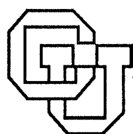
ChemTrains: A Rule-Based Visual Language for Building Graphical Simulations

Brigham Bell

CU-CS-529-92 February 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ChemTrains:
A Rule-Based Visual Language for Building Graphical Simulations

Brigham Bell

CU-CS-529-92 February 1992

This is a manual for the ChemTrains programming language that includes a historical introduction, advice for programming in the language, and example working programs.

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309

bell@cs.colorado.edu

Table of Contents

1. **Introduction** 1

2. **ChemTrains Programming Concepts** 2
   - 2.1 Terminology 2
   - 2.2 Pattern Matching 5
   - 2.3 Conflict Resolution 6
   - 2.4 Rule Execution 7
   - 2.5 User Interaction with the Simulation 7
   - 2.6 Built-in Counting Rules 9
   - 2.7 Hiding Parts of the Simulation 9

3. **Programming Techniques** 10
   - 3.1 Creating Rules 11
   - 3.2 Creating Objects and Containers 14
   - 3.3 Creating Paths 17
   - 3.4 Supporting End User Commands 20
   - 3.5 Controlling the Execution of Rules 21
   - 3.6 Creating Control Structures 23
   - 3.7 Creating Container Grids 24
   - 3.8 Creating Attribute-Value Tables 25
   - 3.9 Defining Sets 27
   - 3.10 Counting Events 28

4. **The Programming Environment** 29
   - 4.1 Menu Operations 30
   - 4.2 Execute and Edit Mode 30
   - 4.3 Drawing the Simulation 31
   - 4.4 Selecting, Copying, and Moving Objects 31
   - 4.5 Hiding and Unhiding Objects 31
   - 4.6 Creating a Rule 32
   - 4.7 Variables 32
   - 4.8 Editing an existing Rule 32
   - 4.9 Rule Ordering and Selection 33
   - 4.10 Creating and Editing Grids 33

5. **Example Problems and Solutions** 34
   - 5.1 Maze Search 34
   - 5.2 A Turing Machine Editor 36
   - 5.3 Tic Tac Toe 38
   - 5.4 Grasshopper Lifecycle Simulation 40

6. **References** 44

1. Introduction

ChemTrains is a visual programming language for describing graphical simulations that have a qualitative behavior model, such as document flow in an organization or the phase change of a substance as temperature varies. ChemTrains models show objects participating in reactions similar to chemical reactions and moving among places on the screen along paths. The name "ChemTrains" was suggested by the chemical reactions and the role of paths, thought of as train tracks.

The ChemTrains family of graphical simulation environments has been evolving over the past few years. The first discussions of a ChemTrains language were between Clayton Lewis and Victor Schoenberg in the fall of 1988. They wanted a language that supported simple graphical solutions to problems with a qualitative rather than a quantitative model. The initial discussions resulted in two main ideas:

- using a computational model based on chemical reactions, and
- allowing reactants to travel between connected places.

The first prototype was built by Schoenberg and John Rieman in the fall of 1989. This prototype pointed out some problems with the ideas, most notably that the resulting language was limited, and that the computational model of chemical reactions conflicted with a separate computational model for allowing reactants to travel between places. In the spring and summer of 1990, Lewis, Rieman, and myself began another iteration of ChemTrains design work. These design discussions focussed on many design issues, most of which were decisions between simple and powerful computational features. In the fall of 1990, I built a second prototype. The two biggest shortcomings of this prototype were that it couldn't support numeric computation and decomposition needed in solving larger problems. This manual describes the language resulting from a third iteration in the evolution of ChemTrains that was done as part of my thesis work.

The language design goals of ChemTrains are still more or less the same as they were in 1988:

- usable by inexperienced programmers who do not want to spend a lot of time hacking on the low-level details of a program,
- applicable to problems that have qualitative solutions,
- graphically-oriented, and
- domain independent.

The main design ideas also still exist:

- the simulation picture is constructed of graphical icons and objects;
- places and paths are drawn in the simulation picture for the purpose of computation;
- graphical objects may move between places along paths;

- the behavior of the simulation is executed by a computational model of reaction rules; and
- the reaction rules are described graphically.

Although the ChemTrains environment may be used by two types of people, *graphical programmers*, who create simulations, and *end users*, who use these simulations, it is suited more for the graphical programmer. The environment enables a graphical programmer to draw a simulation as it would initially appear to an end user, and then to specify the behavior of that simulation by drawing graphical rules. Each rule has two main components: a pattern picture and a result picture. When a simulation is executed, ChemTrains uses a rule interpreter to animate the display. When the rule interpreter recognizes that the pattern of one of the rules is identical to a portion of the main display, it then replaces that portion of the display with the picture in the result of the rule. When trying to recognize whether a pattern matches a portion of the display, the interpreter decides mainly based on whether the topology of the pattern matches the display rather than whether the geometry matches the display. The display will continually simulate as long as a pattern of a rule matches part of the display. When none of the rule patterns match the simulation, the rule interpreter stops.

2. ChemTrains Programming Concepts

This section describes the fundamental concepts useful in understanding and building ChemTrains simulations, concentrating on features of the language rather than user interface features of the environment. Where it is necessary, some details of the supporting environment are described. This section begins with definitions of terms used to describe ChemTrains simulations.

2.1 Terminology

A *simulation display* is a window that displays a graphical simulation. Figure 1 is an example simulation display. A simulation display is analogous to working memory in a production system language. ChemTrains rules operate on the simulation display similar to the way that productions in a production system language operate on the working memory.

The *simulation environment* is the programming environment that enables users to modify the simulation display and the rules.

An *object* is a graphical object of the display. An object may either be a rectangle, an oval, a polyline (a set of connected line segments), a text string (also called a label), or an icon (a pixel array). The simulation environment supports creation of each kind of object.

Figure 1 shows a picture in which each kind of object is used. The mountain scene is a single icon. Each of the houses are copies of a single icon. The

labels on each of the houses are text strings. The two trees are polylines. Five boxes are shown, two that are used to group houses, and three that are used to compose a drawing of a switch at the bottom right side of the picture. Two ovals are also used in the drawing of the switch. The box surrounding the whole picture defines the window boundaries.

Two objects are *identical* if the objects are the same kind (either rectangle, oval, polyline, icon, or text string) and their displays are identical. Each of the houses in figure 1 are identical because they are all copies of the same house icon, even though the text strings overlap the house displays. The two large rectangles are also identical, because they are both rectangle objects with the same dimensions. Two objects that look identical may not actually be identical. For example, the trees in figure 1 would not be identical if one was drawn as an icon and the other was drawn from line segments, and the two rectangles may not be identical if one was constructed as a rectangle object and the other was constructed out of line segments. The best way to make sure that a set of objects are identical is to copy a single version of an object.

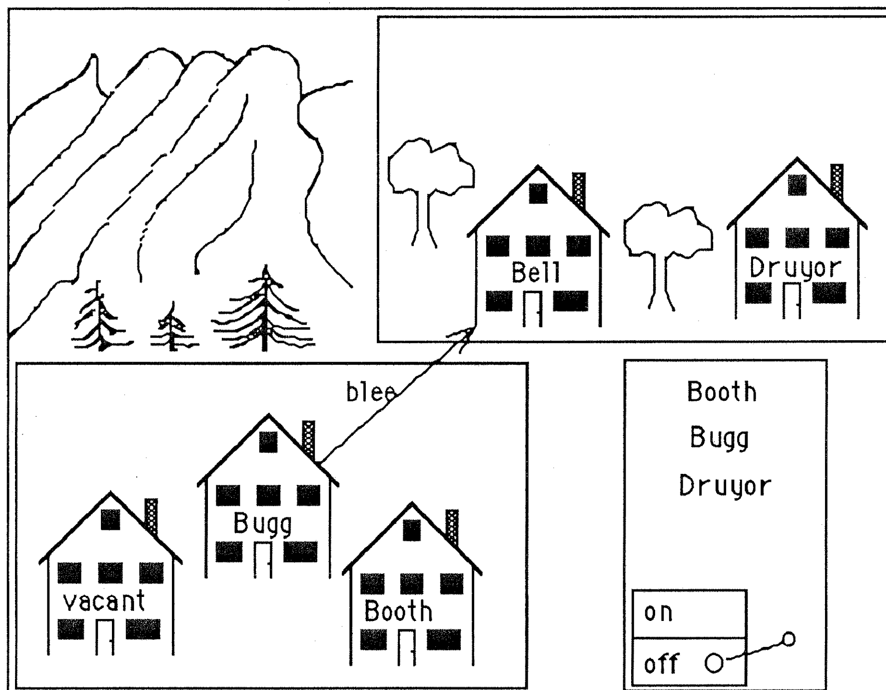


Figure 1: Example ChemTrains Simulation Display: house lighting controlled by a switch.

Objects that are identical are considered to be instances of the same *type*. In most programming languages, there is a strong separation between type and instance, and there is usually a specific syntax for defining types. In ChemTrains however, there is no mechanism for defining types, and the distinction between type and instance is hidden from the programmer. In

creating an object on the simulation display that is different from all other objects, a new type is defined. If the new object happens to be identical to other objects, it is considered another instance of an existing type. When an object is changed on the display (e.g. resizing a rectangle or redrawing the pixels of an icon), the simulation environment recognizes the change as meaning two possible things, that either

- only this object instance is to be changed, defining a new type, or
- all identical object instances are to be changed, redefining the type definition that all the identical objects share.

When an object is modified and there are other identical objects, the system asks whether all identical objects or just this single object should be modified.

An object is *inside* another object if its bounding rectangle encloses the bounding rectangle of the other object. In figure 1, the text strings that label the houses are each inside of the single houses that they label, because they are each completely surrounded by the house. The two trees and the "Bell" and "Druyor" houses along with their labels are each inside one single box.

An object *contains* another object if the inside constraint is met.

A *container object* or a *place object* is an object that is being used in the simulation to contain other objects. Any object may be a container object. The houses are considered container objects for their labels. The two small boxes that contain the "on" and "off" labels and the oval are also container objects.

A *path* is a line connecting one object to another object. Paths must be explicitly created in the simulation environment, and are different from polyline line segments, which are objects.

A path *connects* two objects.

A path can either be *directed* or *non-directed*. A directed path has an arrowhead on one end, and a non-directed path has no arrowhead. The path connecting the "Bugg" house to the "Bell" house is a directed path. The path connecting the two ovals in the bottom right area is a non-directed path.

An object *labels* a path if the object is a text string that overlaps the path. A path can only have one label. The directed path in figure 1 is labeled by the "blee" text string.

A *replacement rule* or *rule* is a mechanism for defining general changes in a picture. A replacement rule has a name, a pattern picture, and a result picture. When the objects of the pattern picture of a rule matches objects in the simulation picture, the matched objects are replaced with objects shown

in the result picture of the rule. Figure 2 shows an example rule called "house on."

A *variable* is an object in either the pattern or result of a rule that may represent any object in the simulation. When an object is variablized, it is displayed in italics if it is a text string, or it is otherwise marked with a big "V." In figure 2 the "name" text strings are variables, and everything else is not a variable.

The environment allows the programmer to be in either *execute mode* or *edit mode*. In execute mode, the programmer views the graphical simulation working as an end user may see it. In this mode, the rule interpreter grabs control over the simulation whenever a change is made to the simulation, and then releases control when no rules apply. In edit mode, the programmer may edit the simulation without the rule interpreter interfering.

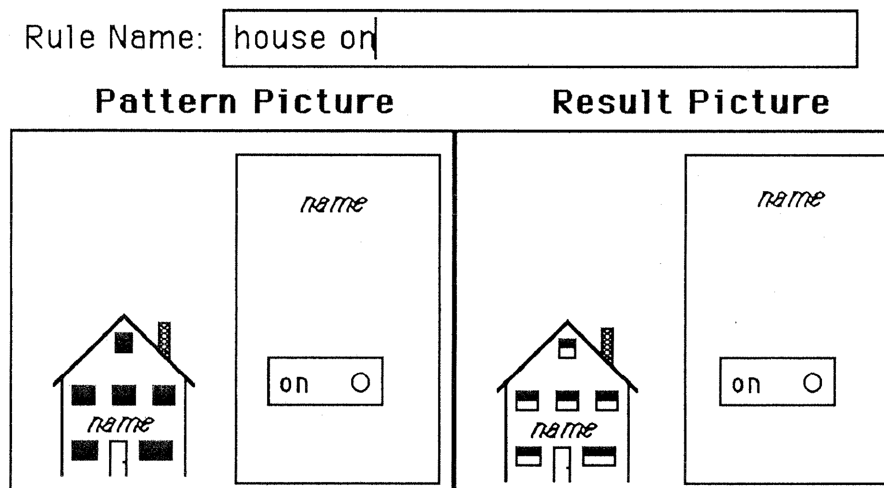


Figure 2: Example simulation rule to turn on house lights.

The rule interpreter of ChemTrains, like other production system, executes a recognize-act cycle that has three phases:

1. a pattern matching phase,
2. a conflict resolution phase, and
3. a rule execution phase.

After the rule execution phase, the rule interpreter starts a new cycle. If no rule matches during pattern matching, the interpreter halts. Each of the three phases are described in detail below.

2.2 Pattern Matching

The ChemTrains pattern matcher does not care that the exact placement of objects in the pattern of a rule matches the exact placement of objects in the simulation picture; however, it does care that some very specific types of graphical constraints are met by matched objects in the simulation. It cares:

- whether objects are inside other objects;
- whether objects are connected to other objects; and
- whether text strings label paths.

Here is a more formal description of pattern matching. The pattern picture of a rule matches objects in the simulation if and only if:

1. Every object in the pattern that is not a variable matches an identical object in the picture.
2. Every variable in the pattern matches an object in the picture.
3. Identical variables in the pattern match identical objects in the picture. For example, two identical variables in the pattern can match any two things in the simulation as long as the two things are identical.
4. For every inside constraint in the pattern, the inside constraint must also be satisfied by the matched objects.
5. For every nondirected path in the pattern, a nondirected path must connect the matched objects.
6. For every directed path in the pattern, a directed path must connect the matched objects and point in the same direction.
7. For every text string object that labels a path in the pattern, the matched text string must label the matched path.

