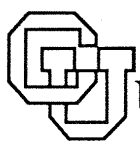
Using Programming Walkthroughs to Design a Visual Language

Brigham Bell

Ph.D. Thesis

CU-CS-581-92 February 1992



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Using Programming Walkthroughs to Design a Visual Language

Brigham Bell

Ph.D. Thesis
CU-CS-581-92 February 1992

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

A thesis submitted to the Faculty of the Graduate School in partial fulfillment of the requirement for the degree of Doctor of Philosophy and prepared under the direction of Professor Clayton Lewis.

Abstract

This thesis describes a search for an expressive yet facile visual programming language for creating and executing graphical simulations. The search for such a language is guided by a set of twenty problems, which include simulating a Turing machine, mice in a maze, grasshoppers in a field, and computer network communication protocols. The programming language, called "ChemTrains," is a rule-based language in which both the condition and action of each rule are described as pictures. A ChemTrains rule will execute when the topology rather than the geometry of a pattern matches a portion of the simulation picture, enabling rules to be drawn at a fairly high level of abstraction. Unique and interesting design alternatives arise because of both the rule-based paradigm and the visual programming aspect.

This thesis also describes a methodology for designing programming languages that are easy to write. The methodology, called "problem-centered design with walkthrough feedback," involves continually re-evaluating the writability of different iterations of the language's design by examining the steps required in creating sample programs and weighing the knowledge required to choose among alternative steps. The walkthrough evaluation is used to compare competing language designs and to uncover a design's shortcomings. In applying the methodology to the design of ChemTrains, the process of doing walkthrough evaluations on the twenty problems uncovered over a hundred design alternatives, and the comparisons between walkthroughs of different designs led the search through the design space.

User testing on a prototype of the final design showed that people with no computer science background and with minimal training could effectively write ChemTrains programs. These results confirm that the walkthrough evaluation analysis was on the mark. However, a few unexpected results in user testing showed that some difficulties in writing programs are hard to predict.

Acknowledgements

I gratefully thank:

Allan Bugg, George Furnas, Rich Gold, Mark Gross, Roland Hubscher, David Maulsby, and Alex Repenning for helpful discussions on this work;

the walkthrough analysis theorists, Wayne Citrin, Clayton Lewis, Peter Polson, John Rieman, Bob Weaver, Cathleen Wharton, Nick Wilde, and Ben Zorn, for discussions about programming and cognitive walkthroughs;

my advisor, Clayton Lewis, for teaching me how to do research and guiding me at crucial points;

the rest of my thesis committee, Wayne Citrin, Michael Main, Peter Polson, Bob Terwilliger, and Ben Zorn, for also helping to guide this research;

anonymous subjects for their time;

Gordon Bell, Gwen Bell, Pamela Bell, Steve Levi, Clayton Lewis, John Rieman, and Nick Wilde for reading and commenting on earlier drafts;

my parents, Gordon and Gwen, my sister, Laura, my wife, Pamela, and her parents, Allan and Valerie Bugg, for much love and support;

the National Science Foundation for funding some of this work; and

Gordon Bell for the hardware.

Table of Contents

1 Introduction

> "To the uninitiated, computers appear to be complicated and boring. As usual, the uninitiated are right. Computers *are* complicated and boring, and nothing here will even come close to making them understandable and interesting, unless you are one of those wimpy types who carry mechanical pencils and do the puzzles in *Scientific American*."
>
> - Dave Barry

This thesis explores the design of a programming language intended for people who have at least some programming experience and need to build automated graphical simulations. These users should not need any advanced programming experience or training. The language, called ChemTrains91, provides a simulation environment that has been designed to be easy to learn and use, while still being powerful enough to enable relatively simple solutions to complicated problems. The language is domain independent and allows programmers to draw and create animated simulations that look similar to the domain that they are working in.

This thesis describes not only the programming language for building simulations, but also the process used for designing the language. This chapter first introduces why ChemTrains91 was designed and built, then introduces the design methodology and why it was used, and finally gives an outline of the rest of the thesis.

1.1 Graphical Simulation in ChemTrains

A *simulation environment* is an automated workbench that provides support for programming the behavior of some physical or abstract phenomenon. A scientist or engineer would use a simulation environment to study, demonstrate, or teach a phenomenon that is hard to show physically because it is either abstract, not visible to the human eye, too slow, or too messy. A simulation environment provides representational tools for describing the components of a phenomenon and provides language support for describing the behavior of the phenomenon. A simulation environment interprets these descriptions to produce a working simulation.

Research in simulation environments is important because scientists and engineers with enough knowledge of a domain often do not want to waste a lot of time programming a simulation. They want to describe the behavior of a simulation in familiar terms.

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 thesis describes the third iteration in the evolution of ChemTrains. The resulting language is referred to as "ChemTrains91" and is not expected to be the end of the road in the search for the best ChemTrains simulation environment.

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.

1.2 Design Methodology Overview

This thesis also explains and evaluates a methodology for designing programming languages. The methodology is called "Problem-Centered Design with Walkthrough Feedback" and is founded in two general principles of good design work:

- design should focus on tasks rather than abstractions; and
- design should be iterative.

The design of ChemTrains91 is shown as a case study in using this methodology. At one level this thesis explores what the best ChemTrains language could be. This design search is described within the framework of the design methodology. At another level this thesis shows how well the methodology works in designing ChemTrains91 and shows how well the methodology might possibly work in designing other languages.

Like ChemTrains91, the design methodology has evolved directly from previous work. It uses a method for evaluating the usability of programming languages called "The Programming Walkthrough," which was used in the previous design of ChemTrains to compare the programmability of three competing designs. (Lewis, Rieman, & Bell, 1990; Bell, Rieman, & Lewis 1990) The programming walkthrough method has also been used to evaluate the DINO parallel programming language (Weaver & Lewis, 1990), and has recently been used to evaluate a language for graphically specifying network communication protocols and to evaluate a general-purpose macro processing language. In all of these cases, the programming walkthrough analysis was done after much of the design work had been completed. (Bell, et. al. 1991) This thesis explores integrating programming walkthrough evaluations more thoroughly into an iterative design process. The programming walkthrough method is employed as the driving force behind every design decision.

Problem-Centered Design with Walkthrough Feedback is an important design methodology to study because:

1. The programming walkthrough method of evaluation has already been proven as quick, but has only proven to be an approximate estimate of usability. Further use of the method will better show its limitations.
2. The programming walkthrough method has already proven to help in generating new design features, but has only been used for the purpose of evaluation. Using this method as the central activity of the design process yields a unique way of searching for easily programmable languages and usable environments. Applying this methodology in the design of ChemTrains91 will show if it is efficient.

1.3 Organization of Thesis

Chapter 2 describes Problem-Centered Design with Walkthrough Feedback. This design methodology is proposed as an improvement to traditional design methodologies and as an outgrowth of previous work.

Chapter 3 first summarizes programming languages and simulation environments similar to ChemTrains. The rest of the chapter is a language manual for ChemTrains91 that includes an abstract description of the language, a description of the programming environment, and detailed advice on how to go about solving simulation problems. The problem-solving advice is required for doing the programming walkthrough analysis.

Chapter 4 and 5 describe how ChemTrains91 was designed within the Problem-Centered Design with Walkthrough Feedback framework. Chapter 4 and 5 are in reverse chronological order, showing the evaluation of the final design (4) before the rationale that supports this design (5).

Chapter 4 is organized by the problems used in guiding the design. For each problem, the solutions and walkthroughs of the final design are shown. This chapter concludes with a summary of the language's shortcomings.

Chapter 5 is organized by the major design issues raised by the walkthrough analyses. For each of the issues, the rationale behind the final design decision is described and supported in terms of the problems and their walkthroughs. A summary of the design alternatives is used to compare the final design and implementation with the previous designs.

Chapter 6 describes user testing of the final prototype and compares the user's performances with the programming walkthrough analysis. The comparison is used to measure the accuracy of the programming walkthrough evaluation method. The results of user testing are also compared with specific points of design rationale.

Chapter 7 compares ChemTrains with OPS5, a typical textual rule-based language, and BITPICT, a rule-based visual language that operates at the pixel level.

Chapter 8 describes some implementation details of ChemTrains. Since the language is a unique sort of production system, details of the rule engine and pattern matcher are described. Possible speed improvements to the pattern matching algorithm are also suggested.

Chapter 9 summarizes what was learned in the design of this language, and compares what was expected from the design methodology with what actually happened.

2 The Design Methodology

The reason for investigating a new design methodology is that current ones do not adequately support usability considerations. The traditional waterfall model of software development is especially ill-suited to addressing issues of usability. System usability is often considered during a typical design process, and designers often cite usability as a primary consideration, but a typical strategy of focussing on abstractions rather than concrete tasks often prevents designers from adequately evaluating usability during design.

This chapter first describes design methodologies for addressing the usability of user interfaces and programming languages. The walkthrough method of analysis is described as an evaluation technique that can be applied in a number of ways within a design process. The previous design process of ChemTrains is described as one possible way to integrate walkthrough analysis into the design process. Finally, Problem-Centered Design with Walkthrough Feedback is introduced as a way of integrating walkthrough analysis that improves the efficiency of the design process.

2.1 Design Methodologies for Usability

One approach to addressing usability considerations early in the design process is by following a set of rules based on usability principles. Several of these have been documented. (Foley and Wallace, 1974; Baecker, 1980) These principles are meant to guide the designer toward a usable interface by providing very general rules (e.g. "The nomenclature used is oriented towards and appropriate for the application." (Baecker, 1980)) A similar approach is to provide a set of guidelines, which are collections of tests which can be applied to an interface to determine if it is satisfactory. Smith and Mosier (1984) provide one such checklist that includes 679 guidelines categorized into different areas. Although principles and guidelines can usefully assist the designer, they can at times be too vague, contradictory, and hard to apply to real design problems. (Mosier and Smith, 1986)

Several approaches address usability considerations by making important changes to the software waterfall model. This is suggested by problems noted by Swartout and Balzer (1982) concerning the intertwining of specification, design, and implementation. Gould and Lewis (1985) recommend three fundamental principals that can be applied to a design methodology, *an early*

focus on users and tasks, empirical measurement, and *iterative design.* Gould, Boies, and Lewis (1991) add a fourth principle: *integrated design.*

Several more specific modifications to the software waterfall model are described by Grudin, Ehrlich, and Shriner (1987) and by Wasserman, Pircher, Shewmake, and Kersten (1986). The latter method integrates prototyping into the design process.

Buxton and Sniderman (1980) describe how prototypes can be used in an iterative design process. Although a prototype based on an early design will give the designers an idea of whether their system will be usable, one of the problems with prototyping is that the cost of building even a prototype is high, even if it is a "rapid prototype." Another problem is that while a prototype gives a great estimate of a design's usability, further design exploration becomes biased towards designs that are similar to the prototype. In order to use prototyping to evaluate a design that is far from the current design, the prototype may have to be completely rewritten. In general, designs closer to the current design will be so much easier to prototype that designers may prefer to continue without exploration of widely varying designs.

Another approach to addressing usability considerations is by implementing a user model which simulates how a typical user would use the possible system. Young (1989) describes a Programmable User Model (PUM) that is a psychologically constrained architecture that gives a designer a framework for describing how a user would interact with a system. The resulting user model may be used as an evaluation of the system, and may draw the designer's attention to the problems in the design. A PUM is not currently working. Young suggests SOAR (Laird, Newell, & Rosenbloom, 1987) as a problem-solving architecture that can be used to build these user models. The problem with this approach is that implementing a model that describes how a user interacts with a system may be nearly as complicated as implementing the system itself.

Another approach to addressing usability before any implementation is the "walkthrough" approach as proposed by Yourdan (1989) and Spencer (1985.) This method involves the designers going step by step through how the product satisfies the needs of the intended users. A group of reviewers questions and probes as the process evolves, unearthing the weaknesses of the design.

Cognitive walkthroughs (Lewis, Polson, Wharton, & Rieman 1990) are a very systematic walkthrough procedure for evaluating features of a user interface. The procedure involves answering specific questions about how the user's goals will map onto actions available from the interface at every step in the sequence of problem solving.

Programming walkthroughs attempt to apply many of these same techniques to the evaluation of programming language designs.

2.2 The Walkthrough Evaluation Method

The *walkthrough* is a technique for evaluating two important aspects of a programming environment: *expressiveness*, the ability to state solutions to hard problems simply, and *facility*, the ability to solve problems easily. In the walkthrough, a programming environment is evaluated by analyzing the mental steps that a programmer would most likely take in solving a specific task. The method is an outgrowth of work on cognitive walkthroughs. Walkthroughs are done on a suite of tasks, called the *target problems*, to give an overall estimate of a language's expressiveness and facility.

The walkthrough analysis focuses on the process of writing programs in a language rather than the programs themselves. In walking through how someone might write a program, the steps of problem solving are recorded. A walkthrough is this written record. The evaluation of expressiveness is simply based on the length of the solution used in the walkthrough. The estimate of facility is a much more complex evaluation that takes into consideration the length and complexity of the walkthrough and the length and complexity of the knowledge that supports mapping problems onto actual working programs. This problem-solving knowledge is called the *doctrine* of the language.

Doctrine includes general concepts of the language and its environment, as well as advice on how to go about solving problems. The problem-solving advice supplies what Soloway (Spohrer, Soloway, & Pope 1985) calls plans, which the programmer needs to bridge between task requirements and individual programming statements or operations. In addition to this high-level problem-solving doctrine, lower levels of doctrine can be used to evaluate the language's programming environment, which also plays a role in programmability. An added benefit of producing doctrine at any level is that it can be used to enhance a language's user manual.

This section first describes the levels of doctrine. Then the generalized walkthrough method is described as a step-by-step process that yields a full analysis of facility. This presentation of the walkthrough method differs from previous descriptions in two important ways:

- the process is defined in a more general way that can be applied to the evaluation of environments in addition to languages; and
- a step of the process, called "the walkthrough summary," is added to more accurately determine facility.

2.2.1 Levels of Doctrine

Figure 2-1 shows the mapping of a problem statement (at the top) into commands in the programming environment (at the bottom). For each level of problem solving, associated doctrine is needed to help bridge from goals at one level to actions at the next level down.

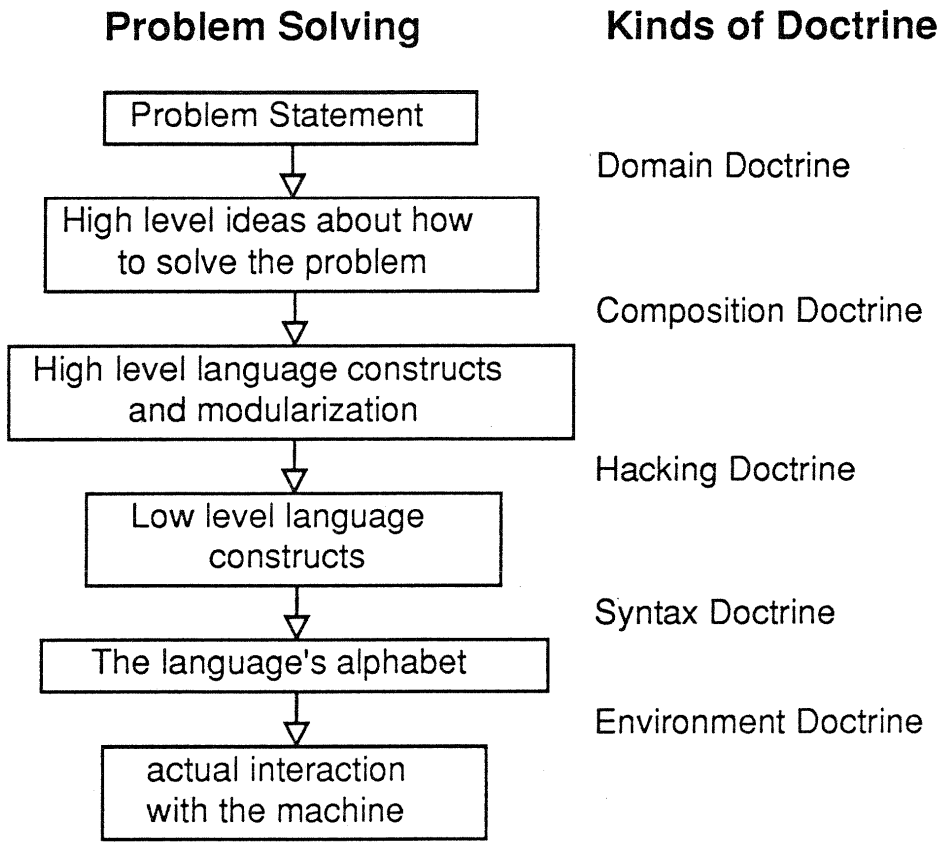


Figure 2-1: Doctrine needed for different levels of problem solving.

Doctrine for any level of problem solving contains the knowledge that an intended user is not expected to know before learning the language but does need to know in order to use the language effectively.

Domain doctrine is knowledge that describes general methods for mapping problems in specific domains onto algorithms. For example, domain doctrine about bridge design would include information about how to apply formulas for computing stress to types of bridge configurations. General-purpose programming languages, like ChemTrains or Pascal, cannot include domain doctrine. In doing a walkthrough analysis for a general-purpose language, this knowledge must be assumed. To ensure a fair estimate of facility, the statement of each target problem must include the domain specific doctrine needed for a solution to that problem. By doing this, the boundary between where the domain knowledge leaves off and the composition knowledge picks up is explicit.

Composition doctrine is knowledge about how to take general algorithms for solving a problem and mapping these algorithms onto high-level constructs of the language. It contains knowledge about how to decompose algorithms into modules. For general-purpose procedural languages this doctrine is typically taught in a college sophomore course on the fundamentals of computer science, or it is learned through experience.

Hacking doctrine is knowledge about how to take high-level algorithmic specifications of a solution and turn them into basic sentences available in the language. Hacking doctrine for a procedural language would contain such things as how to do a summation by using a **for** loop or how to define data structures for hashing. This knowledge is contained in "how to" programming books. The difference between composition doctrine and hacking doctrine is that composition doctrine does not contain the low-level knowledge often needed to get a program working.

Hacking doctrine is necessary in most programming languages because there is a large gap between the specification of a program and the sentences to actually make the program work. The goal of many general-purpose high-level programming languages, such as SETL (Kennedy & Schwartz, 1974), Self (Ungar & Smith, 1987), and Prolog (Clocksin & Mellish, 1981), is to drastically reduce the amount of hacking doctrine. Domain specific high-level languages should not typically need hacking doctrine at all. A language for defining network editors (Bell, 1987) and a language for communications protocol specifications (Citrin, 1990) do not require hacking doctrine.

Syntax doctrine is knowledge about the syntax of the language's constructs that does not include any semantic knowledge about how to use the constructs. While syntax doctrine is typically described in a syntax diagram or BNF grammar written in a language's reference manual, it may also be stated as a set of simple rules to follow. Like the other levels of doctrine, the amount of syntax doctrine can provide an insight into the usability of a language. The syntax of some languages, like Lisp and Prolog, can be stated very concisely, making their syntax doctrine small but not necessarily easy to understand.

Environment doctrine is knowledge about how to communicate in the language. It would include any knowledge about the program editor, the debugger, and other programming tools available in the environment. For example, it would contain general knowledge about how to use a syntax-directed editor for a language.

2.2.2 The Generalized Walkthrough Process

The walkthrough analysis method is not meant to produce an absolute measure of facility upon which all languages may be rated. Because the

method is subjective, only relative facility between two languages can be computed. The method can be applied in two possible ways:

- to uncover shortcomings in a single language, or
- to compare facility and expressiveness of languages that are close in nature.

If the method is used simply to uncover shortcomings, only a rough estimate of facility is needed. If the method is used to compare languages, the estimate of facility should be quantified. In comparing languages, the facility must be evaluated at every level in which the problem solving differs. In order to get an accurate comparison, the languages should not differ substantially in more than two of the five main levels of problem solving. For example, a comparison between Common LISP and Pascal would be very difficult because they differ at all levels of problem solving. A comparison between Common LISP and Scheme would be easier because they only differ substantially at the hacking level of problem solving, since both the lower level syntax doctrine and the higher level composition doctrine are similar.

The following steps of the walkthrough process may be used to uncover shortcomings of a language or to compare languages.

1. *Consider the user's knowledge.* A typical intended user's programming expertise needs to be considered. Do they have computer science background, programming background, or enough mathematics background that help in understanding concepts such as variables? Do they have background that will help them in understanding the model of computation? Will their background help in understanding the syntax? Will their background help in understanding the computing environment? Consideration of the intended user's background knowledge is necessary in order to create appropriate doctrine.
2. *Choose problem-solving level.* If the basic computational model of the language is under consideration, then the walkthrough should be done at a high level. A *programming walkthrough* is a high-level walkthrough of a programming language that ignores the problem solving below the level of the low-level constructs. If the basic computational model of the language is fixed but the syntax and environment are not, then a walkthrough should be done at a lower level.
3. *Select target problems.* The target problems should include a range of tasks that cover all general types of problems that the language is intended for, but should not include a huge number of problems. The target problems should be selected at a level of granularity that is appropriate for the problem-solving level tested. For example, if the environment is being evaluated, the problems do not need to test all kinds of compositional techniques supported by the language; the problems just need to test a variety of the environment features.

4. *Define the language.* A walkthrough analysis requires that the language has at least been designed. To do an evaluation of a language, this language must be fixed at the appropriate level. If a programming walkthrough is done, the language must have a semantic definition but not necessarily a syntactic definition. If an analysis is done at the environment level, the syntax of the language and the programming environment must be defined.
5. *Record solutions.* For each target problem, a reasonable solution must be written down. The solution must be possible within the definition of the language. This step of the process can be done in unison with defining the language, because actually working out the solutions forces a rigorous language definition.
6. *Create doctrine.* The doctrine lists all of the knowledge the intended user needs to know in order to map goals at one level of problem solving to tasks at a lower level. Doctrine is constructed by stepping through each of the problem solutions until each solution is guided from start to finish by pieces of doctrine. Here are some points about how to create individual pieces of doctrine, that might be called "The Doctrine Doctrine:"
   - If a step isn't guided directly by existing doctrine, then write a piece of doctrine by describing the situation that it applies to and the resulting tasks that have to be done.
   - If a piece of doctrine weakly guides a walkthrough step, then try to decompose the doctrine into more specific pieces that provide more direct guidance.
   - If a high-level programming walkthrough is done, the description of tasks should stop at the layer of low-level language constructs.
   - If a piece of doctrine is stated in terms that any intended users may not know, then define the necessary terminology.
   - A piece of doctrine for a general-purpose language may not be stated in domain specific terms.
   - If a piece of doctrine is so specific that it cannot apply to a general class of problems, it must be generalized.
   - If an abstract piece of doctrine can be decomposed into more understandable concrete pieces, then break it into these pieces.
   - If the doctrine is meant to enhance documentation of the language, the pieces of doctrine should be enhanced with examples for readability.
7. *Record walkthrough steps.* For each problem the walkthrough is recorded as a sequence of steps guided by pieces of doctrine. This may be done in unison with the doctrine creation. The final walkthrough record should have no holes.
8. *Write the walkthrough summary.* When a walkthrough of a single problem is complete, the steps should completely proceed from the problem statement to a solution. But all of the steps may

not be equally easy. A *walkthrough summary* is a descriptive summary of a single walkthrough that contains an overall estimate of a walkthrough's complexity and a description of the most complex parts of the walkthrough. The next section describes how to analyze the complexity of the walkthroughs and the doctrine.

9. *Analyze or Compare Languages.* When all the walkthroughs and walkthrough summaries of a language are complete, the expressiveness and facility may be fully described.

2.2.3 Evaluating Facility in a Walkthrough Summary

One metric for the facility of a single walkthrough is the length of the sequence of steps, which approximate the length of time it takes to solve the problem. A simple metric for the facility of a language is the summation of walkthrough lengths plus the length of the doctrine, which approximates the language's learnability. These estimates can be produced easily, but only accurately estimate facility if:

1. steps of a walkthrough are roughly equivalent in complexity,
2. pieces of doctrine are roughly equivalent in complexity, and
3. pieces of doctrine are independent.

These conditions are very hard to meet with such an informal model of a user's possible thinking.

After producing a walkthrough, important aspects of facility may not be apparent at the surface level. The ease with which doctrine is applied in each step of problem solving may vary greatly. A walkthrough step is considered *complex* if there is a possible difficulty during problem solving in selecting appropriate doctrine or applying the selected doctrine in an appropriate way. The process of measuring facility involves judging the complexity of the walkthrough steps.

A walkthrough summary is a listing of descriptions of the most complex parts of the walkthrough. If steps of the walkthrough are unexpectedly simple, they should also be listed in the summary. For each step listed in a walkthrough summary, an explanation of the complexity should be stated. When summarizing a walkthrough and a language's doctrine, here are some issues that should be considered:

- *Is doctrine rooted in common knowledge?* Common knowledge is knowledge at any level of problem solving that a large majority of the intended user's know before learning the language. Doctrine with strong ties to common knowledge should be easier to learn and apply. For example, the doctrine for doing arithmetic in most procedural languages would rely on common knowledge of arithmetic. A language that uses prefix notation would not yield simple syntax doctrine for writing arithmetic because it doesn't fit well with existing common knowledge.

for Pamela

- *Is a piece of doctrine abstract?* A piece of doctrine is *abstract* if it needs many specific examples to fully explain. Since abstract pieces of doctrine are harder to apply to actual problems, abstract doctrine is generally more complex than less abstract doctrine. In some cases it may be possible to break a very abstract piece of doctrine into several more concrete pieces that are easier to understand.
- *Does doctrine exist at different problem-solving levels?* Two pieces of doctrine may not be equally complex because they are at different levels of problem solving. Pieces of doctrine that are closer to the bottom layers of problem solving are typically easier to apply because the tasks more directly map to actual interactions with the interface.
- *Is there interference among doctrine situations?* If the situations described in different pieces of doctrine are similar, walkthrough steps that use these pieces of doctrine are more complex because the application of doctrine in problem solving becomes more difficult.
- *Does doctrine lead to impractical solutions?* An *impractical* solution is a solution that is much longer than the shortest possible solution to a problem. At any given step of a walkthrough, many pieces of doctrine may apply to the state of problem solving. If each of these possible choices leads to a practical solution, the doctrine is fine; however, if one piece of doctrine consistently leads to impractical solutions, then this piece of doctrine should weigh against the facility of the language.
- *Is a step guided by multiple pieces of doctrine?* If a single walkthrough step is guided by multiple pieces of doctrine, each of which support the step in a different way, then the step is simple.
- *Does a step require an "aha?"* A walkthrough step may be complex because it relies on some leap of thinking. An example of an "aha step" would be choosing a B-tree data structure for representing the positions of graphical objects on a two dimensional plane (Bentley, 1979). This step would be particularly tricky because B-tree doctrine would refer to database situations rather than graphical situations, since the B-tree data structure has been primarily used in database implementations. This step would not be a big aha if portions of Bentley's paper are listed as doctrine. A collection of doctrine that included such specific and detailed ideas would need to contain most algorithm literature written to date, in addition to future ideas. Since all possible leaps of thinking cannot be accounted for in doctrine, aha steps should be recognized as such in a walkthrough and should count against a walkthrough.
- *Is a step concrete?* A walkthrough step may be simple because it leads directly from the current problem-solving situation. For

example, composition doctrine for general-purpose programming languages might include something like: "when defining a data structure, build a library of accessing and setting functions for the data structure." Even though this is high-level composition doctrine, this piece of doctrine leads concretely from the step of using a B-tree for the representation graphic objects to the step of building B-tree accessing and setting functions.

The complexity of each step will fall somewhere in the range between concrete and abstract. The final analysis of a language contains an analysis of the doctrine and a summary of common shortcomings based on all the walkthrough summaries.

The types of steps that were hardest to make and the pieces of doctrine that were hardest to follow point to the language's shortcomings. When using the final analysis to compare languages, the relative facility can be measured by examining the difference in the number and the difficulty of the steps between the languages. In order to get an accurate comparison between languages, the same basic solutions to the target problems should be used.

The walkthrough procedure is an imperfect, subjective, but quick analysis of facility. The statement of doctrine is the main source of possible unfairness in judging facility. For example, a single language may have two equally reasonable statements of doctrine that yield walkthroughs of different lengths. The walkthrough summaries provide the evaluator with a way to balance out doctrine inequities.

Walkthrough analysis may be integrated into the larger context of design in a number of ways. The walkthroughs may be done by the designers of the system, user interface experts, possible users, or evaluators comparing systems. Walkthroughs may be done early or late in the design process.

2.3 The Previous ChemTrains Design Methodology

The approach that was used to design the previous implementation of ChemTrains involved two phases of design, problem-centered design and competitive design with walkthrough evaluation. This approach is described more fully in a previous paper (Lewis, Rieman, & Bell, 1990.) An overview of this design process is shown in Table 2-1. Entries in italics are details specific to ChemTrains.

The problem-centered design phase involved informally working through six of ten target problems that were specified as the requirements for the language. Design discussions focussed on working through the problems and generating design alternatives that would possibly facilitate solutions. The target problems were broken into smaller pieces, called *micro problems*, allowing the designers to focus on specific design issues. New design features

were introduced into the design discussion only to address existing shortcomings in a solution to a micro problem. In this way, the search of the design space was directed toward design features that helped solve the most number of target problems, and away from features that either did not help solve any problems or were believed to be too complex.

Table 2-1: Design Process used for previous ChemTrains.

Problem-Centered Requirements Specification
specify tasks (target problems) that can be solved:
problems with a qualitative model:
mail room, petri net, turing machine, bunsen burner, maze, tic tac toe, doorbell, grasshoppers, sunspots, and xerox machine.
specify type of user:
nonprogrammer.

Design Phase 1: **Problem-Centered Design**
elaborate design space by working through subset of problems:
discover possible ChemTrains features.
define design space:
28 ChemTrains design features.

Design Phase 2: **Competitive Design with Walkthrough Evaluation**
design alternative systems:
OPSTrains, ShowTrains, ZeroTrains.
build doctrine by working through subset of problems:
doctrine of 3 designs.
walkthroughs on reserve problems:
*10 problems * 3 designs = 30 walkthroughs.*
compare walkthroughs of designs:
for expressiveness & facility.

The main advantage of this design process was that it enabled us to uncover the most appropriate part of the design space, the alternatives that solve problems stated in the requirements specification. The main disadvantage was that although a lot of design territory was discovered, the search could have been more efficient using better methods of evaluating new design features. Since the exploration of new design alternatives depends directly on what is assumed about a possible current design, a weak evaluation function may lead the design through irrelevant design alternatives. In the iterative process of problem-centered design, the search through bad design space may explode.

The second phase of the design of ChemTrains satisfied what was missing in the first phase: an evaluation function. In this phase of design three widely differing ChemTrains designs were chosen from the design space. The six problems that had been used in design phase 1 were used to create the doctrine. Each of the designs was evaluated by doing programming walkthroughs on the six problems in addition to four larger problems held in

reserve. Programming walkthroughs were used in place of a prototype to estimate the expressiveness and facility of a possible system. This method provided a clear advantage over prototyping: we were able to compare three totally different designs for usability fairly quickly.

The two phases provided an effective design one-two punch: first, elucidate the whole design space, getting an idea of the tradeoffs involved; then, pick some possible winning designs, evaluate them, and choose a winner. One disadvantage with this method is that in the process of evaluating the designs more thoroughly in the second phase, new design alternatives arose. These features were discovered in the process of not only building doctrine with the six main problems but also analyzing the larger, more complex reserve problems. Another disadvantage is that although the winning design may be the best of the chosen designs, it may not be the best possible design from the set of known design choices.

The problems with this design methodology can be solved by modifying the methodology in two ways. First, by making the whole process iterative, new design features may be added after programming walkthrough analysis, and better designs can be discovered by throwing out features in the design that aren't shown to help in a walkthrough analysis. Second, by replacing informal problem-centered design with walkthrough evaluation and analysis, design features that are clear losers can be thrown out quickly enabling a more efficient search through the design space. The resulting design process is similar to the prototyping design process because it gives an early evaluation of the usability of a system, but it offers the designer the freedom to easily change a design and re-evaluate it.

2.4 Problem-Centered Design with Walkthrough Feedback

This design methodology is an iterative process that is analogous to a general search algorithm such as generate and test. The test in this process is the walkthrough test of usability considerations. Generation of new designs in the design search space is usually a subjective matter left up to the designer, but guidelines for reducing the amount of search can be described. Some guidelines based on "cautious" and "curious" approaches to design generation are described below. The full design process is shown in figure 2-2. The termination condition for this loop is not explicitly shown in the figure. This design process may stop for a lot of different reasons that depend on the judgement of the designers:

- The known design space has been explored.
- The evaluation of the best design found so far seems adequate.
- The amount learned on recent design iterations is becoming less fruitful.
- The designers have no more time or patience for design work.

Like any design process the design search may proceed indefinitely if a perfect system is desired. When the amount that is learned on each iteration becomes small, the designers should consider prototyping and user testing.

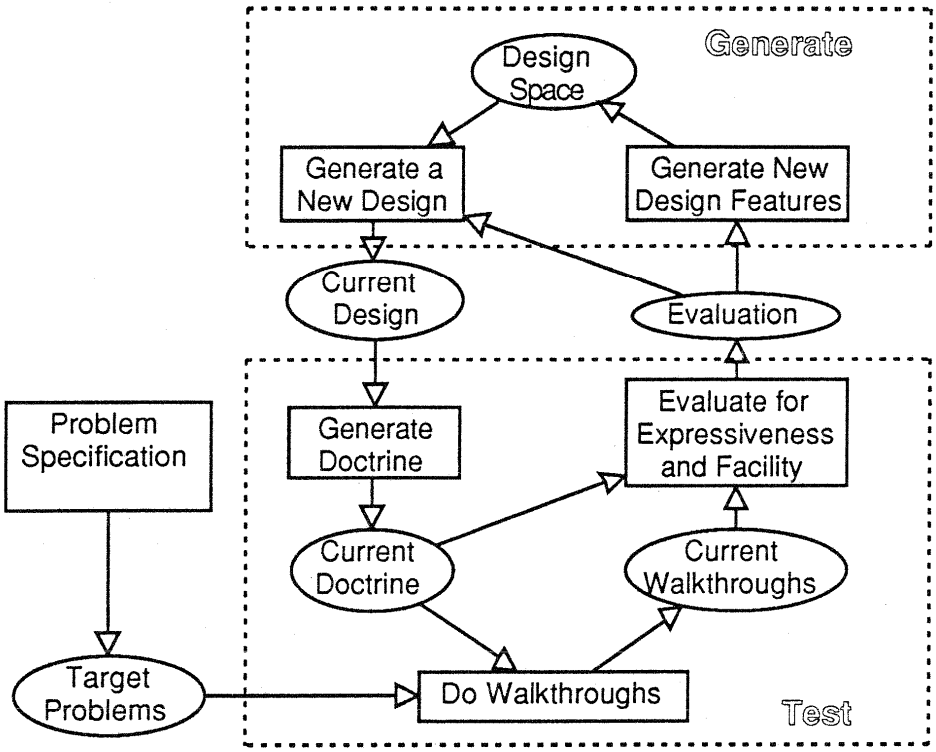


Figure 2-2: Problem-Centered Design with Walkthrough Feedback.
(A dataflow diagram: boxes are procedures, ellipses are data.)

