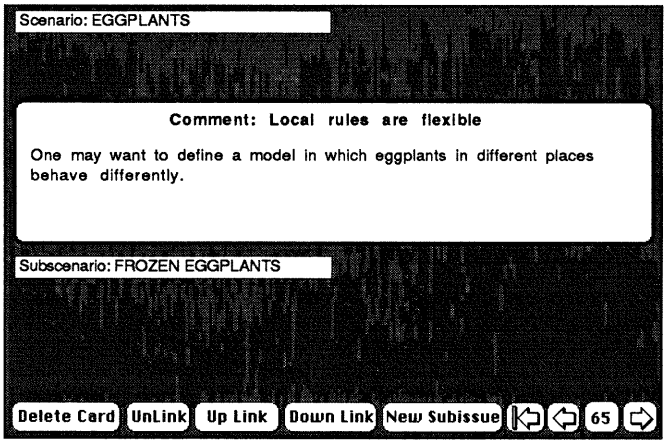

Answer: Rules associated with values
Problem: Scope of rules
What is the relationship between the object modify/create/delete/etc. rules and the collection of objects in the model?
Answer?: Local to objects
Answer?: Local to places
Answer?: Global scope
Influence: C-I model
Delete Card
UnLink
Up Link
Down Link
New Subissue
60

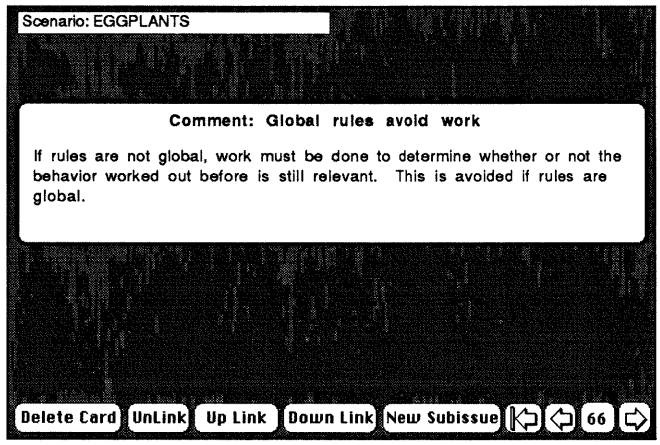

Problem: Scope of rules
Influence: C-I model
The C-I model makes one think about the comprehension process and consider scenarios involving comprehension of models.
Scenario: EGGPLANTS
Delete Card
UnLink
Up Link
Down Link
New Subissue
61

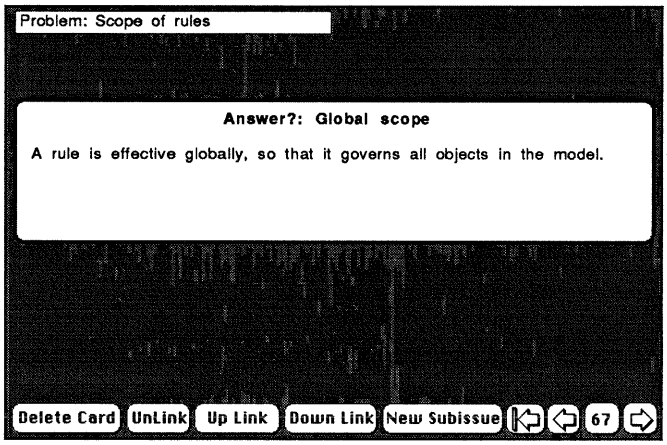

Influence: C-I model
Scenario: EGGPLANTS
Having encountered an eggplant in one place in the model and understood its behavior, one encounters another one elsewhere in the model.
Comment: Global rules avoid work
Comment: Local rules are flexible
Reconcilliation: Global + exceptions
Delete Card
UnLink
Up Link
Down Link
New Subissue
62

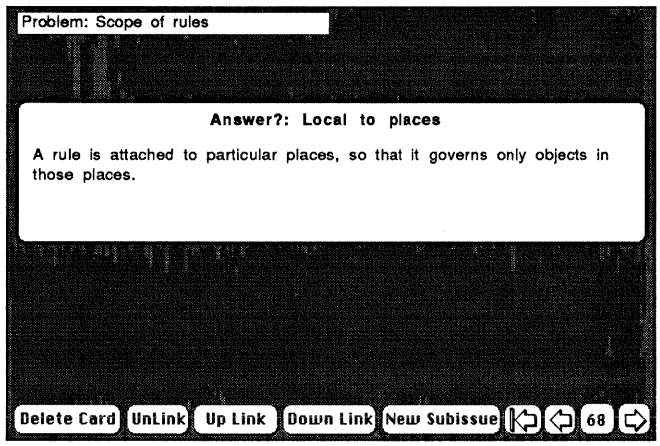

Comment: Local rules are flexible
Subscenario: FROZEN EGGPLANTS
Attribute compressiblity has value=mushy if eggplant is in the garden but value=hard if eggplant is in the refrigerator. Place determines behavior of eggplant (actually, temperature associated with place).
Delete Card
UnLink
Up Link
Down Link
New Subissue
63

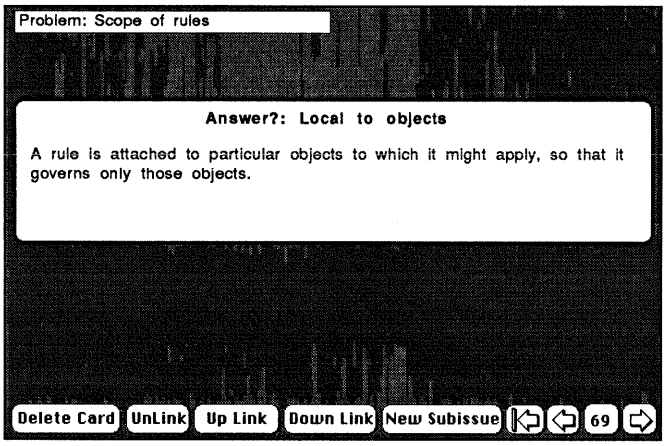

Scenario: EGGPLANTS
Reconcilliation: Global + exceptions
Use global rules for eggplant that test for place characteristics. In principle one can still cache one's knowledge of the behavior of the first eggplant.
Delete Card
UnLink
Up Link
Down Link
New Subissue
64

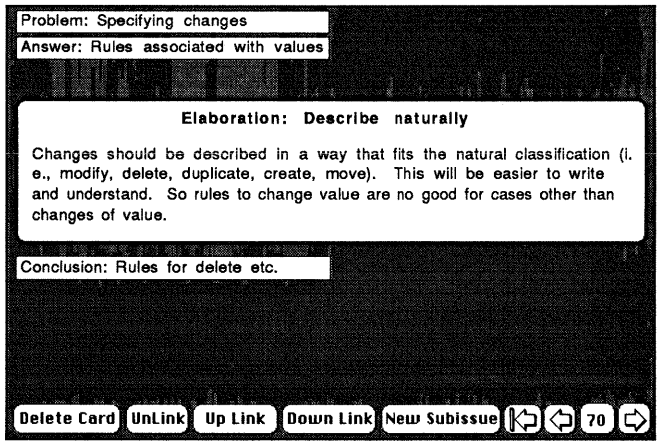

Scenario: EGGPLANTS
Comment: Local rules are flexible
One may want to define a model in which eggplants in different places behave differently.
Subscenario: FROZEN EGGPLANTS
Delete Card
UnLink
Up Link
Down Link
New Subissue
65

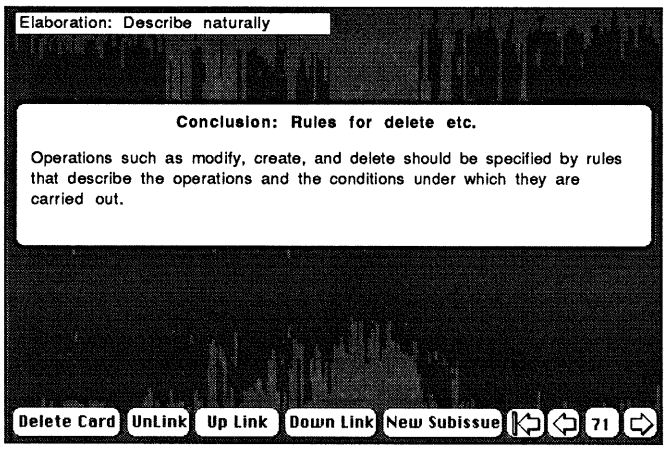

Scenario: EGGPLANTS
Comment: Global rules avoid work
If rules are not global, work must be done to determine whether or not the behavior worked out before is still relevant. This is avoided if rules are global.
Delete Card
UnLink
Up Link
Down Link
New Subissue
66

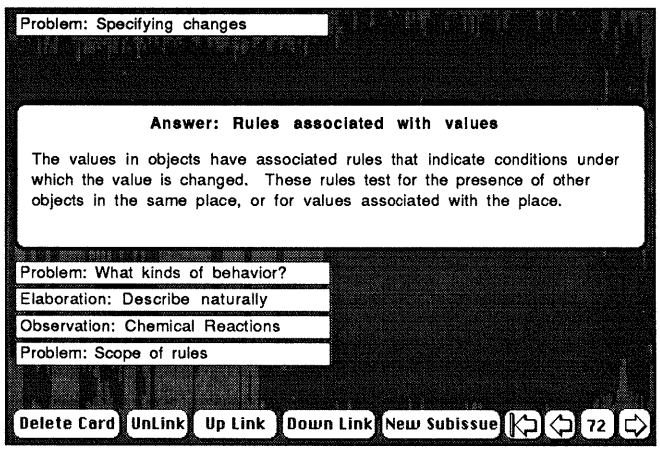

Problem: Scope of rules
Answer?: Global scope
A rule is effective globally, so that it governs all objects in the model.
Delete Card
UnLink
Up Link
Down Link
New Subissue
67