Any other sort of geometric relationship between objects in the pattern picture, such as adjacency, of a rule is ignored in pattern matching. For example, the pattern shown in figure 2 literally means:

- match an unlit house that contains some object, and
- match a rectangle identical to the large rectangle that contains:
  1. an object identical to the object inside the unlit house, and
  2. a rectangle identical to the small rectangle that contains:
     a. a text string named "on," and
     b. an oval identical to the oval shown.

When the switch in figure 1 is turned on, the pattern of this rule matches the picture in three different ways, matching the "Bugg," "Booth," and "Druyor" houses. When interpreting the meaning of a pattern, it is useful to think in terms of the topology of a picture rather than the geometry.

2.3 Conflict Resolution

When more than one rule matches a picture, the rule highest in priority is chosen. The language environment supports a feature for ordering the rules. An example of using rule ordering to control a simulation would be in writing rules to play tic tac toe: a rule to play a win would be placed before a rule to play a block.

ChemTrains adds one enhancement to the rule ordering feature. Rules in the ordering may be specified to be at the same level of priority. In the programming environment, this is referred to as "parallelizing" rules. When two or more rules are parallelized with respect to each other, it means that it picks between those rules randomly if the engine cannot match any rules

higher in priority. When a set of rules is parallelized, the probabilistic weights of choosing each of the parallel rules can be modified by the programmer. This feature is used to describe behaviors that happen randomly with respect to each other, such as grasshoppers jumping and eating at the same time, but jumping at a higher probability than eating.

When a rule is finally chosen and this rule may match multiple combinations of objects in the simulation, it chooses between the possible combinations randomly. Also, the interpreter never executes on the same unchanged combination of data more than once.

2.4 Rule Execution

When a rule is chosen, the system executes the rule by interpreting actions, specified by differences between the pattern and result pictures of the rule. The following kinds of differences are permitted:

- any objects or paths from the pattern may be deleted;
- any paths may be added;
- an object may be added if it is placed inside an object that is in both the pattern and result pictures; or
- an object may be added if it replaces a deleted object from the pattern picture, as the lit house replaces the unlit house in the "house on" rule shown in figure 2.

When the control knob is moved to the "on" position, the "house on" rule executes three times on three consecutive recognize-act cycles, resulting in turning three lights on as shown in figure 3. To enable the house lights to be turned off when the switch is in the "off" position, the "house off" rule, shown in figure 4, is added.

The exact placement of the objects in a result does not have to perfectly match the placement of corresponding objects in a pattern. ChemTrains tries to figure out the mapping from pattern objects to result objects. When there are multiple possible mappings (examples shown in later chapters), the system chooses the mapping that best fits the spatial relationships in the pattern and result pictures.

2.5 User Interaction with the Simulation

After a rule is executed, ChemTrains begins the recognize-act cycle again. If no rule matches in the pattern matching phase, the system stops rule execution and allows the user to modify the simulation. When anything is changed in the picture (e.g. an object is moved, added, or deleted), the system re-starts the recognize-act cycle. Because the rule interpreter will execute as a user interacts with the system, ChemTrains can support construction of simulations that require interactions with the end user during its execution.

The user can also modify the simulation during the execution of the recognize-act cycle. When the system recognizes that the user is trying to

move something by clicking on it, the system halts the cycle and permits the movement of the object clicked by the user. If the user's action interrupts pattern matching or conflict resolution, the cycle is simply halted. If the user's action interrupts the execution of a rule's actions, the actions are fully completed. A rule is never partially executed because this may lead to an illegal or unwanted state in the simulation. After the user's action is completed, the system resumes execution of the recognize-act cycle. Because ChemTrains allows interaction during a simulation and recognizes these changes and acts on them, ChemTrains can be considered a general tool for building user interfaces for bitmap displays.

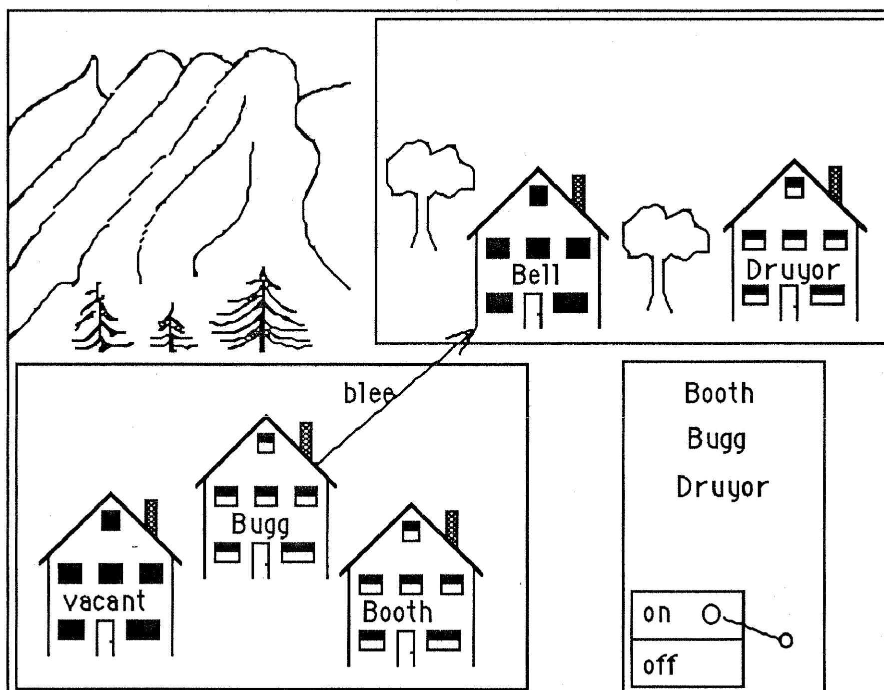


Figure 3: Changes to House Lighting Simulation Picture.

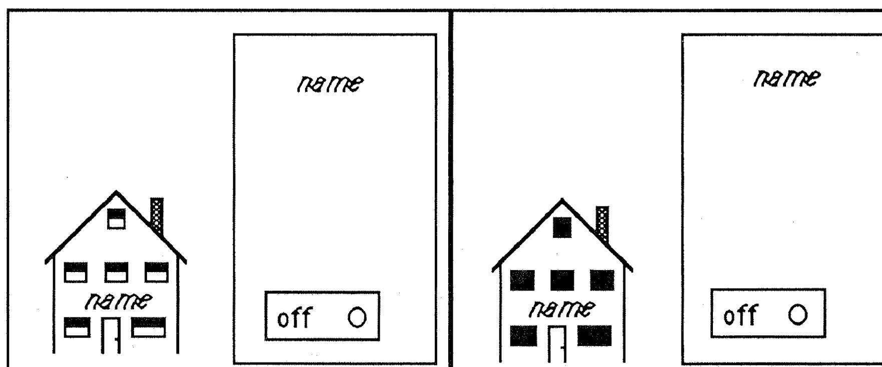


Figure 4: Example simulation rule to turn off house lights.

2.6 Built-in Counting Rules

ChemTrains provides three built-in rules for counting. The rules can execute when either a "incr," "decr," or "clear" text string object is in the same container as a text string object that appears as a number, such as "1495" or "-112.73". When this situation occurs, the numeric text string object will be modified appropriately (incremented, decremented, or changed to "0"), and the operator will be deleted. Since these rules are built into the recognize-act cycle as the rules with highest priority, a counting operation will execute as soon as a count operator appears in a container with a number. The grasshopper simulation discussed in section 5.4 demonstrates the use of counters. When a counter operator exists inside a layered topology of places each with numbers, the system picks the number that exists with the operator in the smallest container.

2.7 Hiding Parts of the Simulation

Objects and paths may be *hidden*. The environment provides interface commands for hiding and unhiding selected objects and paths, and also provides an interface switch for showing all hidden objects, called "Show hidden?." When this switch is on, hidden objects and paths are displayed, and when the switch is off, hidden objects and paths are invisible. The rule interpreter matches on hidden objects as if they are not hidden. Whether an object is hidden or not has no effect on pattern matching.

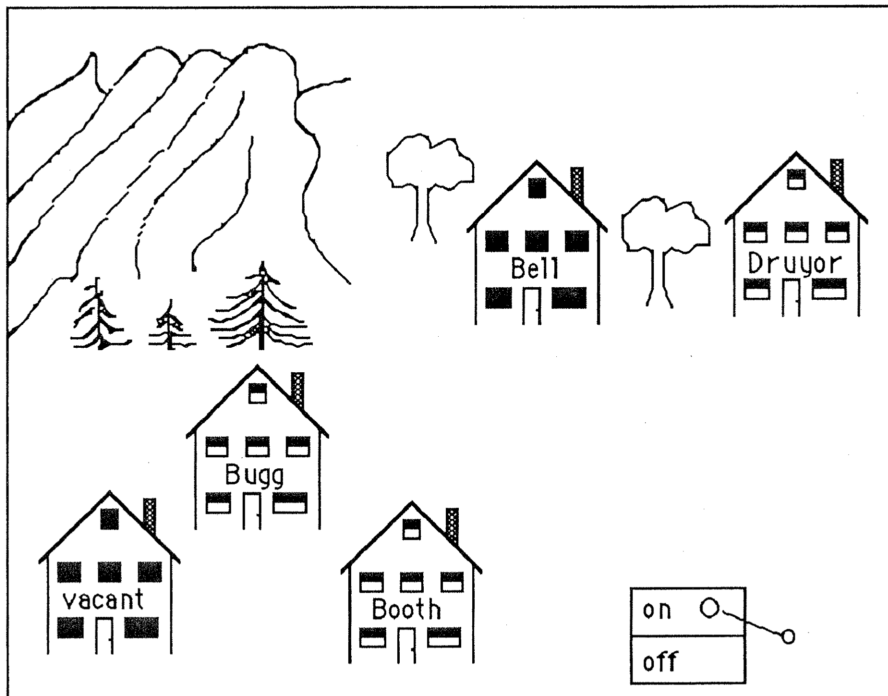


Figure 5: House lighting simulation picture with hidden objects not shown.

The "hide" feature was added so that parts of a simulation can be hidden for some users, such as end users, and displayed to other users, such as graphical programmers. The feature is most usefully applied to components of a simulation that are necessary in order to get a simulation working but are confusing or irrelevant to end users. Figure 5 shows the house lighting simulation picture as seen when the hidden objects are not displayed. The names of houses above the on/off switch are hidden. These invisible objects still exist in the same positions, allowing the simulation to continue working as before.

The tic tac toe simulation (section 5) also illustrates the use of hidden objects. The nine cells of the board and the labeling of those cells must be created in order to get the simulation working, but may be hidden so that the end user doesn't have to see them. These cells could be considered internal data structures of the tic tac toe program.

3. Programming Techniques

This section describes advice that may be useful in constructing simulation displays and in describing the behavior of the simulation. The advice is broken into individual programming techiques, each of which has four parts: a name describing its use, an abstract description of the problem-solving situation in which the technique could be applied, and the actions that should be taken. Many of the techniques are also illustrated with examples directly from the house lighting problem already shown or with possible extensions to the house lighting problem. Here is an example technique:

draw initial picture:
When starting,
draw how the simulation would initially appear to the end user.

The rest of the programming techniques are divided into the following groups:

1. Creating Rules
2. Creating Objects and Containers
3. Creating Paths
4. Supporting End User Commands
5. Controlling the Execution of Rules
6. Creating Control Structures
7. Creating Container Grids
8. Creating Attribute-Value Tables
9. Defining Sets
10. Counting Events

3.1 Creating Rules

This advice describes how to construct rules for animating an existing picture.

create rule:

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 create a rule, following these steps:

1. 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,
2. copy these objects and paths also onto the result picture of the rule,
3. modify the objects in the result picture as desired (advice for describing rule actions follows), and
4. give the rule a name that is appropriate for the task it does. (figure 6 through 11)

deletion action:

If an object or path is to be deleted when a rule is executed,
then remove that object or path from the result picture. (figure 6 & 9)

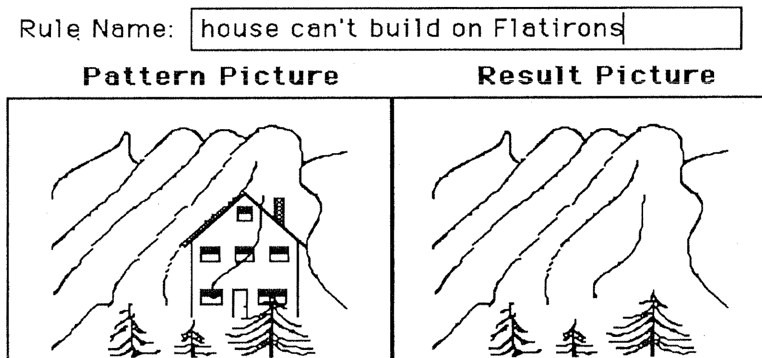


Figure 6: Example of a **deletion action** to delete any house in the Flatirons.

addition action:

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. (figure 7)

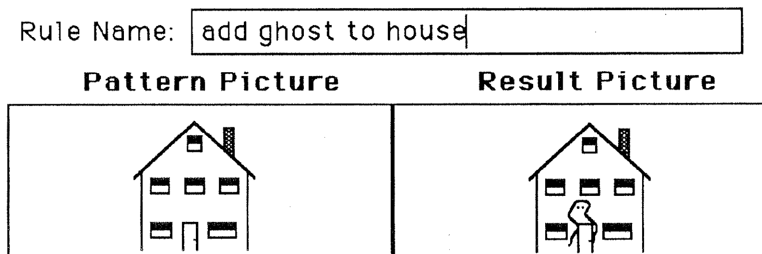


Figure 7: Example of an **addition action** in a rule to add a ghost to every house.

replacement action:

If an object is to be replaced with another object when a rule is executed,
then remove the object from the result and replace it with the new object. (figure 10)

movement action:

If an object is to move from one container to another when a rule is executed,
then move that object in the result picture from its current container to an appropriate position in the destination container. (figure 8)

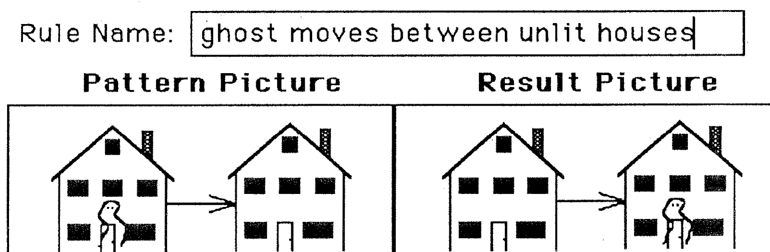


Figure 8: Example of a **movement action** in a rule to move ghosts between unlit houses.

wildcard match:

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. (figure 7 & 9)

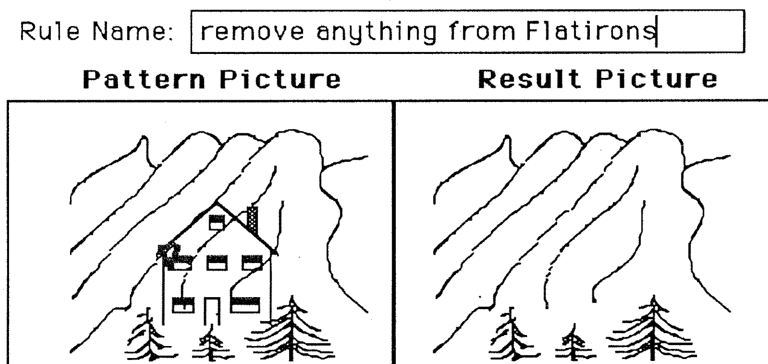


Figure 9: The variablized house is an example of a **wildcard match**, and the rule is an example of a **deletion action**. This rule will delete any object in the Flatirons.

identical variable match:

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. (figure 10)

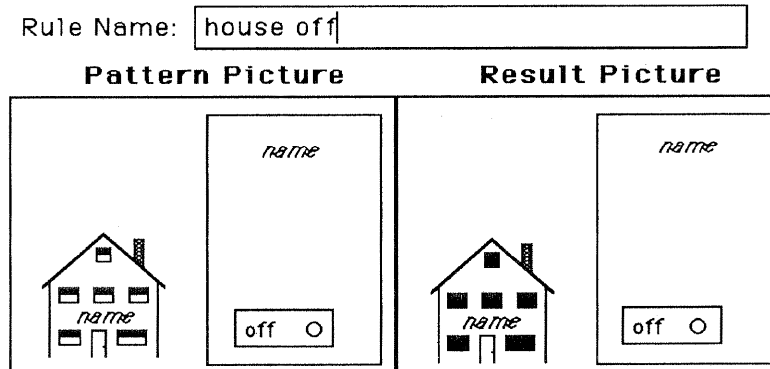


Figure 10: The italicized "names" are an example of an **identical variable match**, and the switch from a lit house to an unlit house is an example of a **replacement action**.

variable addition action:

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. (figure 11)

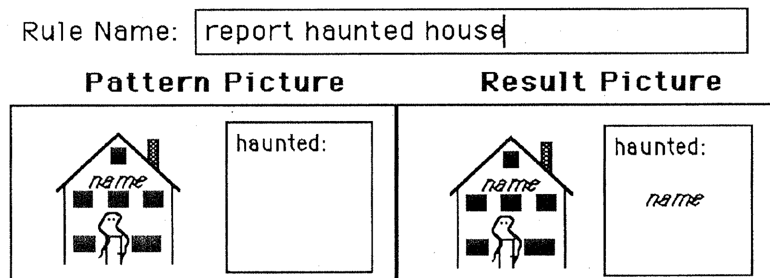


Figure 11: The italicized name is an example of a **wildcard match** and the new name in the result is an example of a **variable addition action**.

3.2 Creating Objects and Containers

This advice describes some principles that can simplify the programming of the simulation's behavior.

draw additional container:

If an object can move or can be placed in a particular area of the interface that is not drawn,

then draw a container object that is big enough to contain the object. (figure 12)

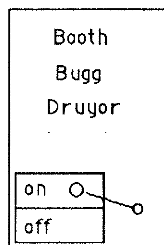


Figure 12: The outer rectangle is an example of an **additional container** needed to contain the names of houses that can have lighting power.

draw position containers:

If an object or set of objects can move around to different positions,

then draw a container object that is big enough to contain the object and copy it to all of the positions that it may be moved to.

draw big enough container:

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.

draw unique identifier:

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), and place a copy inside each object. (figure 13)

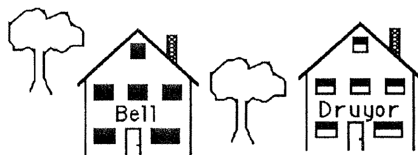


Figure 13: The "Bell" and "Druyor" labels are examples of **unique identifiers**.

draw mode description:

If the simulation should behave differently in different modes, and the modes cannot be determined by pictures on the screen,
then create a container that contains a label defining the current mode. (figure 14)

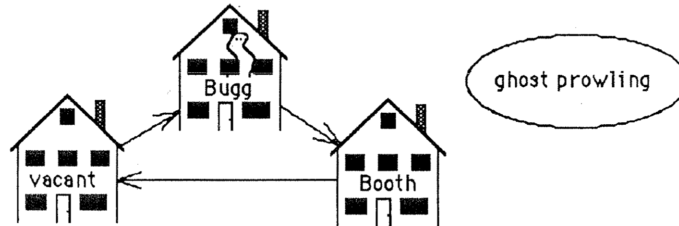


Figure 14: The "ghost prowling" label and associated container is an example of drawing a **mode description**. The connections between the houses are an example of **draw directed path**.

mode condition:

If a rule can only occur during a mode as drawn,
then put the mode description in the pattern and result of the rule. (figure 15)

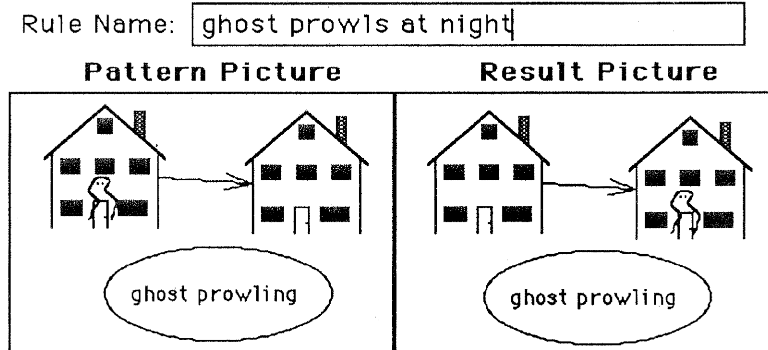


Figure 15: Using the "ghost prowling" text string and the oval is an example of a **mode condition** in a rule.

draw empty container marker:

If a simulation needs to test for the absence of an object within a container,

then create an object to show that a container is empty, and place it in every empty container in the simulation. (figure 16) Follow the absence testing advice to create rules that use the empty container marker.

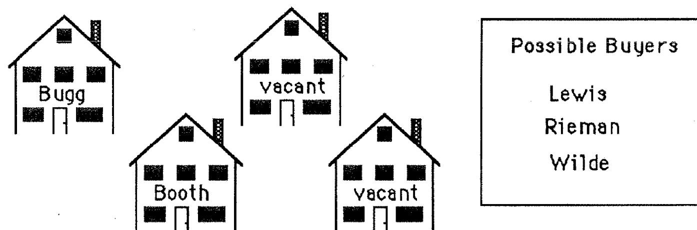


Figure 16: The "vacant" labels in the two houses are text string objects that signify that a house is empty. These labels may be used to determine whether a container is empty.

absence testing:

If a rule needs to test for the absence of an object within a container, and the absence has been represented with empty container markers (from the previous advice),

then test for the presence of the empty container marker, by placing the empty container marker in the pattern of the rule. (figure 17)

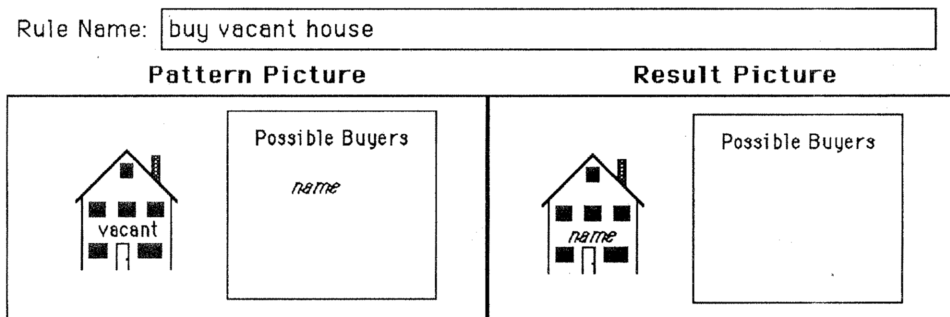


Figure 17: This rule tests for the absence of a family in a house by testing for the presence of the "vacant" string.

hide objects:

If a set of graphical objects and paths has been added only to make the simulation work, and would be irrelevant to an end user,

then hide the objects.

3.3 Creating Paths

This advice describes the situations for creating paths between places.

draw directed path:

If an object can travel between two container objects and can travel only in one direction,
then create a directed path connecting the two objects. (Figure 17)

draw non-directed path:

If an object can travel between two container objects and may travel in both directions under the same conditions,
then create a non-directed path connecting the two objects. (figure 18)

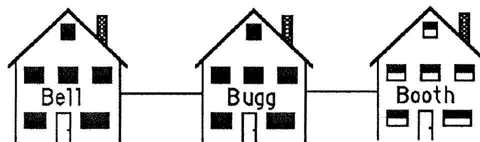


Figure 18: The three houses are connected with **non-directed paths** so that objects may travel between them in either direction.

draw two directed paths:

If an object can travel between two container objects in both directions, and it travels in the two directions under different kinds of conditions,
then create two directed paths connecting the two objects in both directions, and give them each a different label. (figure 19)

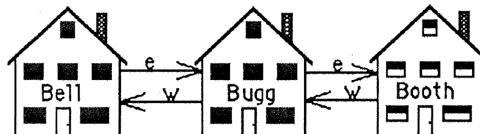


Figure 19: The three houses are connected with **two directed paths**, so that objects may travel east or west between the houses on different conditions.

label path:
If two or more paths are connected to the same object, and the paths are to be used for different purposes,
then create a label for each path with a different name. (figure 19, 20, 21, & 22)

label common paths:
If a set of paths in the display are all used for a common purpose,
then label each of the paths with the same name. (figure 20)

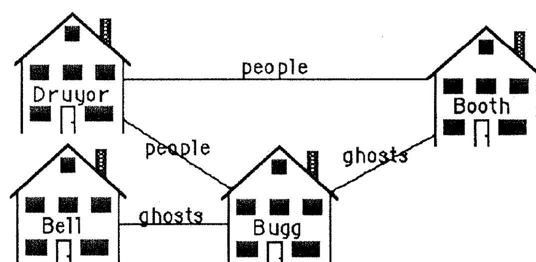


Figure 20: These paths are labeled based on what may pass between them. Rules may be created to only allow ghosts to move along "ghosts" paths, and people to move along "people" paths. This is an example of **labeling common paths.**

draw paths for representation:
If a set of objects have an underlying configuration or relationships between each other, and the simulation needs this representation,
then draw directed or non-directed paths describing the relationships, and label the paths if more than one kind of relationship is described. (figure 21)

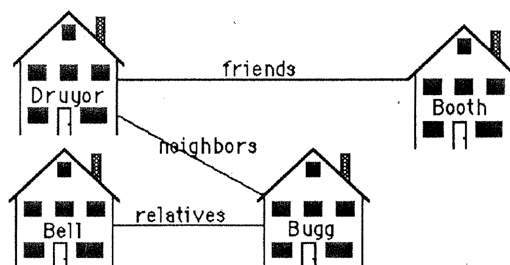


Figure 21: These houses are connected based on relationships between the families. This labeling is an example of **drawing paths for representation.**

draw paths for control:

If one part of the simulation picture is dependent on the state of a control panel, and there are several of these control panels that look alike,

then connect the main container object of the control panel to the parts of the picture that they affect. (figure 22)

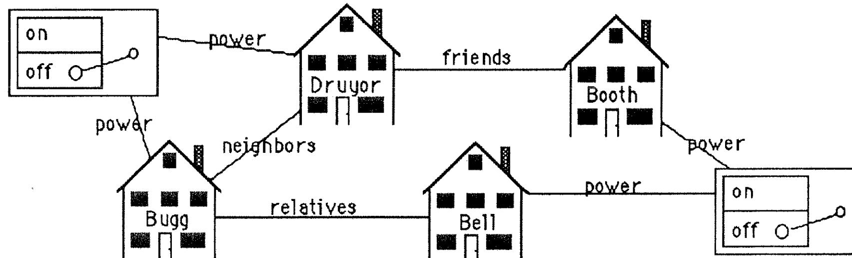


Figure 22: The labeled paths between the switches and the houses represent power lines. These paths are example of **paths that may be used for control** purposes.

draw path with a stopover:

If an object can travel between two container objects, and the simulation requires that objects traveling between these two containers must be processed in some way, such as being counted or being thrown away,

then create an intermediate container between the two containers, and create a path from the source to the intermediate container and from the intermediate container to the destination. (figure 23)

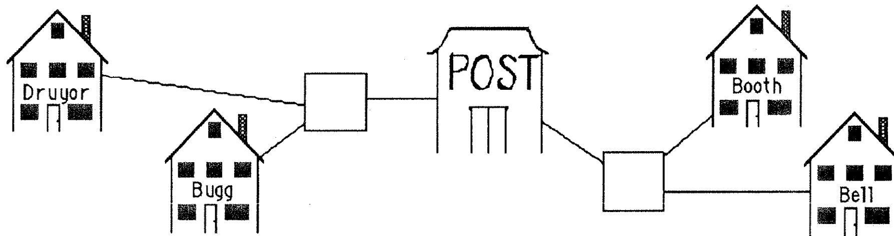


Figure 23: The rectangles between the houses and the post office represent intermediate stopovers in the connection.