The design process should not only result in a single design that is highly usable, but should also provide an analysis of the design space. The analysis can be done by keeping an organized and full record of the search that includes design descriptions, walkthroughs, and evaluations for all designs that have been explored. This record can be useful for current and future design work. The record of the search ought to enable a designer to easily answer questions like:

- What design is the best for doing target problem A?
- What design features have the greatest impact on expressiveness?
- How much impact do these features have on expressiveness?
- Are design features X and Y compatible?
- What are all the pros and cons associated with feature Z?

Being able to answer questions like this, the designer will be more able to explain previous design decisions and continue with further work. MacLean, Young, and Moran (1989) assert that "the product of user interface design should be not only the interface itself but also a rationale for why the interface is the way it is." The record of the design search is intended to help satisfy this requirement. In addition, the record of design can aid the designer with some of the drudgery involved in repeating work from one design iteration

to the next. For example, when generating the doctrine for a design, most of the doctrine can possibly be borrowed from doctrine of a similar existing design.

Each of the steps involved in the design process imposes needs on the record of the design search. The following discussion of the design components includes these needs.

2.4.1 Specification

The first task in this process is the specification, from which the target problems are generated. This task is similar to generating a suite of programs to test a compiler. The problems must be rich enough to test a wide variety of possible situations, yet small and concise enough to enable fairly quick walkthrough analysis. It is unrealistic to expect an ideal set of target problems on the first pass. The suite of target problems may change during the design process for any of the following reasons:

- The general specification has changed, resulting in an additional target problem.
- A target problem is thrown out or reduced because it is covered by one or more other problems.
- As discussed later, the statement of target problems may be augmented with problem-solving information discovered in the design process after the walkthrough analysis.

The disadvantage with changing the target problems is that comparison between old walkthrough evaluations and new ones becomes unfair. When old walkthroughs are significantly effected by a change in the specification, the designer should update the old walkthroughs. Because this is a cumbersome task, changes to the specification should be done infrequently.

2.4.2 The Design Space

The design space is stated as a set of yes/no questions, each of which has an associated detailed explanation. A design is a set of answers to each of the design space questions. The statement of the design space is an important record because:

- it states what features have been considered;
- it is usually extended on each iteration of design;
- it is used for precisely defining the features of a design; and
- it is used for precisely comparing the features of multiple designs.

Since the record of the design space is central to the iterative design process, it is important that the yes/no questions be easily accessible and extendable. The questions should not be listed in the order they were created, but should be placed in a hierarchy. A hierarchical representation enables the designer to quickly find existing design alternatives and to view a set of related design features together.

2.4.3 Generation of New Designs

The generation of new designs should lead the designer efficiently to an optimal design in the space. In picking a design, there are two contradictory but necessary approaches: a cautious approach, in which a design is picked that is similar to an existing good design but includes minor adjustments, and a curious approach, in which a design is picked because it is a totally different style from any existing design that deserves exploring. The curious approach should be used in the beginning of the design search process in order to discover what overall design features fit the problems the best. The cautious approach is similar to hill climbing and should be used more often near the end of the design search. The cautious approach may also be used to fill in details of an existing design. As the design progresses, the walkthroughs may be done at a lower level to determine environment details that were ignored at the higher levels. Using the cautious and curious approaches to zeroing in on a design can be likened to the simulated annealing approach of problem solving (Kirkpatrick, Gelatt, & Vecci, 1983.)

2.4.4 Generation of Doctrine

A first draft of the doctrine can easily be produced by combining all of the explanations of each of the chosen features in the design. The doctrine can be completed by filling in the usage of these features. After the doctrine is completed, pieces of the doctrine that have been derived from specific features should be associated in some way to these features, so that future generation of doctrine becomes simpler. In cases where a combination of features effect a single portion of doctrine, the doctrine should be associated with the combination of features rather than just one of them.

2.4.5 The Walkthrough Procedure with Recycling

On the first iteration of the design process the walkthroughs are done for the first time. There are some problems with doing repeated walkthroughs on following design iterations:

1. Solutions may become increasingly better not because the designs are better but because the target problems are more well understood. (e.g. a better representation may be found using primitives common to all designs.)
2. As the target problems are walked through repeatedly, the designer may not remember some of the insights required to initially solve the problem. In doing walkthroughs over and over, steps that were initially complex may seem more simple over time.
3. Doing walkthroughs on the same target problems repeatedly is a tedious task that begs to be lightened.

The first and second problems are very similar to "carry-over effects" known in experimental psychology when subjects are given a sequence of related experiments. The subjects carry over a bias from one experiment to the next.

These problems are addressed by extending the walkthrough process in the following ways:

- Do not begin the walkthrough of a problem until the statement of the problem and the solution seem stable.
- When a new solution to a target problem is discovered during the design process, either ignore the solution, or introduce the solution in a new statement of the target problem. Since a target problem statement contains an approach to a solution, multiple target problem statements may be listed for one general target problem. A new target problem statement should be added only if it exercises abilities that other problems do not exercise.
- To guard against bias in doing repeated walkthroughs (problem 2), the walkthroughs should contain annotations describing insights that are important. These insights can be described in the walkthrough summaries, and can be reused in future walkthroughs if they are still applicable. Insights that are domain specific should be included in the target problem statement.
- To reduce the work involved in doing repeated walkthroughs (problem 3), parts of previous walkthroughs that are identical or very similar should be reused. This can be done most easily when minor changes have been made in the design (the hill climbing design approach). A recycled walkthrough will be checked for new gaps or holes in the logical progression of steps. The annotations of required insights are also carried along in a recycled walkthrough. Since these insights are explicitly saved for each walkthrough, they remain to possibly be lessened in future designs.
- Another way to reduce the work in doing repeated walkthroughs is to only do new walkthroughs on a selected set of target problems. Although new design changes may affect all walkthroughs, a walkthrough analysis of only the most affected problems will give a quick comparison of facility with previous designs.
- An additional way to address the problem of carry-over effects is to have another person confirm the logic in each walkthrough.

2.4.6 Using Walkthrough Analysis for Design Extensions

The purpose of evaluating the walkthroughs is to compare expressiveness and facility of the current design with previous designs. The evaluation, in turn, is useful in generating new design features and making new design decisions. The walkthrough summary evaluations state complexities in the walkthrough, which point to the problematic features of the language. These shortcomings are addressed by generating either alternative features or extensions to existing features. In either case the proposed features are added as yes/no questions to the existing statement of the design space.

3 ChemTrains Language Description

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.

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.

This chapter has four sections:

1. an overview of similar languages and environments,
2. basic programming concepts and terminology of ChemTrains,
3. the composition and hacking doctrine describing how to go about writing ChemTrains programs, and
4. a description of the programming environment that supports the ChemTrains language.

3.1 Language Background

ChemTrains shares a lot of ideas with previous systems. Since the focus of ChemTrains is on qualitative rather than quantitative solutions to problems,

the language shares ideas with languages for symbolic processing. ChemTrains more specifically shares ideas in common with:

- production system languages,
- visual languages, and
- environments for building graphical simulations.

3.1.1 Production System Languages

Production systems derive from a computational formalism proposed by Post (1943) based on string-replacement rules. The closely related idea of a Markov algorithm (Markov, 1954) involves imposing an order on the replacement rules and using this order to decide which applicable rule to apply next. Newell (1973) used string-modifying production rules with a simple control strategy to model certain types of human problem-solving behavior. The OPS family of languages was evolved in the late 70's and early 80's into a general-purpose programming language. OPS5 (Forgy, 1981) was successfully used to construct the XCON expert system (McDermott, 1984) for configuring VAX computers that contained two thousand rules. Success in using production system languages to build expert systems (Hayes-Roth, Waterman, & Lenat, 1983) led to the creation of several commercial languages for building expert systems, such as ART (Inference Corp.) and Knowledge Craft (The Carnegie Group.) More recently another production language, Soar (Laird, Newell, & Rosenbloom,) has been designed and used for cognitive modeling. Although production system languages have proven to be expressive languages that can model complicated processes, there has not been much research in designing and building production system languages that are substantially easier to learn and use.

The Production System Model of Computation

Production system languages execute rules, called *production memory*, on data, called *working memory*, by running an *inference engine*. The working memory is divided into working memory elements that can be considered individual facts. The inference engine uses the rules of production memory to add or remove facts from working memory. Each rule in production memory has a left hand side, which describes a pattern, and a right hand side, which describes actions to execute when the pattern has been matched. The inference engine executes what is known as a *recognize-act cycle* to have the rules effect the working memory. This cycle has the following steps:

1. *Pattern matching* recognizes rules whose left hand side condition completely matches some portion of the working memory.
2. *Conflict resolution* picks one of these rules and picks a set of working memory elements that matches the rule's condition.
3. *Rule execution* executes the actions of the selected rule, producing output, making changes to working memory, or in some languages making changes to production memory.
4. The cycle repeats.

If no rules can match, the inference engine halts. The specific strategy for conflict resolution varies among different production system languages. Early production system languages used rule ordering to determine which rule to fire. OPS5 uses a strategy based on recency of data and specificity of rule. The conflict resolution strategy has a significant impact on how rules are written and controlled. In the Grapes (Sauers & Farrell, 1982) and HAPS (Sauers & Walsh, 1983) production system languages rules have not only a pattern and action but also a description of the goal within a goal hierarchy. The conflict resolution strategy used by these languages selects rules to execute based on this goal hierarchy, forcing rules to be written within a structured modular framework.

OPS5

OPS5 is a typical textual production system language. In OPS5, working memory is a list of records whose structure is defined by a type definition. The following expressions define the type "person" to have two attributes, define the type "population-computation" to have two attributes, and add four instances of the "person" type to working memory:

```
(literalize person
  name
  age)

(literalize population-computation
  total-age
  group-size)

(make person ^name Fred ^age 34)
(make person ^name Wilma ^age 34)
(make person ^name Barney ^age 32)
(make person ^name Betty ^age 30)
```

A rule has three parts, a name, a pattern, and a list of actions. The following rule, named "announce-oldest," matches the person with the largest age, and will print out a statement of the oldest person in working memory. The pattern of this rule will match both the "Fred" and "Wilma" working memory elements. The first condition matches any person. The second condition is a negation that states there is no person with a greater age.

```
(p announce-oldest
    (person ^age <age> ^name <name>)
    - (person ^age > <age>)
  -->
    (write (crlf) <name> " at " <age>
                  " is one of the oldest people known."))
```

The following rule creates a working memory element for counting the population, if one doesn't exist. This rule shows that new working elements can be added in the action of a rule.

```
(p start-average-computation
    (person)
    - (population-computation)
  -->
    (make population-computation ^total-age 0 ^group-size 0))
```

The following rule will add the age of a person to the total age tallied by the population-computation working memory element. This rule shows that matched objects in a pattern may be referred to and modified in the action.

```
(p add-person-to-average
    (population-computation ^total-age <total-age>
                            ^group-size <group-size>)
    (person ^age <age>)
  -->
    (modify 1 ^total-age (compute <total-age> + <age>)
              ^group-size (compute <group-size> + 1)))
```

This simple OPS5 program would write out the names of the oldest people represented in working memory, and then it would create a "population-computation" element, which is used to sum up the age of each and every person. The program would not infinitely fire the "announce-oldest" rule or the "add-person-to-average" rule, because OPS5 only fires a rule once for each combination of matched data.

BITPICT

BITPICT (Furnas, 1990) is a production system language in which working memory is a bitmap display and production memory is a set of pattern/result rewrite rules. BITPICT rules are drawn at the pixel level, and specify exactly how patterns of pixels should be replaced with a different pattern of pixels.

Figure 3-1 shows an example of a simple BITPICT simulation of balls rolling down ramps, using a naive physics model. The four pictures represent different time slices of the simulation executing. Two rules are needed, one that moves a ball down, and another that slides a ball along a ramp. For a BITPICT rule pattern to match, the field of the pattern bitmap must match a piece of the bitmap display exactly. The BITPICT language permits rules to match with or without regard to rotation in 90 degree intervals and with or without regard to horizontal or vertical reflection. For example, the "fall" rule in figure 3-1 is specified only to match in the shown configuration, but the "ramp" rule is specified to match the shown configuration or a horizontal reflection.

The need for BITPICT rules to match rotations and reflections more generally is shown in the simulation of counting the number of unconnected vines in a tangled forest of vines, shown in figure 3-2. This simulation works by first reducing each vine to a single point, collating the points at the bottom of the screen, and doing roman numeral arithmetic on them. The three rules for

reducing the vines must each work for general rotations and reflections. The other rules must instead be restricted to match only in the one orientation.

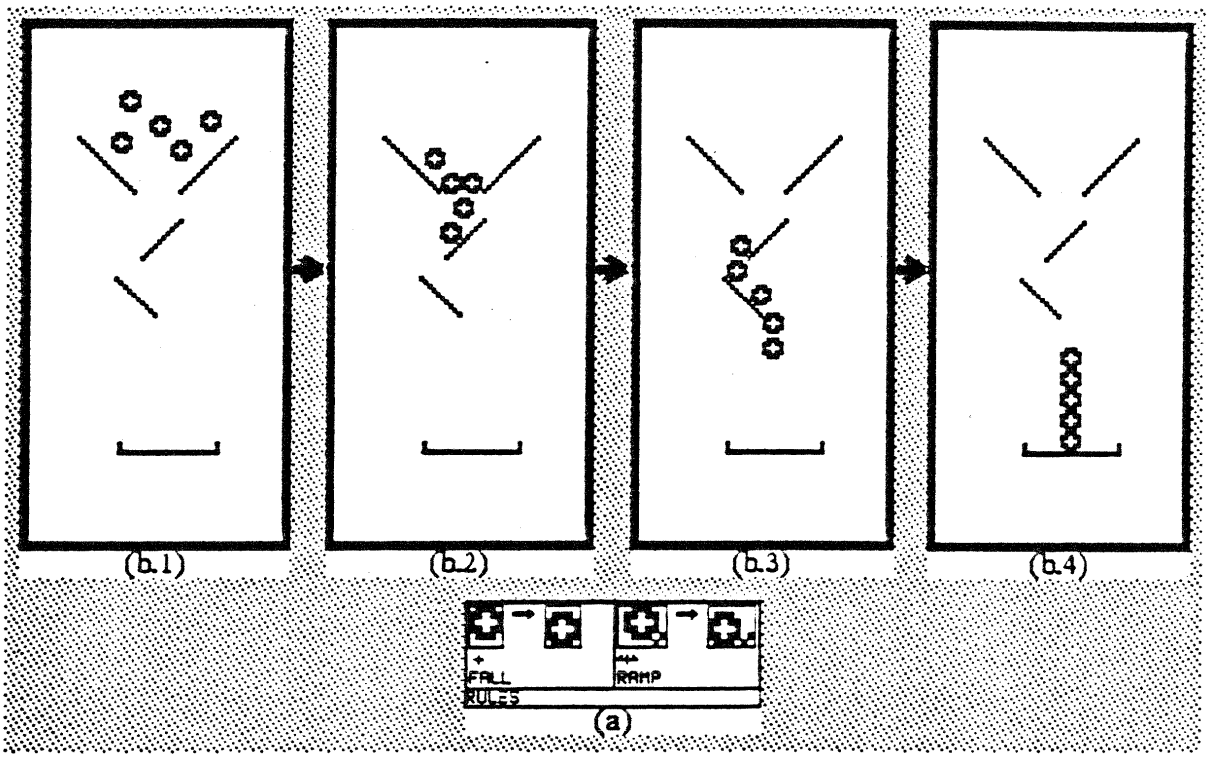


Figure 3-1: BITPICT simulation of balls and ramps: (a) Rules for a straight fall and ramp rolling; (b) Samples of four configurations from beginning to end.

The simulation in figure 3-2 also illustrates an issue of control. In writing rule-based programs in any production system language, the execution of one rule may interfere with the execution of others. In languages such as OPS5 and ChemTrains there is no explicit feature for modularizing and separating the execution of rules. In these languages the programmer must invent separate working memory structures that are used solely for the purpose of control. A language such as GRAPES supplies a feature for modularizing rules by describing how they apply to solving goals in a goal tree. BITPICT supplies a simple feature for separating rules into groups. The rules in figure 3-2 are separated into three groups. After rules in one group are finished executing, the rules of the next group are started and the rules of the previous group are no longer allowed to execute. This feature of modularization is necessary so that simulation of one process cannot accidentally interfere with simulation of another process.

Chapter 7 further describes the differences between OPS5, BITPICT, and ChemTrains.

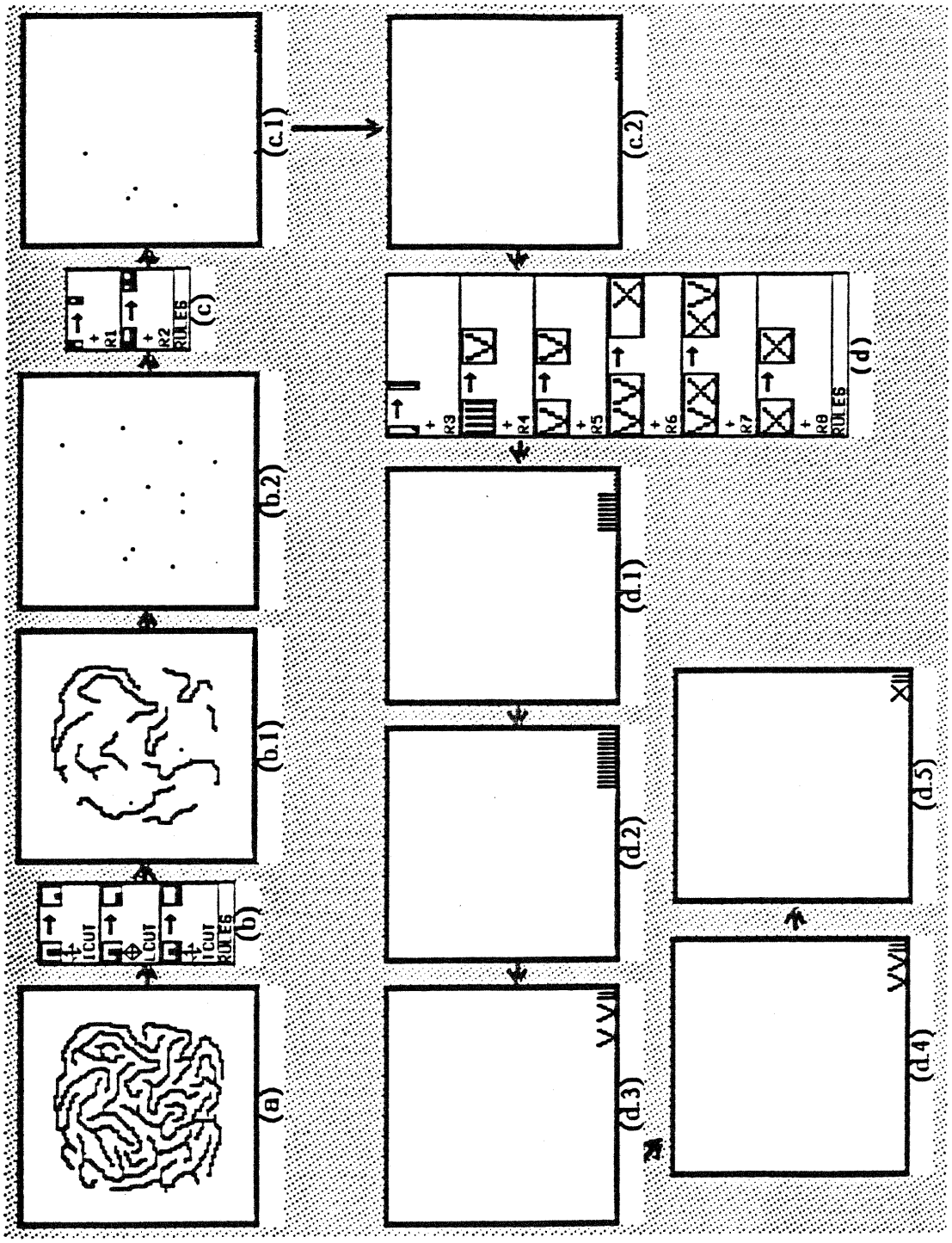


Figure 4. Counting the tangled forest. (a) A tangled forest of bifurcating trees. The rules, (b), nibble at the tips of straight, corner "L", and "T" sections, reducing each component, (b.1), to a dot (b.2). The rules, (c), move the dots down, and to the right, (c.1), into the lower right corner, (c.2). The rules, (d), count the aligned dots in Roman numerals, yielding the answer, XII.

Figure 3-2: BITPICT simulation of counting vines in a tangled forest.

Logic-Based Languages

Logic-based languages such as Prolog (Clocksin & Mellish, 1981) use predicate calculus to represent data and programs. Prolog programs are similar to rule-based programs because the programs consist of a set of implication statements, which are similar to rewrite rules. The following is an example prolog implication statement:

```
grandfather(X,Y) :- parent(Z,Y), father(X,Z).
```

Prolog executes programs proves theorems by applying these implication statements to the stated facts, in much the same way that rules are applied in a production system language.

Ringwood (1989) presents a graphic representation for logic programs, in which facts are represented as directed graphs, and the implications are shown as rewrite rules in the same graph structure. These graphic rules are more similar to ChemTrains than to BITPICT rules because the pattern and result pictures are at a fairly high level representing logic predicates as constellations of objects connected together.

3.1.2 Visual Languages

Visual programming and program visualization are active and old fields that have been surveyed by Myers (1988) and Shu (1988.) *Visual programming* refers to systems that may be programmed in pictures with at least two dimensions. Textual languages are not considered two dimensional because they are read and processed as a string. *Program visualization* is the graphic demonstration of a program running, illustrating either the execution of code or the modification of data. ChemTrains relates to both classes of systems, the first because ChemTrains programs are visual, and the second because the execution of ChemTrains programs is visual. The following are some systems and classes of systems that relate to ChemTrains because the programs are visual.

A large number of visual languages have been designed and built to provide a visual syntax for specifying program control. For example, Nassi-Schneiderman (1973) diagrams were formalized as a modification to flowcharts that enable goto-less structured programs to be drawn. The PICT graphical programming environment (Glinert & Tanimoto, 1984) enables programs to be written and tested by drawing data flow diagrams out of pre-existing icons representing different operations. The Tinkertoy graphical programming environment (Edel, 1986) enables Lisp programs to be constructed similarly by drawing data flow diagrams out of lisp functions represented as icons. HI-VISUAL (Ichikawa & Hirakawa, 1987) and PROGRAPH2 (Cox & Pietrzykowski, 1988) are two other iconic dataflow languages which explore modularization and data abstraction. A version of PROGRAPH is also commercially available. The main difference between these languages and ChemTrains is that these languages have specific

graphical constructs defining the control of a program, while ChemTrains doesn't have them.

Kahn and Saraswat (1990) describe a graphical language based on a textual language for concurrent programming language. Their language, called Pictorial Janus, permits the behavior of graphical animations to be created in the same graphical representation as the animation. An example simulation is shown in figure 3-3. The graphical syntax of this language, like ChemTrains, is based on the topology and the connections of objects in the picture. Unlike ChemTrains, the kinds of graphical objects, such as circles, bullets, and different kinds of arrows, each have different computational meanings, limiting the animations. Also unlike ChemTrains, the rules of behavior, rather than being described as pattern/result rewrite rules, are described in single pictures in which the pattern picture contains a miniature version of the result picture, and the mapping between the pattern and result objects is described with connections.

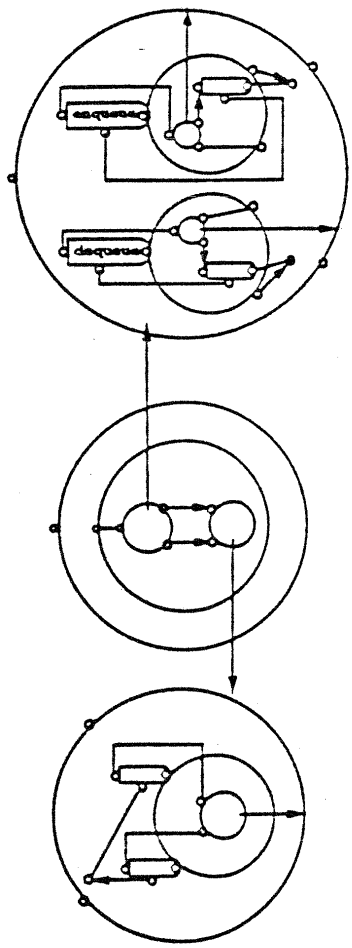


Figure 3-3 : An example Pictorial Janus program that models a queue. The top part of the display describes the behavior of the enqueue and dequeue operations.

3.1.3 Graphical Simulation Environments

The following systems are similar to ChemTrains because they offer simple ways to construct graphical simulations, but these systems do not offer a lot of support for visual programming.

NoPumpG (Wilde & Lewis, 1990) also allows complex graphical simulations to be created easily. Simulations are built in this system by creating unidirectional constraints between points of the graphical objects on the screen. The constraints are viewed and edited within spreadsheet-like cells that are automatically generated and associated with objects on the screen. ChemTrains was developed, in part, because of the awkwardness of doing qualitative problems with equations in NoPumpG.

ThingLab (Borning, 1979) is a system for building complex graphical simulations. There are three fundamental of features of ThingLab:

- a part-whole hierarchy of objects that enables data abstraction,
- an inheritance hierarchy of object classes that enables reuse, and
- specification of behavior as constraints.

ThingLab is very much like NoPumpG, in that positions and sizes of objects on the simulation display and relationships between objects can be easily defined with constraint equations and numeric formulas. The main difference between these two environments is that NoPumpG opts for simplicity of the environment. The main differences between these simulation environments and ChemTrains is the model of computation and the focus of these systems on quantitative models.

AgentSheets (Repenning, 1991) is an object-oriented tool for building simulations that provides an intermediate layer between the high-level building blocks and the low-level programming. In the behavior of objects is described by associating textual code with classes of graphical objects. Simulations in AgentSheets must initially be defined by programmers knowledgeable in the underlying textual language. However, the behavior of a simulation may be extended by casual users because of the ability to easily create new classes of objects by cloning combinations of previously defined classes. In addition to requiring textual code and having an object-oriented typing mechanism, AgentSheets differs graphically from ChemTrains because objects populate a two dimensional grid of cells, facilitating adjacency rather than containment as the natural visual relationship.

3.2 ChemTrains Programming Concepts

The ChemTrains and BITPICT visual rule-based languages are different in precisely the same way the Draw and Paint programs are different. In BITPICT, like Paint, pictures are drawn by changing the pixels of the display. After a circle or rectangle is drawn in these programs, the drawn shape is not accessible as a single object and can only be moved or modified by changing

specific pixels of the display. In ChemTrains, like Draw, pictures are drawn by adding and changing graphical objects on the display. After a circle or rectangle is drawn in these programs, it can be moved or modified easily as a single autonomous object. In both ChemTrains and Draw, the structure underlying a drawing cannot always be inferred from simply looking at the drawing. For example, something that looks like a rectangle may be any number of possible structures, including four independent line segments, one single rectangle, or any number rectangular structures all overlaid. In BITPICT and Paint, since the pixel displays are the structure, there is no ambiguity in them.

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.

3.2.1 Terminology

A *simulation display* is a window that displays a graphical simulation. Figure 3-4 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 3-4 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 3-4 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 3-4 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.

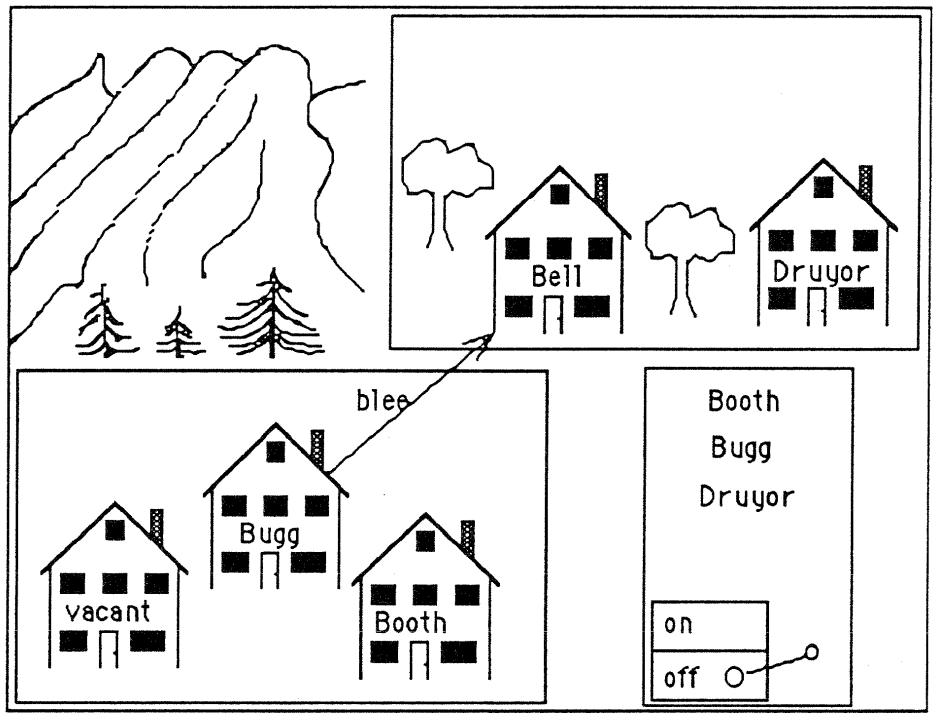


Figure 3-4: Example ChemTrains Simulation Display: house lighting controlled by a switch.

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 3-4, 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 3-4 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 3-5 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 3-5 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.

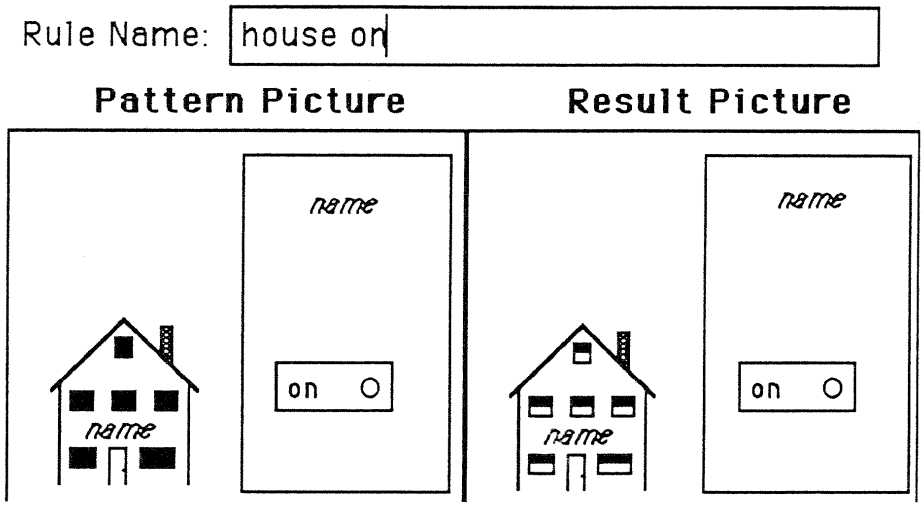


Figure 3-5: Example simulation rule to turn on house lights.

The rule interpreter of ChemTrains, like other production system, executes a recognize-act cycle that has three phases, a pattern matching phase, a conflict resolution phase, and a rule execution phase.

3.2.2 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 3-5 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 3-4 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.

3.2.3 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.

3.2.4 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 3-5.

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 3-6. To enable the house lights to be turned off when the switch is in the "off" position, the "house off" rule, shown in figure 3-7, is added.

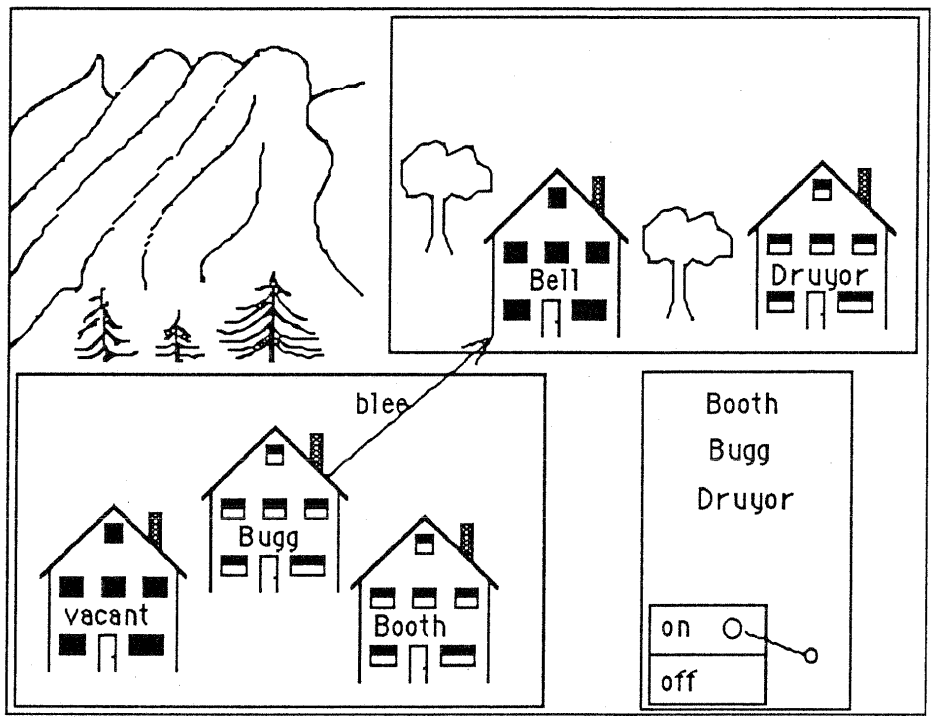


Figure 3-6: Changes to House Lighting Simulation Picture.