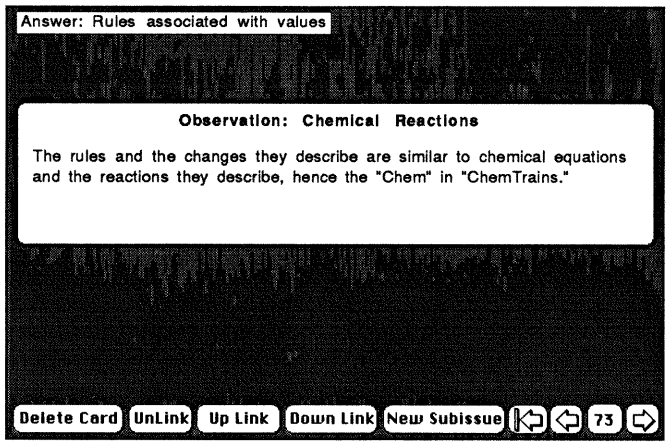

Problem: Scope of rules
Answer?: Local to places
A rule is attached to particular places, so that it governs only objects in those places.
Delete Card
UnLink
Up Link
Down Link
New Subissue
68

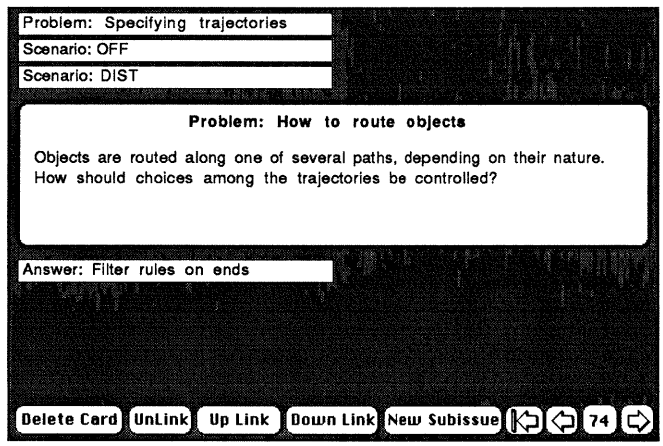

Problem: Scope of rules
Answer?: Local to objects
A rule is attached to particular objects to which it might apply, so that it governs only those objects.
Delete Card
UnLink
Up Link
Down Link
New Subissue
69

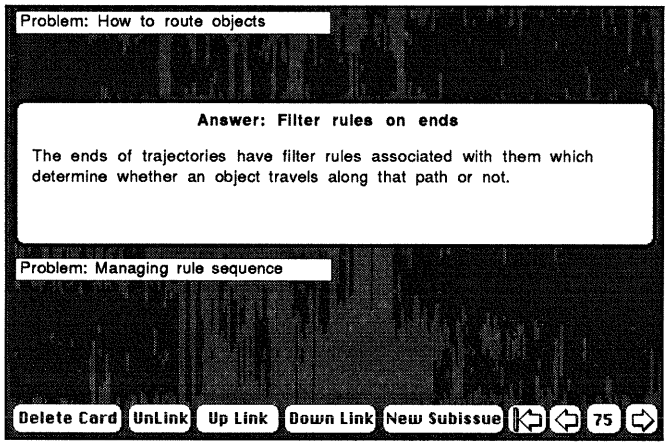

Problem: Specifying changes
Answer: Rules associated with values
Elaboration: Describe naturally
Changes should be described in a way that fits the natural classification (i. e., modify, delete, duplicate, create, move). This will be easier to write and understand. So rules to change value are no good for cases other than changes of value.
Conclusion: Rules for delete etc.
Delete Card
UnLink
Up Link
Down Link
New Subissue
70

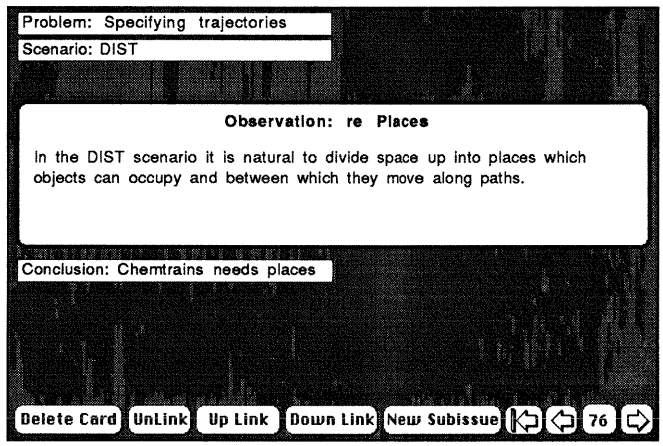

Elaboration: Describe naturally
Conclusion: Rules for delete etc.
Operations such as modify, create, and delete should be specified by rules that describe the operations and the conditions under which they are carried out.
Delete Card
UnLink
Up Link
Down Link
New Subissue
71

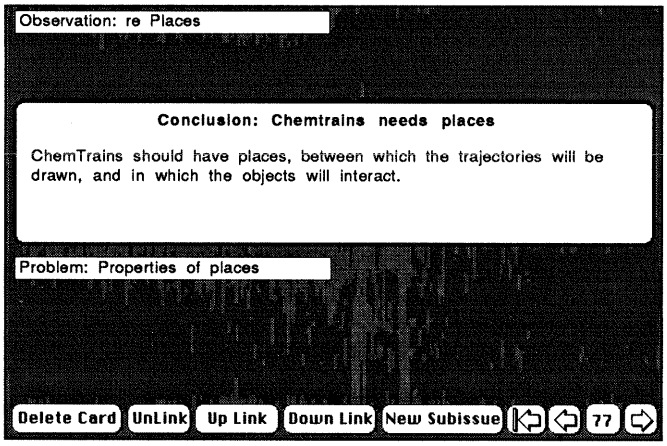

Problem: Specifying changes
Answer: Rules associated with values
The values in objects have associated rules that indicate conditions under which the value is changed. These rules test for the presence of other objects in the same place, or for values associated with the place.
Problem: What kinds of behavior?
Elaboration: Describe naturally
Observation: Chemical Reactions
Problem: Scope of rules
Delete Card
UnLink
Up Link
Down Link
New Subissue
72

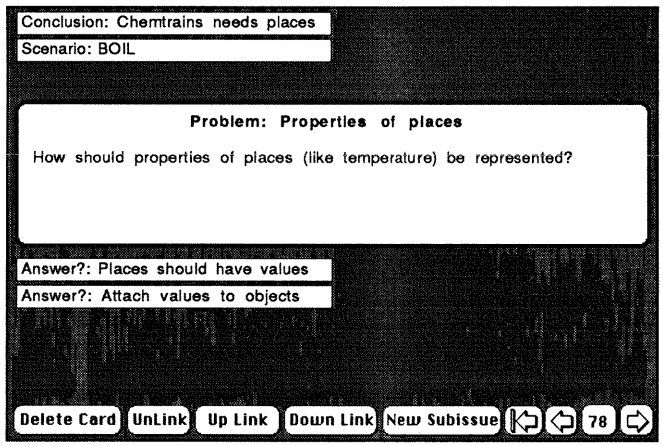

Answer: Rules associated with values
Observation: Chemical Reactions
The rules and the changes they describe are similar to chemical equations and the reactions they describe, hence the "Chem" in "ChemTrains."
Delete Card
UnLink
Up Link
Down Link
New Subissue
73

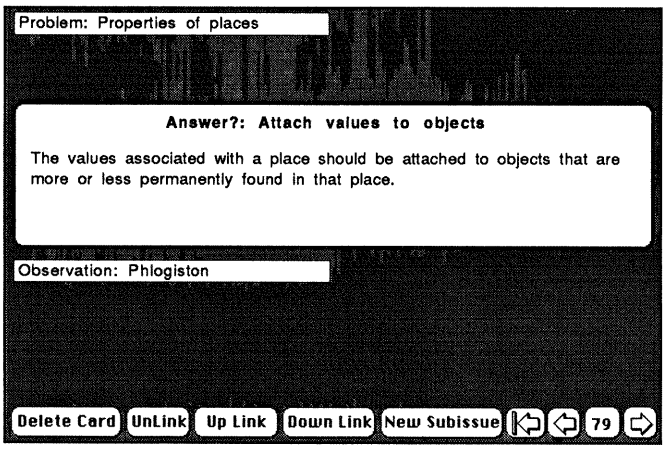

Problem: Specifying trajectories
Scenario: OFF
Scenario: DIST
Problem: How to route objects
Objects are routed along one of several paths, depending on their nature. How should choices among the trajectories be controlled?
Answer: Filter rules on ends
Delete Card
UnLink
Up Link
Down Link
New Subissue
74

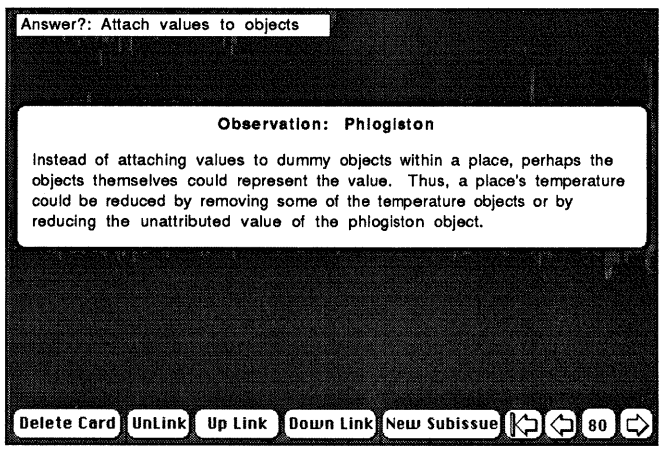

Problem: How to route objects
Answer: Filter rules on ends
The ends of trajectories have filter rules associated with them which determine whether an object travels along that path or not.
Problem: Managing rule sequence
Delete Card
UnLink
Up Link
Down Link
New Subissue
75