3.4 Supporting End User Commands

This advice describes how to build simulations that enable user interaction.

draw control panel:

If the behavior of a simulation can be controlled by a switch,
then draw a set of container objects for each setting of the switch, label each of the container objects with a description of the switch setting, and create a container object that holds all of the switches. (figure 24)

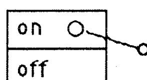


Figure 24: Control panel with two settings for controlling house lighting.

draw command list:

If the simulation should behave in different ways that can be controlled by the end user,
then create a label for each type of behavior, and place the labels in a unique container object. (figure 25)

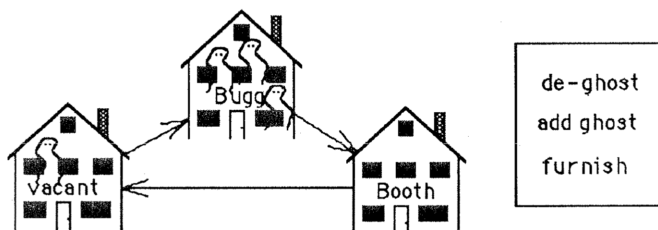


Figure 25: The rectangle and the three commands inside it are an example of a **command list**. These commands may be dragged into the houses by the end user.

create command rule:

If the end user can control the simulation by dragging an object (a command) into a specific container object in the display,
then create a rule whose pattern picture includes the object inside of the container and the other objects to be modified, and whose result picture shows the appropriate modifications to the objects. If the modification completes the command, also move the command back to its source container. (figure 26)

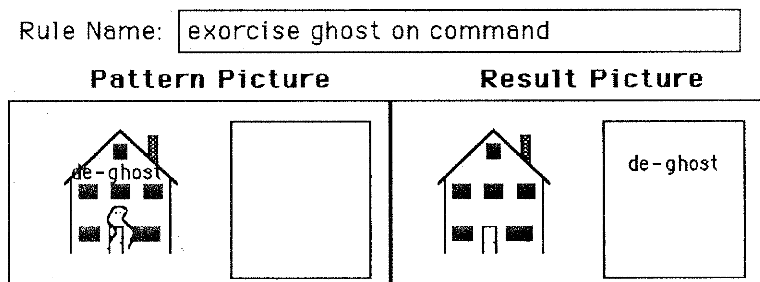


Figure 26: A **command rule** to remove one ghost from a house when the "de-ghost" command is placed in the house. The rule also puts "de-ghost" back in the command list.

3.5 Controlling the Execution of Rules

This advice describes how to have greater control over the way in which rules execute. The execution of rules can be manipulated by either changing the relative ordering and priorities of the rules.

reorder rules:

If one rule is executing, causing a more appropriate rule not to execute,
then in the "rule ordering" list place the name of the more appropriate rule above the rule that shouldn't be firing.

describe random rule behaviors:

If two or more different types of behaviors may happen during a simulation in identical circumstances, and can occur randomly with respect to each other,
then create a rule for each behavior, place the rules next to each other in order, parallelize them, and define what percent chance each has to fire. (figure 27)

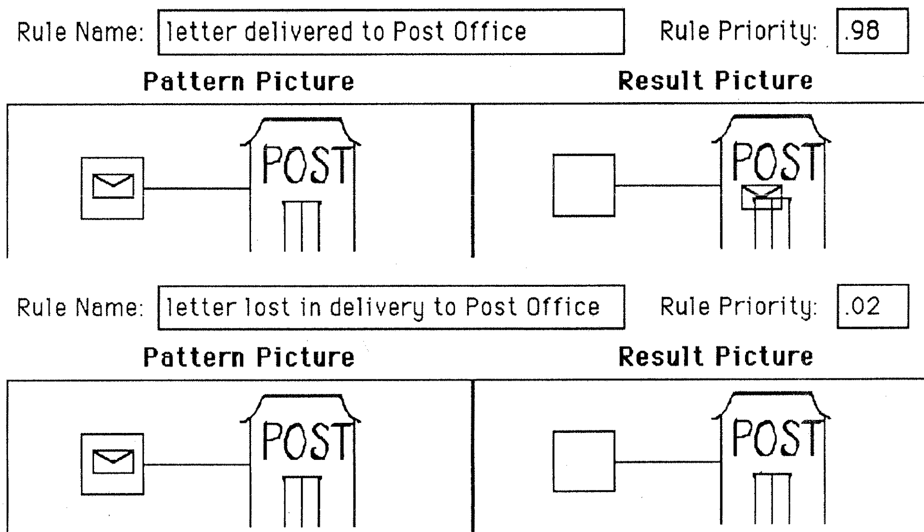


Figure 27: These two rules will randomly deliver 98% of the letters sent to the post office and will lose the remaining letters.

populate randomly:

If a set of containers must be filled with individual objects of different types,

then write individual rules to create each kind of object within a single container, place the rules next to each other in the rule ordering, parallelize them, and define the population percentage with the rule priority percentage attribute. (figure 28)

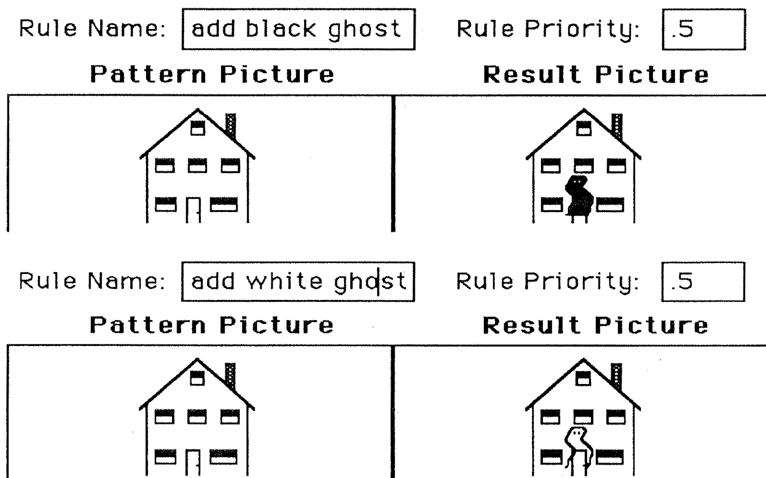


Figure 28: These two rules will populate each house with an individual ghost. Each house has a 50/50 chance of receiving a white or black ghost.

enable rule to continually fire:

If a rule is not firing on the same objects more than once, and the rule needs to fire continually,

then create on the display two identical containers and a label inside one of them, copy these objects onto the pattern and result of the rule, and move the label from one container to the other in the result, thus making a superficial change in the display, allowing the rule to fire again.

3.6 Creating Control Structures

If the description of the high-level behavior a simulation may easily be described as a sequence of steps or states, then by drawing a description of the control structure, the rules may use this structure to force behaviors to occur only in the written sequence.

draw control sequence structure:

If a sequence of tasks must be accomplished in order,
then follow these steps:

1. create a sequence of identical containers,
2. connect the sequence with directed paths,
3. label each container with the name of a task,
4. create a marker to denote the current task and place it inside the container of the starting task,
5. write a rule to move the marker from one container to the next container along the directed path, (place this rule last in rule priority) and
6. if the sequence is a loop, connect the end container to the start container with a directed path.
(figure 29)

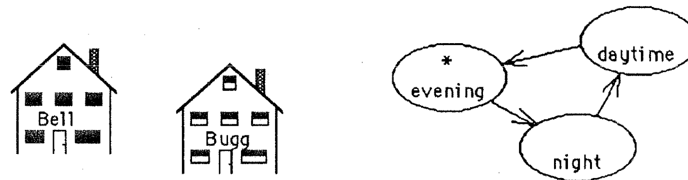


Figure 29: The three ovals with named labels represent a looped control sequence of tasks. The "*" represents the current task.

sequence structure condition:

If a control sequence structure has been built, and the behavior of each task needs to be described,
then create a rule that describes the behavior, and place the task name, its container, and the current task marker in the condition and result of the rule. (figure 30)

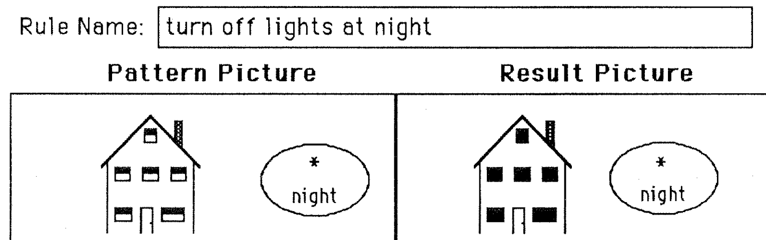


Figure 30: This rule forces a house to turn off its lights at night.

3.7 Creating Container Grids

This advice describes how to build a one or two dimensional grid of rectangular containers. A grid representation is useful when objects of the simulation may move around the screen, and these positions are regularly spaced and connected and too numerous to draw by hand.

draw two dimensional grid:

If objects can move around in a two dimensional space,

then create a grid of connected places, using the path creation advice to decide how the grid of places should be connected. (figure 31)
The "create grid" menu item provides an interface for describing grids.

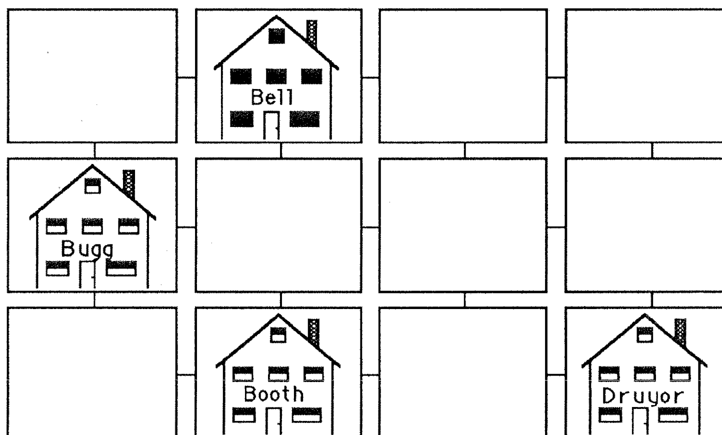


Figure 3 : This is a 3x4 grid of containers that represents lots that houses may be placed on. Adjacent squares are connected in this grid with non-directed paths.

fill grid:

If one kind of object is to be initially placed in all or most of the cells of an existing grid,

then create a rule that adds this object to a grid cell, execute the simulation, delete the rule, and delete any of these objects that are unnecessary.

draw ordered list:

If a list of items has to be processed in a specific order,

then follow these steps:

1. create an ordered list of the items, by using the grid interface (the "create grid" menu item) to draw a sequence of connected containers;
2. place the items in the grid containers in the appropriate order; and
3. create an object to mark the current position in the list. (figure 32)

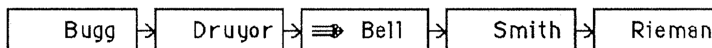


Figure 32: This is a five element sequence connected by directed paths to show the order of processing. A marker has been created and currently points to the "Bell" item.

3.8 Creating Attribute-Value Tables

This advice describes how to describe properties of existing objects. The properties may be described or within the objects themselves (the first advice) or within a drawn table (the second advice).

draw attribute-value pairs:

If there are a bunch of objects with a common set of attributes which have different values, and these objects are big enough to hold the attributes and values,

then draw a container in the object for each attribute-value pair. (figure 33)

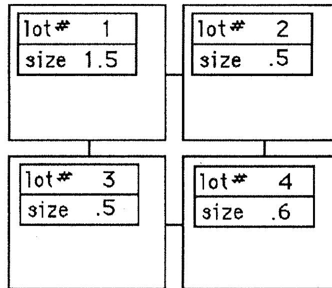


Figure 33: The four main containers represent adjacent house lots. Two properties are defined within each lot container, the lot# and the size of the lot in acres.

draw attribute-value table:

If there are a bunch of objects with a common set of attributes which have different values,

then draw an attribute-value table, by following these steps:

1. draw a row as a rectangle for each data entry,
2. draw a column as a rectangle for each attribute,
3. label the top of each column with an attribute name,
4. if each object is not labeled, give each object a unique label,
5. label each row with the label of each object, and
6. fill in the values of the table. (figure 37)

NAME	LOCATION	PAYS BILLS?
B. Bell	Boulder	no
G. Bell	Boston	yes
A. Bugg	Boulder	yes
I. Druyor	Prarie du Chien	yes

Figure 34: Each row is drawn as an identically sized rectangle. Each column is a rectangle labeled by an attribute name. Each row describes one data entry. The name may be associated with other objects in the simulation (e.g. houses.)

table driven condition:

If the behavior of an object within a rule is described within an attribute-value table, and the object depends on settings made in the table,

then copy into the pattern and result of the rule: the row and column containers of the table, the label of the appropriate column, the label of the object, and the appropriate value of the attribute. The row and column should overlap, the column should contain the column label, the row should contain object label, and the value should be placed at the intersection of the row and column. (Figure 35)

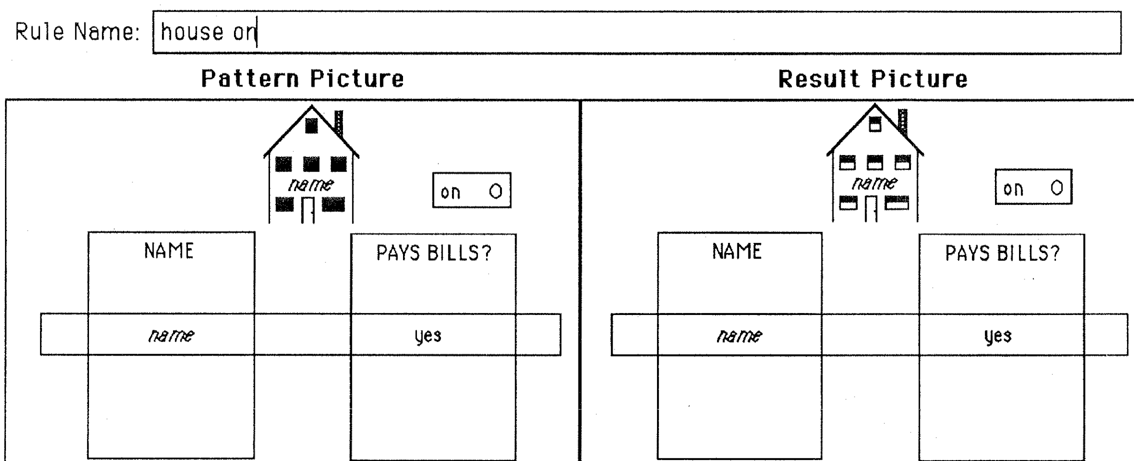


Figure 35: The pattern matches when a button is in the "on" position, there is an unlit house, there is an entry of the table that has the house name, and that entry shows that the person is paying the bills.

draw pairing structure:

If there is a one to one mapping between one set of objects and another,

then create an attribute-value table with two columns that describes the mapping. An example use of pairing structure would be to describe opposites.