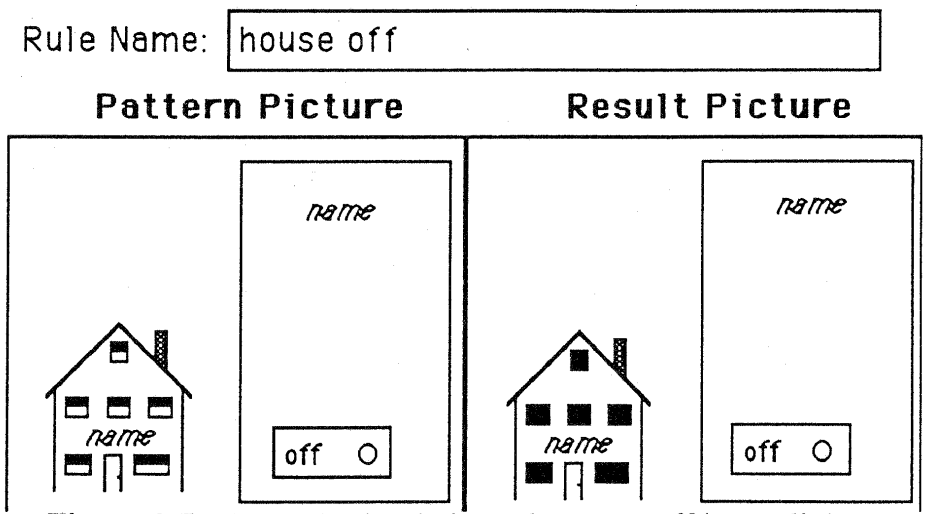


Figure 3-7: Example simulation rule to turn off house lights.

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.

3.2.5 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.

3.2.6 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 4-8 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.

3.2.7 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.

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 3-8 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.

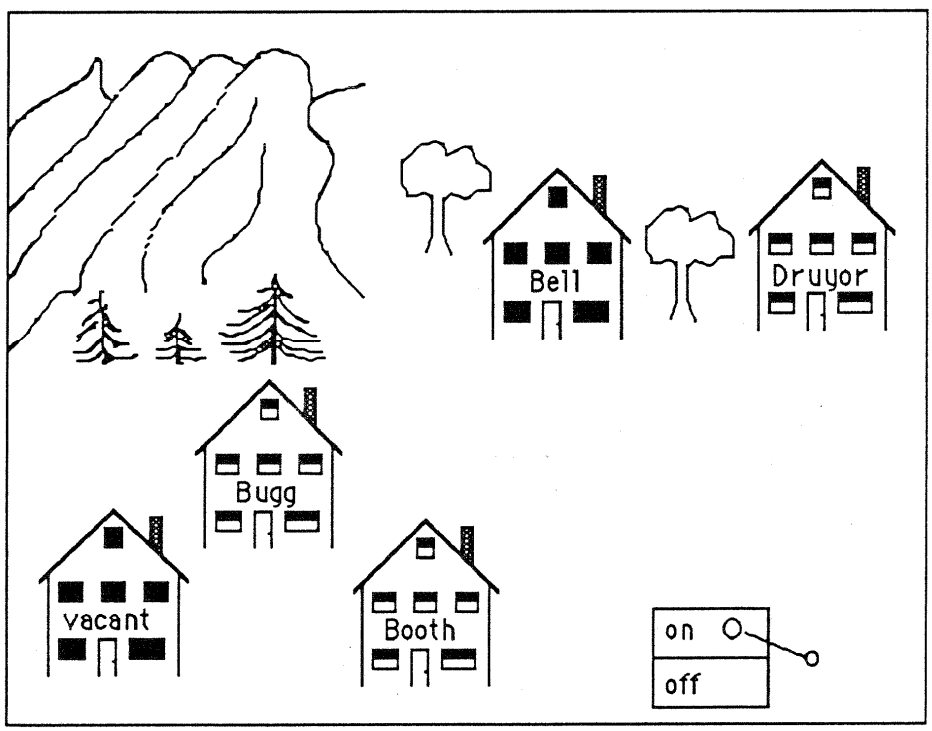


Figure 3-8: House lighting simulation picture with hidden objects not shown.

The tic tac toe simulation (section 4.4) shown in figure 4-8 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.

3.3 Programming Techniques

This section is the composition and hacking doctrine for ChemTrains91. The doctrine is divided into pieces that can be considered individual programming techniques. Techniques are described for constructing the display of the simulation and for describing the behavior of the simulation. The structure of the picture (how objects contain other objects, and how objects connect) plays a major role in the ease with which the simulation behavior can be described.

Each technique description has four parts: a name describing its use, a list of target problems that it has been applied to, an abstract description of the problem-solving situation in which the technique can 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 for programming in ChemTrains91:

draw initial picture: [all simulations]
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

3.3.1 Creating Rules

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

create rule: [all simulations]

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 3-9 through 3-14)

deletion action: [most simulations]

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 3-9 & 3-12)

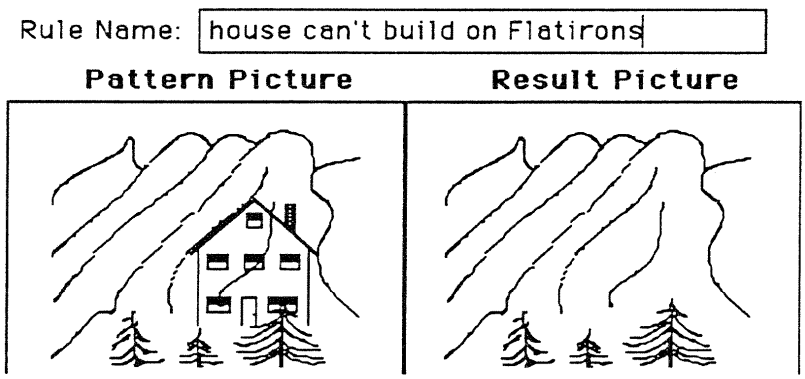


Figure 3-9: Example of a **deletion action** to delete any house in the Flatirons.

addition action: [most simulations]

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 3-10)

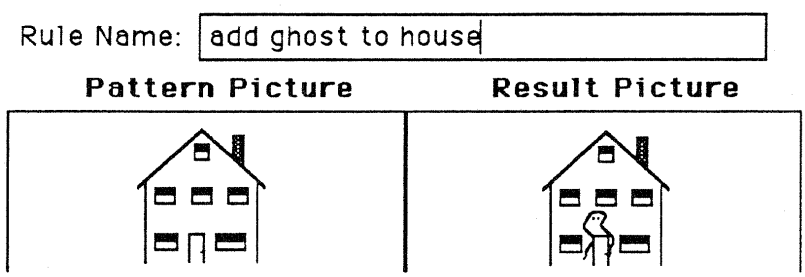


Figure 3-10: Example of an **addition action** in a rule to add a ghost to every house.

replacement action: [most simulations]

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 3-13)

movement action: [most simulations]

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 3-11)

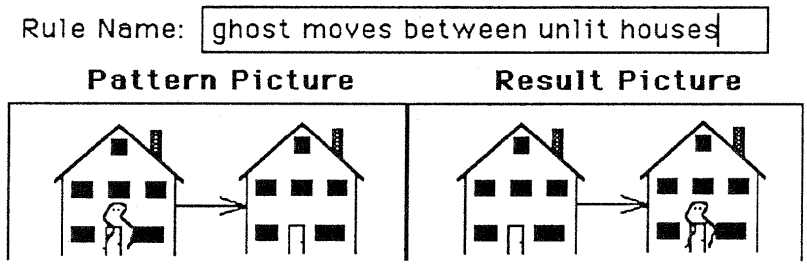


Figure 3-11: Example of a **movement action** in a rule to move ghosts between unlit houses.

wildcard match: [bburner]

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 3-10 & 3-12)

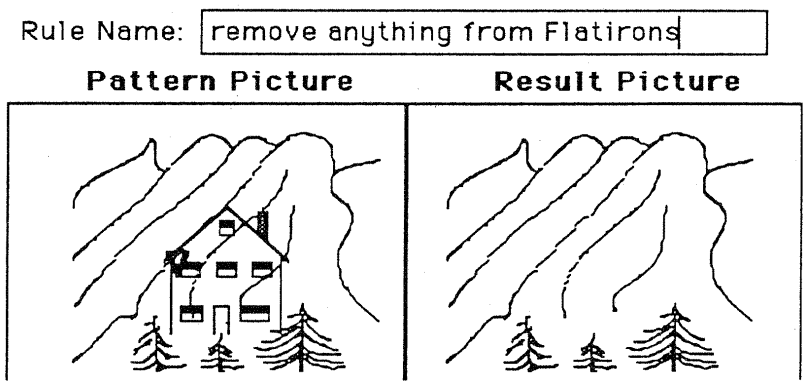


Figure 3-12: The variablized house is an example of a **wildcard match**, and the rule is an example of a **deletion action**. This rule will delete any object in the Flatirons.

identical variable match: [most simulations]

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 3-13)

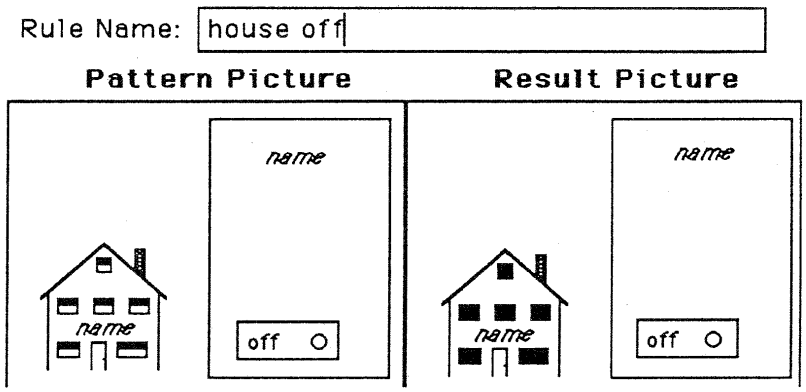


Figure 3-13: The italicized "names" are an example of an **identical variable match**, and the switch from a lit house to an unlit house is an example of a **replacement action**.

variable addition action: [most simulations]

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 3-14)

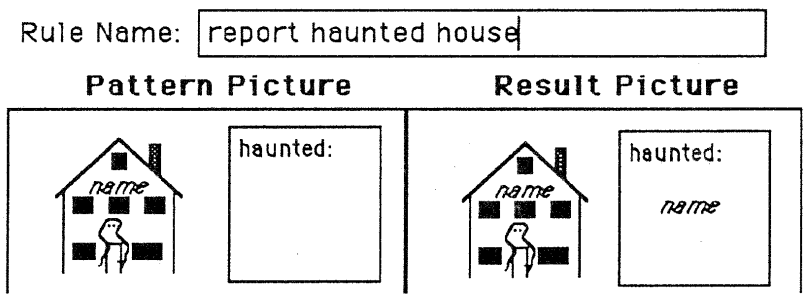


Figure 3-14: The italicized name is an example of a **wildcard match** and the new name in the result is an example of a **variable addition action.**

3.3.2 Creating Objects and Containers

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

draw additional container: [bburner, ttt, checkers, dice]

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 3-15)

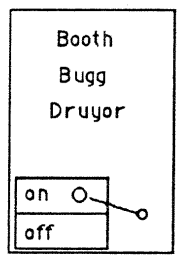


Figure 3-15: The outer rectangle is an example of an **additional container** needed to contain the names of houses that can have lighting power.

draw position containers: [maze, grasshoppers]

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: [tm]

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: [ttt]

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 3-16)

Figure 3-16: The "Bell" and "Druyor" labels are examples of **unique identifiers**.

draw mode description: [ttt, checkers]

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 3-17)

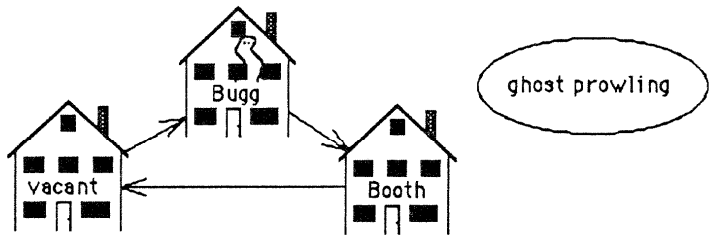


Figure 3-17: The "ghost prowling" label and associated container is an example of drawing a **mode description**. The connections between the houses are an example of **draw directed path**.

mode condition: [ttt, checkers]

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 3-18)

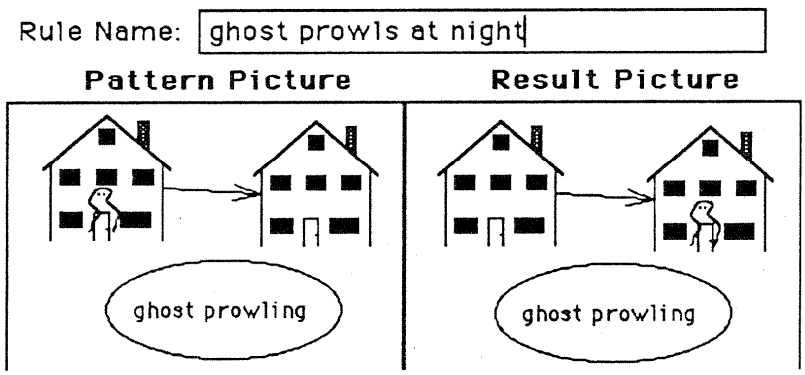


Figure 3-18: Using the "ghost prowling" text string and the oval is an example of a **mode condition** in a rule.

draw empty container marker: [bburner, ttt, checkers, count traffic, maze]

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 3-19) Follow the absence testing advice to create rules that use the empty container marker.

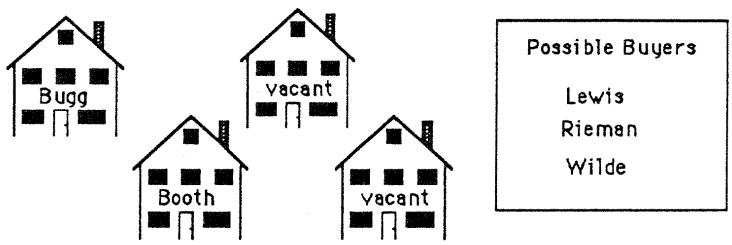


Figure 3-19: The "vacant" labels in the two houses are text string objects that signify that a house is empty. These labels may be used to determine whether a container is empty.

absence testing: [bburner, ttt, checkers, count traffic, maze]

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 3-20)

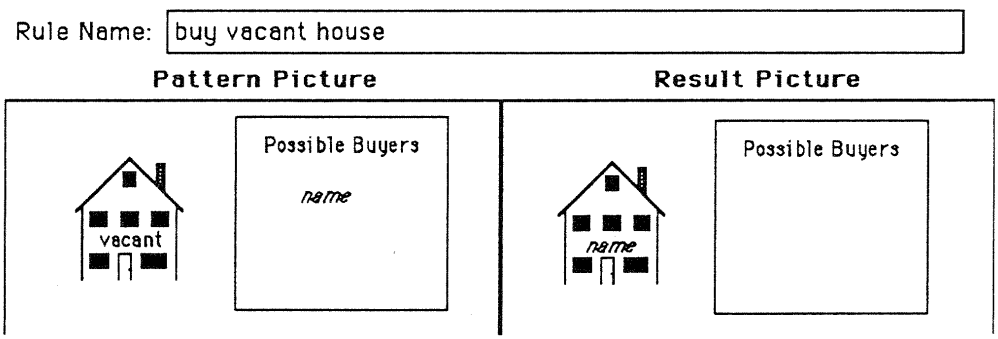


Figure 3-20: This rule tests for the absence of a family in a house by testing for the presence of the "vacant" string.

hide objects: [most simulations]

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.

3.3.3 Creating Paths

This advice describes the situations for creating paths between places.

draw directed path: [protocol, tm, fsr, logic circuit, counter]
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 3-20)

draw non-directed path: [maze, houses connected, grasshoppers]
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 3-21)

Figure 3-21: The three houses are connected with **non-directed paths** so that objects may travel between them in either direction.

draw two directed paths: [tm, arrows]
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 3-22)

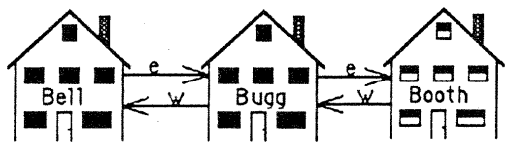


Figure 3-22: The three houses are connected with **two directed paths**, so that objects may travel east or west between the houses on different conditions.

label path: [tm, fsr, arrows]

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 3-22, 23, 24, & 25)

label common paths: [tm, arrows]

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 3-23)

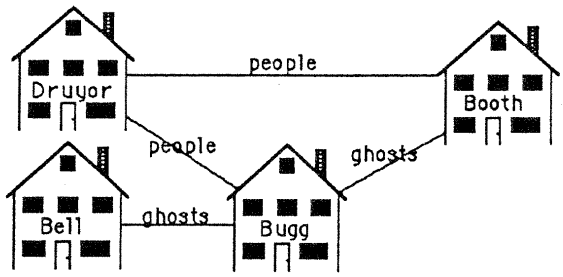


Figure 3-23: These paths are labeled based on what may pass between them. Rules may be created to only allow ghosts to move along "ghosts" paths, and people to move along "people" paths. This is an example of **labeling common paths.**

draw paths for representation: [arrows]

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 3-24)

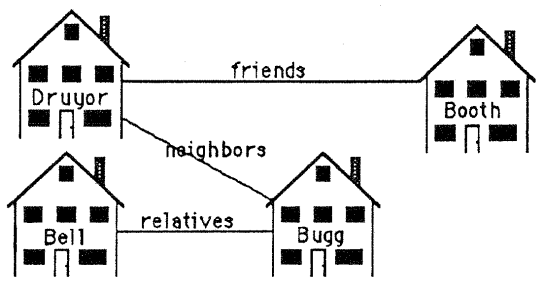


Figure 3-24: These houses are connected based on relationships between the families. This labeling is an example of **drawing paths for representation.**

draw paths for control: [houses connected]

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 3-25)

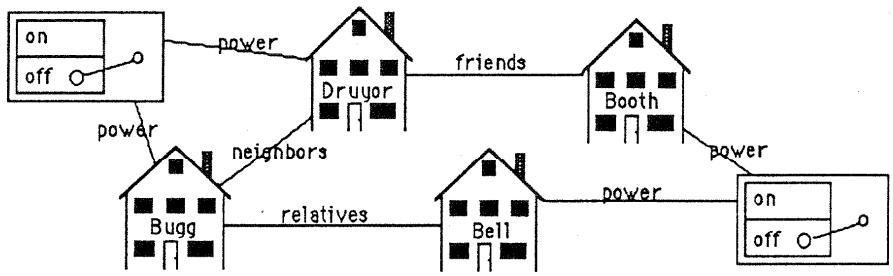


Figure 3-25: The labeled paths between the switches and the houses represent power lines. These paths are example of **paths that may be used for control** purposes.

draw path with a stopover: [protocol]

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 3-26)

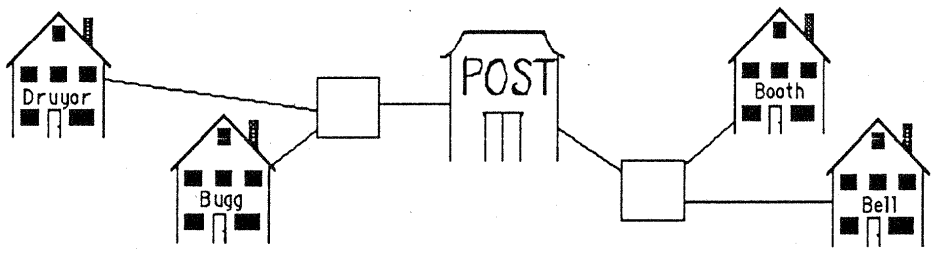


Figure 3-26: The rectangles between the houses and the post office represent intermediate stopovers in the connection.

3.3.4 Supporting End User Commands

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

draw control panel: [bburner, houses]

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 3-27)

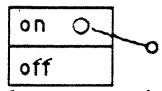


Figure 3-27: Control panel with two settings for controlling house lighting.

draw command list: [ttt, counter, dice]

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 3-28)

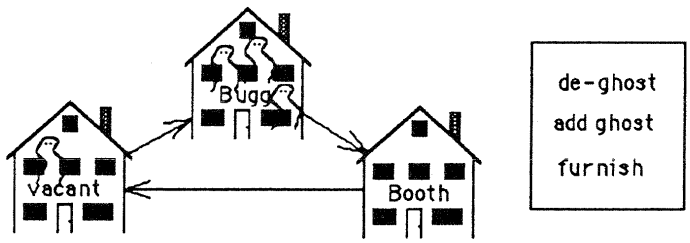


Figure 3-28: The rectangle and the three commands inside it are an example of a **command list**. These commands may be dragged into the houses by the end user.

create command rule: [most simulations]

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 3-29)

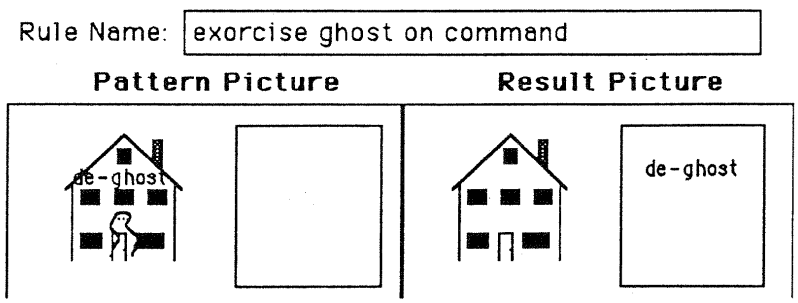


Figure 3-29: A **command rule** to remove one ghost from a house when the "de-ghost" command is placed in the house. The rule also puts "de-ghost" back in the command list.

3.3.5 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: [most simulations]

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: [protocol, dice]

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 3-30)

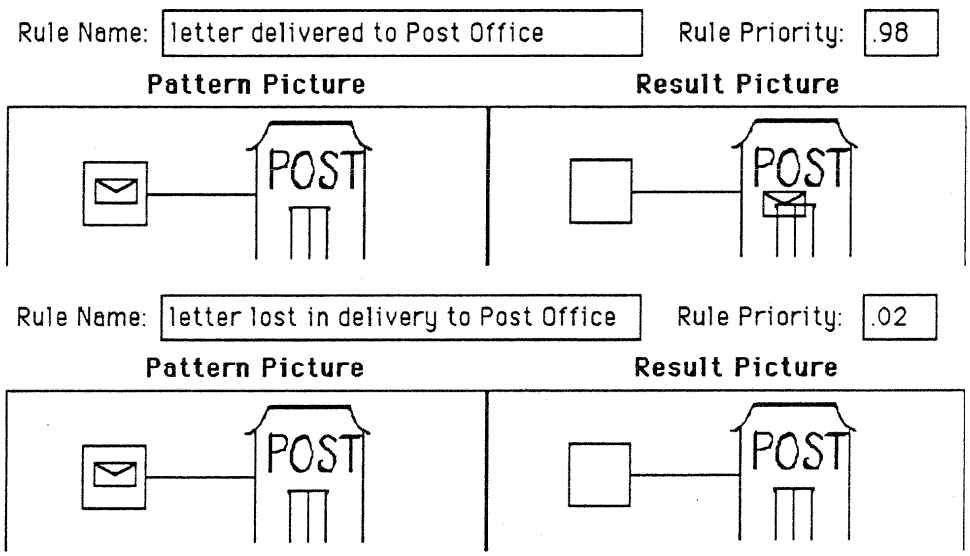


Figure 3-30: These two rules will randomly deliver 98% of the letters sent to the post office and will lose the remaining letters.

populate randomly: [grasshoppers]

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 3-31)

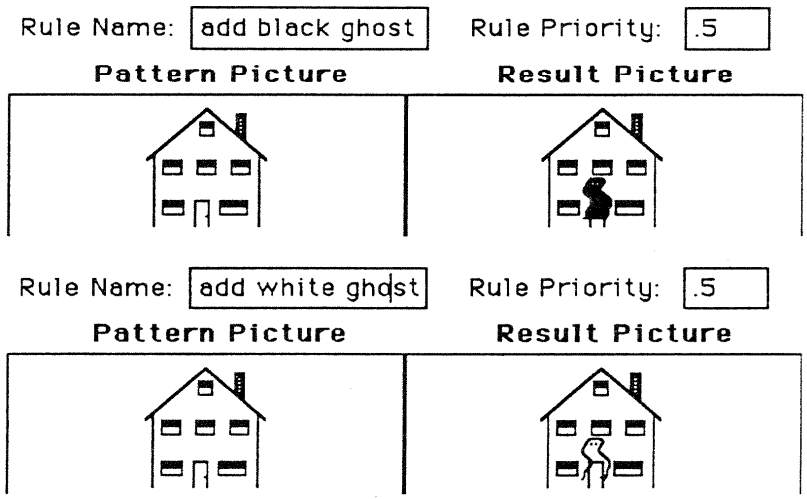


Figure 3-31: These two rules will populate each house with an individual ghost. Each house has a 50/50 chance of receiving a white or black ghost.

enable rule to continually fire: [protocol]

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.

3.3.6 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: [grasshoppers]

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 3-32)

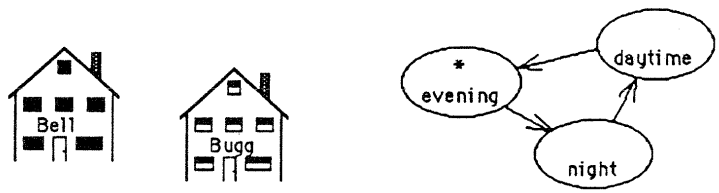


Figure 3-32: The three ovals with named labels represent a looped control sequence of tasks. The "*" represents the current task.

sequence structure condition: [grasshoppers]

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 3-33)

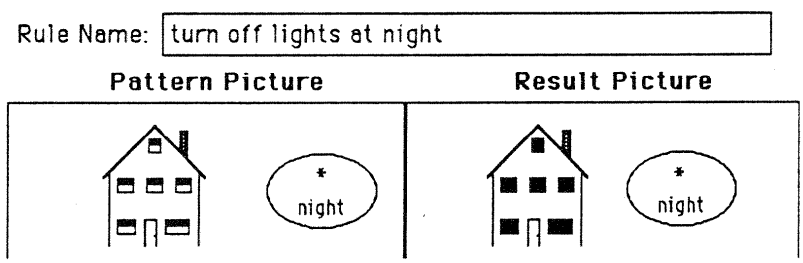


Figure 3-33: This rule forces a house to turn off its lights at night.

3.3.7 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: [arrows, checkers, grasshoppers]

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 3-34)
The "create grid" menu item provides an interface for describing grids.

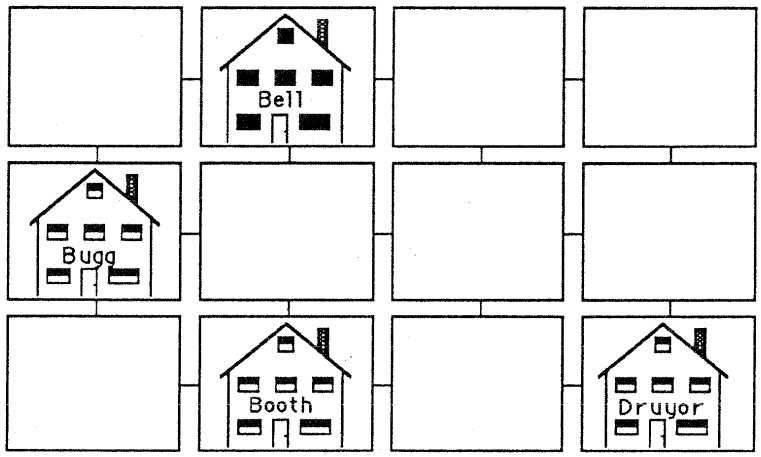


Figure 3-34: This is a 3x4 grid of containers that represents lots that houses may be placed on. Adjacent squares are connected in this grid with non-directed paths.

fill grid: [arrows, grasshoppers, checkers]

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: [fsr, tm, counter, protocol]

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 3-35)

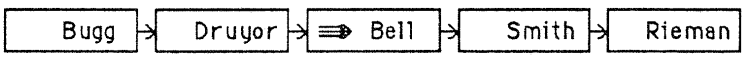


Figure 3-35: This is a five element sequence connected by directed paths to show the order of processing. A marker has been created and currently points to the "Bell" item.

3.3.8 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: [protocol]

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 3-36)

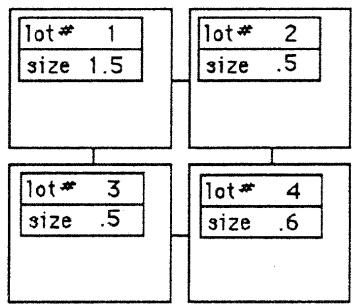


Figure 3-36: The four main containers represent adjacent house lots. Two properties are defined within each lot container, the lot# and the size of the lot in acres.

draw attribute-value table: [tm, houses with switch box]

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 3-37)

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

Figure 3-37: 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: [tm, houses with switch box]

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 3-38)

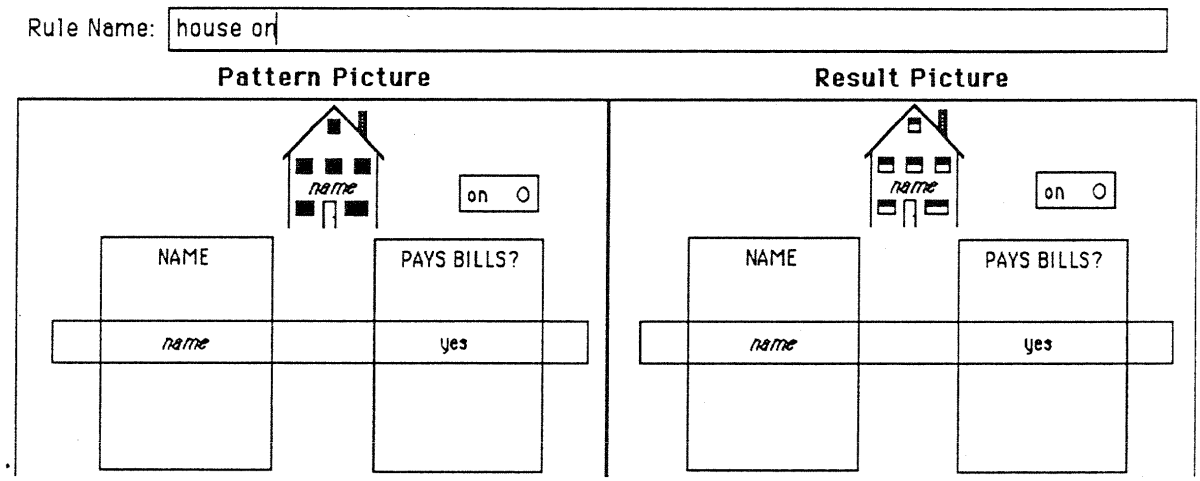


Figure 3-38: The pattern matches when a button is in the "on" position, there is an unlit house, there is an entry of the table that has the house name, and that entry shows that the person is paying the bills.

draw pairing structure [arrows, dice, protocol]

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.

3.3.9 Defining Sets

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

draw set definition: [checkers]

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 3-39)

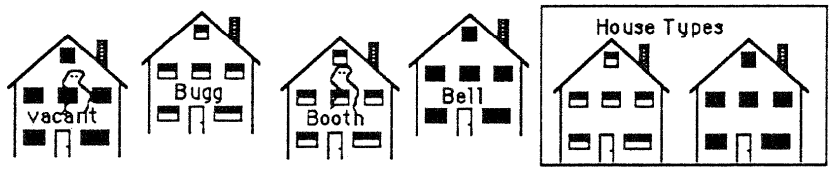


Figure 3-39: The rectangle and its contents is an example of a set definition. The rectangle contains both kinds of houses, defining the set of houses.

set condition: [checkers]

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 3-40)

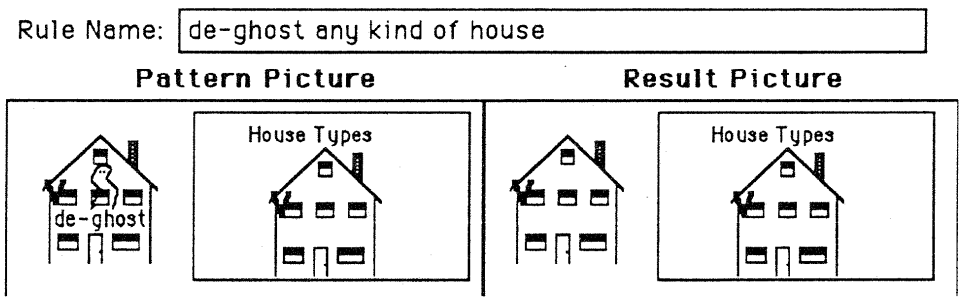


Figure 3-40: This rule removes a ghost from any kind of house, as long as that kind of house is defined in the "House Types" rectangle. The lit houses in this rule are all variablized, allowing them to match either type of house.

3.3.10 Counting Events

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

count occurrences: [count traffic, protocol, grasshoppers]

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 3-41) The "decr" and "clear" labels may be used in the same way to decrement or clear.