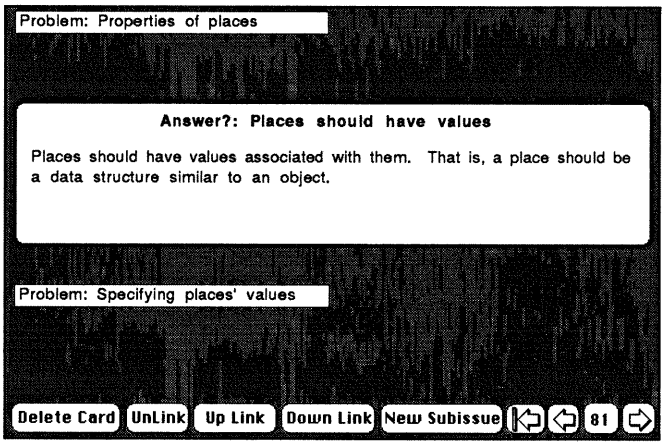

Problem: Specifying trajectories
Scenario: DIST
Observation: re Places
In the DIST scenario it is natural to divide space up into places which objects can occupy and between which they move along paths.
Conclusion: Chemtrains needs places
Delete Card
UnLink
Up Link
Down Link
New Subissue
76

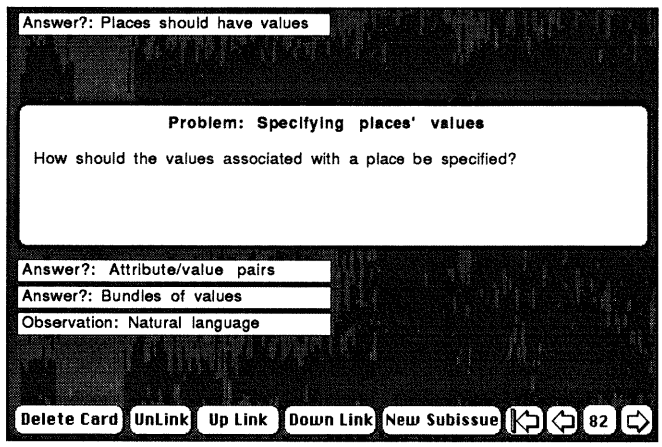

Observation: re Places
Conclusion: Chemtrains needs places
ChemTrains should have places, between which the trajectories will be drawn, and in which the objects will interact.
Problem: Properties of places
Delete Card
UnLink
Up Link
Down Link
New Subissue
77

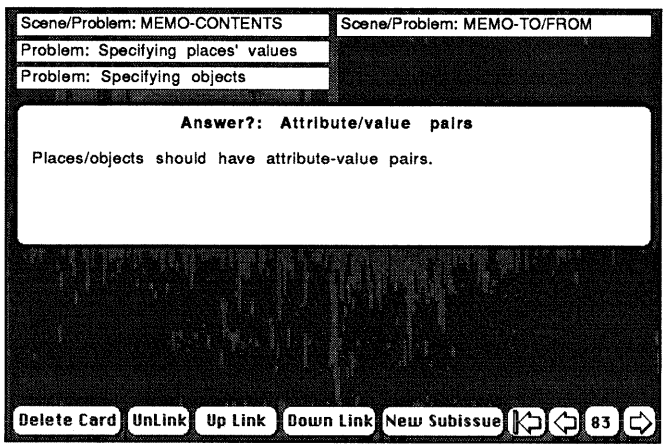

Conclusion: Chemtrains needs places
Scenario: BOIL
Problem: Properties of places
How should properties of places (like temperature) be represented?
Answer?: Places should have values
Answer?: Attach values to objects
Delete Card
UnLink
Up Link
Down Link
New Subissue
78

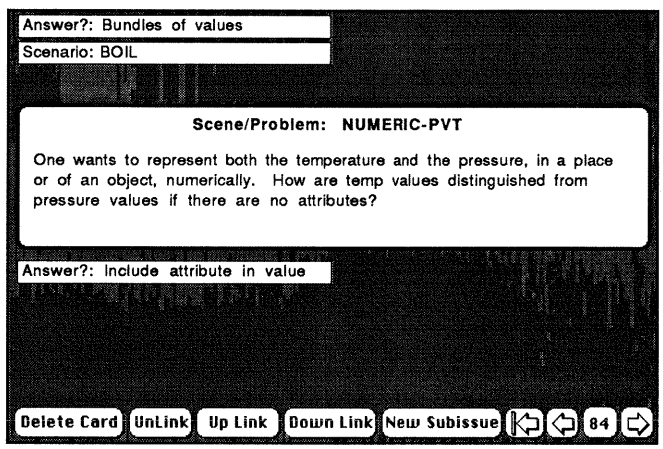

Problem: Properties of places
Answer?: Attach values to objects
The values associated with a place should be attached to objects that are more or less permanently found in that place.
Observation: Phlogiston
Delete Card
UnLink
Up Link
Down Link
New Subissue
79

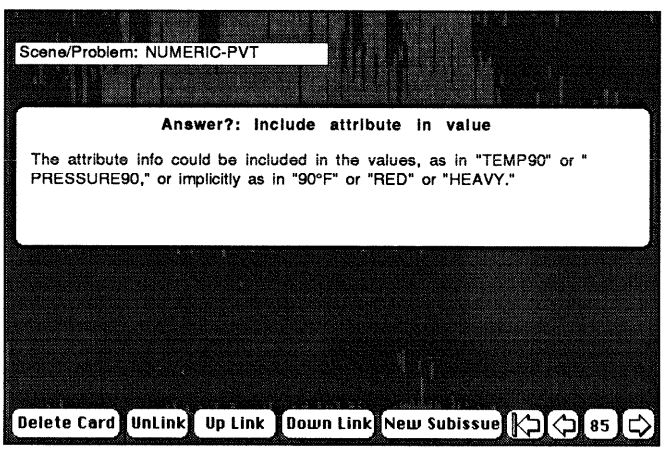

Answer?: Attach values to objects
Observation: Phlogiston
Instead of attaching values to dummy objects within a place, perhaps the objects themselves could represent the value. Thus, a place's temperature could be reduced by removing some of the temperature objects or by reducing the unattributed value of the phlogiston object.
Delete Card
UnLink
Up Link
Down Link
New Subissue
80

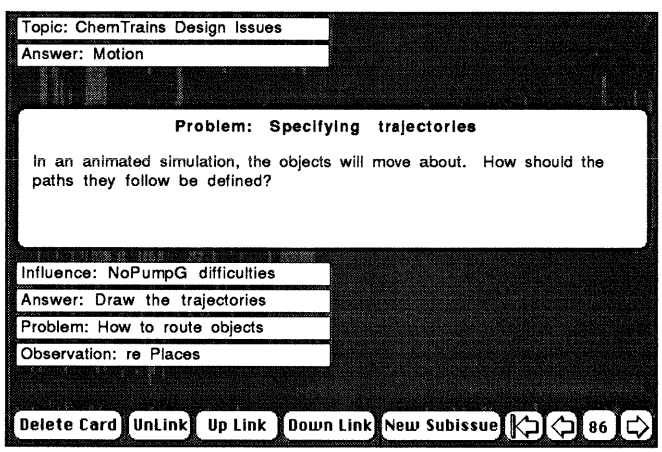

Problem: Properties of places
Answer?: Places should have values
Places should have values associated with them. That is, a place should be a data structure similar to an object.
Problem: Specifying places' values
Delete Card
UnLink
Up Link
Down Link
New Subissue
81

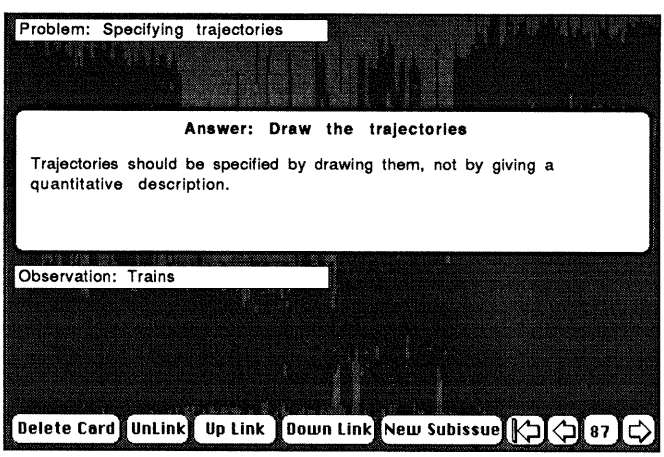

Answer?: Places should have values
Problem: Specifying places' values
How should the values associated with a place be specified?
Answer?: Attribute/value pairs
Answer?: Bundles of values
Observation: Natural language
Delete Card
UnLink
Up Link
Down Link
New Subissue
82