3.9 Defining Sets

This advice describes how to constrain the rules to act only on objects that are defined within a set.

draw set definition:

If a specific behavior that may occur is restricted to a set of objects,
then create a unique container object in an unused part of the display, and place a copy of each of the possible objects in the container. (figure 36)

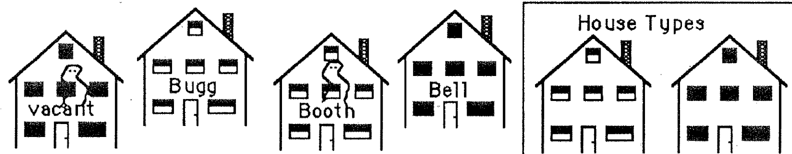


Figure 36: The rectangle and its contents is an example of a set definition. The rectangle contains both kinds of houses, defining the set of houses.

set condition:

If one of the objects in the pattern of the current rule is restricted to be one of a set of objects,
then specify that this object is a variable, define the set of objects, place the set container in the pattern and result of the rule, and place a copy of the variable object in the set container in the pattern and result of the rule. (figure 37)

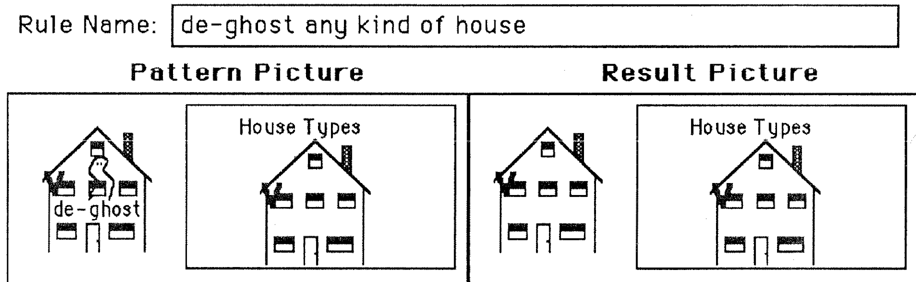


Figure 37: This rule removes a ghost from any kind of house, as long as that kind of house is defined in the "House Types" rectangle. The lit houses in this rule are all variablized, allowing them to match either type of house.

3.10 Counting Events

This advice describes how to count object populations or event occurrences.

count occurrences:

If a count is needed of particular events happening in the simulation,
then follow these steps to have the event counted:

1. import the rules and the display of the simulation called "counter,"
2. test the ability to increment, decrement, or clear the counter, by copying the commands into the counter box,
3. resize and label the count container to a size appropriate for the current simulation, and
4. create a counting rule, whose pattern picture displays the event's conditions, the counter and its containing box, and whose result picture displays the same as the pattern with the addition of an "incr" label placed in the counter box. (figure 38) The "decr" and "clear" labels may be used in the same way to decrement or clear.

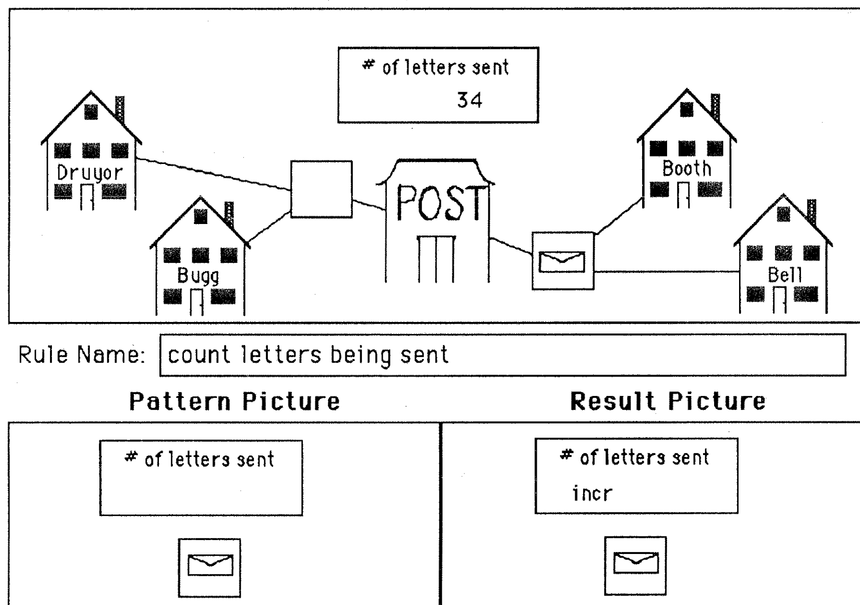


Figure 38: The simulation display shows a container that displays the number of letters that have been sent. The rule counts letters by placing the "incr" text string in this container, whenever a letter appears in the box connecting to the post office.

count multiple occurrences:

If multiple events have to be counted,
then create a unique label within each counter container, and use this unique label in the pattern of the counting rules.

use bar graph counter:

If a simulation needs to show a count as a bar graph,
then import the rules and display of the bar graph counter simulation.

4. The Programming Environment

This section describes some of details and idiosyncrasies of the ChemTrains programming environment. Figure 39 shows the full ChemTrains environment running the "houses-switch-box" simulation. The "house off" rule is displayed, and a trace of the rule execution is shown. This section describes the usage of the commands on the right hand side of this figure.

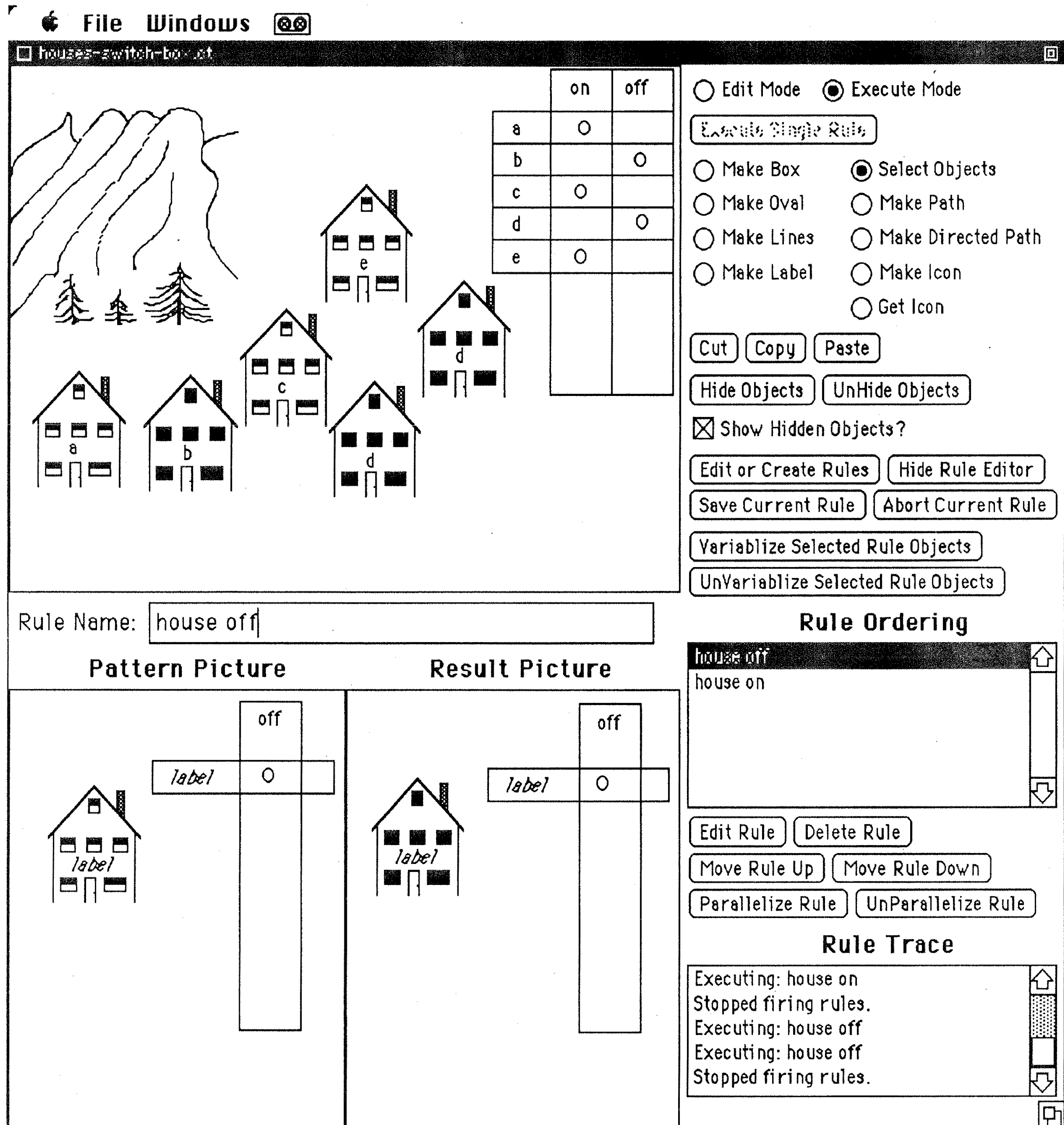


Figure 39: The ChemTrains Programming Environment.

4.1 Menu Operations

The "File" menu contains the following operations:

- "New Simulation" creates an empty ChemTrains window.
- "Open Simulation ..." opens up a chosen simulation in a new ChemTrains window.
- "Open Simulation Standalone ..." opens up a chosen simulation, but does not show or enable the simulation creation actions, so that the simulation can be viewed solely as an end user may see it.
- "Close Simulation" closes the current ChemTrains window.
- "Save Simulation" saves the current ChemTrains window to its appropriate file.
- "Save Simulation as ..." saves the current ChemTrains window to a new filename.
- "Import Simulation ..." reads the rules and display of a chosen simulation, appends the rules to the rules of the current simulation, and outputs the display to a selected location of the current simulation. Importing simulations is useful when a general-purpose simulation exists to accomplish a subtask needed in the current simulation. For example, existing simulations for doing bar graph display and doing numeric counting are generally useful and can be imported.
- "Create Grid" displays an interface for creating a grid of places on the simulation. The grid creation interface is described in detail in section 3.4.10.
- "Edit Grid ..." displays the grid interface with the grid information from a previously edited grid.
- "ChemTrains Help" displays a text file containing a little help.
- "Quit" exits out of ChemTrains.

Existing ChemTrains windows may be selected from the "Window" menu.

4.2 Execute and Edit Mode

"Execute Mode" allows ChemTrains to execute rules whenever anything changes on the screen. When no rules apply, the rule interpreter stops. The execution of rules may be interrupted by the user in one of two ways:

1. by typing Command-period (the Macintosh standard way of aborting the execution of a program), which halts rule execution immediately, or
2. by selecting and moving objects in the simulation, which halts rule execution after the current cycle of pattern matching and executing a single rule. The simulation continues with rule execution after the user action has been made).

The rule interpreter may be restarted, after being stopped, in one of two ways:

1. by reselecting "Execute Mode," or
2. by changing the simulation in some way, moving, creating, or deleting an object on the screen.

"Edit Mode" disables the rules from executing. This mode is useful in creating and editing the simulation before debugging it. Rule execution may be single stepped in this mode with the "Execute Single Rule" action.

4.3 Drawing the Simulation

Boxes, ovals, polylines, text strings, and icons can be drawn on the display screen. New icons can be created by selecting "Make Icon." Old icons can be retrieved by selecting "Get Icon." Once an icon is created or retrieved, the icon can be placed on the simulation by clicking the desired location in the main display. Likewise, a box can be created by selecting "Make Box" and clicking the box's corner positions in the main display.

Whenever objects are expected to exhibit an identical behavior on the screen, it is best to make their displays identical, so that multiple rules are not needed to describe a common behavior. The easiest way to create identical objects is to create one object and copy it.

4.4 Selecting, Copying, and Moving Objects

Objects may be selected by clicking on them. When an object is selected it is highlighted by eight small surrounding squares. Multiple objects may be selected by dragging a rectangle around the objects, or by clicking on the objects with the shift key held down. (Mac standard)

Once an object or set of objects is selected it may be moved by pulling one of the objects to the desired location. The selected objects may be copied by holding down the option key, (also Mac standard) and dragging the objects to a new location. In this case the original objects stay in the original location and the copied objects are moved. Objects may be moved or copied between different windows of the ChemTrains interface.

The selected objects may be moved by one pixel in any direction with the arrow keys (also Mac standard.)

Another way to copy objects is using the cut and paste commands. Selected objects may be cut or copied using the "cut" and "copy" commands or the command-x (cut) and command-c (copy). To paste objects click the desired location, and select the "paste" command or command-v.