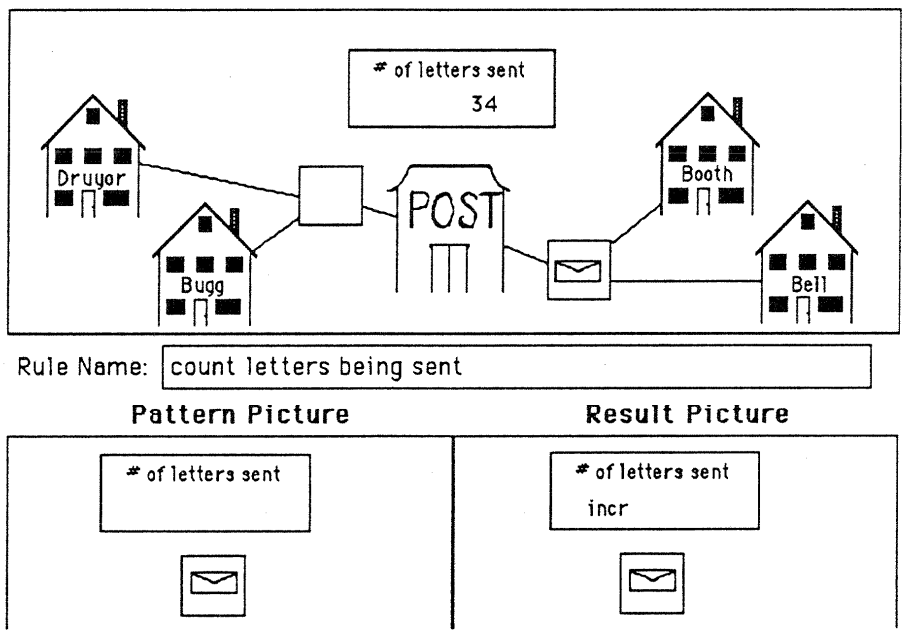


Figure 3-41: The simulation display shows a container that displays the number of letters that have been sent. The rule counts letters by placing the "incr" text string in this container, whenever a letter appears in the box connecting to the post office.

count multiple occurrences: [count traffic, grasshoppers]

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: [count traffic with bar graph output]

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

3.4 The Programming Environment

This section describes some of details and idiosyncrasies of the ChemTrains programming environment. Figure 3-42 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.

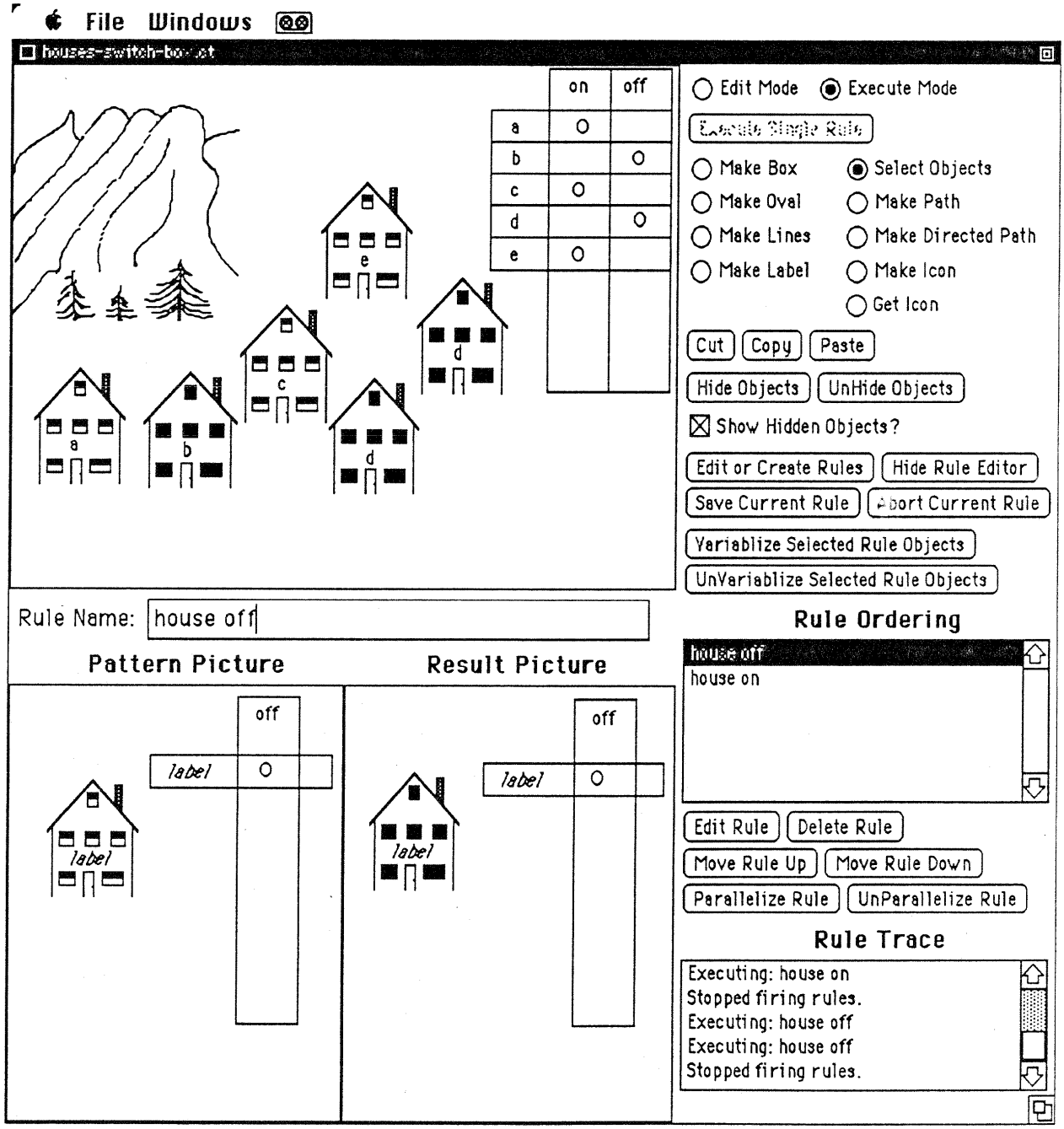


Figure 3-42: The ChemTrains Programming Environment.

3.4.1 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.

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.4.5 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.

3.4.6 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.

3.4.7 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.

3.4.8 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.

3.4.9 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.

3.4.10 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 3-43 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 bouncing arrows problem (figure 4-4) is an example of a grid with paths. The "[Re]Display Grid" action redisplays the grid if there are any changes. The "Done" action exits from this interface.

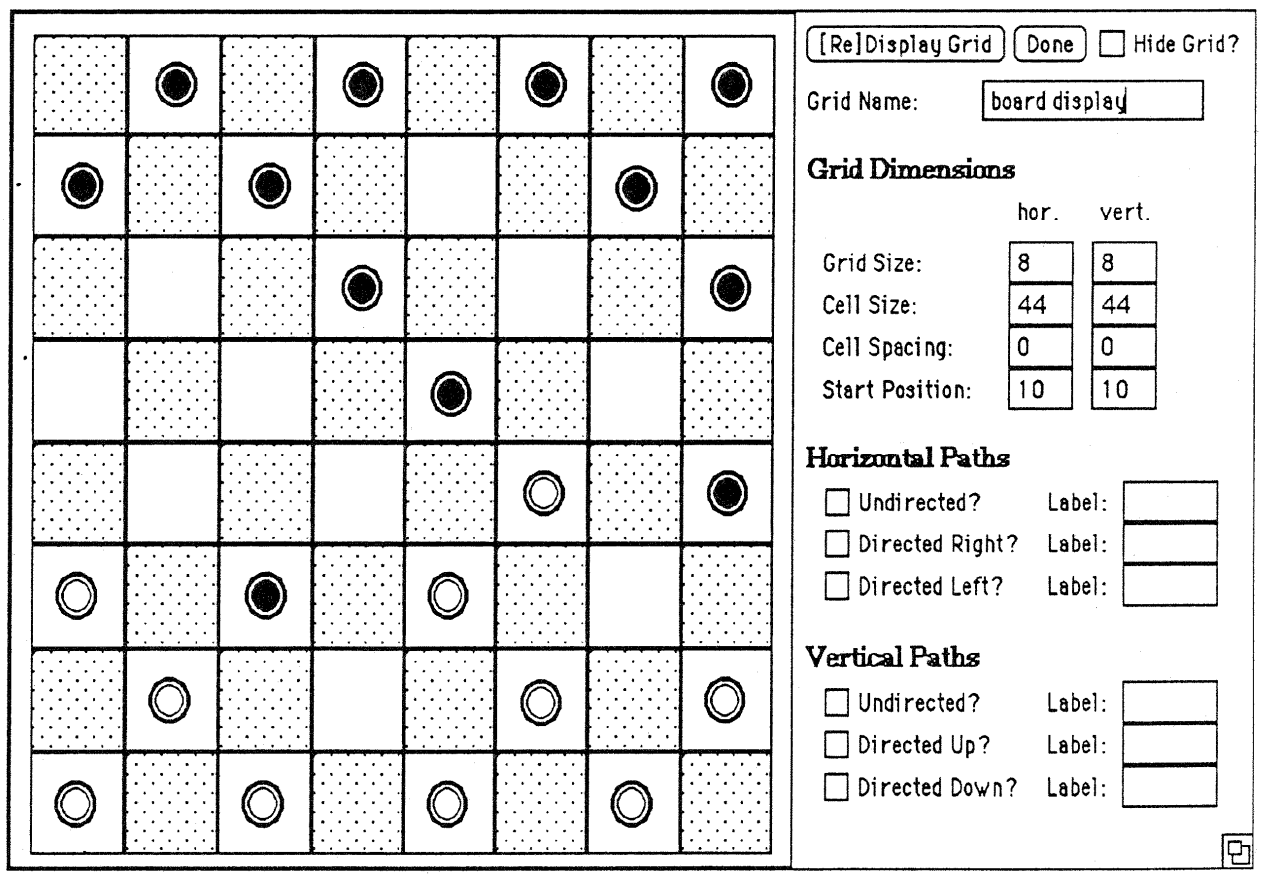


Figure 3-43: Environment for editing a single grid.

4 Problems, Solutions, and Walkthroughs

This chapter is a complete programming walkthrough evaluation of the final design of the ChemTrains91 language as described in the previous chapter. The complete evaluation includes individual evaluations on each of the target problems used in guiding design. Each problem description includes:

- the statement of the target problem,
- a likely solution to the problem, and
- a walkthrough summary.

In addition, some of the problem writeups also include the complete programming walkthrough as an example. This chapter concludes with walkthrough summaries that point out shortcomings in the language.

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.

1. **Simple switch problems** test the ability to control different parts of the simulation through switches.
2. **Maze problems** test the ability of objects to move around the screen with various strategies for wandering or searching.
3. **Automata problems** test for computational power.
4. **Game playing problems** test medium scale problems that require a variety of simulation techniques.
5. **Simple counting problems** test the ability to count things as they occur and show the numeric results.
6. **The rolling dice problem** is a simple problem that tests the ability to simulate random behavior.
7. **Network protocol problems** tests the ability to simulate a complicated procedure.
8. **The grasshopper problem** tests another complicated behavior.

The walkthroughs of the simple counting problems (section 4.5) are done on an intermediate design rather than the final design.

4.1 Simple Switch Problems

These target problems test a simple and important aspect of a simulation language: the ability to describe a direct relationship between the setting of a switch and the display of other things on the screen. Variations of the house lighting problem, shown in the previous chapter, test the ability to control

multiple pictures from a single two-way switch. The bunsen burner problem tests the ability to simulate a three-way switch.

4.1.1 House Lighting Problems

House Lighting Problem: Show a bunch of houses and a single on/off switch. When the switch is turned on, lights are turned on in all the houses. When the switch is turned off, lights are turned off in all the houses.

House Lighting with wired switches: Show a bunch of houses and a few on/off switches. Allow different switches to govern the lighting of different sets of houses. Switches are wired to houses that they control.

House Lighting controlled by switch box: Show a bunch of named houses and a single switch box whose switches are named. Switches control the lighting of the houses when the name of the house is the same as the switch name.

4.1.2 House Lighting Solutions

Figure 4-1a shows the screen of a solution to the simple house lighting problem. This solution has two rules: one to change a lighted house with an unlighted house when the switch is off, and another to change an unlighted house with a lighted house when the switch is on.

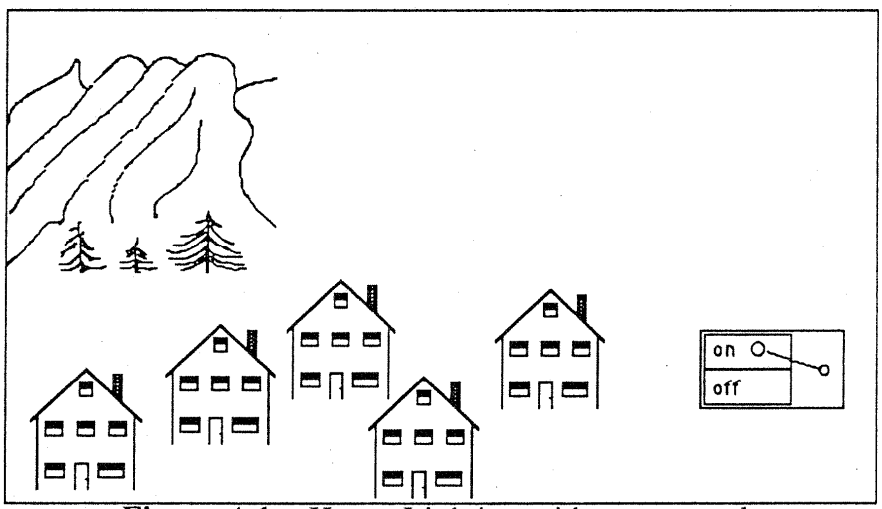


Figure 4-1a: House Lighting with one control.

Figure 4-1b shows a solution to the house lighting with wired switches problem. Paths connect between the switches and the houses that they control. Figure 4-1c shows one of the two needed rules, a rule to turn a house light off when a switch in the off position is connected to it.

Figure 4-1d shows the solution to the house lighting controlled by switch box problem. The lighting of each house is controlled by the identically labeled switch. Figure 4-2e shows the rule to turn a house light on. The label of the house and switch in the rule is a variable. The two columns of the switch box are rectangles, and each row of the switch box is an individual rectangle.

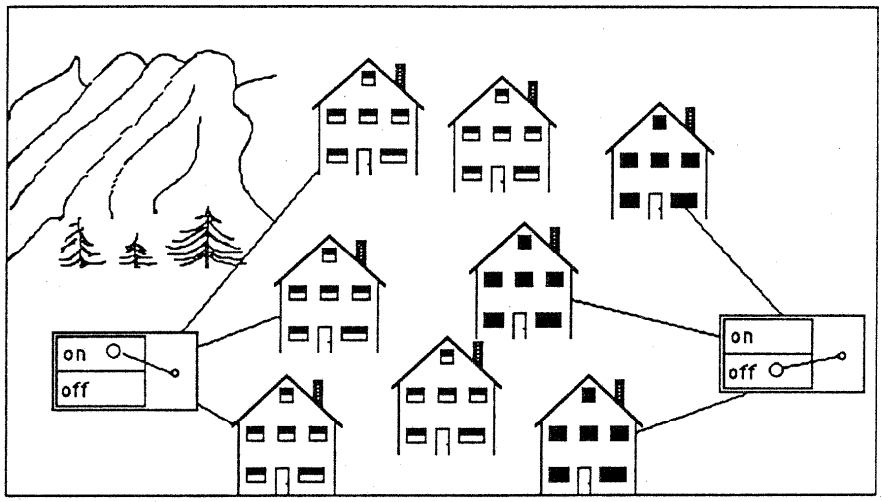


Figure 4-1b: Connected House Lighting.

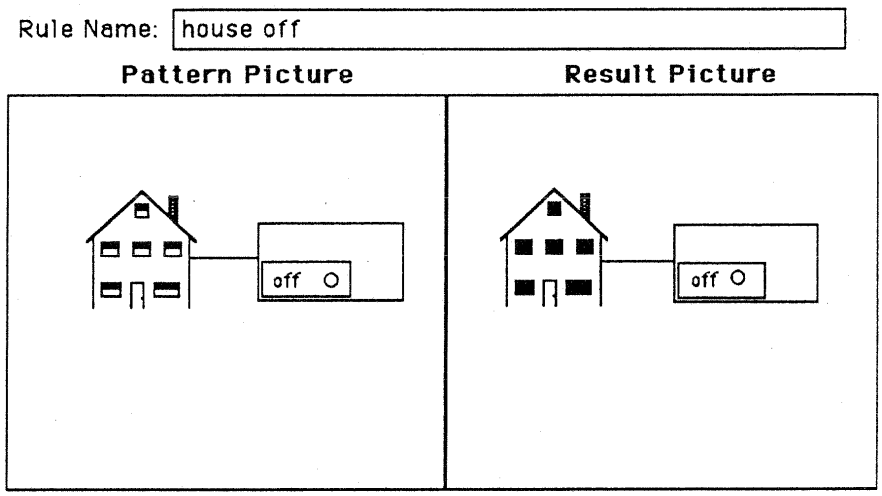


Figure 4-1c: Connected House Lighting Rule.

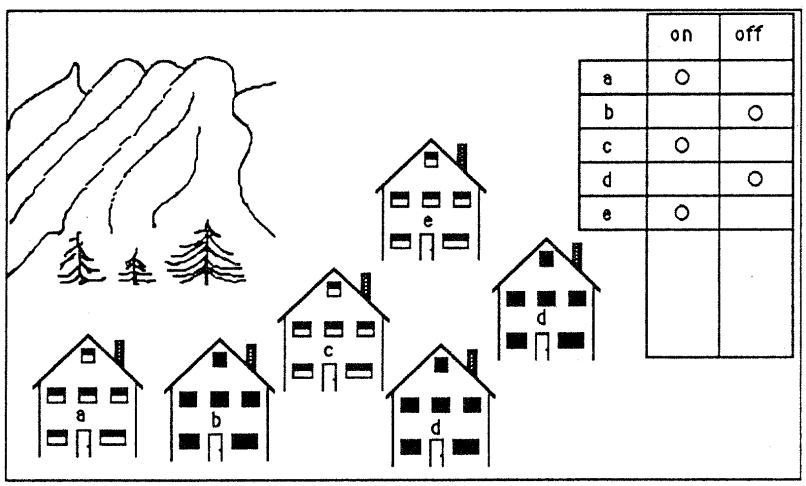


Figure 4-1d: House Lighting controlled by switch box.

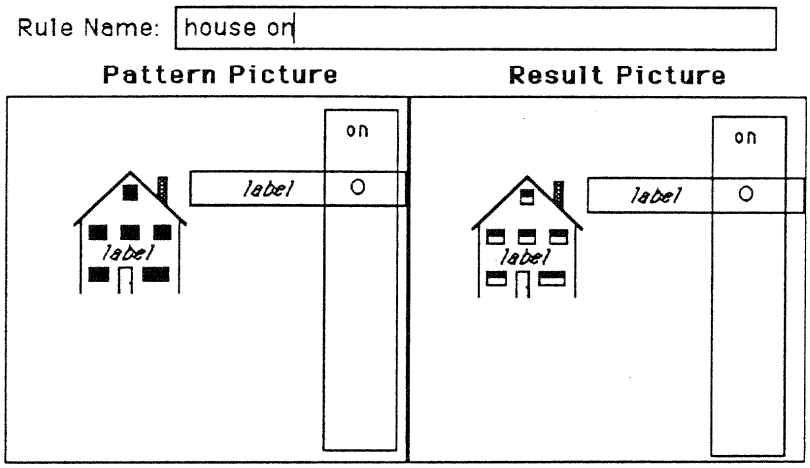


Figure 4-1e: House Lighting switch box controlling rule.

4.1.3 House Lighting Walkthrough and Summaries

Here is a walkthrough of the house lighting problem with wired switches (the solution shown in figure 4-1b&c).

draw initial picture: the houses, with lights on.
draw control panel: an on/off switches with a container holding each whole switch.
draw paths for control: connect the switches to the houses that they control.
create rule: named "house off," copy a house, a connected switch, and the connection into the pattern and result of the rule.
  replacement action: replace the lighted house with an unlighted house.
  addition action: connect the switch to the new house.
create rule: named "house on," copy a shaded house, a connected switch, and the connection into the pattern and result of the rule.
  replacement action: replace the unlighted house with an lighted house.
  addition action: connect the switch to the new house.

This programming walkthrough shows that simple doctrine applied to the problem leads to a good solution. The step of drawing a path between houses and switches is additionally supported by a strong similarity between the domain (wires) and the language (paths).

A programming walkthrough of the simple house lighting problem is also trivial.

A programming walkthrough for the house lighting controlled by switch box problem (shown in figure 4-1d&e) is trickier than the previous walkthrough because doctrine does not adequately support the solution. Doctrine for creating tables exists, but no doctrine explains how tables can be used as control switches. This doctrine could be added, but it may only benefit this specific solution.

4.1.4 The Bunsen Burner Problem

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 inside 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.

4.1.5 Bunsen Burner Solution

Figure 4-2 shows the main display of the solution and two of the six rules needed in the solution. The main display contains a control panel, a container for the flame, a beaker, and a substance in the beaker. The end user may control the flame by moving the control knob between the three settings. The first rule, "turn-flame-to-hi," recognizes when the knob is in the "hi" position, and changes the flame in the flame holder to a high flame. The "V" labeling the low flame in the pattern denotes that this is a variable object, allowing this rule to work when anything is in the flame container. The second rule shown here changes the substance in the beaker to steam when there is a high flame.

4.1.6 Bunsen Burner Walkthrough and Summary

draw initial picture: the bunsen burner, the beaker, and a substance in the beaker.
draw control panel: made up of three separate rectangles.
draw additional container: that is to contain the flame.
draw empty container marker: the text string "no flame" inside of the flame holding rectangle.
create rule: named "Turn Flame to Low"; copy the "low" rectangle of the control panel and the rectangle holding the flame onto the pattern and result pictures.
wildcard match: specify that the "no flame" object in the flame holder is a variable object.
replacement action: replace the "no flame" marker with a low flame in the result picture.
create rule: named "Turn Flame to high";
follow the same steps as in making "Turn Flame to low."
create 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.
wildcard match: specify that the "no flame" object in the flame holder is a variable object.
create rule: named "Change to liquid"; copy the beaker and the rectangle holding the flame onto the pattern and result pictures.
wildcard match: specify that the ice cube object in the beaker is a variable object.
replacement action: replace the ice cube in the beaker with an object that looks like liquid in the result of the rule.
create rule: named "Change to gas";
follow the same steps as in making "Change to liquid" ...
create rule: named "Change to solid"
copy the beaker and the rectangle holding the flame onto the pattern and result pictures.
wildcard match: specify that the ice cube in the beaker is a variable.

The toughest parts of this walkthrough are:

- **draw additional container** guides the creation of a flame container, but it is tricky because the container is an artificial part of the display that is only needed to hold a changing flame;
- **draw empty container marker** guides the creation of another artificial object, the "no flame" marker; and
- **wildcard match** abstractly guides the variablization of the flame in each rule.

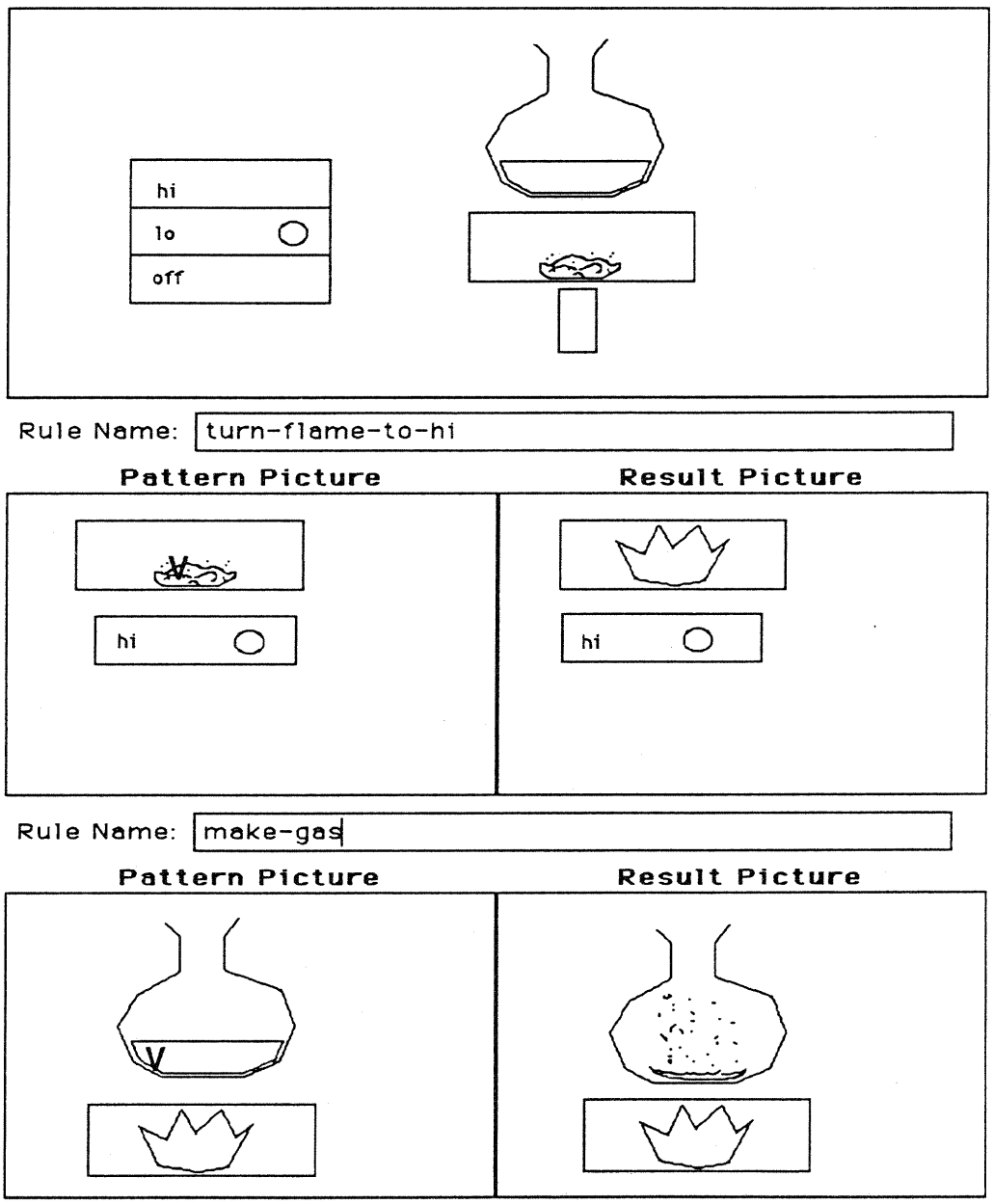


Figure 4-2: Bunsen Burner Simulation & hi flame rules.

4.2 Maze Problems

These problems test the ability for the language to simulate things moving around on the screen with different kinds of behaviors. Three target problems are given here: the maze search problem, the maze wander problem, and the bouncing arrows problem. The maze search problem tests whether a mouse can easily keep track of where it's been in a maze. The maze wander problem tests whether the mouse can behave more similarly to a real mouse. The bouncing arrows problem can be easily described in BITPICT, providing a useful comparison between the two languages.

4.2.1 The Maze Search 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.

4.2.2 Maze Search Solution

Figure 4-3a shows a mouse in the process of searching a maze for cheese. The mouse had started in the lower right hand corner.

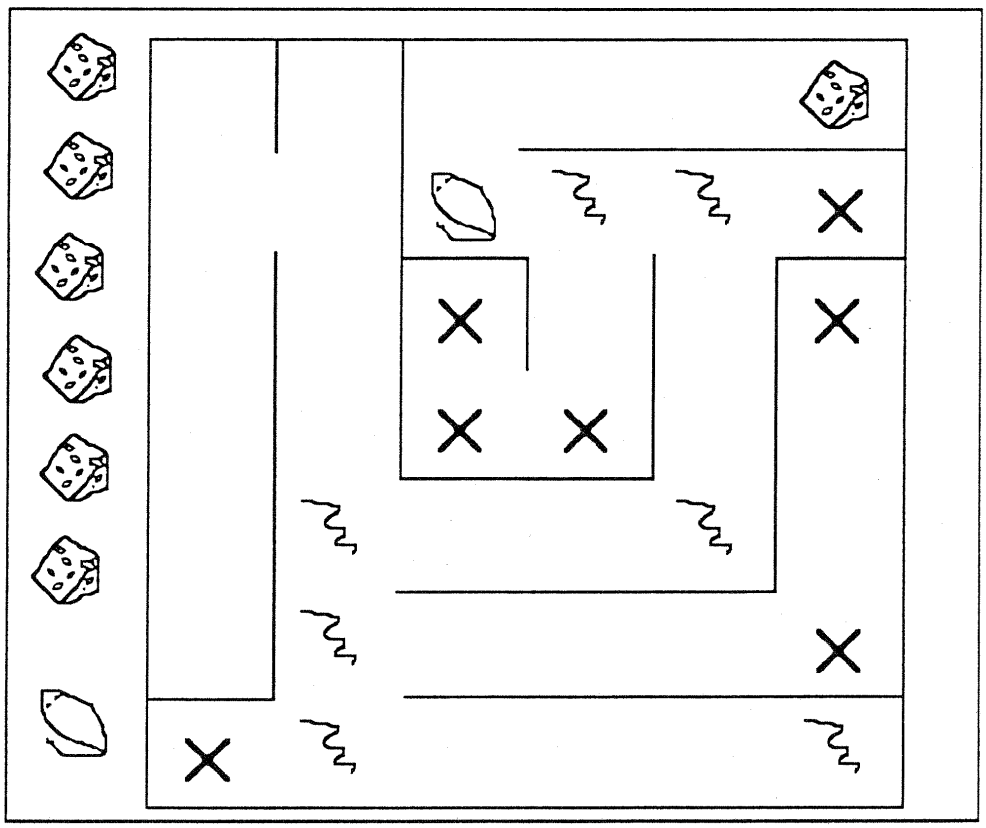


Figure 4-3a: Display of Maze Simulation during execution.

The solution to the maze search problem simulation shown in figure 4-3a 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 4-3b shows the maze simulation when the mouse has completed the search, and shows the hidden places and connections in the maze.

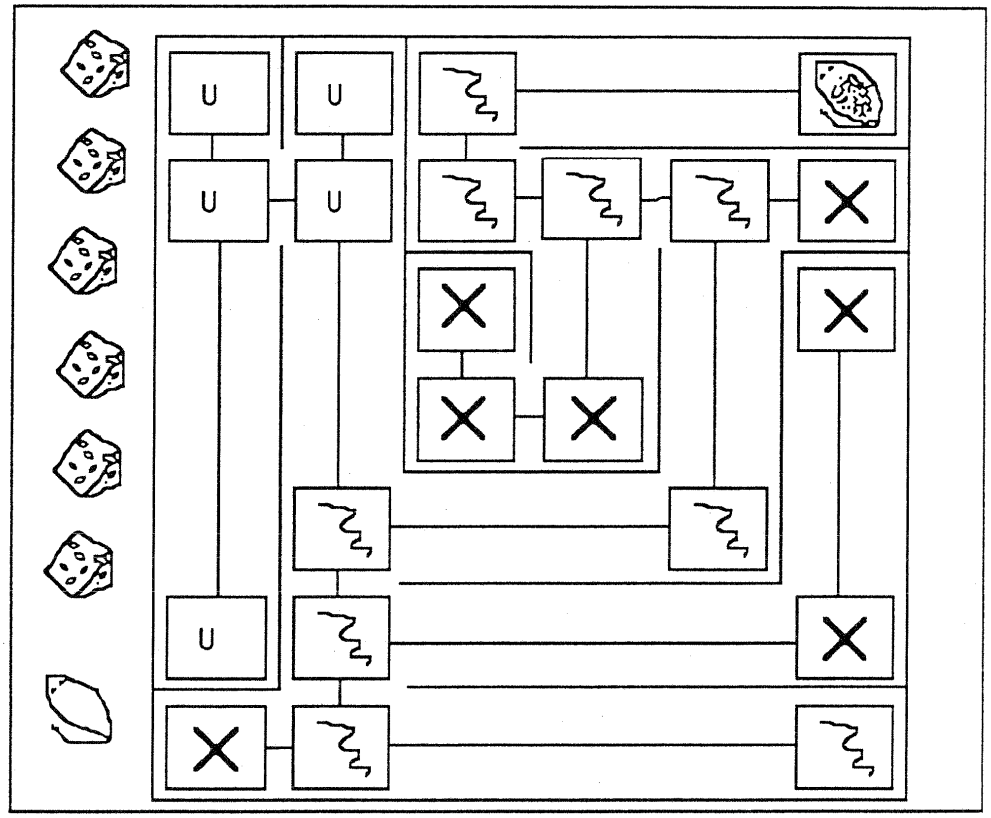


Figure 4-3b: Display of Maze with hidden places and paths.

The three rules required in the maze search problem are shown in figure 4-3c. 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.