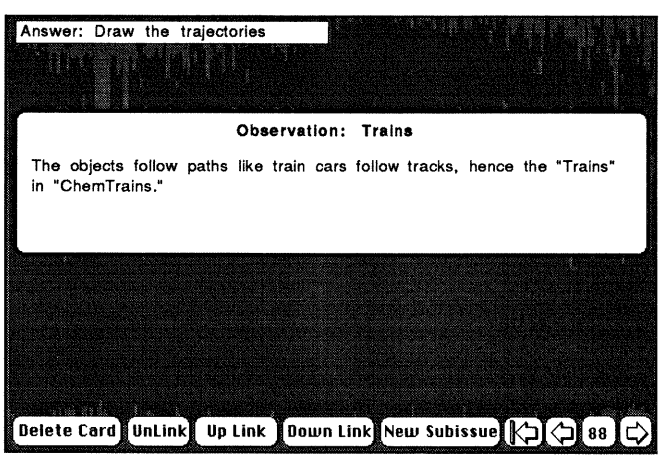

Scene/Problem: MEMO-CONTENTS
Scene/Problem: MEMO-TO/FROM
Problem: Specifying places' values
Problem: Specifying objects
Answer?: Attribute/value pairs
Places/objects should have attribute-value pairs.
Delete Card
UnLink
Up Link
Down Link
New Subissue
83

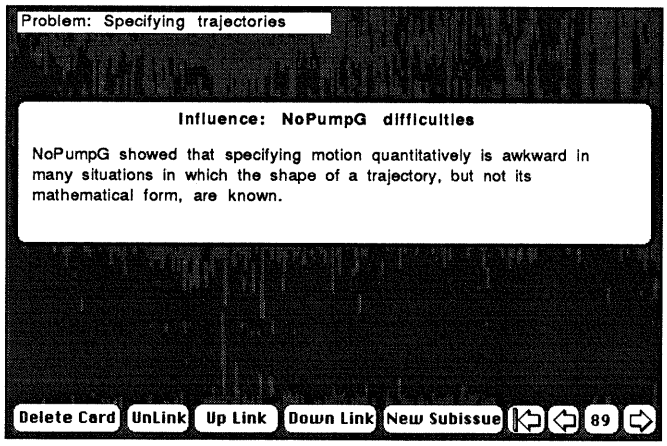

Answer?: Bundles of values
Scenario: BOIL
Scene/Problem: NUMERIC-PVT
One wants to represent both the temperature and the pressure, in a place or of an object, numerically. How are temp values distinguished from pressure values if there are no attributes?
Answer?: Include attribute in value
Delete Card
UnLink
Up Link
Down Link
New Subissue
84

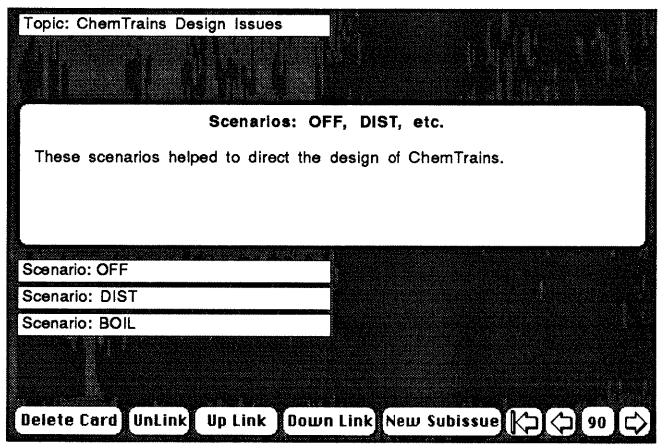

Scene/Problem: NUMERIC-PVT
Answer?: Include attribute in value
The attribute info could be included in the values, as in "TEMP90" or " PRESSURE90," or implicitly as in "90°F" or "RED" or "HEAVY."
Delete Card
UnLink
Up Link
Down Link
New Subissue
85

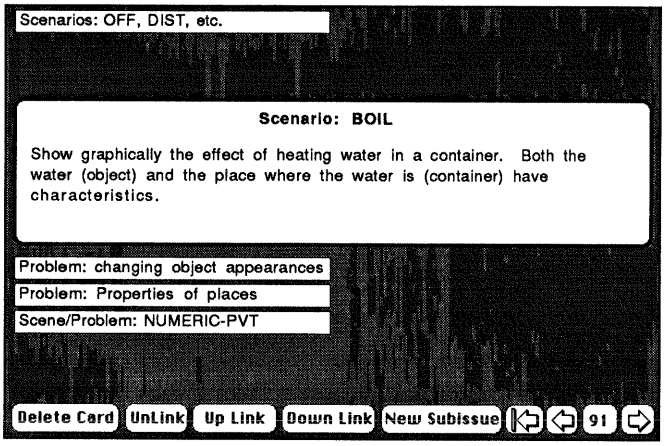

Topic: ChemTrains Design Issues
Answer: Motion
Problem: Specifying trajectories
In an animated simulation, the objects will move about. How should the paths they follow be defined?
Influence: NoPumpG difficulties
Answer: Draw the trajectories
Problem: How to route objects
Observation: re Places
Delete Card
UnLink
Up Link
Down Link
New Subissue
86

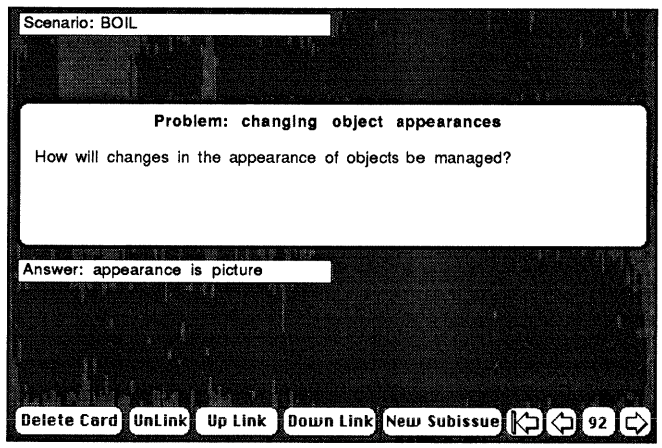

Problem: Specifying trajectories
Answer: Draw the trajectories
Trajectories should be specified by drawing them, not by giving a quantitative description.
Observation: Trains
Delete Card
UnLink
Up Link
Down Link
New Subissue
87

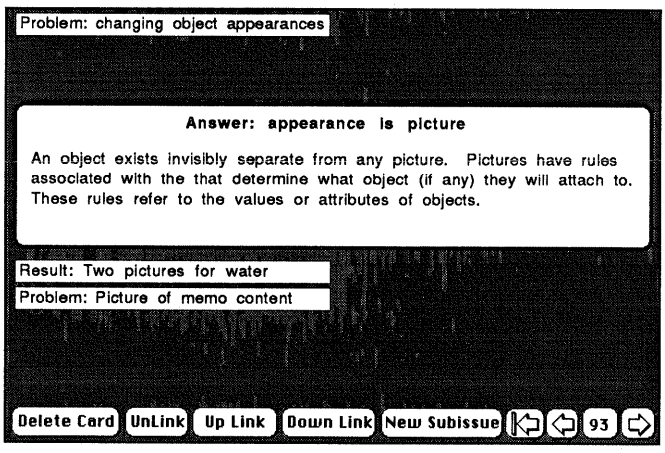

Answer: Draw the trajectories
Observation: Trains
The objects follow paths like train cars follow tracks, hence the "Trains" in "ChemTrains."
Delete Card
UnLink
Up Link
Down Link
New Subissue
88

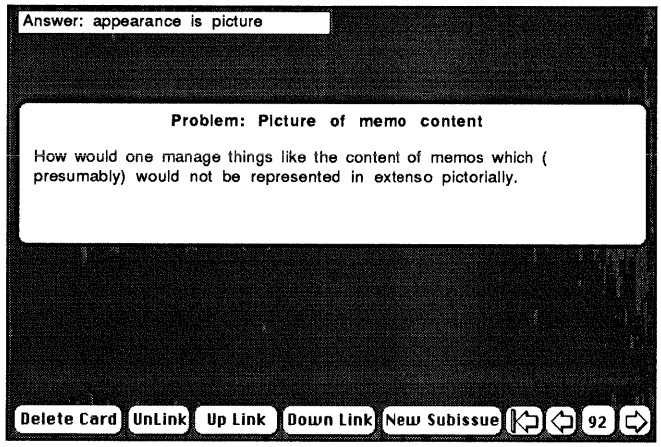

Problem: Specifying trajectories
Influence: NoPumpG difficulties
NoPumpG showed that specifying motion quantitatively is awkward in many situations in which the shape of a trajectory, but not its mathematical form, are known.
Delete Card
UnLink
Up Link
Down Link
New Subissue
89

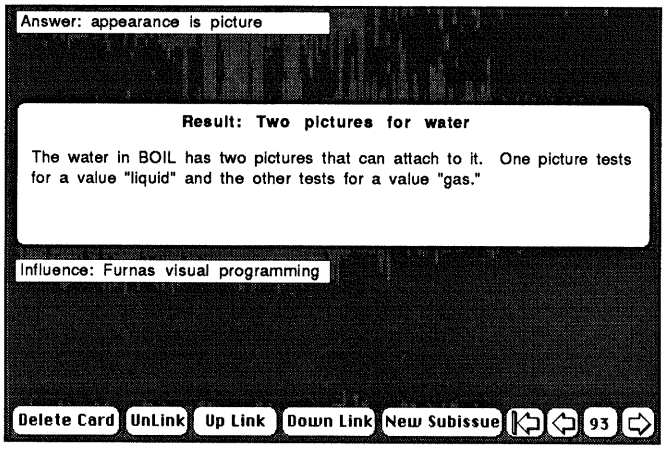

Topic: ChemTrains Design Issues
Scenarios: OFF, DIST, etc.
These scenarios helped to direct the design of ChemTrains.
Scenario: OFF
Scenario: DIST
Scenario: BOIL
Delete Card
UnLink
Up Link
Down Link
New Subissue
90