4.5 Hiding and Unhiding Objects

Objects may be hidden by selecting them and executing "Hide objects." Objects are only really hidden in the display if the "Show Hidden Objects?" toggle is off. So if the "Show Hidden Objects?" toggle is on, the "Hide Objects" command will appear to do nothing.

4.6 Creating a Rule

The command "Edit or Create Rules" opens up the rule editor and permits new rules to be created and old rules to be viewed and edited. When creating a rule, there are three parts that must be specified: the rule name, the pattern picture, and the result picture. Rules are given names so that they can be referred to easily in a list. The easiest way to create a rule is to copy items from the simulation to the pattern picture and the result picture, and then making appropriate changes (additions, deletions, or movements) to the result picture. Graphic objects may be dragged between the main display, the pattern picture, and the result picture. A newly created rule or an edited old rule may be saved with the "Save Current Rule" command.

To decide what graphic objects should be placed in the pattern picture, select the objects needed to describe the condition and the objects that may be modified when the rule executes. Copy these objects from the simulation display to the pattern. For example, when writing a rule to produce steam in a beaker when a high flame exists, the pattern picture needs: the high flame because that determines the state of the substance in the beaker, the beaker because that is where the phase change will occur, and the previous substance in the beaker because that has to be replaced by steam. In this example, the programmer would copy these objects also to the result picture, and replace the beaker's substance with steam. This rule should appropriately execute when a high flame exists, and replace the substance in the beaker with steam.

4.7 Variables

Objects in the pattern or result of a rule may be variablized by using the "Variablize Selected Rule Objects" command. A variablized object is displayed in italics if it is a text string, or else it is displayed with an overlapping big "V." A good way to incorporate variables into a rule is to draw the rule first without variables, debug it as a specific case, and then after the rule is working for this case, generalize it by variablizing objects that may match any object.

4.8 Editing an existing Rule

An old rule may be edited either by double clicking on a rule listed in the "Rule Ordering" or "Rule Trace" display, or by using the "Edit Rule" command. An edited rule can be modified and resaved. If the name of a rule is changed, the interface asks whether a new rule should be created leaving the old rule alone, or the rule should replace the old rule. A new rule that is similar to an existing rule can be created by editing the existing rule and changing its name.

The "Hide Rule Editor" command will close the bottom half of the programming environment which shows the rule editor and its associated commands.

4.9 Rule Ordering and Selection

The "Rule Ordering" display shows the rules in their order of priority. The rules may be ordered with the "Move Rule Up" and "Move Rule Down" commands. A set of rules that are next to each other in ordering may be "parallelized" with the "parallelize rules" and "unparallelize rules" command. When a set of rule is parallelized, that means that the rule interpreter will select randomly among them, if they are applicable. The chance of selection between them may be set by the "Rule Priority" attribute of the rule editor.

4.10 Creating and Editing Grids

Sometimes it is necessary to create either a one or two dimensional grid of places, such as a long row of connected things or a field of objects that move around. Figure 40 shows the interface as it is used to create the 64 squares of a checkers board. The interface permits the dimensions and the exact positions of the grid to be specified, either by filling out the numbers under "grid dimensions" or by moving and stretching the existing cells of the grid. Paths connecting the cells either vertically or horizontally may be specified. The "[Re]Display Grid" action redisplays the grid if there are any changes. The "Done" action exits from this interface.

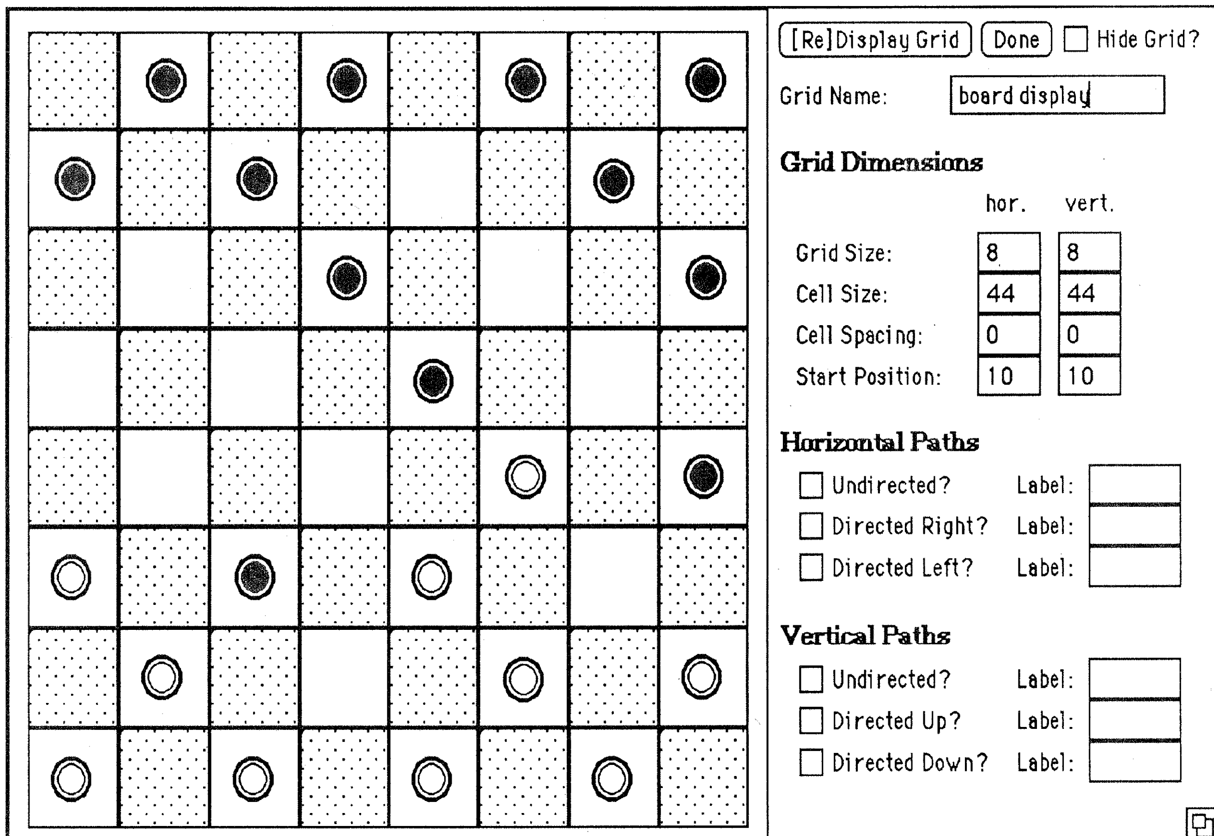


Figure 40: Environment for editing a single grid.

5. Example Problems and Solutions

The suite of target problems were chosen to include a range of qualitative problems. Small problems are included to test very specific kinds of computation. Larger problems are included to test scalability of the language. This section illustrates the use of ChemTrains in solving four simulation problems: a mouse travelling in a maze, a turing machine, a tic-tac-toe player, and a simulation of grasshoppers.

5.1 Maze Search

The maze search problem demonstrates the ability to simulate things moving around on the screen in an organized behavior. Here is the statement of the problem:

> Show a simple maze, with a mouse at its entrance and cheese at some distant point. The mouse should move through the maze leaving a string behind it as it moves. If it reaches a dead end, it should backtrack and try a different route. The mouse should never look in the same place more than once, except to backtrack. When the mouse sees the cheese it should stop and eat it.

Figure 41 shows a mouse in the process of searching a maze for cheese. The mouse had started in the lower right hand corner.

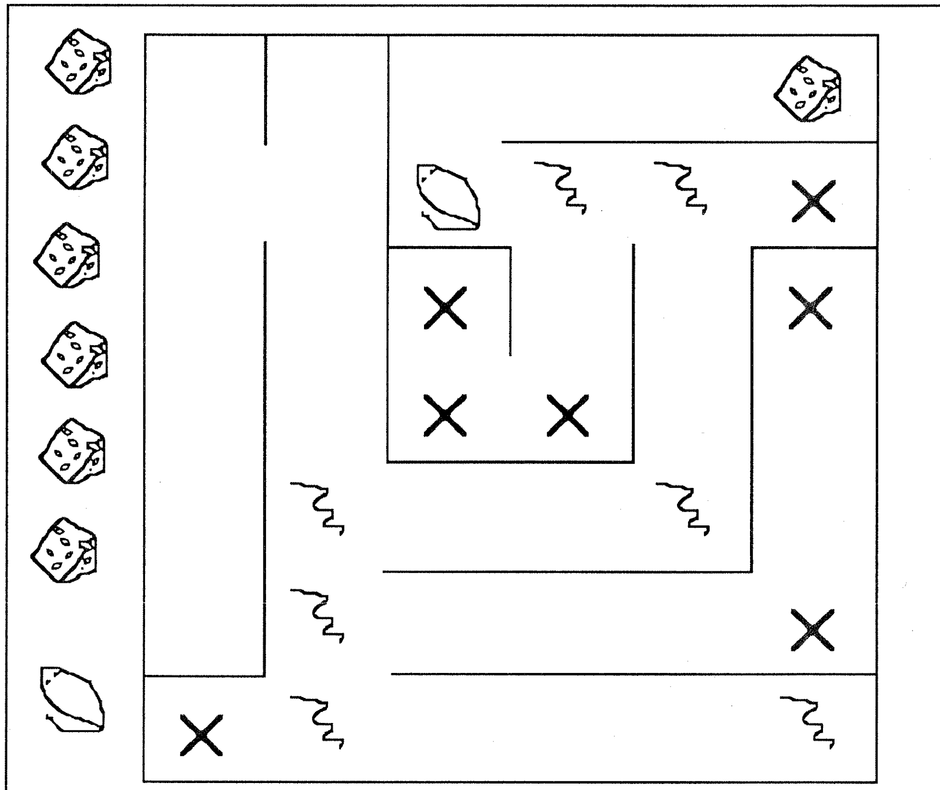


Figure 41: Display of Maze Simulation during execution.

The solution to the maze search problem simulation shown in figure 41 requires that hidden rectangles exist for every corner, dead end, and decision point in the maze. These rectangles are appropriately connected with hidden paths. Figure 42 shows the maze simulation when the mouse has completed the search, and shows the hidden places and connections in the maze.

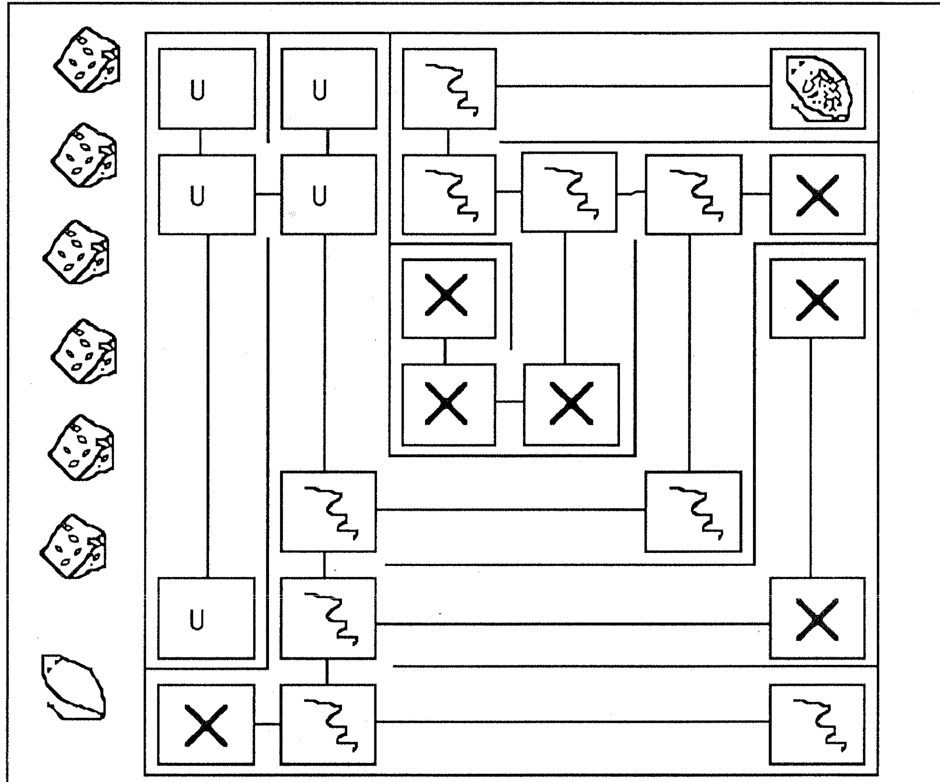


Figure 42: Display of Maze with hidden places and paths.

The three rules required in the maze search problem are shown in figure 43. One rule allows the mouse to consume the cheese if they're in the same place. The second rule allows the mouse to go to a connected unseen place leaving string behind it. And the third rule allows the mouse to reel in a piece of string, and leave an "X" marker in the previous place. The mouse will always pursue unseen places before backing up because the second rule takes precedence over the third rule.