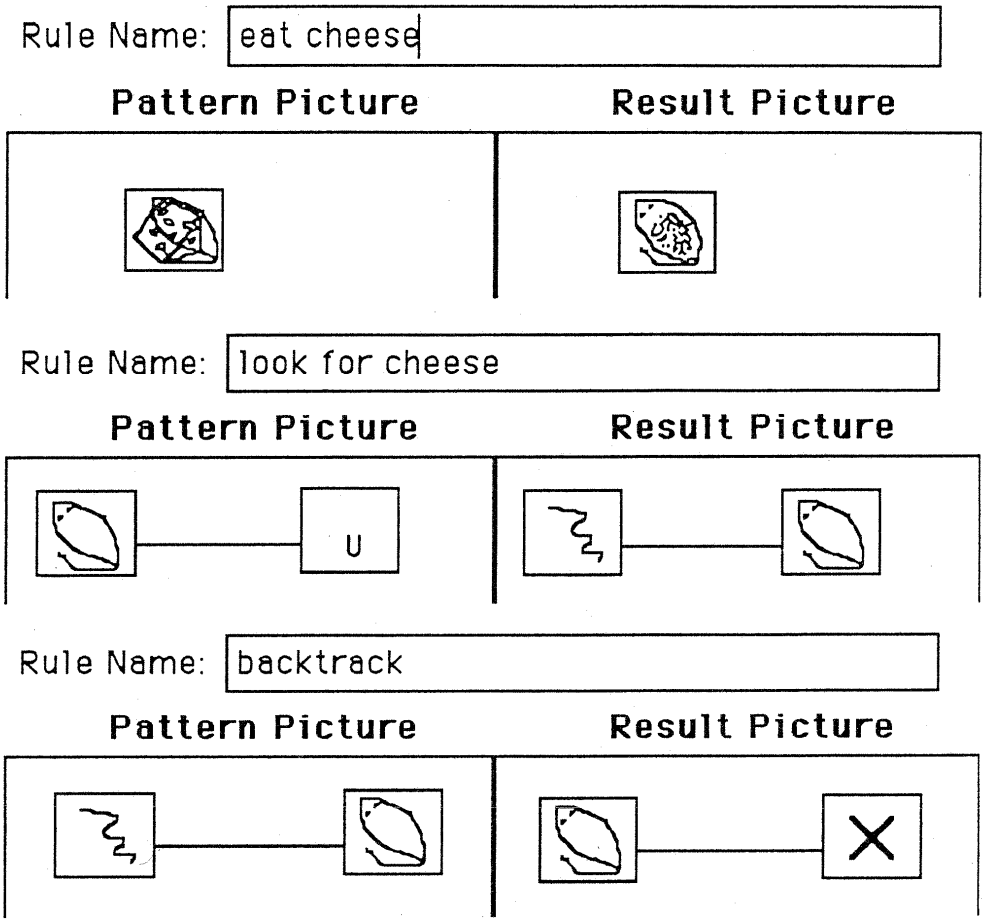


Figure 4-3c: Maze Searching Rules.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw position containers** guides the creation of the maze decision places;
- **draw non-directed path** guides the creation of paths between the places; and
- **draw empty container marker** guides the creation of the unseen marker.

The creation of the appropriate rules is guided by advice contained in the target problem.

4.2.3 The Maze Wander Problem

Show a simple maze, with one or more mice and one or more pieces of cheese. The mouse should move randomly through the maze. If it reaches a dead end, it should turn around, and should only turn around at a dead end. When a mouse sees cheese it should eat it. Mice should also avoid each other.

4.2.4 Maze Wander Solution

The solution to this problem requires a representation of the maze identical to the last problem. In order to force a mouse not to backup over its previous position, a marker needs to follow each mouse to keep track of this. The following rules are needed:

- mouse eat cheese,
- move a mouse forward,
- move a mouse away from another mouse, and
- move a mouse out of a dead end.

The tricky part of this solution is inventing the previous position marker, which is not guided by any particular doctrine.

4.2.5 The Bouncing Arrows Problem

Show any number of arrows in a display traveling in the direction that they are pointed. An arrow may point right, left, up, or down. When an arrow hits a blockade it must reverse direction. For example, a right arrow switches to a left arrow when it is blocked on the right.

4.2.6 Bouncing Arrows Solution

One possible solution to the bouncing arrows problem is shown in figure 4-4. The display includes a grid of places that may contain an arrow or a blockade. Each place in the grid connects to its four adjacent places with labeled directed paths. The display also contains two tables. One table defines what arrow displays are to be used when an arrow bounces (the up & down arrows are together, and the right and left arrows are together.) The other table defines what path direction each of the arrows should follow. Two rules are needed: one for moving an arrow forward, and one for reversing an arrow.

The ChemTrains solution is not very intuitive especially considering how simple the problem is. This solution can be compared with the BITPICT solution, described in chapter seven, Language Comparisons. The tricky parts of the ChemTrains solution are guided by the following doctrine:

- **draw grid** guides the creation of a two dimensional grid;
- **draw two directed paths** guides the creation of directed labeled paths going in different directions from each cell in the grid;
- **draw attribute-value table** loosely guides the creation of a table for defining the path direction that an arrow should follow;
- **draw pairing structure** loosely guides the creation of the table describing arrow opposites;
- **table driven condition** guides the use of the table in creating rules for moving an arrow forward along the appropriate path and reversing an arrow when it runs into a wall.

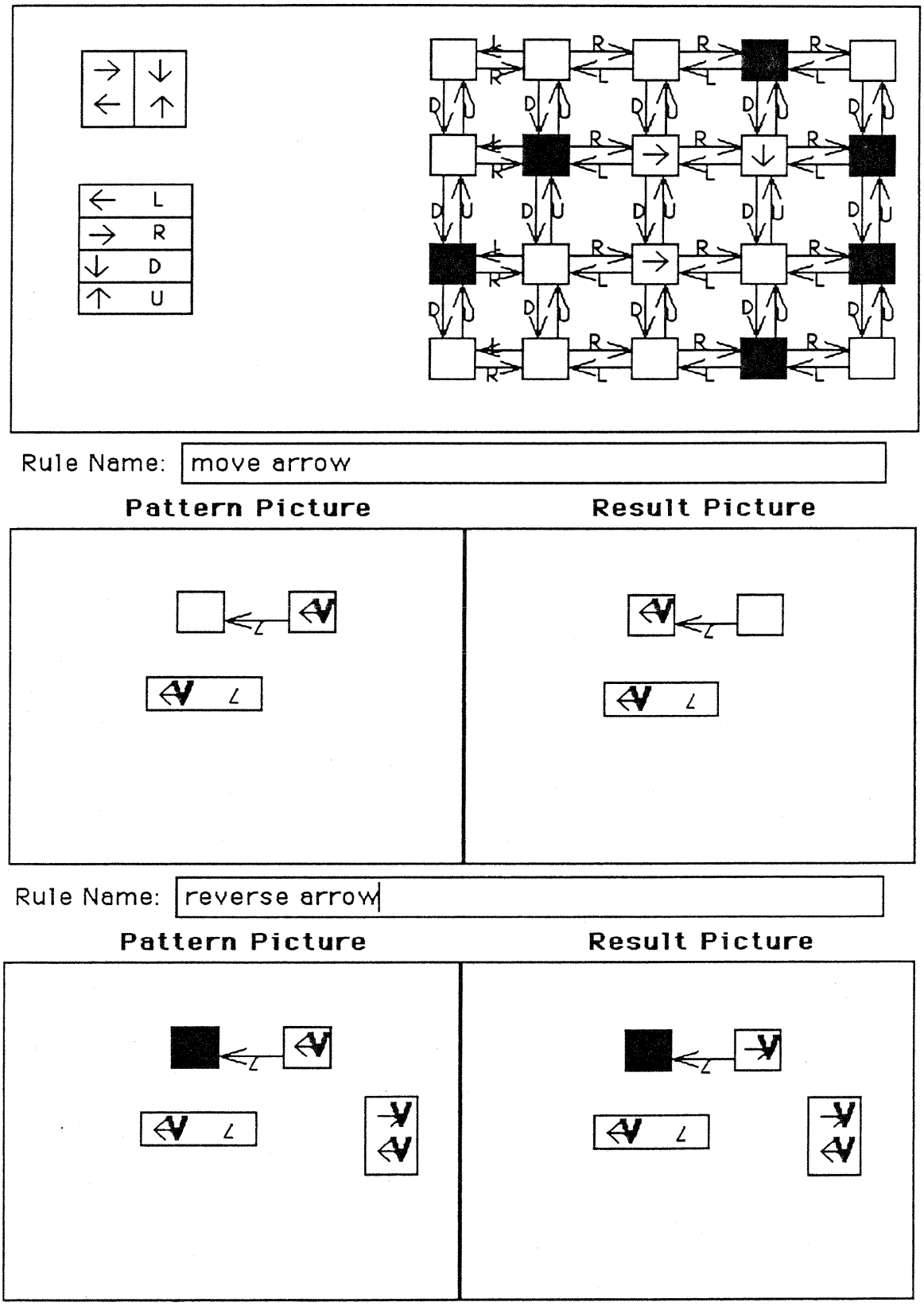


Figure 4-4: Bouncing arrows solution.

4.3 Automata Problems

These target problems test the ability to mimic general computer science automata. The finite state recognizer problem is included here because it is a common and useful automata for describing regular languages. The Turing Machine problem is added here not only to ensure that the language is Turing equivalent, but also because a Turing Machine simulator is a complex interface. The logic circuit problem is included because this problem is addressed by other visual programming languages, such as ThingLab and BITPICT, and the solutions may be compared.

4.3.1 The Finite State Recognizer Problem

Show a Finite State Recognizer that recognizes sentences from the language (001)*. Also display an example sentence in the language. The simulation should show the symbols of the sentence on an input tape being read and matched with symbols on the arcs of the FSR. A marker should display the current position within the input tape, and another marker should display the current state.

4.3.2 DFSR Solution

The solution to a deterministic finite state recognizer is straightforward. Figure 4-5 shows a finite state recognizer and an input tape. A marker is needed to keep track of the current state and the current position on the input tape. One rule traverses the input tape and a labeled arc if the label matches the current input symbol, and another rule traverses the input tape if a label in the current state matches the current input symbol. Since paths cannot connect from an object to the same object, a label inside a state is used to represent self arcs, and the second rule matches on these.

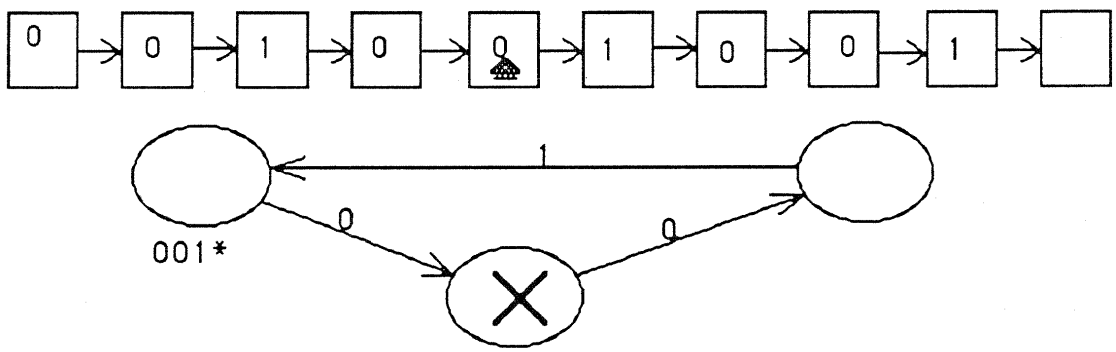


Figure 4-5: Deterministic Finite State Recognizer.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw ordered list** guides the creation of the input tape and its associated tape marker;
- **draw unique identifier** and solution information in the target problem guide the creation of the current state marker;
- **identical variable match** abstractly guides the use of variables to describe matching any arc to a matching input symbol; and

- the creation of a second rule for supporting transitions that stay in the same state is a hard step, which would be simplified if paths could connect from a place to itself.

4.3.3 The Nondeterministic Finite State Recognizer Problem

Demonstrate an NDFSR working. A nondeterministic recognizer differs from a deterministic by allowing multiple transitions to be traversed on a single input and by allowing multiple states to be current at any point in time. As the input is accepted, a single marker should display the current position within the input tape as in the deterministic recognizer, but multiple markers should display all current states.

4.3.4 NDFSR Solution

The solution to the nondeterministic fsr problem is not as straightforward as the fsr solution. The simulation must allow multiple current state markers to traverse the fsr in parallel, and should allow multiple arcs to be traversed from any given input symbol. A solution to this problem requires two different kinds of state markers. One kind of state marker is used to signify states that can be traversed on the current input symbol, and another kind of state marker is used to signify states that are to be the new current states after the current input symbol has been traversed. Additional rules are needed to delete current state markers once all the arcs have been traversed and to change the new state markers into current state markers. In addition, after traversing all appropriate arcs, there is a difficulty in simultaneously:

- deleting all current state markers,
- changing all new state markers to current state markers, and
- moving the input tape marker across one symbol.

The problem is that as current state markers are created they may be deleted because they could be considered current state markers for the previous iteration of arc traversal. A solution is to create a mode marker that forces one set operations to occur before another set of operations.

The solution to this NDFSR target problem is particularly ugly, and the doctrine does not provide much guidance:

- **draw unique identifier** abstractly guides the creation of two different kinds of state markers; and
- **draw mode description** abstractly guides the creation of a representation of the mode for deleting all current state markers and a separate mode for changing new state markers to current state markers; and
- **reorder rules** more concretely guides the user in ordering the rules so that everything happens appropriately before the input tape marker is moved across one symbol.

4.3.5 The Turing Machine 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 $0^n1^n2^n$.

General Turing Machine Problem: 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.

4.3.6 Turing Machine Solution

A possible solution to the general Turing Machine shown in figure 4-6a. 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. 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 4-6b 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.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw attribute-value table** guides the creation of the fsm arc table;
- **table driven condition** guides the use of a table row within the rules for traversing arcs;
- **draw two directed paths** guides the creation of the Turing tape representation;
- **identical variable match** abstractly guides the use of the many variables within each rule; and
- **identical variable match** very abstractly guides the use of variables in describing the direction to move the Turing tape head.

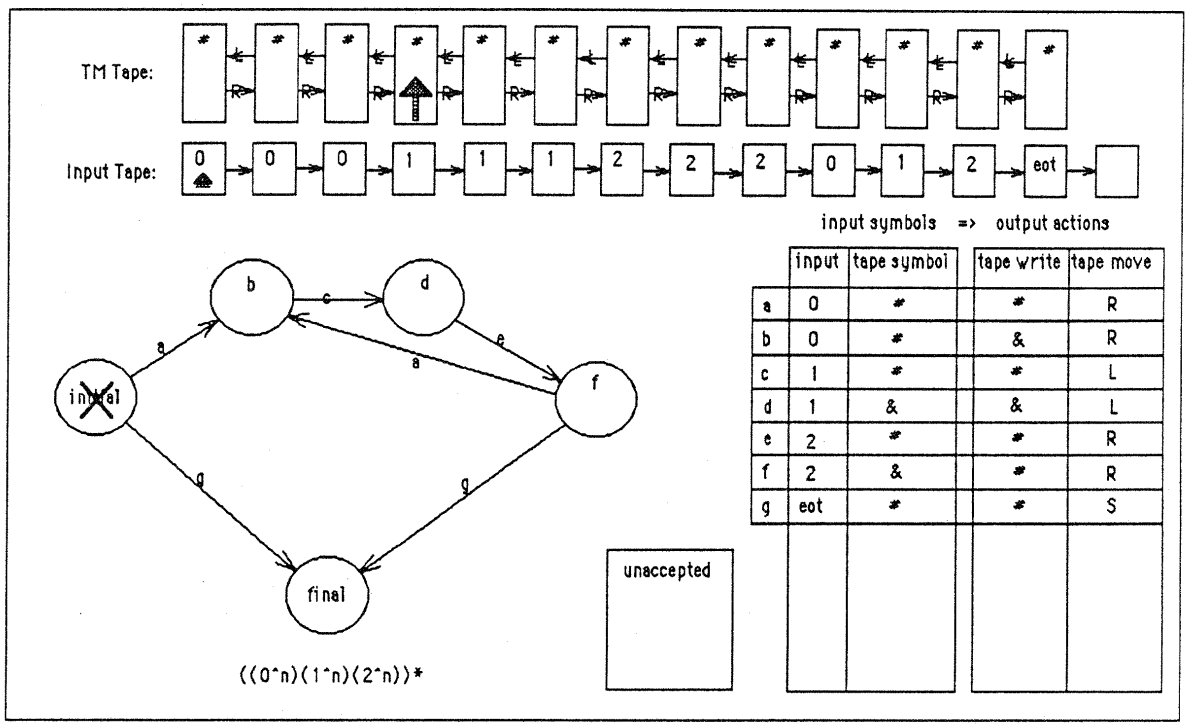


Figure 4-6a: General Turing Machine Simulation.

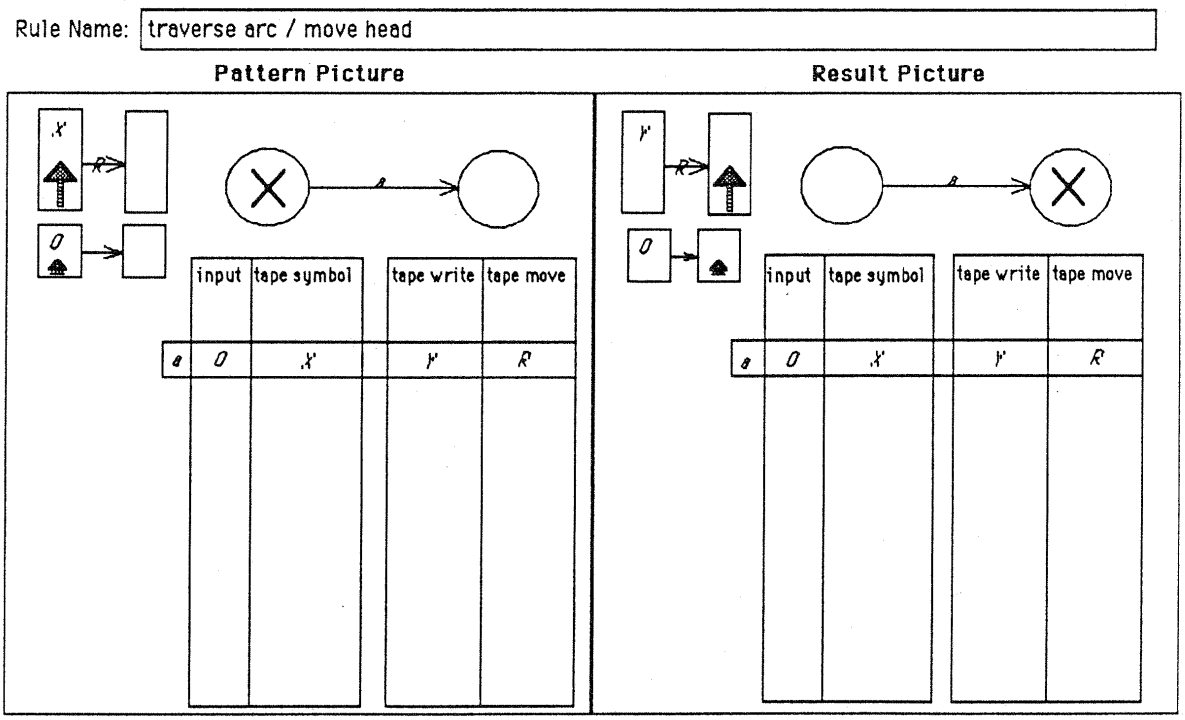


Figure 4-6b: Turing Machine Rule to traverse one transition that matches a table row, one piece of Turing tape, and one input symbol.

4.3.7 The Logic Circuit Problem

Simulate the behavior of "and," "or," and "not" gates and a simple clock. The "and" and "or" gates should receive two inputs and output one result. The "not" gate should receive one input and output one result. Inputs and outputs of the gates must be 0's (false) and 1's (true). An "and" gate should produce a 1 on output if both inputs are 1, and should produce a 0 otherwise. An "or" gate should produce a 0 on output if both inputs are 0, and should produce a 1 otherwise. A "not" gate should produce a 1 on the output if the input is a 0, and should produce a 0 if the input is a 1. A simple clock should flicker between 0 and 1. To test whether these components work, construct an xor circuit and an RS flip flop circuit.

4.3.8 Logic Circuit Solution

Figure 4-7 shows a working RS flip-flop with a simple boolean clock, and the two rules that describe the behavior of the "and" gate. The key to this solution is that the inputs and outputs of the gates are represented by places that can hold a single boolean value. A rule is created for each of the cases described in the problem. The rules describe the cases in which the output of a gate is not set correctly based on the settings of its inputs.

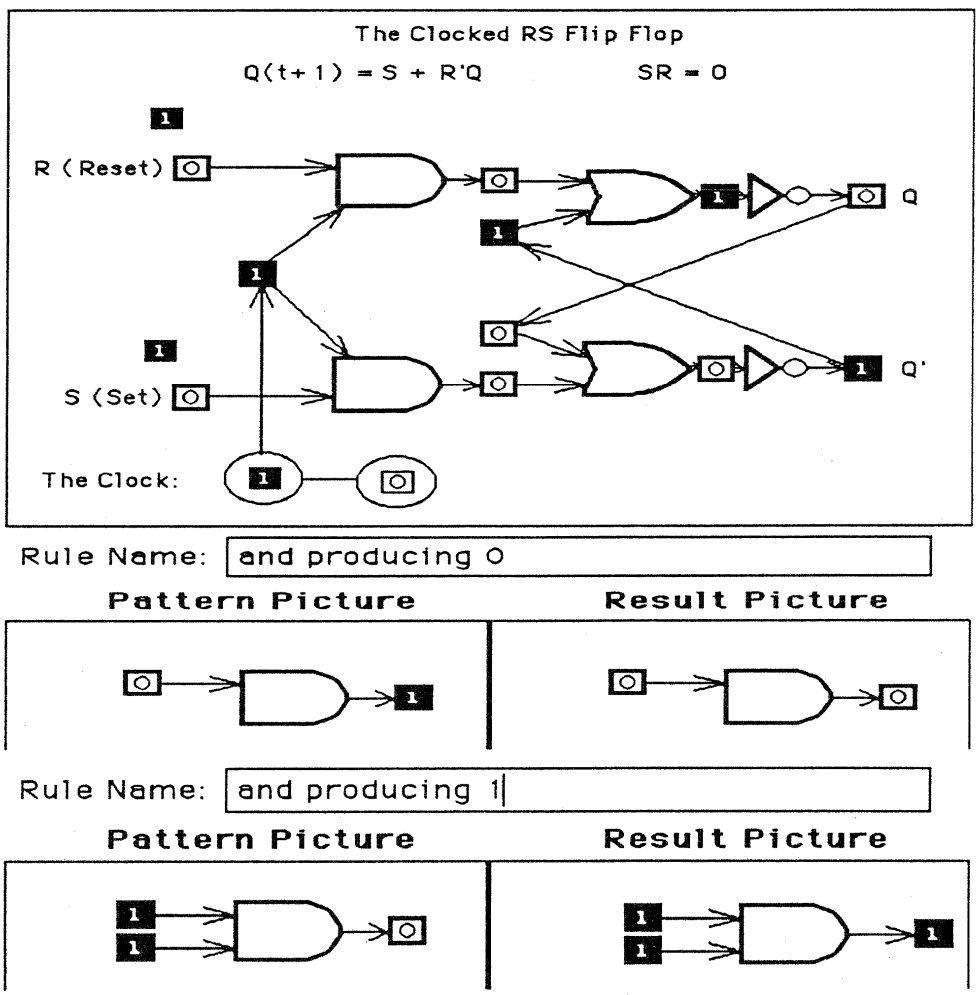


Figure 4-7: Logic circuit simulation display (top) and two rules governing "and" gates (bottom).

4.4 Game Playing Problems

The tic tac toe and checkers playing problems are added because the simulations involve much interaction with the end user and require relatively complex computations to produce good play.

4.4.1 The Tic Tac Toe 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.

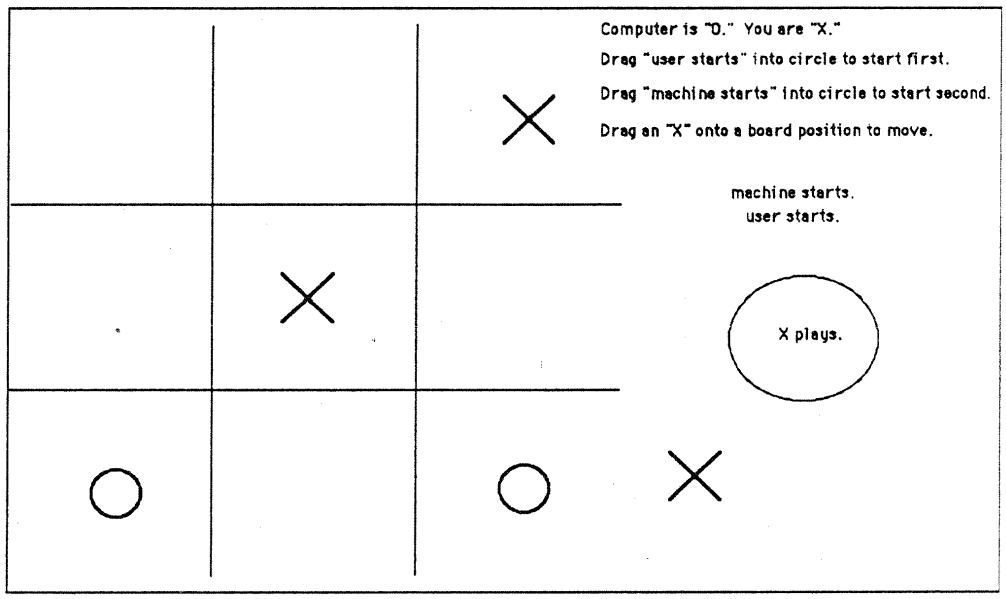


Figure 4-8a: The Tic Tac Toe Simulation.

4.4.2 Tic Tac Toe Solution

Figure 4-8a 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 4-8b 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 4-8c 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 4-8d 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.

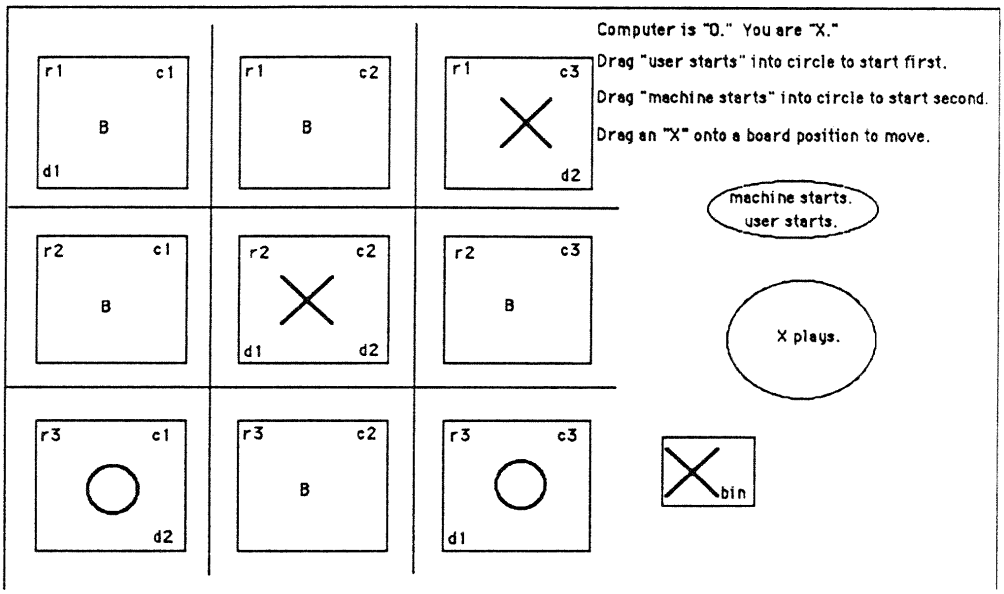


Figure 4-8b: The Tic Tac Toe Simulation with hidden objects shown.

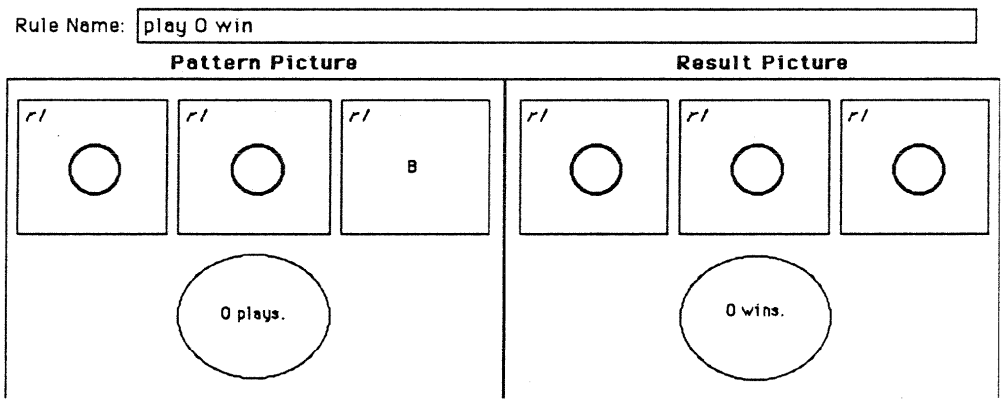


Figure 4-8c: Tic Tac Toe Rule to play a win.

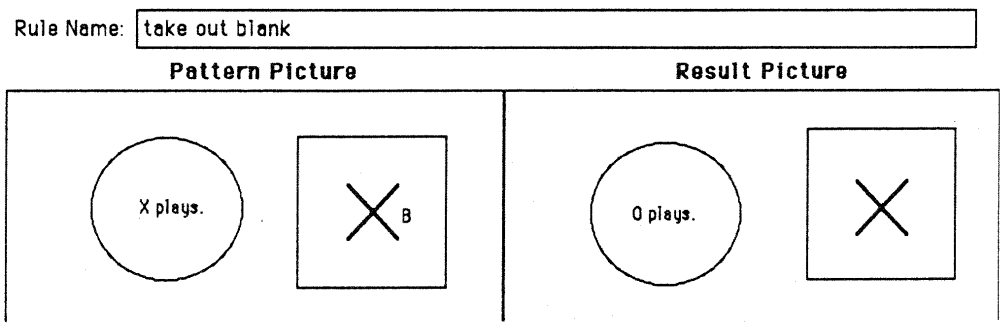


Figure 4-8d: Tic Tac Toe rule to switch turn mode.

4.4.3 Tic Tac Toe Walkthrough

draw initial picture: of the tic tac toe board, the vertical and horizontal lines.
draw command list: create an X and an associated container that may be dragged onto the tic tac toe board as a command.
draw command list: create a command "user starts" and associated container.
draw mode description: create a container that holds an "X plays" label.
draw additional container: no container objects exist for the individual squares of the board, so 9 squares are created on the board.
draw empty container marker: draw a "b" for blank in each of the 9 squares.
draw unique identifier: 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.
draw unique identifier: create three objects "c1," "c2," and "c3" to denote the difference in columns. place the appropriate one of these objects in each square.
draw unique identifier: 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.
hide objects: hide the cells, the blanks, and the row and column identifiers.
hide objects: hide the command and mode containers.
create command rule: named "Clear X tokens." Copy onto the pattern and result pictures the mode container containing "user starts."
  replacement action: replace "X" with "b."
  addition action: add an "X" to the bin.
create command rule: named "Clear O tokens" analogous to "Clear X tokens."
create command rule: named "start new game." Copy onto the pattern and result the mode container containing "user starts" and the command container for the "user starts" command.
  movement action: move "user starts" from the mode container back to its original position.
create command rule: named "Take out blank." Copy onto the pattern and result pictures a cell that contains a "b" and an "X," the mode container containing "X plays."
  deletion action: remove the "b."
  replacement action: replace "X plays" with "O plays."
create rule: named "Detect X win." Copy onto the pattern and result pictures the mode container containing "O plays," and three board squares with an "X" and a an "r1" in each square.
  identical variable match: specify that the row name "r1" is variable for the three squares.
  replacement action: replace "O plays" with "X wins."
create rule: named "Play O win." Copy onto the pattern and result pictures the mode container containing "O plays," and three board squares with an "r1," two containing an "O" and one containing a "b."
  identical variable match: specify that the row name "r1" is a variable for the three squares. specify that the row names in the result are also variable.
  replacement action: replace "O plays" with "O wins."
  replacement action: replace "b" with an "O."
create rule: named "Play a block move" analogous to "Play a win move."
create rule: named "Play a force win" analogous to "Play a win move" except that two rows that have a common square are detected.
create rule: named "Play center" which is also similar, only recognizes the middle cell as a cell with a "d1" and a "d2" in it.
create rule: named "Play anything" which recognizes
reorder rules: The order of the rules are created in this walkthrough in a way that will work. The rules to choose a move must be ordered in their order of priority. The rules to clear tokens must be ordered before the rule to start a new game.

The tricky parts of the tic tac toe walkthrough are summarized here:

- **draw additional container** loosely guides the creation of the cell containers;
- **draw unique identifier** very abstractly guides the creation of unique identifiers for each row, column, and diagonal, and
- **identical variable match** guides the variablization of the row labels within a rule.

The main language weakness pointed out in this walkthrough is that the language relies on the creation and generation of subtle representations.

4.4.4 The Checkers Problem

On the screen, show a Checkers board with the checkers placed on the board in the initial board position. Allow the user to start a new game at any time, and to move the checkers on the board. If a checker is placed on the last row it should automatically be kinged. The user is expected to note when they are done with a move. When this occurs the machine should respond by making a legal play on the board. The machine should be able to play to take pieces, play to be kinged, or move a random piece if there are no other alternatives. Assume the user is honest - don't worry about preventing illegal moves.

4.4.5 Checkers Solution

Figure 4-9a shows the end user interface needed in checkers. It provides a way to move the checkers, to complete a move, and to restart a game. Black is played by the computer, and white by the user.

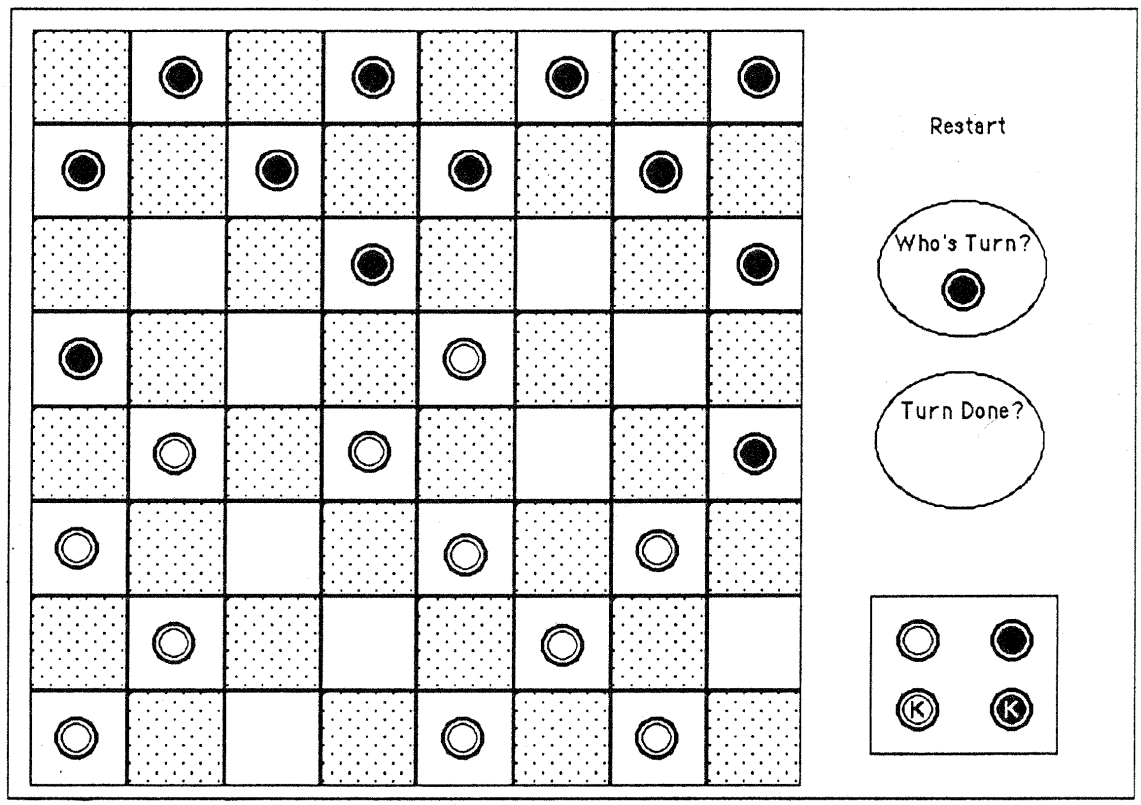


Figure 4-9a: Checkers Simulation.