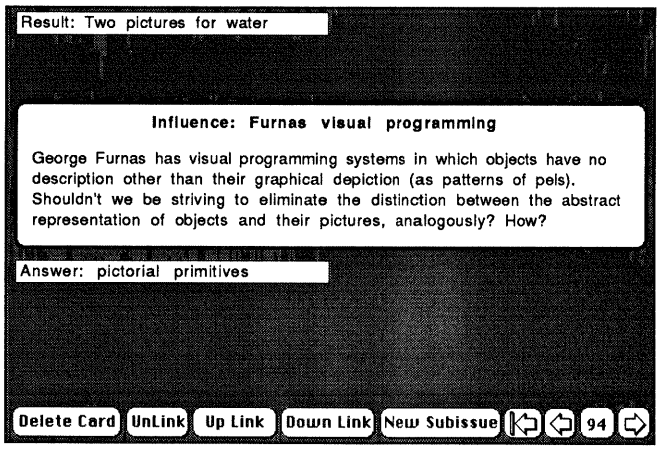

Scenarios: OFF, DIST, etc.
Scenario: BOIL
Show graphically the effect of heating water in a container. Both the water (object) and the place where the water is (container) have characteristics.
Problem: changing object appearances
Problem: Properties of places
Scene/Problem: NUMERIC-PVT
Delete Card
UnLink
Up Link
Down Link
New Subissue
91

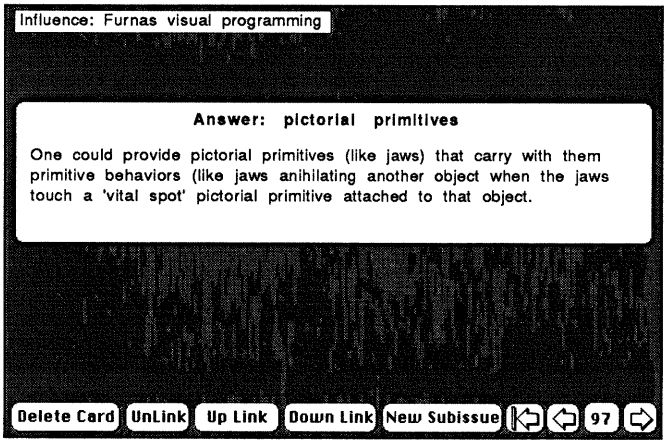

Scenario: BOIL
Problem: changing object appearances
How will changes in the appearance of objects be managed?
Answer: appearance is picture
Delete Card
UnLink
Up Link
Down Link
New Subissue
92

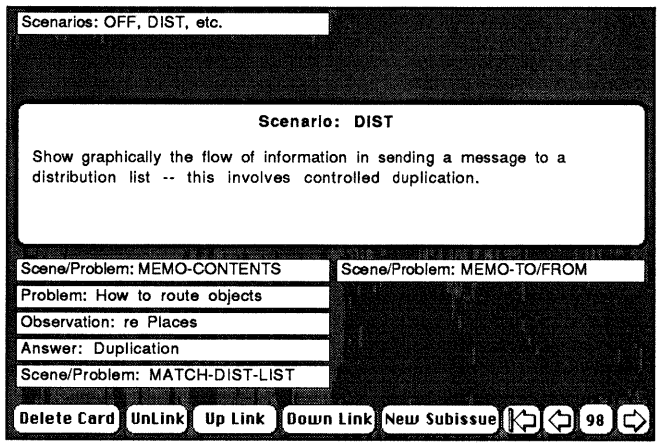

Problem: changing object appearances
Answer: appearance is picture
An object exists invisibly separate from any picture. Pictures have rules associated with the that determine what object (if any) they will attach to. These rules refer to the values or attributes of objects.
Result: Two pictures for water
Problem: Picture of memo content
Delete Card
UnLink
Up Link
Down Link
New Subissue
93

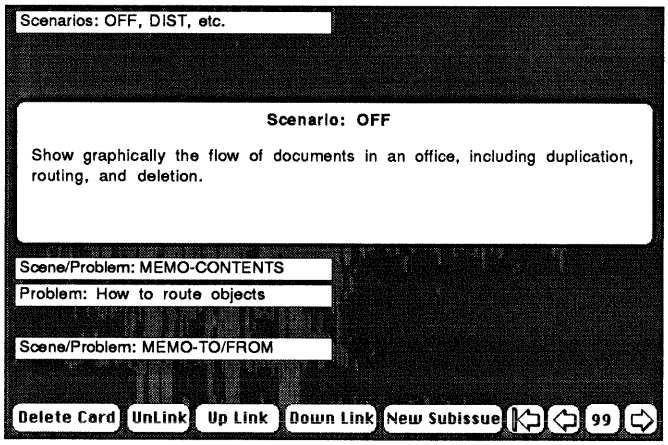

Answer: appearance is picture
Problem: Picture of memo content
How would one manage things like the content of memos which (presumably) would not be represented in extenso pictorially.
Delete Card
UnLink
Up Link
Down Link
New Subissue
92

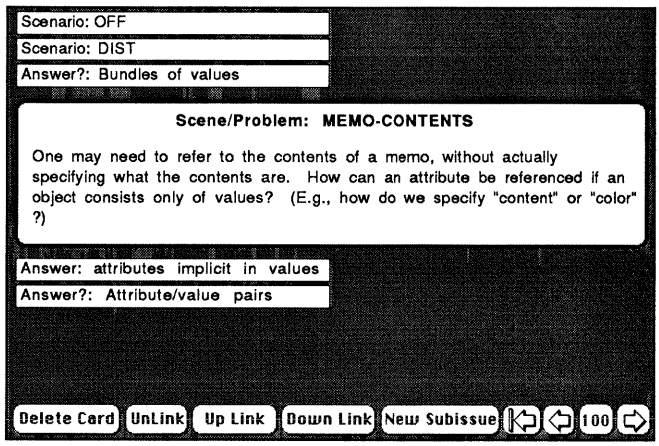

Answer: appearance is picture
Result: Two pictures for water
The water in BOIL has two pictures that can attach to it. One picture tests for a value "liquid" and the other tests for a value "gas."
Influence: Furnas visual programming
Delete Card
UnLink
Up Link
Down Link
New Subissue
93

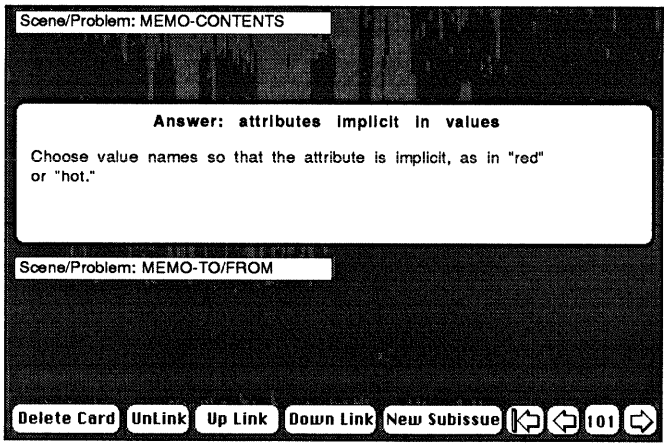

Result: Two pictures for water
Influence: Furnas visual programming
George Furnas has visual programming systems in which objects have no description other than their graphical depiction (as patterns of pels). Shouldn't we be striving to eliminate the distinction between the abstract representation of objects and their pictures, analogously? How?
Answer: pictorial primitives
Delete Card
UnLink
Up Link
Down Link
New Subissue
94

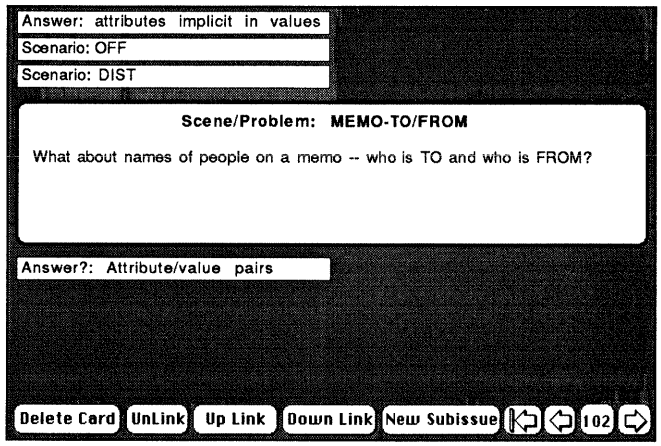

Influence: Furnas visual programming
Answer: pictorial primitives
One could provide pictorial primitives (like jaws) that carry with them primitive behaviors (like jaws anihilating another object when the jaws touch a 'vital spot' pictorial primitive attached to that object.
Delete Card
UnLink
Up Link
Down Link
New Subissue
97

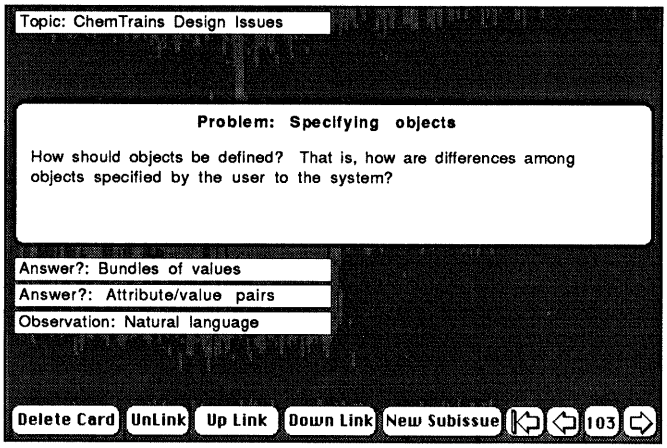

Scenarios: OFF, DIST, etc.
Scenario: DIST
Show graphically the flow of information in sending a message to a distribution list -- this involves controlled duplication.
Scene/Problem: MEMO-CONTENTS
Problem: How to route objects
Observation: re Places
Answer: Duplication
Scene/Problem: MATCH-DIST-LIST
Scene/Problem: MEMO-TO/FROM
Delete Card
UnLink
Up Link
Down Link
New Subissue
98