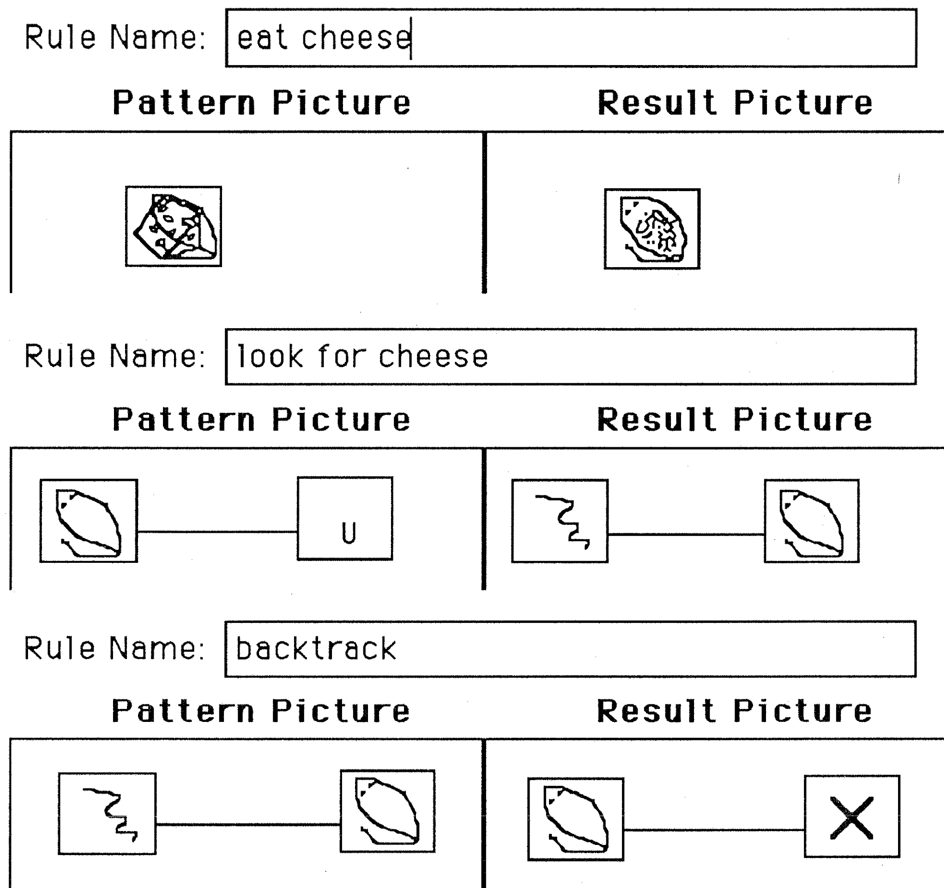


Figure 43: Maze Searching Rules.

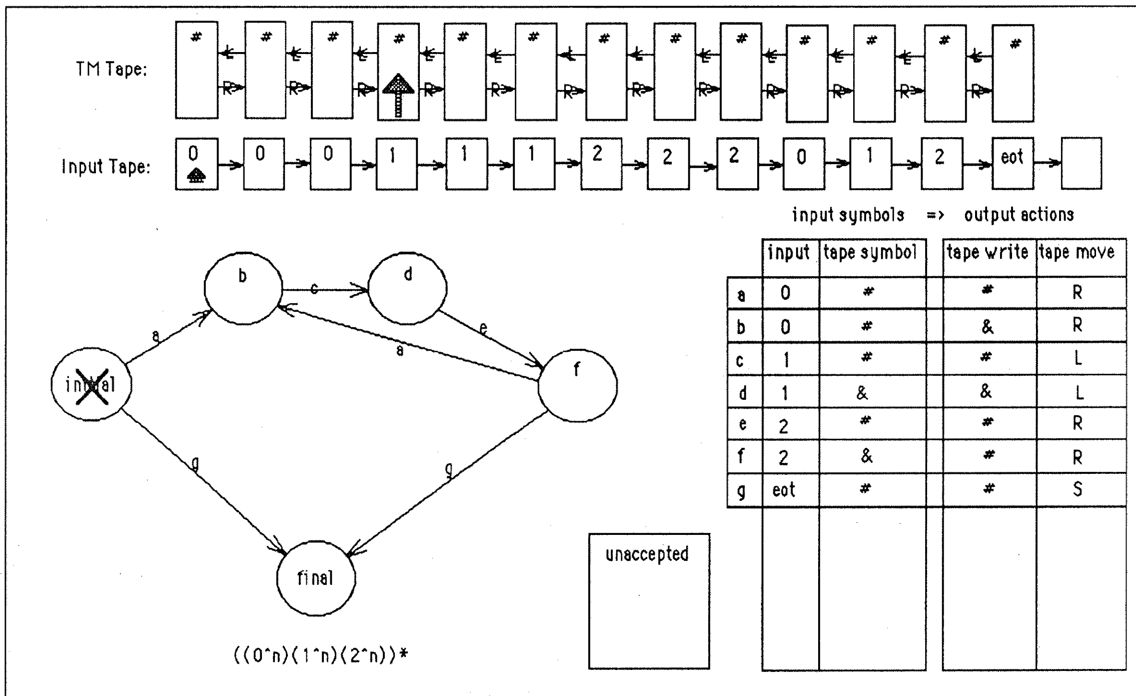
5.2 A Turing Machine Editor

The Turing Machine problem is added to show that the language is Turing equivalent. The problem involves not just simulating a particular Turing program but simulating an editor of Turing machine programs. Here is a statement of the problem:

> Show a Turing Machine tape containing blanks, and an input tape containing 3 0's followed by 3 1's followed by 3 2's. Simulate the Turing Machine program that can recognize any input string that is a sentence in the language $0^n1^n2^n$.
>
> In addition to displaying the Turing tape and the input tape, display the finite state machine and a table defining each arc (the input symbol & tape symbol needed, the tape symbol to write, and the direction to move the tape head.) The solution should allow the end user to draw and simulate any Turing Machine program.

A possible solution to the general Turing Machine shown in figure 44. The Turing tape and the input tape are displayed as places connected with directed paths. The meanings of the finite state machine arcs are specified by labels which are defined by the table to the right of the FSM. The table defines

under what conditions an arc may be traversed, and defines what actions should be taken when it is traversed.



	input	tape symbol	tape write	tape move
a	0	#	#	R
b	0	#	&	R
c	1	#	#	L
d	1	&	&	L
e	2	#	#	R
f	2	&	#	R
g	eot	#	#	S

Figure 44: General Turing Machine Simulation.

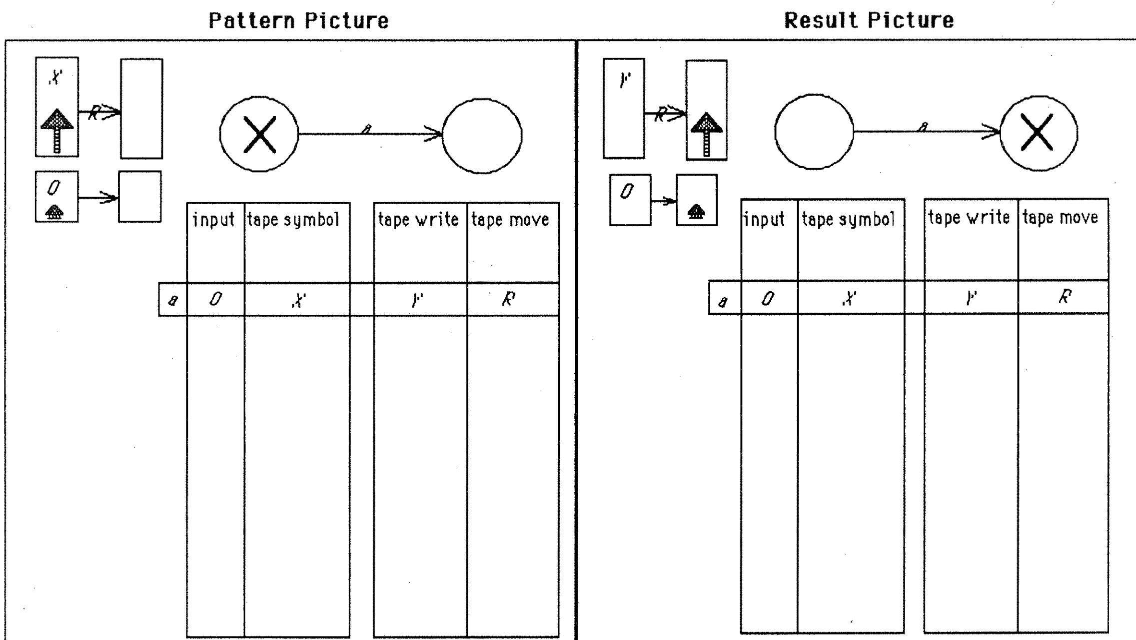


Figure 45: Turing Machine Rule to traverse one transition that matches a table row, one piece of Turing tape, and one input symbol.

For example from the initial state, arc "a" may be traversed if there is a "0" at the current location in the input tape and a "#" at the current location in the Turing Machine tape, and in traversing arc "a" a "#" will be written to the current position of the TM read head and the TM read head will move to the right, along a path on the TM tape labeled by an "R." The simulation is governed by five ChemTrains rules: four rules for two different parameters: whether an arc connects states or stays in the same state (e.g. arc "b") , and whether an arc moves the TM read head right or left or keeps the read head in a static position; and one rule to recognize when the input should be "unaccepted." Figure 45 shows the rule to traverse an arc that goes from one state to another and to traverse a cell of the turing machine, when the head of the Turing tape and the head of the input tape appropriately match the traversed arc. Note that many of items in the rule are variable: tape symbol, the input symbol, the arc label, and all of the items in the table row.

5.3 Tic Tac Toe

The tic-tac-toe playing problem is shown becuase the simulation requires interaction with the end user and requires relatively complex computations to produce good play. Here is a statement of the problem:

> On the screen, show a Tic Tac Toe grid. Allow the user to start a new game at any time, and to 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" on the board. The machine should be able to play for a win, block a win, play the center, or play a random place on the board if there are no other alternatives. The machine should detect when either player has won or there is a draw. Assume the user is honest - don't worry about preventing illegal moves.

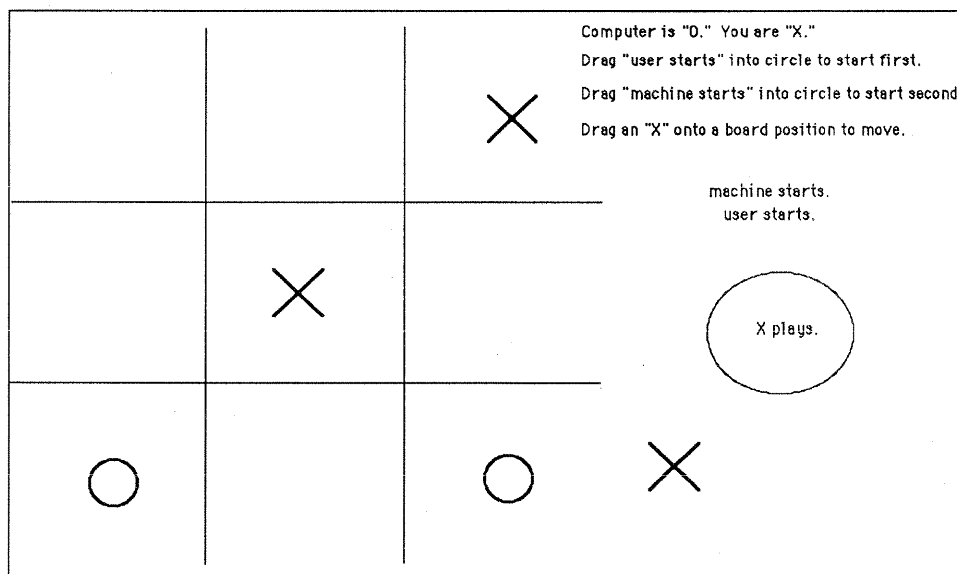


Figure 46: The Tic Tac Toe Simulation.

Figure 46 shows the desired end user interface needed in tic tac toe. However, to get this interface to work correctly in ChemTrains other objects are needed. Figure 47 shows one possible representation that enables a ChemTrains solution. A cell is needed for each position on the board, so that actions can take place in them individually. The cells are labeled with a unique name for each row, column, and diagonal. The rule shown in figure 48 takes advantage of the cell labeling to match any row, column, or diagonal containing two O's and a blank when it is O's turn. The rule shown in figure 49 changes the turn from being X's turn to O's turn and takes out a blank, when X has made a move. Other rules are needed for a clearing a board when a game is restarted, and for playing different kinds of moves.

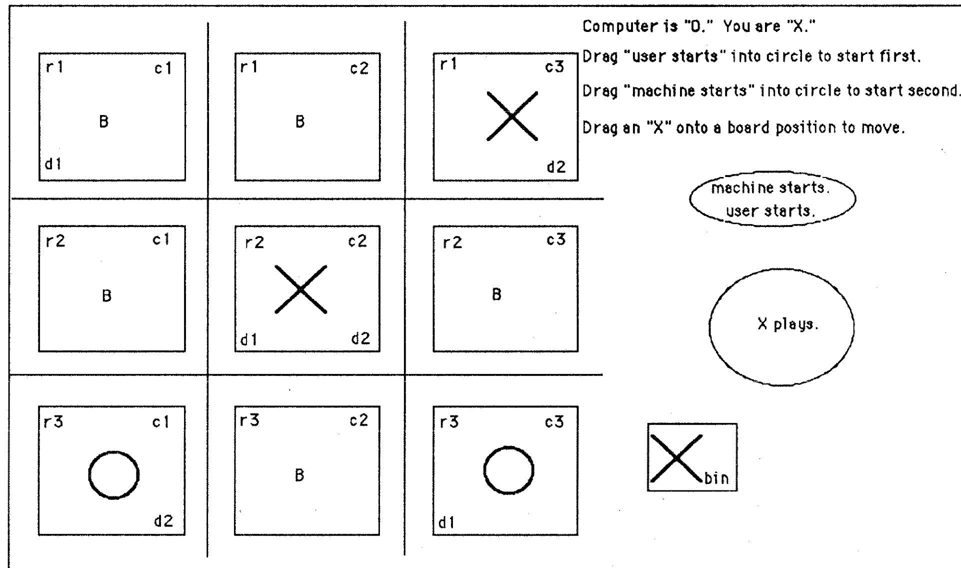


Figure 47: The Tic Tac Toe Simulation with hidden objects shown.