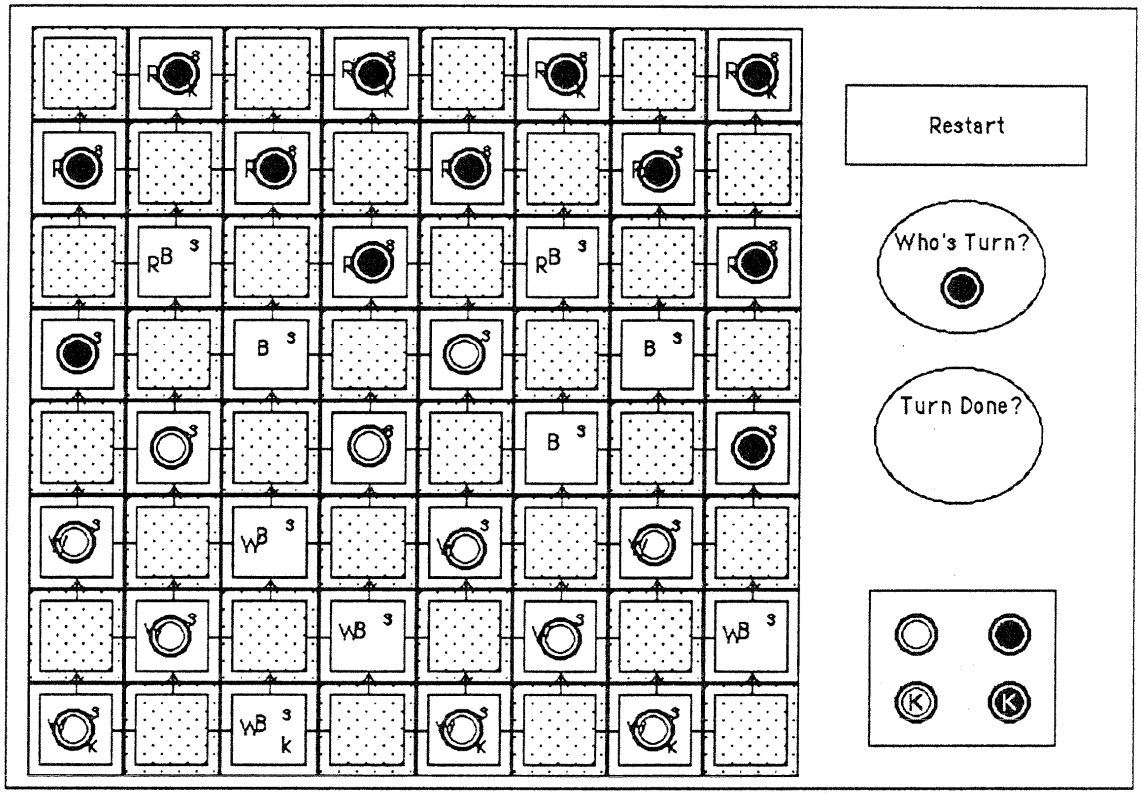


Figure 4-9b: Checkers simulation with hidden objects shown.

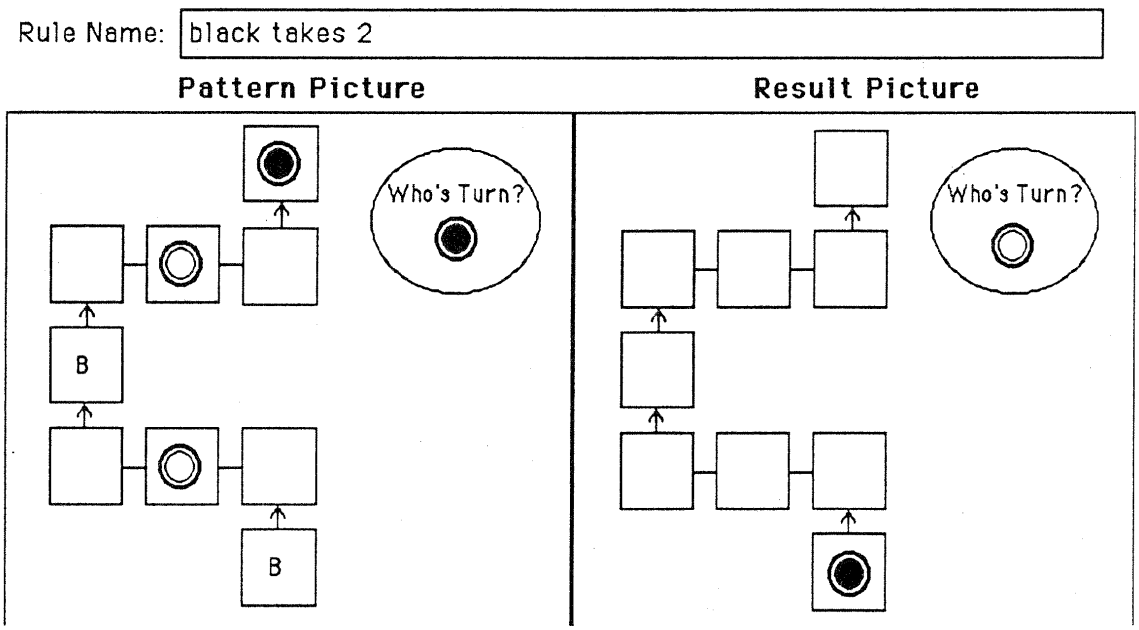


Figure 4-9c: Checkers rule to take two white pieces.

Figure 4-9b shows the checkers simulation with all hidden objects shown. The representation of the board is an 8x8 grid of connected cells. Cells are connected by directed paths from bottom to top because there is an important distinction between moving in one direction and moving in the other direction. Cells are connected by undirected paths from side to side because this distinction doesn't matter. The cell connections give a structure to the playing board that the playing rules can exploit. Figure 4-9c shows a rule that enables a black piece to jump two white pieces. This rule works in any configuration of two jumpable white pieces because the undirected side paths constrain the appropriate adjacencies but do not constrain the adjacencies to be in a particular direction (left or right). The rules for playing black change "Whose Turn" to white on play completion. The rules for playing are:

- jump three white with regular piece,
- jump two white with a regular piece,
- if a king can jump, place a marker on that king,
- if a king has a jump marker, take a jump forward,
- if a king has a jump marker, take a jump backward,
- if a king has a jump marker, end turn in "Who's Turn,"
- jump one white with regular piece,
- move forward with regular piece,
- move backward with king, and
- move forward with king.

The rules for playing black depend on a "B" marker being placed only in vacant playable cells. When white makes a play by shuffling a piece, and taking off any black pieces, the "B" markers need to be updated so that a "B" is in every blank cell but not in a cell with a piece. A few rules are needed to re-setup the blanks. Other rules are needed to reconfigure the starting board when a game is restarted.

There are a lot of rough parts of this walkthrough. They are guided by the following doctrine:

- **draw grid** concretely guides the creation of an 8x8 grid of cells to represent the board;
- **draw paths for representation** abstractly guides the creation paths to represent the structure of the board;
- **draw non-directed path** and **draw directed path** very loosely guide the creation of the specific path representation
- **draw two directed paths** strongly guides a more complicated path representation (not used in this solution);
- **draw unique identifier** guides labeling of the checker cells: "s" for cells that are playable, "k" for cells that are kingable, "W" for cells that are starting cells for white pieces, and "R" for cell that are starting cells for black (red) pieces;
- since the cells have other identifying markers, **draw empty container marker** weakly guides the creation of the "B" markers;

- **create control rule** guides the creation of rules that play when the "Who's Turn" mode has changed;
- **create control rule** and **create rule** abstractly guide the creation of rules that match configurations for making multiple jumps; and
- no doctrine guides the creation of the rules and the jump marker that enable a king to make an arbitrary number of jumps that may include jumps in both directions.

4.5 Counting Problems

These problems test the ability in ChemTrains to count in a simple domain. The solutions shown here are done without the benefit of the counter feature that was accepted in the final design. These walkthroughs illustrate the shortcomings of an intermediate design that had no specific counter feature.

4.5.1 The Counter Problem

Display the digits of a counter. The simulation should enable the end user to increment, decrement, or clear the counter by giving it commands.

4.5.2 Counter Solution

Figure 4-10 shows a solution to the counter problem. The main display includes three main parts: a list of operators that may be applied to the counter, the counter itself which is a string of connected boxes containing 0's, and a representation of the 10 digits which is used as a table defining the order of digits. Two of the three rules needed to describe how to increment the counter are shown in the figure. The rule "increment counter" increments the digit that is currently highlighted with an additional rectangle when the label "incr" is placed in the same container. The rule "increment counter on 9" changes a '9' to a '0' in the highlighted digit place, and moves the highlight rectangle one digit to the left. A third rule is needed to move the highlight rectangle to the right after the counter has been appropriately incremented. So if the counter contains a "299," the "increment counter on 9" rule will execute twice moving the highlight two digits left and replacing the 9's with 0's, the "increment counter" rule will then execute changing the 2 to a 3, and then the rule to shift the highlight rectangle to the right will execute twice.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw command list** guides the creation of the counter operators;
- **draw ordered list** guides the creation of the digits of the counter, and the creation of a marker for the counter;
- **draw big enough container** guides the creation of a container for the counter;
- **draw ordered list** loosely guides the creation of the digit list, given that the programmer has invented the representation; and
- **create command rule** loosely guides the creation of the rules to control the counter.

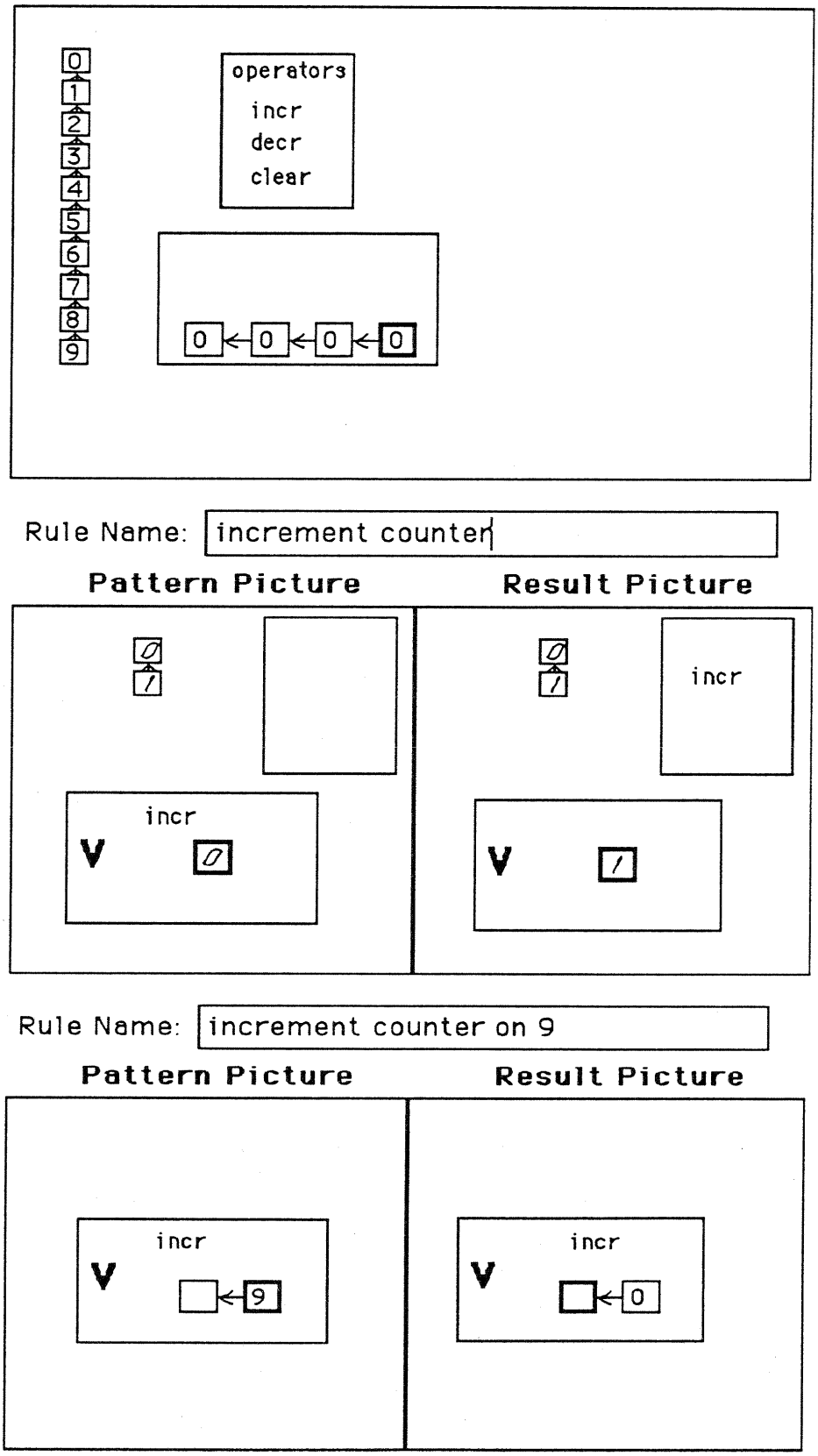


Figure 4-10: Counter simulation display (top) and two rules for incrementing (below).

4.5.3 The Network Traffic Counter Problem

Display a directed graph (nodes and directed links) that hold tokens. The tokens may travel between nodes of the graph along the directed links, but a token may not move to a node that already contains a token. Create two counters that count the number of tokens that pass through two different nodes.

4.5.4 Network Traffic Counter Solution

Figure 4-11 shows a solution to this problem. The main display contains a directed graph with nodes and directed links, and contains a couple of counters which had been imported from the counter simulation. The "traverse arc" rule pushes a token along an arc if there is nothing at the other end. A "nothing there" marker is needed to represent that nothing exists in a node. The "count traffic" rule places an "incr" label in a counter that has the same label as a node that contains a token. Since the counter rules have been imported, the appropriate counters will be incremented when tokens travel through labeled nodes.

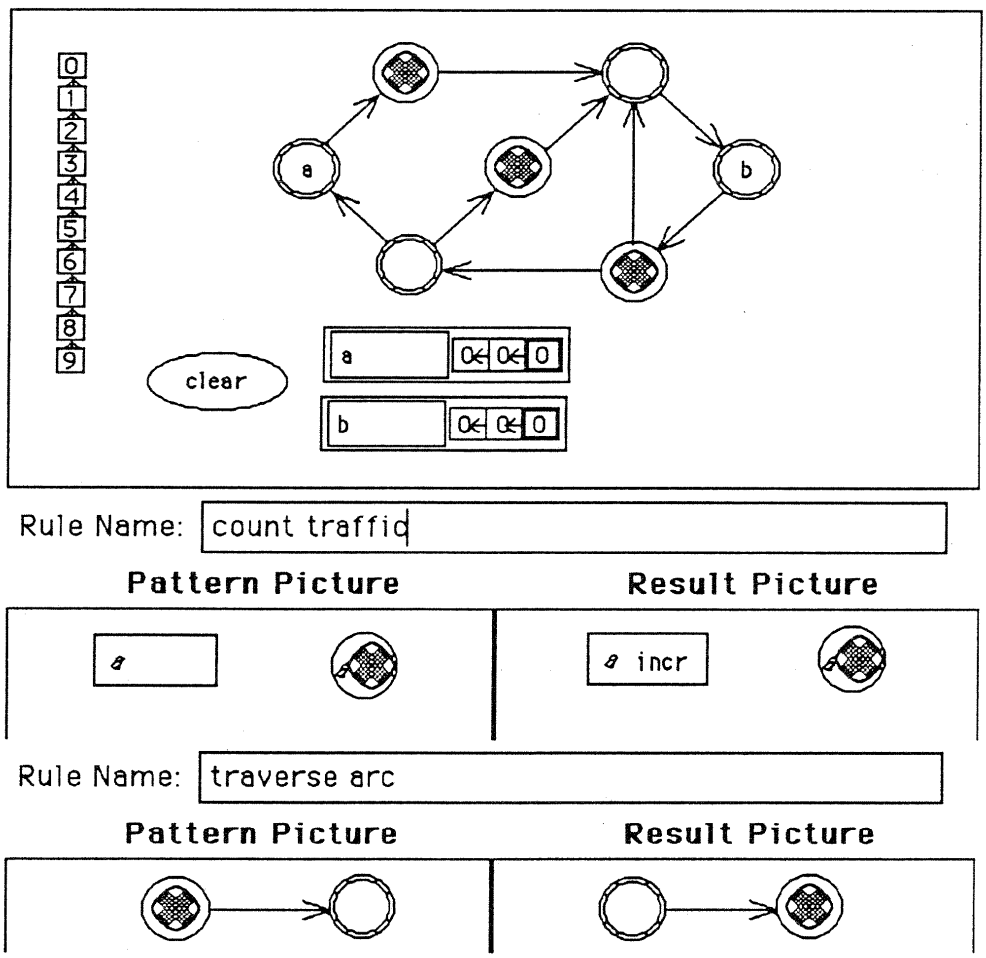


Figure 4-11: Count Traffic Simulation Display (top) and two rules (bottom).

The tricky parts of the solution are guided by the following doctrine advice:

- **draw directed path** guides the use of directed paths in the graph;
- **draw empty container marker** guides construction of an empty marker for the nodes of the graph;
- **count occurrences**, guides the importing of the counter; and
- **count multiple occurrences** guides the use of labels on the counters and the nodes of the graph.

4.5.5 The Network Traffic Problem with Bar Graph Output

Create a simulation of tokens moving within a directed graph as in the previous problem. Instead of displaying the count of network traffic as integers, display the count as a bar graph.

4.5.6 Network Traffic with Bar Graph Solution

Figure 4-12a shows the solution of the bar graph simulation after the network has had some traffic. The structures needed to make this work are hidden. Figure 4-12b shows the simulation with all the bars initialized and with the hidden representation shown. Each bar of the bar graph is shown as a sequence of cells connected by directed paths. Each bar has a single marker that starts at the bottom. Figure 4-12c shows the rule for incrementing a bar of the graph, by moving the marker up and placing a filled rectangle in its place. As in the previous problem, the simulation updates the counter with a rule that places an "incr" command at the base of a bar when a token passes through the node with the same name. Another rule exists for clearing a bar when a "clear" command is placed at the base of the bar.

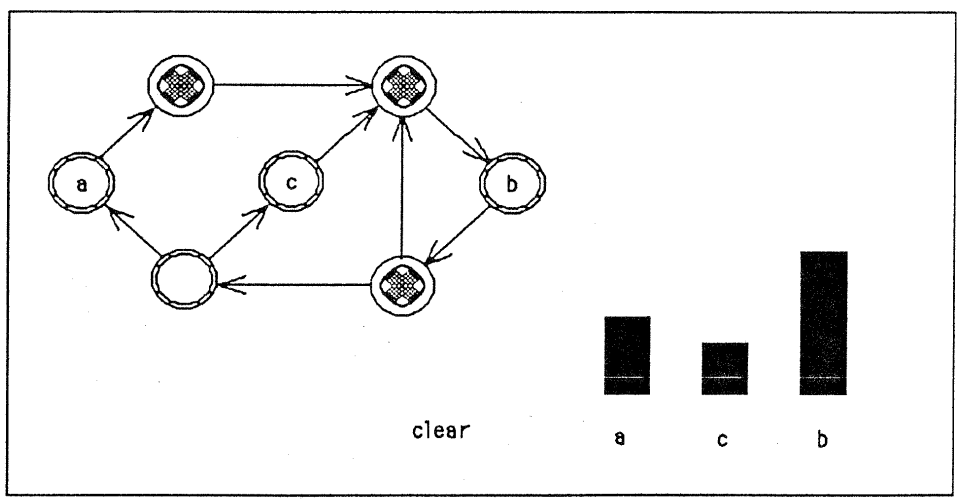


Figure 4-12a: Traffic Simulation with Bar Graph Output.

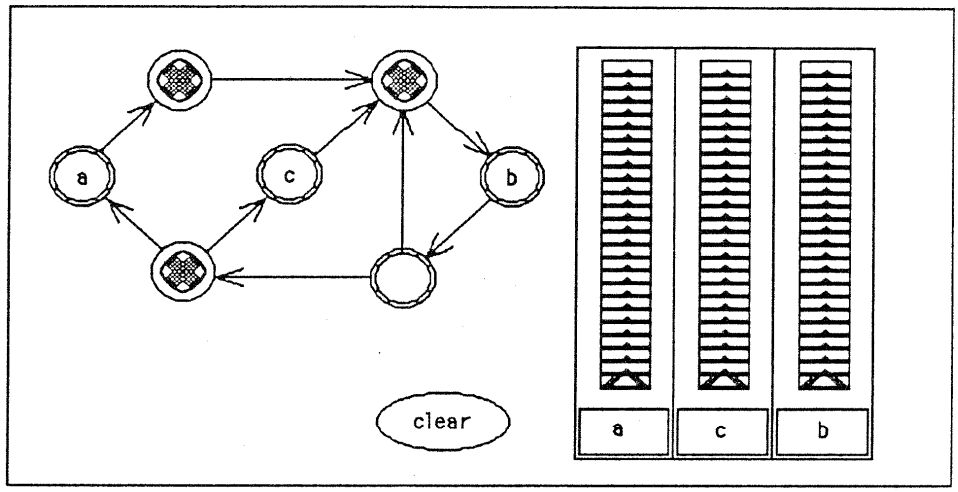


Figure 4-12b: Traffic Simulation with Bar Graph Output: hidden bar graph objects shown.

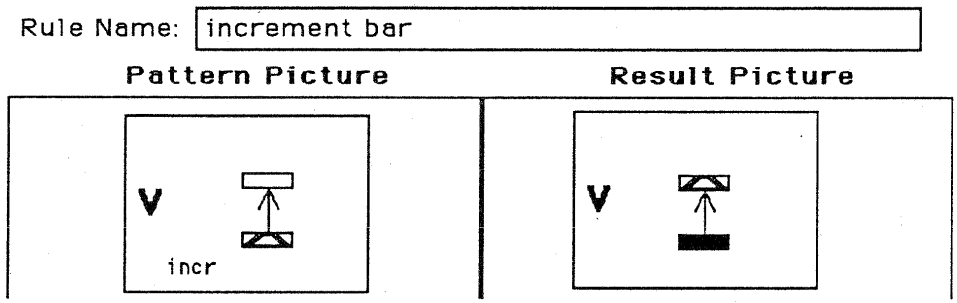


Figure 4-12c: Rule to draw one increment of a single bar.

The creation of bar graph structure is guided by very abstractly by the **draw ordered list** doctrine. Like the simple counter, the simple bar graph simulation can be imported alone without the traffic simulation part.

4.6 The Rolling Dice Problem

The simple rolling dice problem is included to test the ability to describe simulations that randomly exhibit different kinds of behavior at any given time.

Rolling Dice Problem: Display three adjacent sides of a a six sided die. The die should have 1 & 6 on opposite sides, 2 & 5 on opposite sides, and 3 & 4 on opposite sides. The end user should be able to roll the die by placing a roll command near it. The die should tumble a random number of times over 10. For each tumble the die should either roll on one of its 12 edges, or spin around one of its 8 corners.

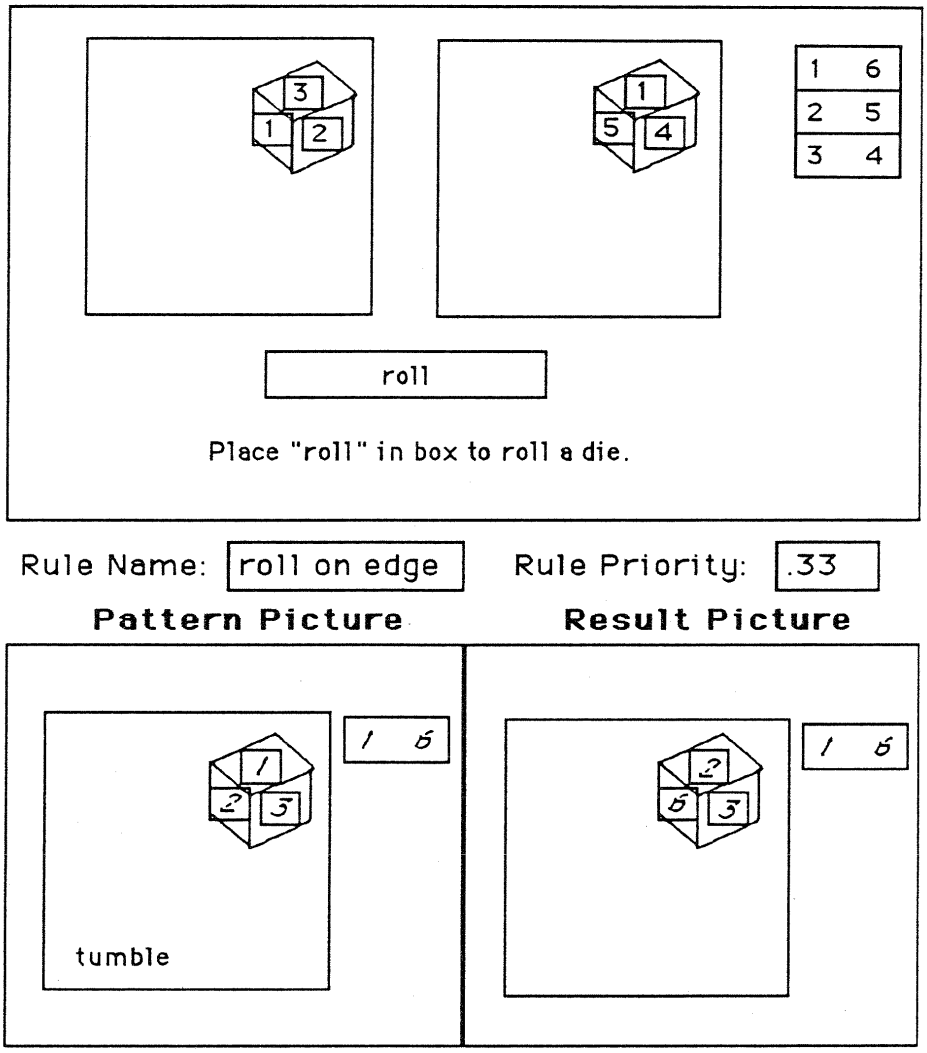


Figure 4-13: Dice simulation display (top) and rule to roll on edge (bottom).

4.6.1 Rolling Dice Solution

Figure 4-13 shows one possible solution to this problem. A die is displayed as a cube. The label of each displayed side of the cube is shown inside a hidden rectangle (not hidden in the figure). One rule turns a roll into a bunch of tumbles, and four rules describe what may occur with a tumble: either a roll along an edge (shown in the figure), a rotation around the front corner, a rotation around a side corner, or a generation of more tumbles. These rules are parallelized so that any one of the 4 rules may be executed for any of the tumbles, resulting in completely random roll of the die. A representation of what faces are opposite is needed so that the appropriate new faces are displayed when a die rolls or spins. For example in the "roll on edge" rule shown in the figure, the right face displaying the "3" is held steady while the die rolls one turn clockwise around this face, the "2" is moved to the top face, the "1" is removed because it has rotated to a hidden side of the die, and the opposite of the "1" is created on the left side of the die. The face markers are all variablized, so that this rule will roll with any set of faces. Since the face

rectangles are identical, the general roll on edge rule will work rotating around any of the three dimensions and rotate in either direction.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw position containers** guides the creation of the identical containers on each face;
- **draw pairing structure** and **table driven condition** guide the creation and use of the table defining opposite face markers;
- **draw command list** and **create command rule** guide the creation and use of a "roll" command for rolling a die; and
- **describe random rule behaviors** guides the programmer in parallelizing the rules that govern a rolling die.

4.7 Network Protocol Problems

These problems are for simulating communicating entities in a network using different communication protocols. The two protocols described here are the alternating bit protocol and the sliding window protocol. For each protocol the simulation must demonstrate a sender sending a string of characters to a receiver. The simulations should demonstrate lost transmissions along wires, and should also keep a count of the amount of message traffic that has travelled between the network entities.

4.7.1 The Alternating Bit Protocol Problem

The sender and receiver communicate by sending a message and an associated bit, that keeps track of whether they are in synchronization. Both the sender and the receiver keep track of the current bit (a 0 or 1). The sender's behavior should be governed by the following protocol:

- send a message with the current bit state and flip the bit state, if the first message is being sent, or if there is an acknowledgement from the receiver for the previous bit state;
- resend the previous message with the previous bit state at any time; and
- ignore any acknowledgement from the receiver for the current bit state.

The receivers behavior should be governed by the following protocol:

- if there is a message whose bit state matches the bit state that is being waited for, send an acknowledgment labeled with the bit state, and switch the bit state;
- resend an acknowledgement for the previous message at any time; and
- ignore any incoming message whose bit state does not match the bit state being waited for.

4.7.2 Alternating Bit Protocol Solution

Figure 4-14a shows three possible network entities named Fred, Ethel, and Wilma. In this figure, Fred is sending messages for the string "atest$" to Wilma. Wilma has already received "at" and has sent an acknowledgement back to Fred. Fred is now ready to send a new message, the "e." The representation of a network entity holds the following attributes and commands: its name, its current bit state, a command describing whether it's sending or receiving, the name of the network entity it's communicating with, a place for incoming messages, and a string of places holding its contents. The entities are connected with mail slots. These are needed to simulate messages continuing or being lost along a transmission lines. Figure 4-14b shows the two rules governing network traffic, "continue message" and "loose message." Together these rules will loose messages 10% of the time. Figure 4-14c shows 2 of the 6 rules needed to describe the protocol. The rules directly correspond to the 6 rules in the target problem.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw initial picture** and **draw additional container** loosely guide the creation of a set of network entities as separate boxes;
- **draw attribute-value pairs** guides the creation of places in each network entity that hold the attributes and commands of a network entity;
- **draw command list** guides the creation of commands needed;
- **draw pairing structure** and **table driven condition** guide the creation and use of a box holding the bit values,
- **draw ordered list** guides the creation of the message strings,
- **draw directed path** guides the creation of direct connections between network entities (a bad solution) rather than connections with intervening mail boxes needed to loose traffic part of the time; (perhaps additional doctrine is needed here, but this may be too domain specific, as shown below)
- **count occurrences** guides the creation and use of the counter;
- **reorder rules** guides the ordering of rules so that sending messages takes precedence over resending messages;
- **describe random rule behaviors** guides specifying rule priorities on the "loose message" and "continue message" rules; and
- **enable rule to continually fire** guides the creation and use of objects (the "X" and two rectangles at the top of figure 4-14a) that allow the rules "resend message" and "resend acknowledgement" to fire more than once on the same piece of data.

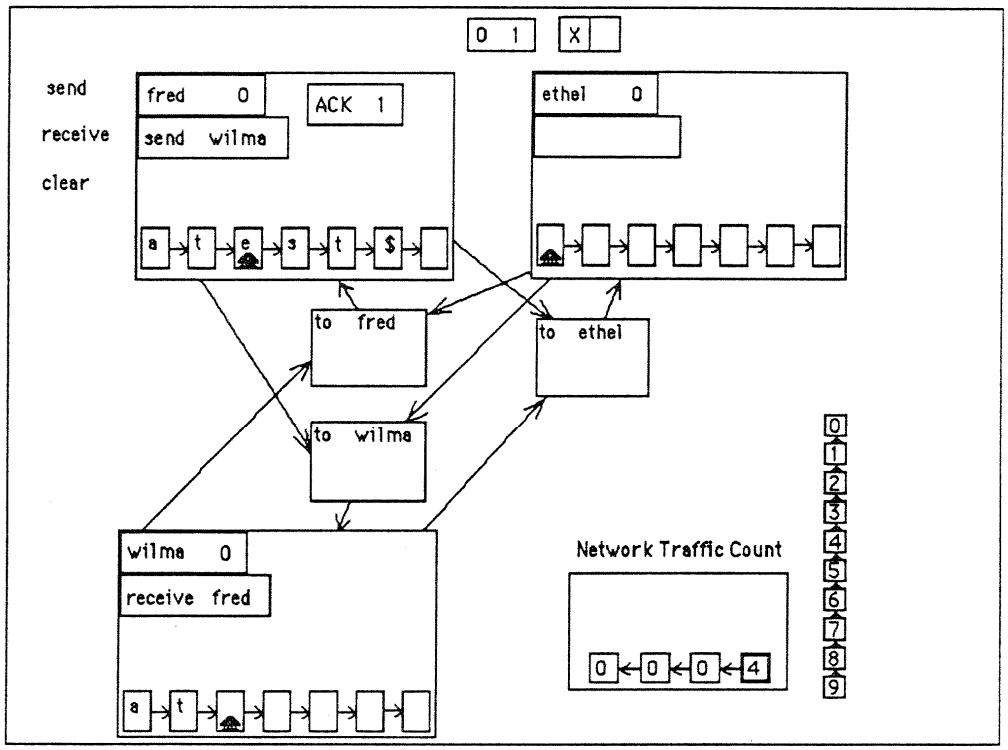


Figure 4-14a: "Alternating Bit" Protocol Network Simulation.