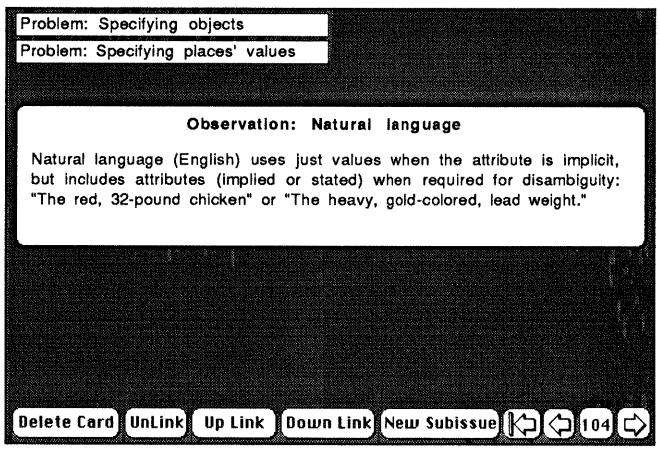

Scenarios: OFF, DIST, etc.
Scenario: OFF
Show graphically the flow of documents in an office, including duplication, routing, and deletion.
Scene/Problem: MEMO-CONTENTS
Problem: How to route objects
Scene/Problem: MEMO-TO/FROM
Delete Card
UnLink
Up Link
Down Link
New Subissue
99

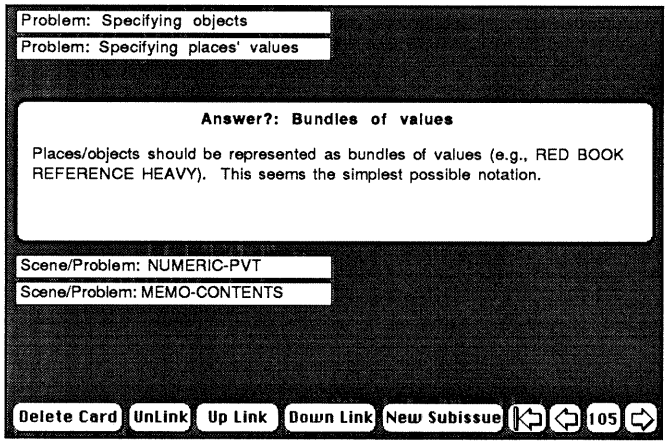

Scenario: OFF
Scenario: DIST
Answer?: Bundles of values
Scene/Problem: MEMO-CONTENTS
One may need to refer to the contents of a memo, without actually specifying what the contents are. How can an attribute be referenced if an object consists only of values? (E.g., how do we specify "content" or "color" ?)
Answer: attributes implicit in values
Answer?: Attribute/value pairs
Delete Card
UnLink
Up Link
Down Link
New Subissue
100

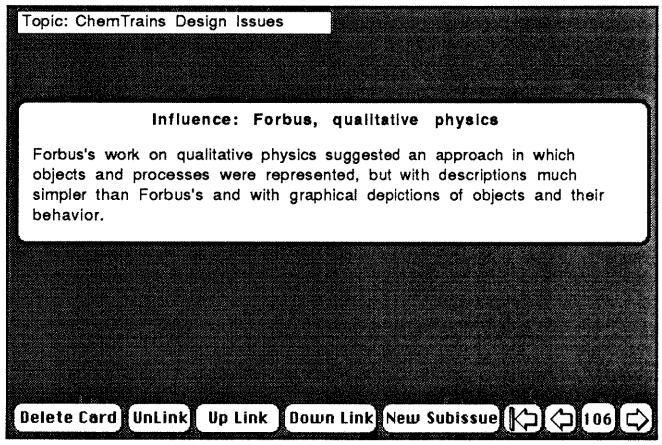

Scene/Problem: MEMO-CONTENTS
Answer: attributes implicit in values
Choose value names so that the attribute is implicit, as in "red" or "hot."
Scene/Problem: MEMO-TO/FROM
Delete Card
UnLink
Up Link
Down Link
New Subissue
101

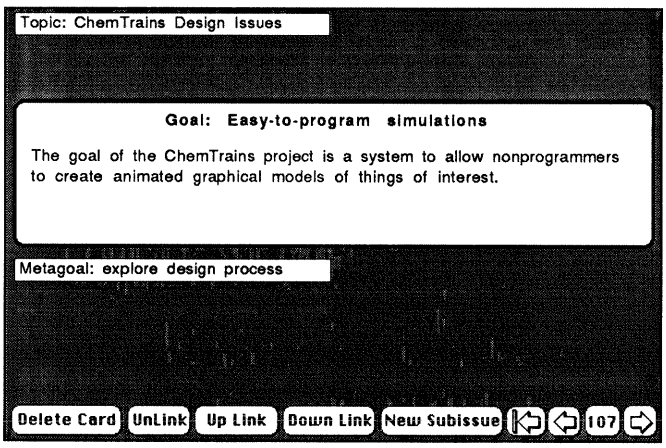

Answer: attributes implicit in values
Scenario: OFF
Scenario: DIST
Scene/Problem: MEMO-TO/FROM
What about names of people on a memo -- who is TO and who is FROM?
Answer?: Attribute/value pairs
Delete Card
UnLink
Up Link
Down Link
New Subissue
102

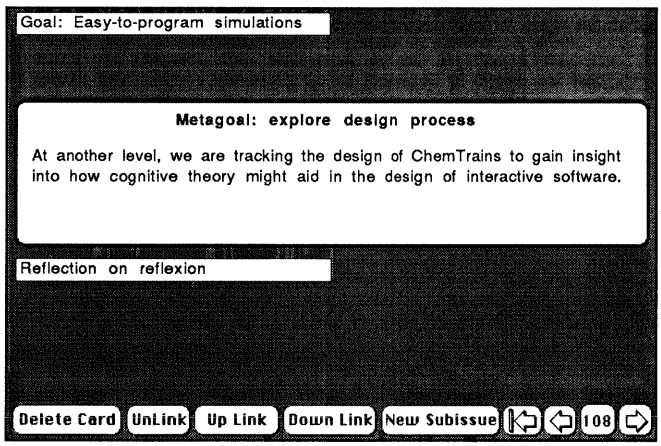

Topic: ChemTrains Design Issues
Problem: Specifying objects
How should objects be defined? That is, how are differences among objects specified by the user to the system?
Answer?: Bundles of values
Answer?: Attribute/value pairs
Observation: Natural language
Delete Card
UnLink
Up Link
Down Link
New Subissue
103

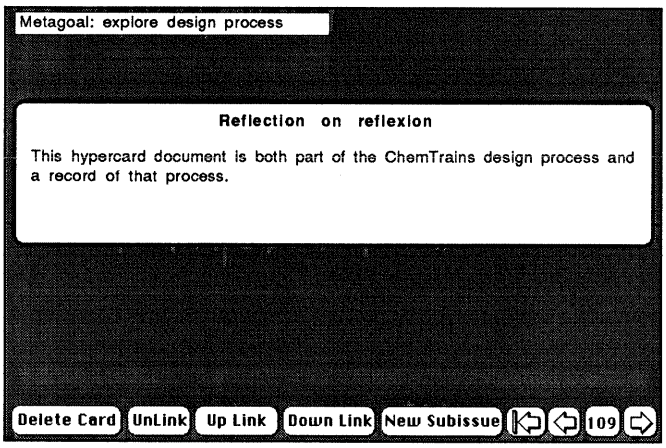

Problem: Specifying objects
Problem: Specifying places' values
Observation: Natural language
Natural language (English) uses just values when the attribute is implicit, but includes attributes (implied or stated) when required for disambiguity: "The red, 32-pound chicken" or "The heavy, gold-colored, lead weight."
Delete Card
UnLink
Up Link
Down Link
New Subissue
104


Problem: Specifying objects
Problem: Specifying places' values
Answer?: Bundles of values
Places/objects should be represented as bundles of values (e.g., RED BOOK REFERENCE HEAVY). This seems the simplest possible notation.
Scene/Problem: NUMERIC-PVT
Scene/Problem: MEMO-CONTENTS
Delete Card
UnLink
Up Link
Down Link
New Subissue
105


Topic: ChemTrains Design Issues
Influence: Forbus, qualitative physics
Forbus's work on qualitative physics suggested an approach in which objects and processes were represented, but with descriptions much simpler than Forbus's and with graphical depictions of objects and their behavior.
Delete Card
UnLink
Up Link
Down Link
New Subissue
106