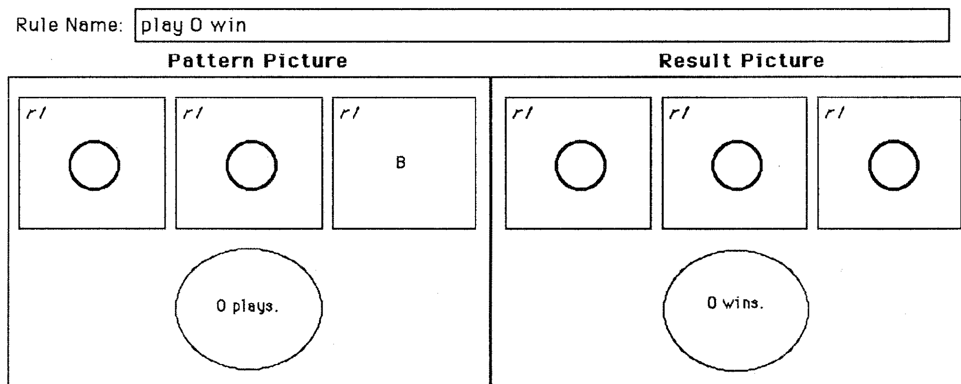


Figure 48: Tic Tac Toe Rule to play a win.

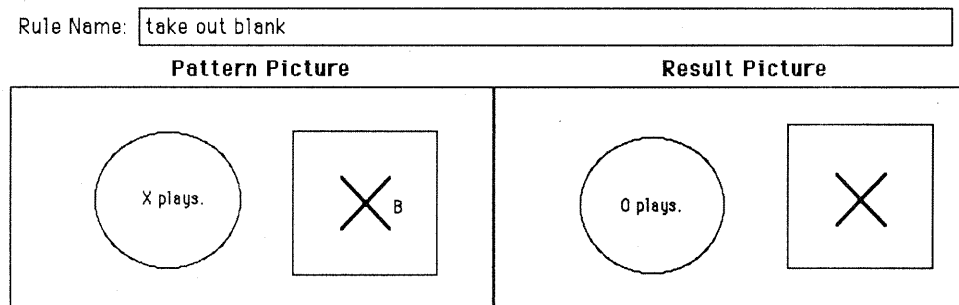


Figure 49: Tic Tac Toe rule to switch turn mode.

5.4 Grasshopper Lifecycle Simulation

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. 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.

Figure 50 shows the simulation running in the summer when all the grasshoppers are feeding. The display shows a population of juvenile grasshoppers (the small ones), adult grasshoppers (the bigger ones), a single adult that may lay eggs, and three types of grass, grass for juveniles (thin blades), grass for adults (thick blades), and poisonous grass (thickest blades). The simulation display also shows a clock to keep track of seasons, and a count of the yearly population.

Figure 51 shows the simulation with the underlying representation pictures. The hidden objects include:

- a grid of connected cells that represent locations of a field,
- pieces of food that the grasshoppers have eaten (shown inside the grasshoppers),
- empty place holders for cells that aren't occupied, and
- a sequence of rectangles to hold the yearly population figures.

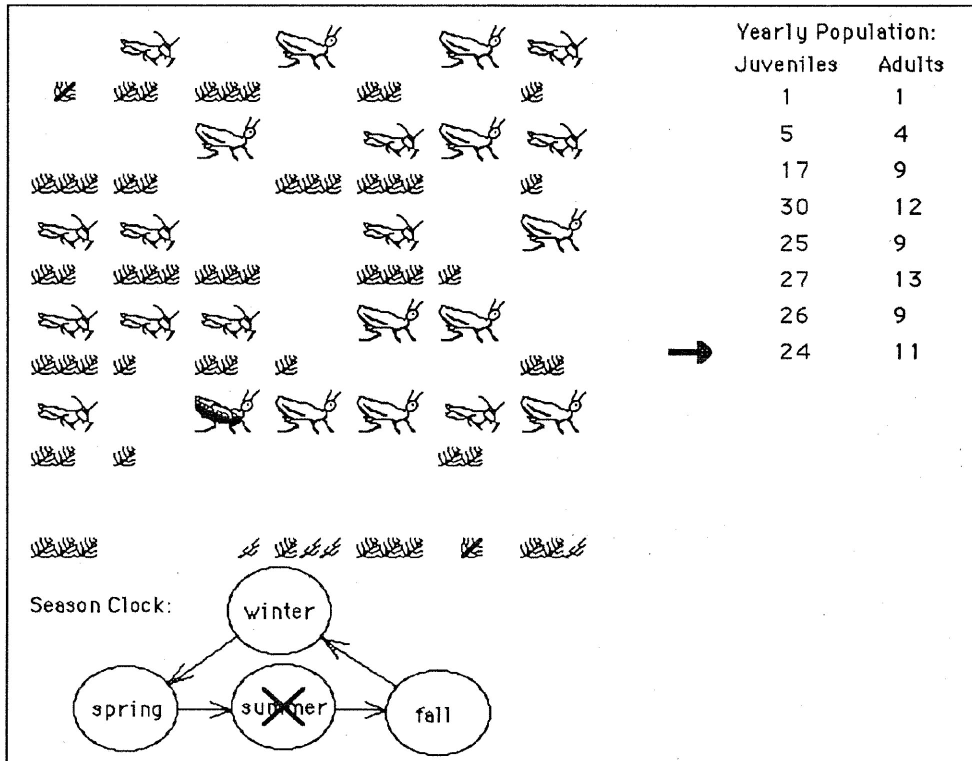


Figure 50: Grasshopper Simulation.

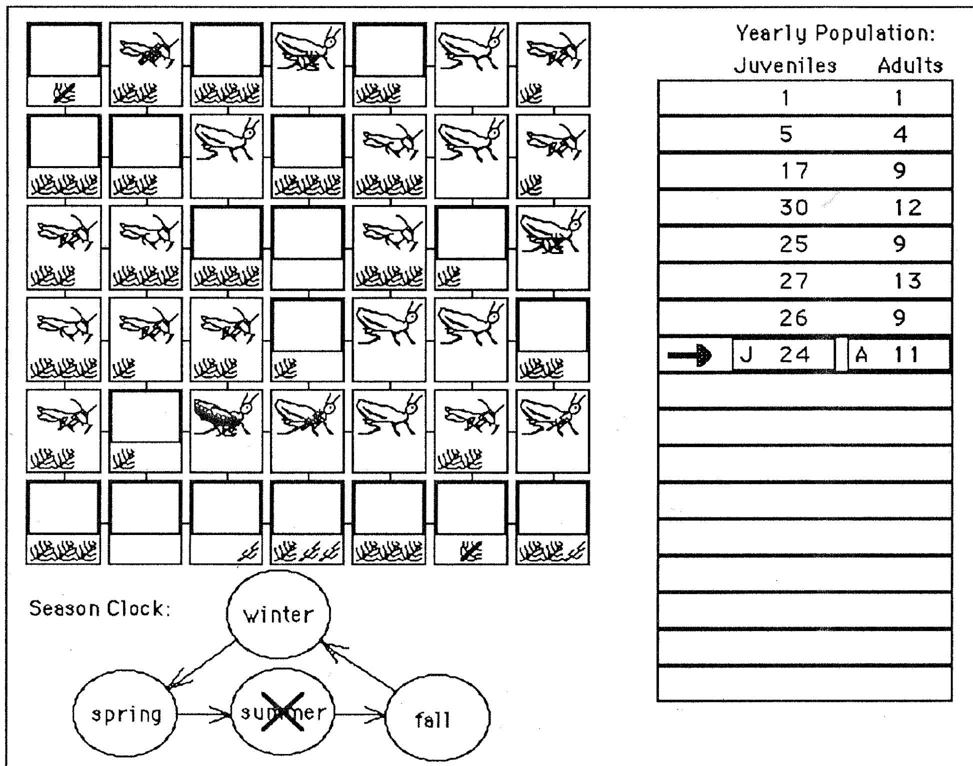


Figure 51: Grasshopper Simulation (hidden objects shown).

This solution uses a simplified version of the grasshopper target problem. Each grasshopper activity occurs only at specific times of the year. The simulation uses a cyclic state diagram to represent the changing seasons.

- In the spring, the three kinds of grass grows, and the eggs hatch into juvenile grasshoppers. Hatching eggs compete for vacant cells. Any egg that can't find a nearby cell will die.
- In the summer, juvenile grasshoppers jump around, eat, and compete for their food. Juveniles with enough food will be promoted into adults, otherwise they will die.
- Also in the summer, adult grasshoppers jump around, eat, and compete for their food. Adults with enough food will be promoted into egg bearing adults, otherwise they will die.
- In the fall, egg bearing adults lay eggs and die.
- In the winter, all remaining grass dies.

There are 31 rules for describing the behavior: eight for spring, sixteen for summer, one for fall, three for winter, one for changing the seasons, one for changing the year marker, and one for halting the simulation when the sequence of years is complete. Here are descriptions of some representative rules.

Figure 52&53 show two of the eight rules that govern an adult's behavior during the summer: eating food and jumping toward food. When a grasshopper eats a piece of food, it is moved inside the body of the grasshopper and is hidden. The grasshopper needs to carry around all of the food it has eaten, in order to represent its nourishment. The move rule places an "M" marker on the grasshopper and on the newly vacant cell. These markers are used by two other rules to carry over the adult's stomach food. This behavior cannot be described with one rule because a grasshopper may hold an arbitrary number of pieces of food. Other rules needed in the summer include: grasshopper dies from poison, grasshopper dies from undernourishment, grasshopper fulfills nourishment, and grasshopper moves two cells to food. Eight more rules are needed to describe the juvenile grasshopper's behavior during summer.

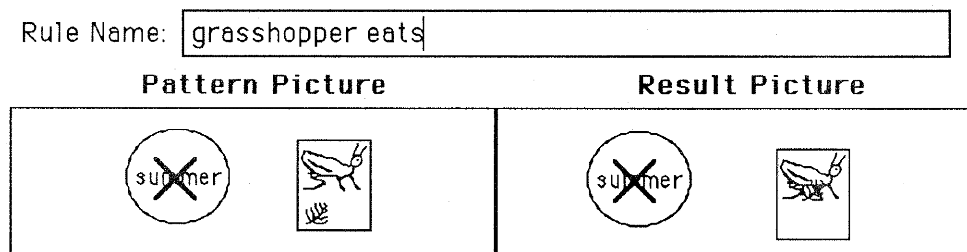


Figure 52: Rule for adult to eat piece of grass in the summer.

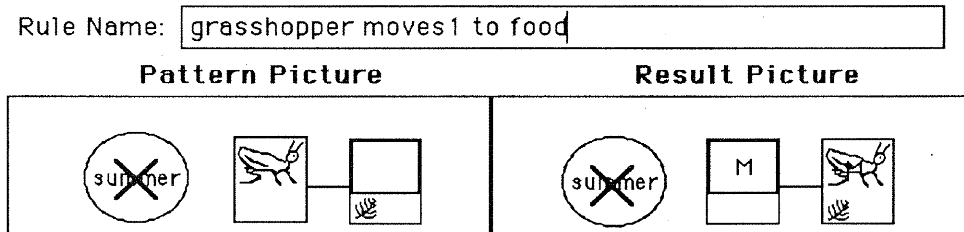


Figure 53: Rule for adult to jump toward grass in the summer.

Figure 54 shows the rule for hatching a single egg in the spring time. This rule also increments the counter in the currently active "J" counter. This rule takes advantage of a feature of ChemTrains that didn't exist for the earlier counter problems. The new feature is a set of built-in rules that recognize "incr," "decr," and "clear" as commands that can change a numeric label. The command is applied to the numeric label if they are both inside the same container. When the command is executed the command label is removed, similarly to the previous counter solutions. Instead of multiple objects to represent a single number, only one label is needed.

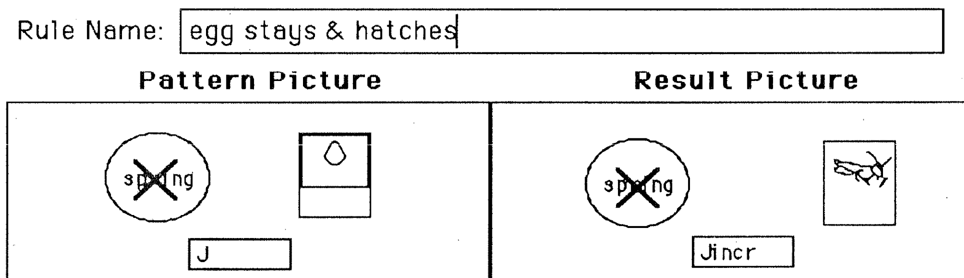


Figure 54: Rule for egg to hatch in the spring.

Figure 55 shows the rule for changing from any season to the next season. This rule is placed last in the rule priority, so that every rule that may apply in any season has a chance to fire. If no more rules apply in a season, this rule will fire, moving the current season marker to the next season.

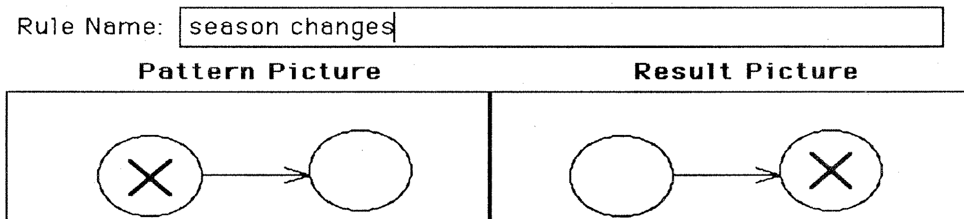


Figure 55: Rule to change seasons.

6. References

For further information about ChemTrains or the design rationale behind the language refer to the following publications:

Bell, B., Using Programming Walkthroughs to Design a Visual Language. Ph.D. Dissertation, Technical Report CU-CS-581-92, University of Colorado, January 1992.

Bell, B. & Citrin, W., Specifying Network Communication Protocols with a Graphical Transformation Rules. Submitted to the International Workshop on Advanced Visual Interfaces, February 1992.

Bell, B., Citrin, W., Lewis, C., Rieman, R., Weaver, R., Wilde, N., & Zorn, B. The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages. Technical Report CU-CS-577-91, University of Colorado, January 1992.

Bell, B., Rieman, J., & Lewis, C., Usability Testing of a Graphical Programming System: Things we missed in a programming walkthrough. *Proceedings of CHI'91*, 7-13, October 1990.

Lewis, C., Rieman, J., & Bell, B. Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System. *Human-Computer Interaction 6*, 319-355, July 1990.