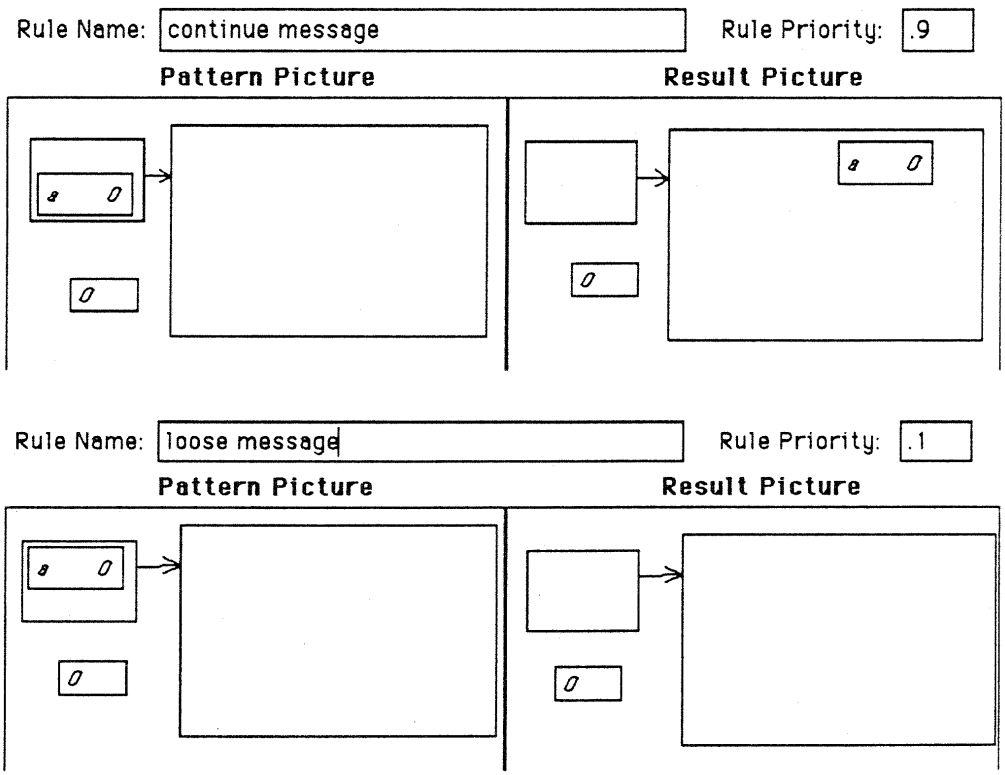


Figure 4-14b: Network Message Travelling Rules.

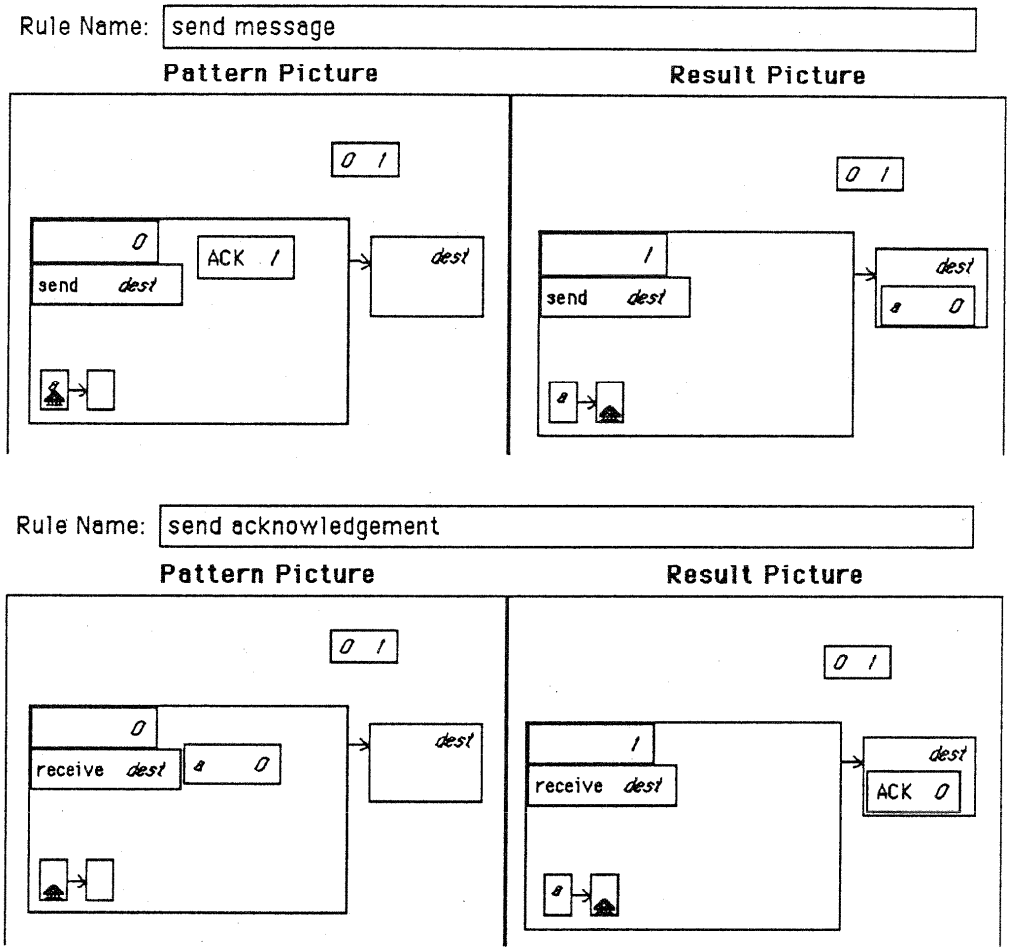


Figure 4-14c: Example Alternating Bit Protocol Rules.

4.8 The Grasshopper Simulation Problem

This problem is added because it is a larger simulation, and is a good example of the type of problem that a nonprogramming scientist would work on. The problem was derived from a book on insect population (Varley, Gradwell, & Hassell, 1973.)

Grasshopper Simulation Problem: 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.

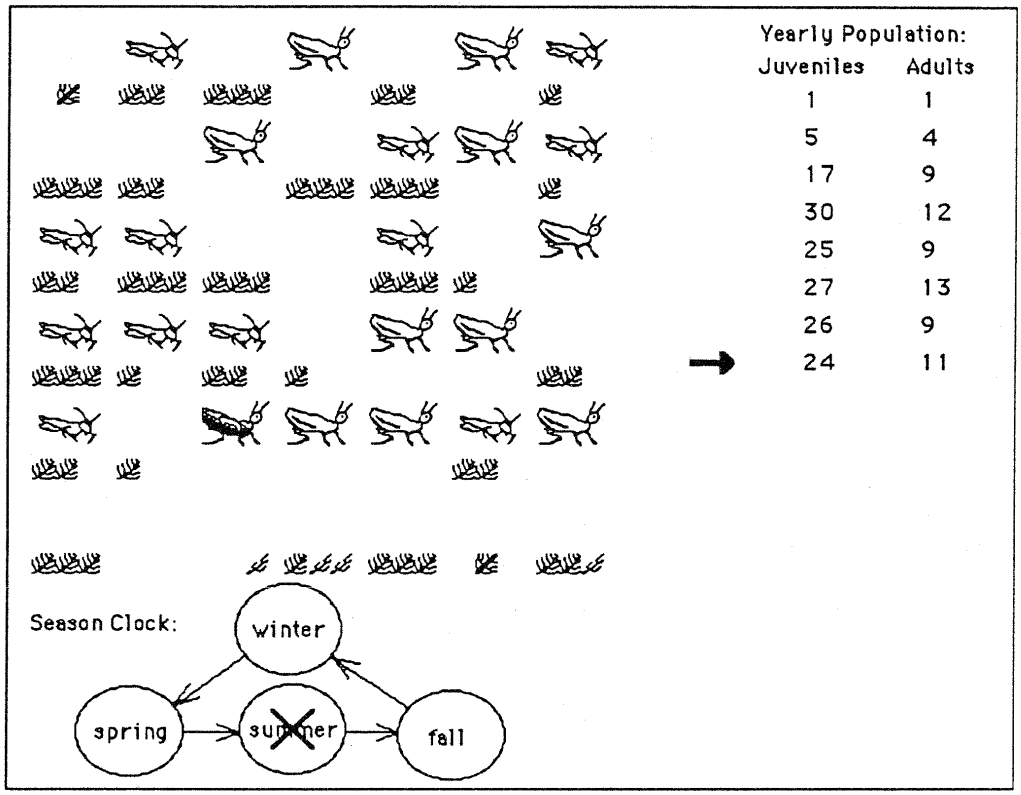


Figure 4-15a: Grasshopper Simulation.

4.8.1 Grasshopper Solution

Figure 4-15a 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 4-16b 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.

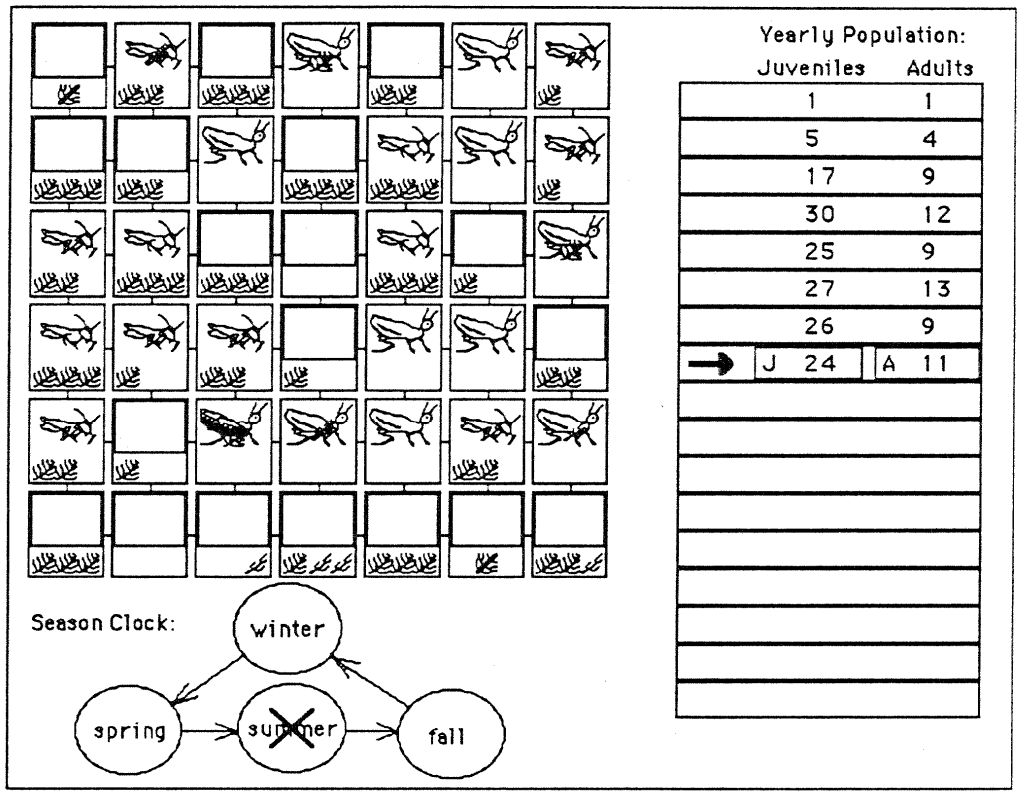


Figure 4-15b: Grasshopper Simulation (hidden objects shown).

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 4-15c&d 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.

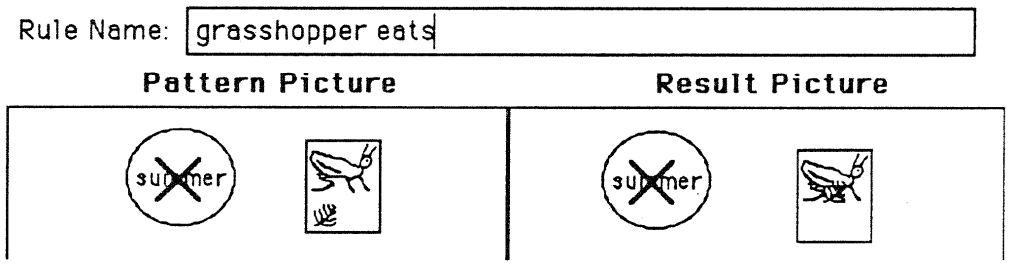


Figure 4-15c: Rule for adult to eat piece of grass in the summer.

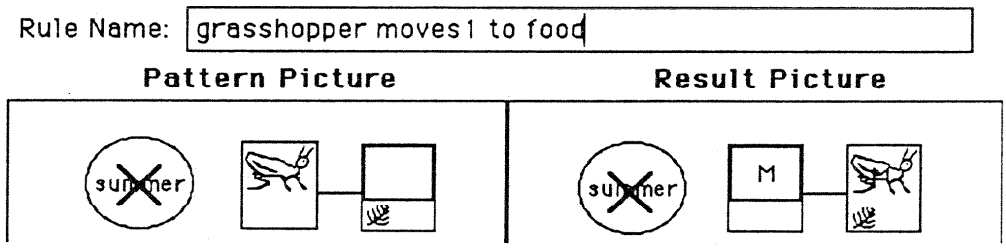


Figure 4-15d: Rule for adult to jump toward grass in the summer.

Figure 4-15e 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 ChemTrains91 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 4-15f 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.

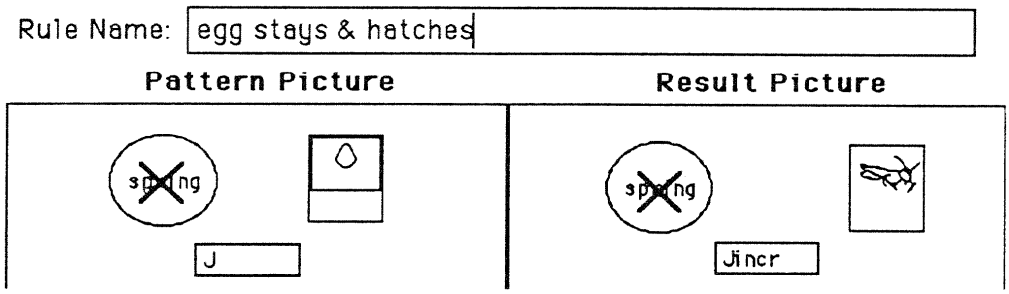


Figure 4-15e: Rule for egg to hatch in the spring.

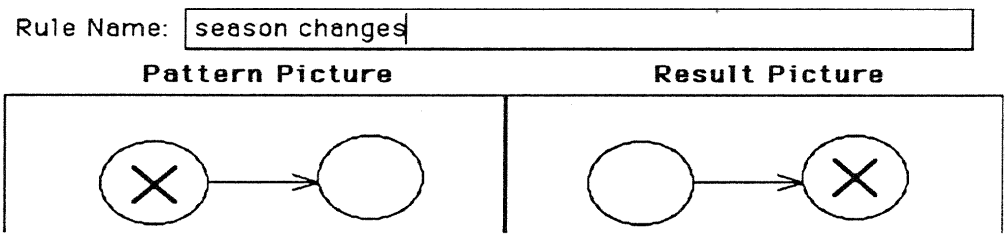


Figure 4-15f: Rule to change seasons.

The tricky parts of the solution are guided by the following doctrine advice:

- **draw grid** guides the creation of a grid for holding the grasshoppers;
- **draw non-directed path** guides the linking of the grid cells with non-directed paths;
- **draw empty container marker** guides the creation of an object that signifies a grid cell is vacant;
- **draw unique identifier** does not adequately guide the creation of the "M" marker signifying that a grasshopper may move its belongings (perhaps more direct doctrine is needed here);
- **draw ordered list** guides the creation of the yearly population report,
- **draw position containers** and **draw unique identifier** abstractly guide the creation of the individual containers and the "J" and "A" markers for the juvenile and adult population labels;
- **populate randomly** guides the creation of rules to grow different kinds of grass randomly in the grass places;
- **draw control sequence structure** guides the creation of the season clock, and the rule to change seasons; and
- **season structure condition** guides the use of the season marker in the rules for describing grasshopper behavior.

This is one of the few solutions to a target problem that does not involve any use of variables. All the rules are stated in terms of constants. Variables could have been used to reduce the number of rules describing juvenile and adult summer behavior. **Draw attribute-value table** and **table driven condition** would have guided the creation of an attribute-value table that

defines the juvenile and adult grasshopper "data types." The use of this table would have reduced the number of rules from sixteen to eight, but variables would have been required in these rules.

4.9 Walkthrough Analysis Summary

The walkthroughs and their summaries showed that the hardest parts of building ChemTrains91 simulations involved inventing and creating picture representations that are needed to make the simulation work. The choice of representation is important not only because some representations may be required for a working solution, but also because the choice directly affects the number and complexity of the necessary rules. A bad choice will result in much work either in writing rules or in recovering and searching for other possible representations. In contrast, a mistake in the construction of the rules can be easily fixed by editing the rules or changing the rule ordering.

Each piece of doctrine can be categorized along two dimensions (described more fully in chapter two):

- whether it is abstract or concrete, and
- whether it is high-level composition doctrine or lower level hacking doctrine.

Table 4-1 shows the ChemTrains91 doctrine for building representations rated on the two scales based on the walkthrough analysis.

Table 4-1: ChemTrains91 Doctrine for Building Picture Representations.

	Abstract Doctrine	Concrete Doctrine
Composition Doctrine	draw attribute-value table, draw attribute-value pairs, draw command list, draw control sequence structure, draw pairing structure, draw set definition	draw initial picture, draw control panel, draw two dimensional grid, draw ordered list,
Hacking Doctrine	draw additional container, draw position containers, draw unique identifier, draw empty container marker, draw mode description, label path, label common paths, draw paths for control, draw path with a stopover, draw paths for representation	hide objects, draw directed path, draw two directed paths, draw non-directed path count occurrences, count multiple occurrences

Doctrine is deemed abstract if its description is very general and hard to apply during problem solving. A walkthrough analysis can help determine a piece of doctrine's abstractness. For example, since the "draw two dimensional grid" is applied in a straightforward manner in the arrows, checkers, and grasshoppers target problems, it is deemed concrete. Since the "draw control sequence structure" doctrine is applied in a more general way in the grasshoppers problem, it is deemed abstract.

Composition doctrine is knowledge that helps map the problem onto high-level structures and configurations of a simulation picture. For example, the "draw attribute-value table" composition doctrine is a description of how to map a stat sheet of facts onto a tabular pictorial representation. The "draw additional container" hacking doctrine is a lower level description of how to build a single piece of the simulation.

In addition to the trouble with building ChemTrains91 representations, there are also some troubles with writing a working set of rules. The creation of rules as guided by doctrine is fairly straightforward, except when variables are required. Use of variables in rules is guided by three abstract pieces of doctrine: **wildcard match**, **identical variable match**, and **variable addition action**. Use of variables is also referred to in more concrete doctrine: **table driven condition** and **set condition**. The walkthrough analysis of many target problems (the house lighting controlled by switch box, the finite state recognizer, the Turing machine, the tic tac toe, the checkers, the network traffic counter, and the rolling dice target problems) all directly took advantage of **identical variable match**, which may be the most powerful and most abstract piece of doctrine for the language. Variables and its associated doctrine give ChemTrains91 most of its power, supporting simple solutions to the target problems, but it is applied very abstractly and could cause difficulties for nonprogrammers. User testing on the previous ChemTrains prototype showed that variables were hard to use even when only the simplest variable doctrine, **wildcard match**, is needed.

The walkthrough analysis showed that there should be no other difficulty with creating rules and controlling the execution of rules, since the doctrine seems to be applicable in straightforward ways. The walkthrough analysis of all of the target problems showed that the main difficulties involved various aspects of building the appropriate picture representation.

5 Design Rationale

This chapter describes the reasoning behind the decisions that were made during the design of ChemTrains91. The design rationale is stated in terms of differences in programming walkthrough summaries that were generated in the process of Problem-Centered Design with Walkthrough Feedback. Rather than stating the evolution of programming walkthroughs of all of the target problems, this chapter states the rationale along the dimension of design issues. The discussion of each issue includes:

- an overview,
- a description of possible design alternatives,
- references to the target problems and portions of these problems that most heavily influenced the rationale,
- a comparison of the walkthrough analysis on these problems, and
- some rationalization in support of the final design choice.

The most complex design decisions are divided into eight design issues:

1. Conflict Resolution Strategy
2. Negation
3. Numeric Computation and Display
4. The Resize Problem and Object Typing
5. Grid Creation
6. Hideable Layers of Abstraction
7. Mouse Interactions
8. Rule Actions

This chapter concludes with two summaries: one summarizes the full design space and the general kinds of decisions that were made; the other summarizes the aspects of the design methodology that proved most useful in the design process.

5.1 Conflict Resolution Strategy

In the design of any production system language, such as ChemTrains, the choice of conflict resolution strategy affects how rules are written and organized. As previously described, a production system language interpreter executes one main loop, called the recognize-act cycle. In the recognize part of the cycle, the interpreter decides which rules can match the current data. The act part of the cycle executes the action of the rule. The interpreter starts

another cycle after completing the rule actions. If no rules match in the recognize part, the interpreter stops.

At any given time many different rules may be applicable, and each rule may apply to the data in many different ways. For example, a single rule to match a grasshopper and a piece of grass may match any grasshopper / grass combination on the display. The process of picking among many possible rules and rule applications is called conflict resolution. The method used by the interpreter to make this decision is called the *conflict resolution strategy*.

The overall conflict resolution strategy has a significant affect on the problem solving involved in getting a set of rules to work appropriately. This section first describes the rationale for the overall strategy and then describes the rationale for some minor adjustments to the strategy.

5.1.1 Overall Conflict Resolution Strategy

Three basic strategies were considered:

- *OPS5 strategy: recency of data and specificity of rule* (Forgy, 83.) This strategy involves always picking a rule that matches the most recently added or modified objects. If two rules match the same objects, the rule that most specifically specifies the constraints is chosen. McDermott and Forgy (1978) suggest that these strategies help manage the control of the rules.
- *random strategy: random selection of rule and random selection of data.* This strategy forces a random selection of both the rules and data when there is a conflict. This strategy will select any applicable rule at any time, and the programmer has no way to control the priority of execution. This strategy was considered because a random selection process seems like it should emulate natural phenomena better.
- *rule ordering strategy: prioritized selection of rule and random selection of data.* This strategy picks the first rule applicable in a rule list, which can be ordered by the programmer. If a rule may fire on a number of different combinations of data, the combination is chosen randomly rather than by recency or any ordering of the data. This strategy was considered because it enables the programmer to control the execution of rules by arranging their priority.

In doing the walkthroughs the target problems provided different requirements on the overall conflict resolution strategy:

- Some target problems (maze, dice, network traffic, and grasshoppers) require that rules execute randomly on the data. For example, a mouse wandering around a maze should travel as randomly as possible. If any non-random strategy of data

selection is used (e.g. recency), the mouse will exhibit behaviors that the programmer did not intend.
- Most target problems (maze, NDFSR, tic tac toe, checkers, alternating bit protocol, and count traffic grasshoppers) require that some parts of their rule sets be prioritized. For example, the tic tac toe simulation requires that the playing rules are ordered by their priority, so that a win move is played before a block move. The maze solution requires that looking forward has priority over moving backward.
- Some target problems (rolling dice, alternating bit protocol, and grasshoppers) require that different behaviors should be executed randomly. For example, in the rolling dice problem, the behaviors of rolling on a side and turning on a corner should happen randomly with respect to each other.

The OPS5 strategy was considered because it had been shown to be useful in controlling the execution of an arbitrary number of rules. (McDermott, 1984) The OPS5 strategy was not chosen because it could not be directly and easily applied in solving the target problems. In contrast, the rule ordering strategy could be applied much more simply and also enabled enough simple solutions to larger problems, such as the grasshopper problem, which required a fairly large rule set (31 rules). The **draw control sequence structure** and **sequence structure condition** doctrines generally describe how to manage the control of a large set of rules. This doctrine supports the most complex rule-based control structures needed for the target problems. More complex control structures, such as traversing goal hierarchies, which the OPS5 strategy supports, can be supported with additional doctrine within the rule ordering strategy.

The rule ordering strategy was chosen over the random strategy because too many problems required that rules be prioritized and only a few problems need to exhibit random behavior. In those problems, random rule execution is necessary for a very limited subset of the rules. Based on the sufficiency of the rule ordering strategy and the simplicity of its associated doctrine, this strategy was chosen as an overall conflict resolution strategy. Two other design issues arise from specific target problems that aren't addressed by the chosen overall conflict resolution strategy:
- Whether to match and execute a combination of data that has already been previously matched and executed.
- Whether to restart the pattern matching and conflict resolution process after every rule execution or to continue to execute rules lower in the rule ordering.
- How to allow the programmer to describe probabilities of intended behaviors.

5.1.2 Repeated Execution on the Same Data

The solutions to many target problems require that the rule interpreter never execute on the same data more than once. For example, in the grasshoppers problem, rules are needed to populate the cells with grass every spring. A rule for putting grass in a cell must fire for each and every cell only once. If the rule to grow grass may execute on the same data more than once, the rule will never stop creating grass.

In the alternating bit protocol problem, the rule interpreter needs to execute on data more than once. In this problem the rule to resend a message needs to send the previous message at any given time during the execution. The protocol must be able to resend the message more than once if the messages keep getting lost. In order to fire more than once on the same objects, the rule has to be enhanced to make some meaningless change to the data.

These conflicting strategies result in the following design alternatives:

1. Fire rules on one combination of data only once, and provide doctrine for allowing a rule to continually fire. This feature is a common production language feature, called *refraction,* that is added so that some rules cannot infinitely fire.
2. Fire rules even if they match previously matched objects, and provide doctrine for stopping rules from firing, when a rule is infinitely firing.
3. Or allow an editable property that specifies whether individual rules obey refraction or not.

The first alternative requires the following doctrine:

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.

The second alternative requires the following doctrine:

enable rule to fire once:

If a rule is firing on the same objects more than once, and the rule needs to fire only once,

then draw a marker that denotes that the action has not happened (like draw empty container marker), and place this marker in the pattern of the rule but not in the result.

The first alternative was chosen over the second alternative because it yields a simpler solution and walkthrough for more problems. The alternating bit

protocol problem is the only problem that isn't well supported. The third alternative was considered because it yields a simple solution to all problems. This alternative was not chosen because it adds to the complexity of the pattern/result rule construct without a lot of benefit. If more future target problems need rules to fire rules continually on data, this design decision should be reconsidered.

5.1.3 Interpretation of Rule Ordering

There are two ways to interpret how rule ordering affects the execution of rules. The first way is to execute all applicable rules and all applicable rule matches before starting another recognize-act cycle. The second way is to do pattern matching from the beginning of the rule ordering after any rule has fired. The first way would force the programmer to think of the rule ordering as a sequence of if-then statements bounded by a loop. The second way would force the programmer to think of the rule ordering as a strict ordering of priority. The second way was chosen because it enabled solutions that would be very hard the first way. For example, the solution of the grasshopper problem requires that specific events happen in certain seasons and that all events of a season should occur before the simulation changes season. The "change season" rule in this solution may only fire if nothing else may fire. If rule ordering is used to define rule priority (the second alternative), the simulation will work as intended. If rule ordering is used to describe a sequence of rules to execute (the first alternative), the "change season" rule will fire before it ought to fire. The use of rule ordering is guided by the following piece of doctrine:

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 in front of the rule that shouldn't be firing.

5.1.4 Nondeterministic and Probabilistic Rule Selection

Some of the target problems require random rule behavior. Consider the following scenarios in which different events can happen with different probabilities: a simulation in which 10% of grasshoppers that eat Poinsettia should die, (similar to loosing 10% of network messages in the network protocol target problems) and a simulation of traffic in which any car entering an intersection will turn right 20% of the time, turn left 15% of the time, and go straight 65% of the time. In order to solve these problems, the programmer of a simulation must be able to:

1. execute an action with a specified probability, and
2. fire a subset of rules nondeterministically.

Two different design ideas are discussed here. These strategies are extensions to the overall conflict resolution strategy of rule ordering.

Strategy 1: Probabilistic Rule Firing
Allow probabilities to be associated with individual rules. A probability on a rule would mean that the rule fires on possible matches with that probability. For example, associating .1 with a "grasshopper dies of poinsettia poisoning" rule will fire with a 10% likelihood on any grasshopper that has eaten poinsettia. This feature would require the following doctrine:

randomly fire rule:
If an existing rule should only fire part of the time that it is applicable,
then give the rule a probability assignment.

A problem arises with this feature when combined with the existing aspect of the system that forces the rule interpreter to start from the beginning after a single rule instance has been matched and fired. To illustrate the problem in the grasshopper problem suppose that four grasshopper/poinsettia pairs exist at the same time (a, b, c & d). When the rule mechanism does pattern matching, it appropriately finds a match and then flips a .10 weighted coin. Suppose a, b, & c are matched but the coin flip deems that the rule should not fire, and then d also is matched but it is accepted and the rule fires. The rule interpreter then starts from scratch again and reflips for a, b, & c, thus raising the probability that any given grasshopper will die. (a, b, or c could die on the first or second pass.)

In order to enable a possible solution to this, the system must fire on all possible rule firings before restarting the rule interpreter from the beginning. The possible design alternatives are:

1. restart the rule interpreter from the beginning after a single rule matches,
2. restart the rule interpreter from the beginning after all applicable rule matches have been tested and executed, or
3. make alternative a, the default for all rules, and allow alternative b to be associated with a subset of rules, as specified by the programmer.

To solve the traffic problem a "turn left" rule is created with probability .15, a "turn right" rule is created with probability .20, and a "go straight" rule is created. All applicable matches must be executed within this group of rules before the rule interpreter restarts from the beginning. There is one remaining problem: 20% of cars at intersections that haven't already turned left is not 20% of all cars at intersections.

Strategy 2: Probabilistically Scrambling the Rule Ordering
The programmer is allowed to select a set of rules in the rule ordering to be re-prioritized randomly at the beginning of each recognize-act cycle. Each of the rules may be given a probability of being placed first in the ordering among the rule set. After the rules are re-prioritized, the rule interpreter picks a rule from the newly ordered rules as usual. After a rule is selected and

fired, the rule interpreter begins another recognize-act cycle and re-prioritizes the rules again. The use of this feature is described by the following doctrine:

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.

In using this design feature to kill 10% grasshoppers that eat Poinsettia, two rules should be made. One rule that might be called "grasshopper dies of poinsettia poisoning" deletes a grasshopper if it has eaten a poinsettia. One rule that might be called "grasshopper recovers from poinsettia poisoning" just deletes the poinsettia if a grasshopper has eaten it. The two rules are "parallelized" and the first rule is given a .1 probability and the second rule is given a .9 probability. These two rules are applicable in the exact same case, but the rule to kill a grasshopper is prioritized before the rule to recover a grasshopper 10% of the time. Therefore 10% of grasshoppers on average that eat poinsettia will fire the "dies" rule, and the rest will recover.

In order to simulate cars turning in traffic, three rules are created similarly to the two grasshopper rules described above. The three rules all are applicable when a car reaches an intersection. Each rule is given a different priority.

An additional complexity arises when two groups of rules each have to be randomly selected, and rules in the first group must take precedence over rules in the second group, and the rules within both groups have probabilistic weights. The rules in each group have to be parallelized with respect to each other.

In comparing the two design alternatives, the second alternative involves more complicated doctrine. Although the first alternative could be described with very simple doctrine as shown, it does not adequately enable solutions to the problems. Therefore, the second design alternative was chosen. Perhaps a working version of the first design or a simpler version of the second design could be invented.

5.2 Negation

Many target problems require rules that check the absence of an object or configuration of objects. There are two approaches that were considered in testing for absence:

- allowing a user to explicitly test for absence in a rule by allowing a negation operator in the pattern of a rule; or

- providing guidance in testing for the presence of "empty object markers" that are used to denote absence.

For example, in the Bunsen Burner example a marker such as a "no flame" text string is placed in the flame holder to mark the fact that no flame exists. Without this marker and without a negation feature, it is impossible to write a rule that tests specifically whether the bunsen burner is in the state of not having a flame.

Here is doctrine for supporting the negation feature:

draw negation pattern:

If a rule can only execute if a particular object does not exist in region of the display,

then copy this object to the appropriate place in the pattern of the rule, and specify that the object is "negated."

The "empty marker" solutions are guided by the following two pieces of doctrine:

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. Follow the absence testing advice to create rules that use the empty container marker.

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 and result of the rule.

In working through the target problems, it was discovered that: (1) in many problems the empty marker solutions are quite simple; (2) in only the checkers problem does the lack of negation present a problem; and (3) in the bunsen burner problem the addition of doctrine supporting negation may lead to a more complicated solution.

1. In many problems, such as the Tic Tac Toe problem, a blank is used to represent the fact that a container is empty. When a tic tac toe rule has to test for the non-existence of an X or an O, it tests for the presence of a blank. The use of this blank yields a simple solution to the problem of controlling the player's and computer's turn sequence, which cannot be solved as easily if the programmer uses the negation feature.
2. In the checkers problem, the empty marker doctrine leads to a similar representation of the board. Using empty markers for checkers, these markers have to be updated when the end user moves checkers from place to place on the checkers board. The

solution, which is shown in the earlier checkers walkthrough, requires rules to delete all the markers, place them again, and then erase all the markers that coexist with checker pieces. This solution and its associated walkthrough are clearly poor. The use of negation in the checkers problem yields a solution that isn't as complex as this solution.

3. In the bunsen burner solution, a "no flame" marker is used to mark the fact that no flame exists below the beaker. This representation enables the bunsen burner rules to treat the "no flame" marker as if it were just any other kind of flame. The rule to make a low flame treats the existing object in the flame place as a wildcard and can match when the flame is either high or the "no flame" marker. If negation is used instead, additional rules must be added for the cases in which nothing exists in the flame place. Using negation rather than the empty container marker yields a longer solution and walkthrough.