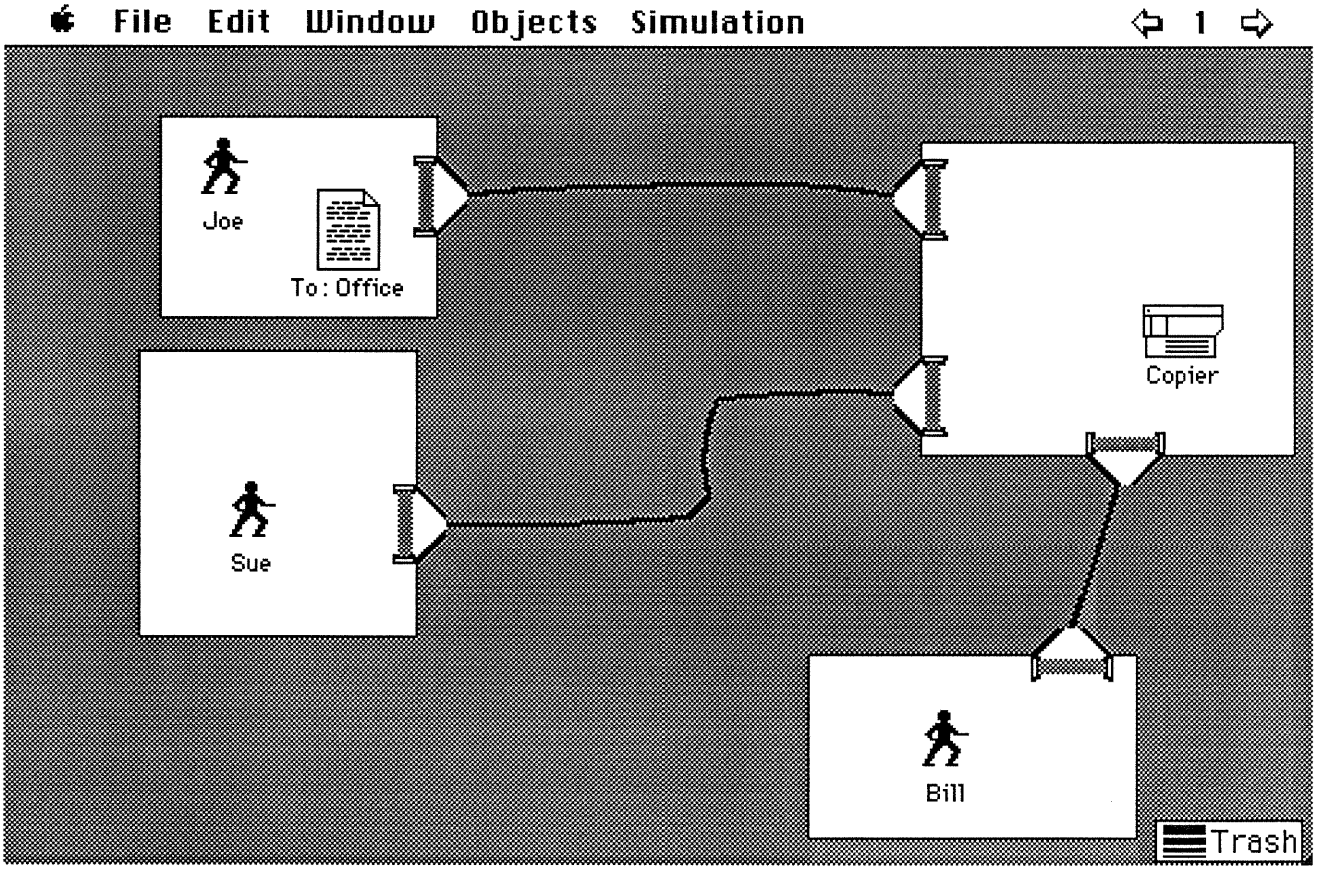

Topic: ChemTrains Design Issues
Goal: Easy-to-program simulations
The goal of the ChemTrains project is a system to allow nonprogrammers to create animated graphical models of things of interest.
Metagoal: explore design process
Delete Card
UnLink
Up Link
Down Link
New Subissue
107


Goal: Easy-to-program simulations
Metagoal: explore design process
At another level, we are tracking the design of ChemTrains to gain insight into how cognitive theory might aid in the design of interactive software.
Reflection on reflexion
Delete Card
UnLink
Up Link
Down Link
New Subissue
108


Metagoal: explore design process
Reflection on reflexion
This hypercard document is both part of the ChemTrains design process and a record of that process.
Delete Card
UnLink
Up Link
Down Link
New Subissue
109

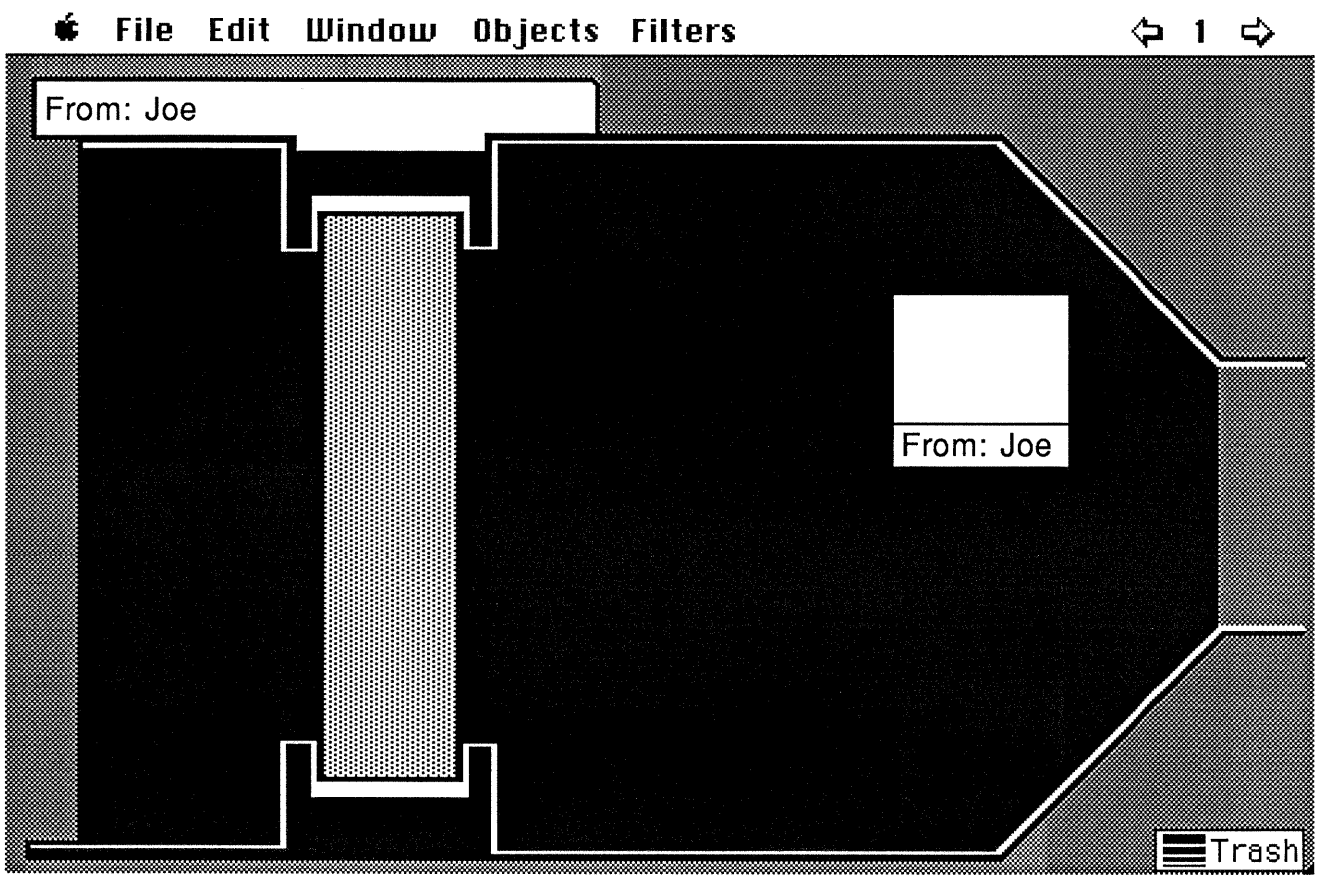
8. Screen Dumps of Early ChemTrains Prototype

Early in the ChemTrains design process, a prototype was produced using HyperCard. The screen dumps show the animation screens, rules, and filters (which control travel along paths) for versions of the Office and Turing Machine raw problems.



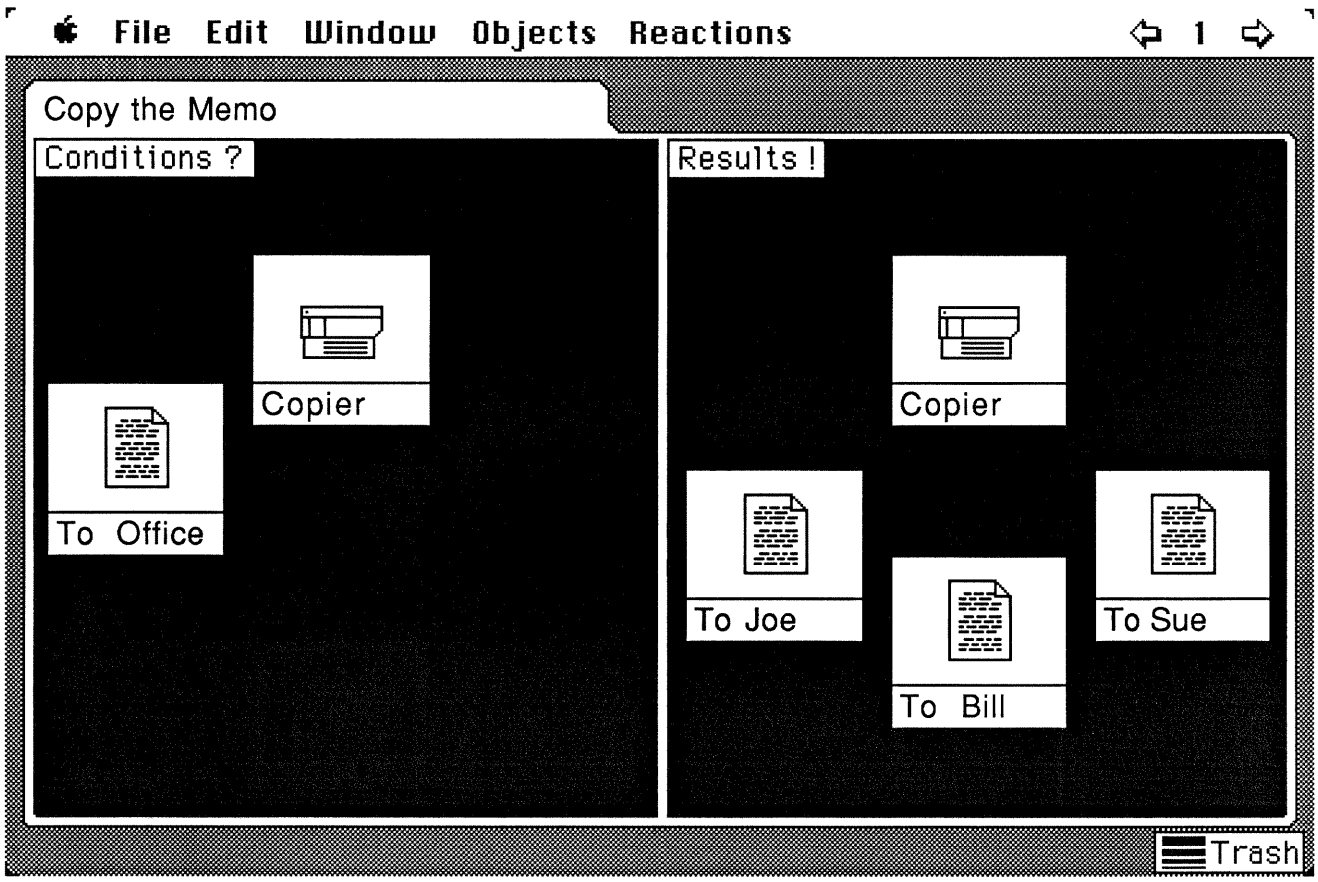
Simplified version of The Office Problem.

The memo marked "To: Office" is grabbed by the triangular filter and placed on the path to the copier room. In the copy room, a Reaction changes the "To: Office" memo into memos marked "To: Joe," "To: Bill," and "To: Sue." The filters attached to the copy room grab the new memos and put them on the appropriate paths.



A Filter, Close Up.

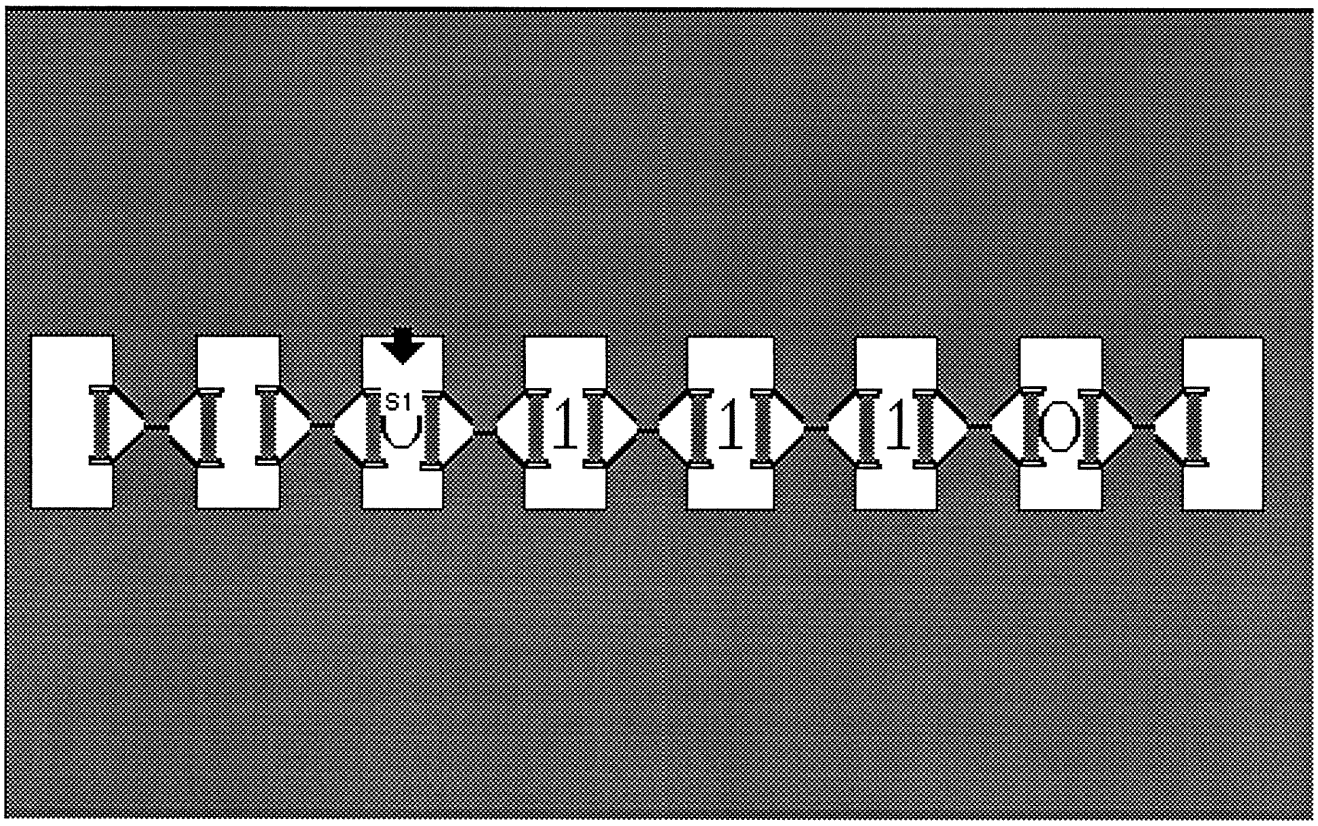
This filter is programmed, using direct manipulation, to pass any object marked "From: Joe."



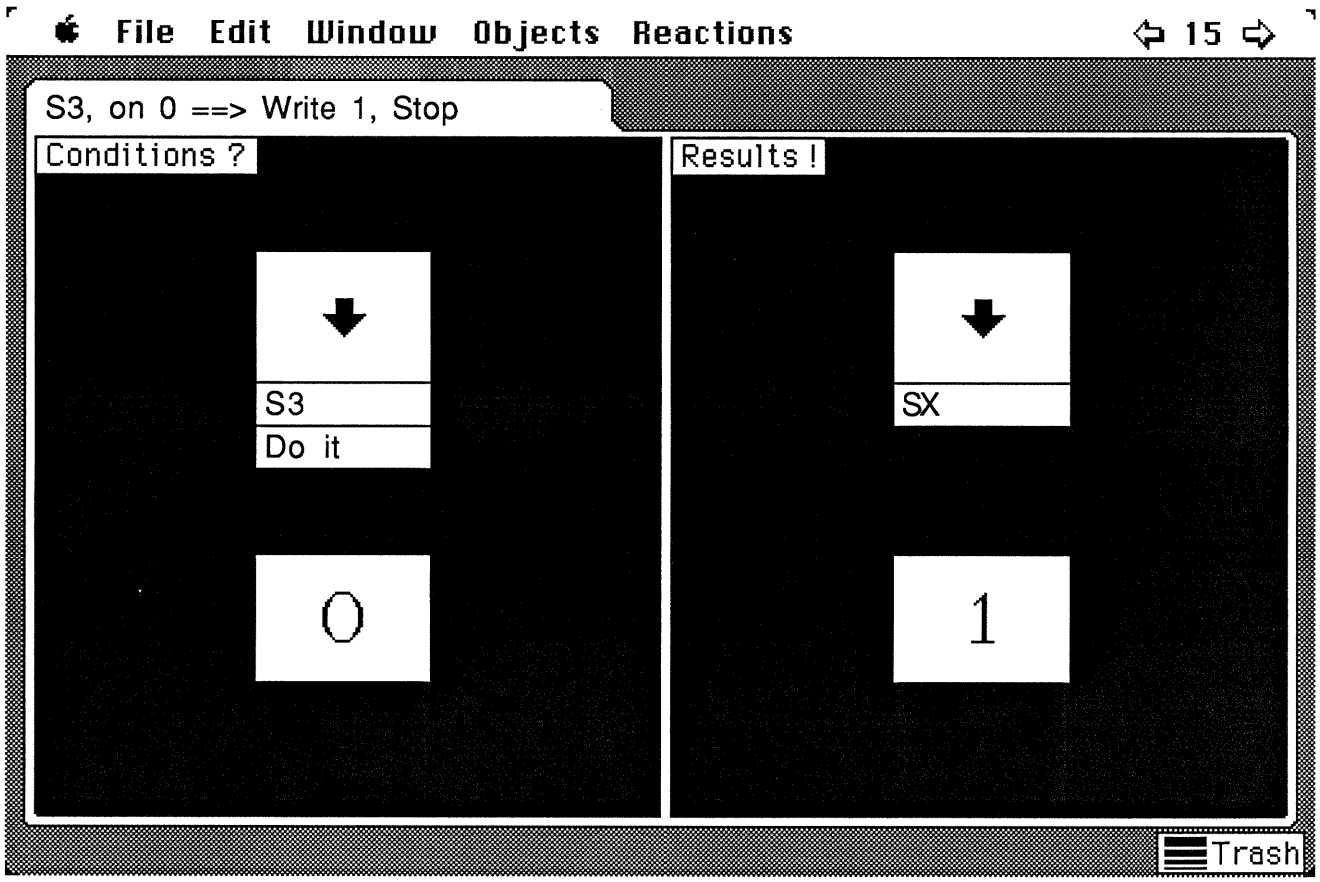
A Reaction Rule.

This rule specifies that when an object with the memo icon and the name "To Office" appears in the same place as an object with the copier icon and the name "Copier," the memo should be changed into memos to Joe, Bill, and Sue.

Reaction rules are programmed using direct manipulation.



The Turing Machine Simulation.



A Reaction Rule from the Turing Machine Simulation.