The experience with the problems suggests that:

a. the negation feature requires slightly less complex doctrine;
b. the "empty container" doctrine leads to simpler solutions and walkthroughs more than the negation doctrine;
c. both the negation feature and the "empty container" doctrine are useful in solving problems; and
d. the existence of both doctrines may require additional doctrine to explain their differences.

Since these conclusions support both adding negation (in a) and leaving the feature out (in b), the decision was very close. The final decision of not adding negation was based on the fact that "empty markers" are a good representational device for problem solving and that the negation feature may lead programmers away from cleaner solutions.

5.3 Numeric Computation and Display

Some of the target problems require numeric computation.

- **The grasshopper problem** requires that the simulation display the population of juvenile and adult grasshoppers over time. The display may be a graph or a list of numbers.
- **The network traffic counter problem** requires that the simulation display the number of occurrences that a token passes through marked nodes of the graph.
- Some solutions to **the sliding window protocol problem** require that a numbered count be associated with objects on the screen, so that network protocol messages will be sent and accepted only within the sliding window.

The decision evolved in four distinct steps. The first design alternative was a set of operators that counted objects and did numeric computation when

named commands appeared in a simulation. The second design alternative was a mechanism for counting and displaying populations of objects over time by monitoring simulations. The third alternative was to enable programmers to import and use either a numeric counter or a bar graph counter made up from ChemTrains symbolic primitives. The fourth and finally accepted alternative was a set of primitive operators that could be applied to text objects that appear as numbers. This finally accepted feature was directly influenced by the third alternative. Each of the design alternatives is described with reasons for why it was or wasn't accepted.

5.3.1 Numeric Computation Operators

These features depend on a number being represented as a text string object that can be parsed as a number. The system would count objects if the text string "count" appeared inside a container with a specification of what to count. When the count operator is applied, the configuration of objects is replaced with a number. For example in figure 5-1a, the "count," the oval, and the containing rectangle is replaced with a count of ovals. Similarly, other arithmetic functions could be provided. The appearance of "+" and two numbers within the same container will replace the container, the "+," and the numbers with the result of the addition. When more than one of these operators can be applied at the same time, the operators are executed from the inside out. In this way, boxes are similar to parentheses. Figure 5-1b shows a more complex configuration of operators in a simulation.

Figure 5-1c shows how the operators can be used in the rules of a simulation program. This rule matches three boxes of a particular size. When the rule fires, the result puts an operator to count the number of "G"s in the smaller open box, and puts an operator to change the height of the column to the number of "G"s counted. After the rule executes, the "G"s will be counted, and then the height of the original striped rectangle will be changed to reflect the number of "G"s.

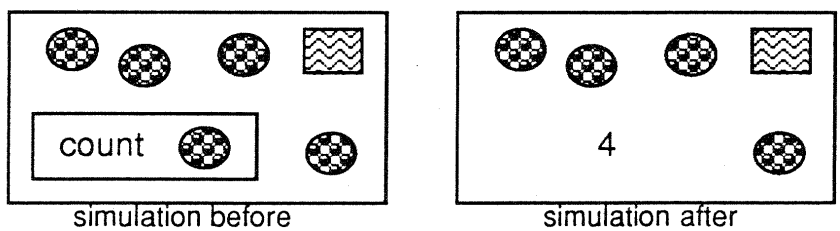


Figure 5-1a: The "count" operator.

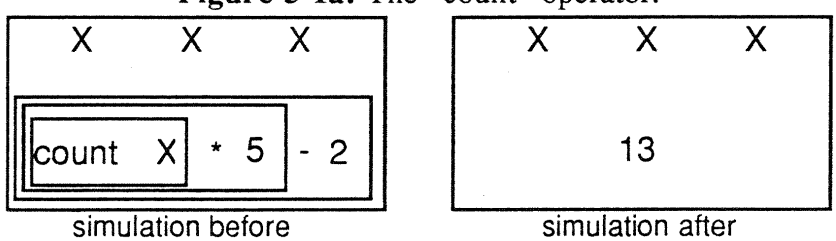


Figure 5-1b: Combination of operators.

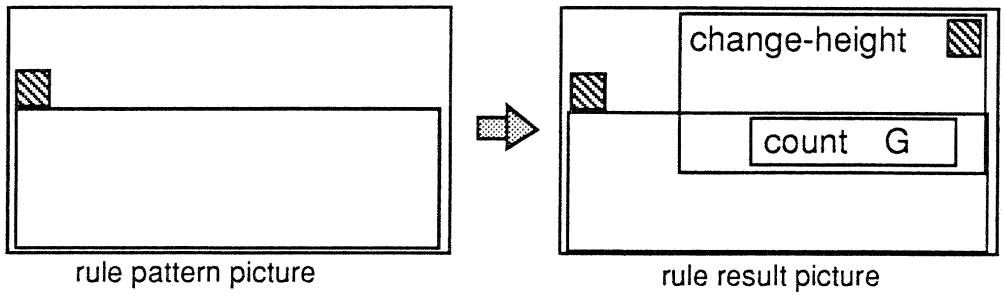


Figure 5-1c: Using "count" and "change-height" operators in a rule.

The combination of the "count" and "change-height" operators can be used to display the current number of grasshoppers with a bar graph. The solution would require a lot of extra containers and objects to be created on the screen. The shortcomings with these features are that:

1. the exact configuration of operators and containers may be confusing to the programmer;
2. all of the added containers and objects may overlap and interact with existing objects of the simulation, resulting in unexpected behaviors; and
3. the features provide a fairly direct way to count static objects, but no way to directly count events, which is necessary in the network traffic counting problem.

5.3.2 External Monitoring of the Simulation

This is a direct approach to the problem of displaying a graph based on the population of objects on the simulation display. The system would provide a facility external to the main simulation display, that would allow the programmer to define the class of objects to be monitored, and would automatically report the population in a separate window. This approach would enable the user to specify the information at a high level without worrying about exactly how it is to be displayed.

The advantage of this solution is that the programmer has a simple facility for monitoring populations or event occurrences that does not interfere with the workings of the simulation. The main disadvantage with this solution is that the facility does not offer numeric operators usable within a ChemTrains program, which is needed in solving the sliding window protocol problem. In addition, the facility seems conceptually different from the rest of the system. A solution that fits in better with the rule-based model of computation may be easier to learn once the ChemTrains basics are learned.

5.3.3 Numeric Computation with Symbols

A solution to the network counting problem shown in the previous chapter was created from basic primitives of the language. The counter rules use a connected sequence of digits to represent a number, and another connected sequence to represent the 10 possible digits. The system symbolically operates

on these rules to implement counting. Likewise, for drawing bar graph output, no features needed to be added. A symbolic solution was invented that required that a single bar be represented as a sequence of connected containers each of which possibly contained a filled element of a bar. Both the digit counter and the bar counter can be incremented, decremented, and cleared by placing operator-like text strings in a container with them. Rules can then be created to do counting of either object populations or event occurrences.

The advantages of these solutions are that they solve all of the target problems, and the solutions do not require additional features that deviate from the basic model of computation. The disadvantage is that the solutions required a lot of trickery.

Because these solutions were usable but hard to invent and create, the general "import simulation" feature was considered and added. This feature enables not only these solutions to be imported and reused but also future devious solutions to be imported and reused.

A remaining disadvantage with importing the digit counter is that the solution requires a lot of extraneous pictorial garbage that can clutter a simulation. The bar graph counter does not have as much clutter.

5.3.4 Primitive Numeric Computation Operators

This design alternative borrows ideas from both the first and third design alternatives:

- use operators to control the setting of numbers, (borrowed from first alternative)
- do numeric computation on text string objects that can be parsed as numbers, (borrowed from first alternative and from LISP) and
- use operators for incrementing, decrementing, and clearing a counter rather than higher level numeric operators. (borrowed from third alternative)

The resulting feature provides a set of built-in rules that modify a number when the number exists in the same container as an operator. When the built-in rule fires, the operator is erased and the number is appropriately changed. This feature can be used to count not only populations, as shown in the grasshopper solution, but also can be used to count event occurrences.

This feature enables solutions to the target problems without much deviance from the computational model (as with alternative 2), and without many extraneous or confusing object configurations (as with alternative 1 and 3). Built-in numeric operators were not added for operating on bar graphs because the bar graph solutions were relatively simple and seem easy to import, as shown by the solution in section 4.5.6.

5.4 The Resize Problem and Object Typing

One of the details of the ChemTrains system is that in order for an object in a rule pattern to match an object on the main display, the objects must appear identical. In using the previous prototype to build the tic tac toe simulation, if the game board is resized after the rules have been specified, the rules will no longer work appropriately because the sizes of the objects have changed. Furthermore, the cells of the board all have to be resized individually.

A related problem is that many ChemTrains solutions depend on a set of objects being identical, which means that the programmer has to either make copies or has to create exactly identical objects, which may be hard to do. No language feature existed for creating objects that have one single definition or for updating the definition of objects that are all identical. This is typically done in most languages with constructs for *typing*: a type is declared in one place in a program, and identical instances are created by referring to the name of a single type.

Two design alternatives for solving these problems were considered:

1. resizing all identical objects when an object changes shape, and
2. providing a palette of object types in use, allowing the user to change the global definition of an object by changing its representation in the palette.

The first alternative directly solves the main problem without any additional doctrine needed, but does not address the problem of creating objects that must be initially identical. This shortcoming can be addressed by directing the programmer with additional environment doctrine to copy objects in most cases and to create new ones objects only when necessary. The typing design alternative solves both problems directly, but additional doctrine is necessary to describe interaction with the palette of types. Because additional doctrine may be needed, the resizing alternative was chosen.

5.5 Grid Creation

In at least three of the target problems (grasshoppers, bouncing arrows, and checkers), a two dimensional grid of interconnecting places is needed in solving the problem. Even if doctrine is provided that guides the programmer in constructing these grids, the task of constructing a large grid is tedious, and an aid is needed. In other problems (counter, turing machine, and fsr) a one dimensional grid is needed, and a grid creation interface would also be useful. The requirements for such a grid creation tool was derived from possible solutions to the problems.

- **Bouncing Arrows:** Two possible solutions can be produced (figure 4-4 & 5-1). The first solution requires that every cell in the grid has 4 directed and labeled paths that point in each direction.

The second solution requires that all cells are connected by undirected and unlabeled paths.

- **Checkers:** There are also two possible grid representations for checkers. The first solution requires that only the playable squares are created, and they are connected diagonally by labeled and directed paths. The second solution requires that all 64 squares are created, vertically adjacent cells are connected by directed paths, and horizontally adjacent cells are connected by undirected paths. The first solution is a more natural representation for the problem.
- **The Turing Machine:** A solution to this requires that the Turing tape be represented as a one dimensional sequence of cells that is connected in both directions by directed and labeled paths.
- **Counter:** The solution requires that a string of cells is connected by directed and unlabeled paths.
- **Grasshoppers:** This solution requires that the cells of one main grid are interconnected by undirected and unlabeled paths. The paths don't have to be directed because the grasshoppers may move freely between the places. The solution also requires that each cell of the grid must contain other cells.

These solutions require a variety of design features in a tool for creating grids.

These features are stated as design questions:

1 Can the grid be positioned?
2 Can the cell size be specified?
3 Can the dimensions of the grid be specified?
4 Can paths connect adjacent cells?
4.1 Can paths be directed in either direction?
4.2 Can paths be undirected?
4.3 Can horizontal and vertical paths be specified separately?
4.4 Can diagonal paths be specified?
4.5 Can paths be labeled?
5a Can a cell be specified as multiple objects?
5b Can multiple grids be overlaid in the same display?
6 Can a grid be re-edited?
7 Can grid display be hidden?
7.1a Can different components of grid be hidden independently?
7.1b Can the grid be hidden as a whole?

Most of these features are necessary in order to solve the problems. Three decisions in this design space needed further analysis.

- Question 4.4 (whether to allow diagonal paths) is considered because it is required by one of the solutions to the checkers problem. This solution in addition to creating diagonal paths requires that every other cell in the field be somehow deleted. Since this may be either complicated or time consuming, and there is another solution, diagonal path creation was not added.

- Question 5a&b are alternative features for supporting an early grasshopper solution which required multiple overlapping grids. Design alternative 5a would allow the programmer to put together a complicated combination of objects within a cell. Design alternative 5b limits the programmer to creating different grids and overlaying them. The first alternative involves a more complicated interface, and the second alternative involves creating two different grids that in some way interact. The simpler alternative (5b) was chosen.
- Question 7.1a&b are alternative features for hiding parts of the grid. Alternative 7.1a would allow parts of the grid to be hidden separately. Alternative 7.1b would only allow the grid to be hidden or not. Since the flexibility of the more complicated first alternative was not shown to be necessary for any of the target problems, alternative 7.1b was chosen.

5.6 Hideable Layers of Abstraction

The current design allows objects in the display to be hideable. This enables the programmer to make a distinction between objects that are visible to the end user and internal representation objects necessary for simulating the appropriate behavior. One of the design alternatives related to the grid representation suggests that different grids need to be treated individually. Both of these language features enable the programmer to separate different layers of abstraction, but only in a limited way. These ideas could be generalized with the following design decisions:

1.5 Can groups of objects be lumped into one layer?
1.5.1 Can a layer be named?
1.5.2 Can objects in one layer interact with objects in another?
1.5.2.1 Can an object be limited to interaction with objects in its own layer?
1.5.3 Can a layer be hideable?

In contrast to the other design decisions, these alternatives did not directly arise from shortcomings involved in solving the target problems. These features may be only useful in simulations that are much more complicated. Since use of these features may only cause confusion, they were not added.

5.7 Mouse Interactions

The current design of ChemTrains enables the programmer to define simulations that are controlled by dragging objects around on the screen. Doctrine is already provided for programming this type of interaction (**create command rule**). This type of interaction is adequate for many applications but does not always provide an intuitive interface for the end user. Also the programmer has to understand this limitation of the language. In order to specify a wider class of interactions the language needs a way to specify

behavior associated with any user mouse click. Here are two design alternatives for specifying interaction associated with mouse clicks.

5.7.1 Specifying a Clickable Point in a Rule Pattern

One alternative is to enable the programmer to create a "click" in the pattern of a rule that would be displayed as a small icon such as an arrow with a flash of streaks coming out of it, or a point with a flash of streaks. One such click object may be placed anywhere within a pattern of a rule, and would mean that the rule may only fire in the specified region. For example, to specify that a particular icon may be deleted if it is clicked, the pattern would contain the icon and a click object within it, and the result would be empty. This feature would allow the programmer to specify that a click must occur in the intersecting area of multiple objects. This feature would be supported by the following doctrine:

create user click rule:
If the end user can control the simulation by clicking on area of the simulation display,
then create a rule that describes the appropriate action, place the objects that determine the clickable area into the pattern and result, and place a "click" pattern in the appropriate region in the rule pattern.

5.7.2 Specifying a Clickable Object in a Rule Pattern

Another alternative is to enable the programmer to specify that an object in a pattern must be clicked in order for the rule to fire. The doctrine supporting this feature is shown below. In using this feature to specify that a particular icon should be deleted when it is clicked, the pattern of a rule contains the icon and is specified as "clicked" and the result of this rule contains nothing. There is no way with this feature to specify that a click that must occur within the intersecting area of two objects.

create user click object rule:
If the end user can control the simulation by clicking on an object,
then create a rule that describes the appropriate action, place the object to be clicked in the pattern and result, and specify that the object is clicked.

The first feature described is more general and can be applied directly in more problems than the second feature. However, the second feature may be easier to understand and apply. Neither feature was added in the final prototype of ChemTrains. A future implementation should have one of these features.

5.8 Interpreting Rule Actions

One interesting set of design issues arises from the early decision that each rule should be written as a pattern picture/result picture pair, as opposed to a pattern picture/action description pair, or as a demonstration of the behavior.

With the pattern/result rule form, the explicit actions to be executed must be inferred from the differences between the two pictures. For example, the rule compiler interprets the rule shown in figure 5-2 to mean that the ghost moves from one house to a connected house, instead of interpreting the rule as no-op rule. The rule compiler makes this inference by comparing the relative positions of the objects in the pattern and result pictures. Since differences in the absolute positioning of the objects are considered irrelevant, the rule specifies no change in the positioning of the houses.

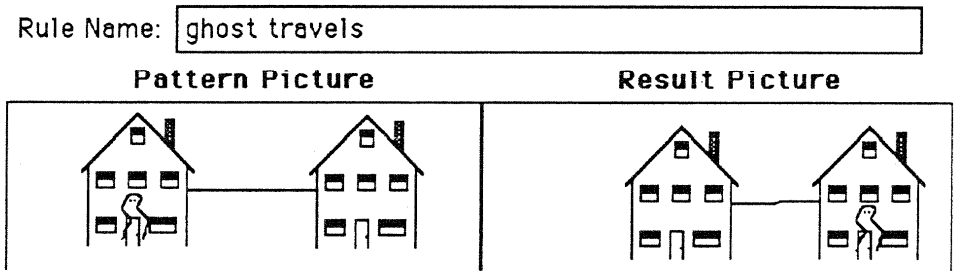


Figure 5-2: A rule to move a ghost among connected houses.

This is a fairly simple ambiguity compared to possible ambiguities that can occur if the differences between patterns and results are completely unrestricted. Figure 5-3 is a rule that points out the kinds of action ambiguities that can occur. In this rule there are are multiple interpretations of where the new house, the new ghost, the new rectangle and the new connection should be added, and whether any of the new result objects simply are added or replace objects matched in the pattern.

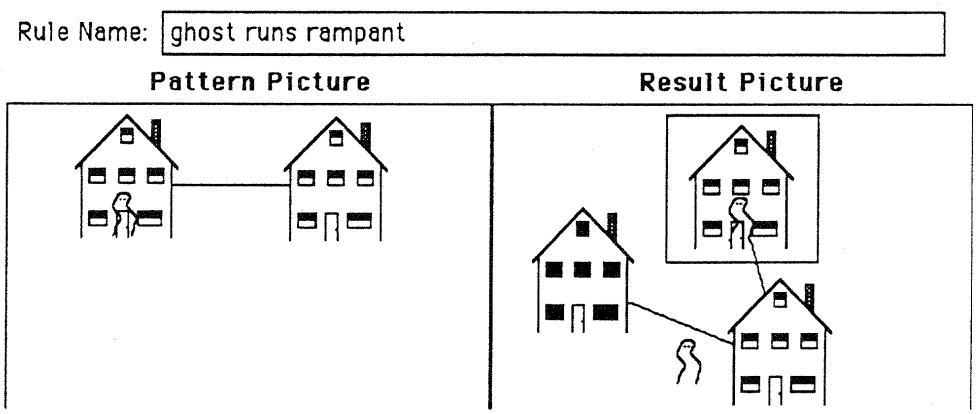


Figure 5-3: A rule filled with ambiguity.

Action ambiguities occur in rules when new objects are introduced into the result and when objects from the pattern are moved to a different position in the result. The amount of ambiguities that the system deals with can be reduced by not allowing certain types of differences between the pattern and result. The types of differences that were considered in the design were:

1 Can a rule add or remove an object?
1.1 Can a rule add an object inside a matched place?
1.2 Can a rule add an object not inside a matched place?
1.2.1 Can a rule add an object that replaces a matched object?
1.2.2 Can a rule add an object that's connected to a matched object?
1.2.3 Can a rule add an object that's adjacent to a matched object?
1.2.4 Can a rule add an object that's a container for a matched object?
1.2.5 Can a rule add a label to a path?
1.2.6 Can a rule add an object on a path?
2 Can a rule add or remove a place?
2.1 Can a rule add a place that contains a matched object?
3 Can a rule add or remove a path?

The design decisions were made based on the number of solutions that were enabled. The final system allows new objects to be created either inside a matched container object, or replacing a matched object from the pattern if the new object is not already inside another object. Objects may not be freely created if they are simply adjacent, connected to, or containing matched objects from the pattern picture. Paths may also be added in a rule. Enabling an object to be moved or to be created inside another object is a fundamental aspect of the system that is necessary in most of the solutions. Enabling an object to replace an object from the pattern is used in the house solutions to switch a house from appearing lit to appearing unlit.

The addition of some of these features will help the expressiveness of the language by providing a less restricted syntax for the rules. For example, the ability to create an object connected to an existing object would enable rules in the Turing machine simulation to create additional tape containers when the tape head has reached either end of the finite Turing tape.

5.9 Design Alternative Summary

This section summarizes all the design alternatives. The ChemTrains prototypes and the designs that have been considered from January 1990 to July 1991 are defined in terms of answers to the design questions. The design space is described as a hierarchy of yes/no questions that is shown in the table 5-1. The question numbers show the decision's place in the design hierarchy. Numbers that share a common prefix number are only relevant questions if the question with the common prefix has been answered as true. For example question 1.5.1 & 1.5.2 are only relevant if the answer to question 1.5 is true. Questions that are mutually exclusive are labeled with letters rather than numbers. For example, questions 1.4a and 1.4b are mutually exclusive. Other interdependencies between the design alternatives are not shown. 't' denotes a yes answer to a question, and '-' denotes a no. In some cases, '-' may mean the designer was not aware of the possibility at the design time, even though the logical progression of the design may require it.

Seven different designs are displayed in the design space table. The first four preceded the work that is described in this thesis.

- **IP** is the Initial Prototype of ChemTrains as completed by John Rieman in the fall of 1989.
- **ZT** is the ZeroTrains design of June 1990.
- **ST** is the ShowTrains design of June 1990.
- **OT** is the OPSTrains design of June 1990.
- **ID** is an Intermediate Design of ChemTrains considered in January 1991 as this thesis work was beginning.
- **FD** is the Final Design of ChemTrains91 as prescribed by the decisions described in this chapter.
- **FP** is the Final Prototype of ChemTrains91 as directed by the final design.

The design questions are separated into 8 main areas:

1. objects,
2. places,
3. paths,
4. rules,
5. abstractions,
6. grids,
7. graph output, and
8. importing.

The hierarchy shown is one possible way to decompose the design space. Some separations that are natural for one design may not be natural for another design. For example, the distinction between objects (Q1.*) and places (Q2.*) is very relevant in early designs but not for later designs.

Table 5-1: The ChemTrains Design Space (part 1 of 3)

Q#	Design Decision	IP	ZT	ST	OT	ID	FD	FP
1	Are graphical objects allowed in the simulation?	t	t	t	t	t	t	t
1.1	Are objects nameable?	t	-	t	-	-	-	-
1.2	Are objects hideable?	-	-	t	t	t	t	t
1.3	Can objects be grouped together as a single object?	-	-	-	-	t	-	-
1.4a	When an object changes shape does it try to update objects like it?	-	-	-	-	t	t	t
1.4b	Are objects typed and editable on a palette?	-	-	-	-	-	-	-
1.4b.1	Do object types have inheritance?	-	-	-	-	-	-	-
1.5	Can groups of objects be lumped into one layer?	-	-	-	-	-	-	-
1.5.1	Can a layer be named?	-	-	-	-	-	-	-
1.5.2	Can objects in one layer interact with objects in another?	-	-	-	-	-	-	-
1.5.2.1	Can an object be limited to interaction with objects in its own layer?	-	-	-	-	-	-	-
1.5.3	Can a layer be hideable?	-	-	-	-	-	-	-

Table 5-1: The ChemTrains Design Space (part 2 of 3)

Q#	Design Decision	IP	ZT	ST	OT	ID	FD	FP
2	Are graphical places allowed in the simulation?	t	t	t	t	t	t	t
2.1	Are places nameable?	-	-	t	-	-	-	-
2.2	Are places hideable?	t	-	t	t	t	t	t
2.3	Can places have any shape?	-	-	t	t	t	t	t
2.4	Can places be nested?	-	-	t	t	t	t	t
2.5	Can places overlap?	-	t	-	t	t	t	t
2.6	Can any place be treated as an object? (are objects = places?)	-	-	-	t	t	t	t
3	Are paths that connect places allowed in the simulation?	t	t	t	t	t	t	t
3.1	Are paths nameable?	-	-	t	t	t	t	t
3.2	Are the path ends nameable?	t	t	-	-	-	-	-
3.3	Are paths hideable?	t	-	t	t	t	t	t
3.4	Can paths be restricted to only allow one-way traffic?	t	-	t	t	t	t	t
3.5	Can paths allow two-way traffic?	t	t	-	t	t	t	t
3.6a	Can an object travel a path by being placed onto the path?	-	t	t	t	t	-	-
3.6b	Can an object travel by being placed in an "ante-room?"	t	-	-	-	-	-	-
3.6c	Is the path to be travelled inferred from differences in pattern & result?	-	-	-	-	-	t	t
3.7	Can a "filter" on a path control when objects may cross?	t	-	-	-	-	-	-
3.8	Can a path be directed from a place back to the same place?	t	-	-	-	-	t	-
4	Is simulation behavior described with graphic rules?	t	t	t	t	t	t	t
4.1	Can a rule be created by demonstration?	-	-	t	-	-	-	-
4.2	Can the pattern and action of a rule be viewed and edited?	t	t	-	t	t	t	t
4.3	Can objects be matched in a rule pattern?	t	t	t	t	t	t	t
4.3.1	Is the "inside" constraint used to match objects in places?	t	t	t	t	t	t	t
4.3.1.1	Can separate multiple places be used in a rule pattern?	-	-	t	t	t	t	t
4.3.1.2	Can an arbitrary nesting of places be used in a rule pattern?	-	-	-	t	t	t	t
4.3.2	Can paths connecting objects be matched in a rule pattern?	-	-	-	t	t	t	t
4.3.3	Is the "adjacency" constraint used in pattern matching?	-	-	-	-	-	-	-
4.3.4	Can objects in a rule pattern be variablized?	-	-	t	t	t	t	t
4.3.4.1a	Is variablization explicitly described by the programmer?	-	-	-	t	t	t	t
4.3.4.1b	Is variablization of objects implied by the rule description?	-	-	t	-	-	-	-
4.3.4.2	Can objects in a rule action or result be variablized?	-	-	t	t	t	t	t
4.3.4.3	Can places also be variablized?	-	-	-	t	t	t	t
4.3.5	Can objects in a rule pattern be negated?	-	-	t	t	t	-	-
4.4	Can a click be recognized in the pattern of a rule?	-	-	-	-	-	t	-
4.4.1a	Can a region be specified as clickable with a click object?	-	-	-	-	-	t	-
4.4.1b	Can an object in the pattern be specified as clickable?	-	-	-	-	-	-	-
4.5	Is a rule specified as pattern picture / action pair?	-	-	-	-	-	-	-
4.6	Is a rule specified as pattern picture / result picture pair?	t	t	t	t	t	t	t
4.6.1	Can a rule add or remove an object?	t	t	t	t	t	t	t
4.6.1.1	Can a rule add an object inside a matched place?	t	t	t	t	t	t	t
4.6.1.2	Can a rule add an object not inside a matched place?	-	-	-	-	-	t	t
4.6.1.2.1	Can a rule add an object that replaces a matched object?	-	-	-	-	-	t	t
4.6.1.2.2	Can a rule add an object that's connected to a matched object?	-	-	-	-	-	t	-
4.6.1.2.3	Can a rule add an object that's a container for a matched object?	-	-	-	-	-	t	-
4.6.1.2.4	Can a rule add an object that's adjacent to a matched object?	-	-	-	-	-	t	-
4.6.1.2.5	Can a rule add a label to a path?	-	-	-	-	-	t	-
4.6.1.2.6	Can a rule add an object on a path?	-	-	-	-	-	-	-
4.6.2	Can a rule add or remove a place?	-	-	-	t	t	t	t
4.6.2.1	Can a rule add a place that contains a matched object?	-	-	-	-	-	t	-
4.6.3	Can a rule add or remove a path?	-	-	-	-	t	t	t

Table 5-1: The ChemTrains Design Space (part 3 of 3)

Q#	Design Decision	IP	ZT	ST	OT	ID	FD	FP
4.7	Is there a conflict resolution strategy?	t	t	t	t	t	t	t
4.7.1a	Does conflict resolution pick rules randomly?	-	t	t	-	-	-	-
4.7.1b	Is rule ordering used in conflict resolution?	t	-	-	t	t	t	t
4.7.1b.1a	Is rule ordering used to define sequence of rules to fire?	t	-	-	-	-	-	-
4.7.1b.1b	Is rule ordering used to define rule priority?	-	-	-	t	t	t	t
4.7.1b.2	Does conflict resolution pick the data randomly?	-	-	-	-	t	t	t
4.7.1c	Is conflict resolution similar to OPS5 (recency of data)?	-	-	-	-	-	-	-
4.7.2	Can a probabilistic weight be given to a rule?	-	-	-	-	-	t	t
4.7.2a	Is the probabilistic weight used to choose percentages of data?	-	-	-	-	-	-	-
4.7.2b	Is the probabilistic weight used to choose between rules of a group?	-	-	-	-	-	t	t
4.7.2b.1	Is their a way to separate probabilistically weighted rule groups?	-	-	-	-	-	t	-
4.7.3a	Can rules fire on the same data more than once?	t	-	-	-	-	-	-
4.7.3b	Does refraction prevent rules from firing more than once?	-	-	-	t	t	t	t
4.7.3c	Can rules fire on the same data more than once, if specified by user?	-	-	-	-	-	-	-
4.8	Is numerical computation of any sort supported?	-	-	-	-	t	t	t
4.8.1a	Is counting supported by a "count" operator?	-	-	-	-	t	-	-
4.8.1b	Is counting supported and displayed by an external monitor?	-	-	-	-	-	-	-
4.8.1c	Is counting supported by primitive operators?	-	-	-	-	-	t	t
4.8.1c.1a	Are primitive operators supported by importing symbolic versions?	-	-	-	-	-	-	t
4.8.1c.1b	Are primitive operators supported with built-in rules?	-	-	-	-	-	t	t
4.8.2	Is arithmetic (+, -, /, *) supported?	-	-	-	-	t	-	-
4.8.3	Can numbers in the simulation effect the size or position of an object?	-	-	-	-	t	-	-
4.8.4a	Can symbolic versions of a bar graph counter be imported?	-	-	-	-	-	t	t
4.8.4b	Are primitive bar graph operators supported by built-in rules?	-	-	-	-	-	-	-
5	Are object abstractions available?	-	-	-	-	t	-	-
5.1	Can abstractions be arbitrarily nested?	-	-	-	-	t	-	-
6	Is a grid representation provided?	-	-	-	-	t	t	t
6.1	Can the grid be positioned?	-	-	-	-	t	t	t
6.2	Can the cell size be specified?	-	-	-	-	t	t	t
6.3	Can the dimensions of the grid be specified?	-	-	-	-	t	t	t
6.4	Can paths connect adjacent cells?	-	-	-	-	t	t	t
6.4.1	Can paths be directed in either direction?	-	-	-	-	-	t	t
6.4.2	Can paths be undirected?	-	-	-	-	t	t	t
6.4.3	Can horizontal and vertical paths be specified separately?	-	-	-	-	-	t	t
6.4.4	Can diagonal paths be specified?	-	-	-	-	-	-	-
6.4.5	Can paths be labeled?	-	-	-	-	-	t	t
6.5a	Can a cell be specified as multiple objects?	-	-	-	-	-	-	-
6.5b	Can multiple grids be overlaid in the same display?	-	-	-	-	-	t	t
6.6	Can a grid be re-edited?	-	-	-	-	-	t	t
6.7	Can grid display be hidden?	-	-	-	-	t	t	t
6.7.1a	Can different components of grid be hidden independently?	-	-	-	-	-	-	-
6.7.1b	Can the grid be hidden as a whole?	-	-	-	-	t	t	t
7	Is a monitoring facility provided to display simulation data?	-	-	-	-	t	-	-
7.1a	Is the graph display controlled by operators available to the user?	-	-	-	-	-	-	-
7.1b	Is input to the graph display automatically controlled by the system?	-	-	-	-	t	-	-
7.2a	Is the graph display displayed within the simulation?	-	-	-	-	-	-	-
7.2b	Is the graph display displayed on a separate window?	-	-	-	-	t	-	-
8	Can a simulation be imported into an existing simulation?	-	-	-	-	-	t	t
8.1	Can the simulation picture and rules be imported independently	-	-	-	-	-	t	t

This design space table displays the evolution of ChemTrains designs of the last two years. The design rationale of the designs of June 1990 (ZeroTrains, ShowTrains, and OPSTrains) are discussed in a previous design document. (Lewis, Rieman, & Bell, 1990) During the design of ChemTrains91 the design space table expanded from 28 design questions to the 111 questions shown above. Of the 83 added questions, 22 were added to more clearly define the known ChemTrains design space, and the remaining 61 questions were added as possible extensions. Of these 61 design questions, many are closely related to previous design work, such as some subtle decisions about the conflict resolution strategy, and many other questions explore more radical extensions, such as object abstracts, grids, and numerical computation. Here is a summary of the kinds of decisions that occurred during the design of ChemTrains91.

Abstract features that were borrowed from computer science ideas usually were not added in the final design. These features were considered because they seemed like they might generally simplify solutions. These ideas failed to pass the programming walkthrough tests for a variety of reasons:

- the features required too much additional doctrine;
- alternative solutions were found that used existing features; and/or
- the feature did not really apply to any of the target problems.

Here are features that were derived from computer science ideas:

- The conflict resolution strategy of recency of data and specificity of rule (Q4.7.1c) was considered because the strategy has been shown to be useful in controlling the execution of rules in large complex rule-based programs, such as XCON. (McDermott, 1982) The simpler strategy of choosing rules based on rule ordering more easily applied to the problems and provided an appropriate amount of power.
- A feature for typing objects (Q1.4b) was considered because most programming languages have a feature for defining types. The alternative of automatically updating objects when any instance is resized, proved to be more economical. Object typing offered no additional benefits than the simpler alternative.
- Negation (Q4.3.5) was considered because most rule-based and logic-based languages include **and, or**, and **not** logic. **Or** and **and** logic are both supported implicitly in ChemTrains. A feature for **not** logic would have to be explicit. The advice for supporting such a feature was weighed against alternative advice for supporting solutions that did not require **not** logic.
- A feature for defining object abstractions (Q5) was considered so that problems requiring more complex behaviors could be broken down into more manageable parts. This feature borrowed ideas from macro processors in languages such as Common Lisp. The object abstraction feature overlapped conceptually with rules in

the same way that the concepts of how to write a function and how to write macro overlap in procedural languages. The cost of explaining not only abstractions but also the subtle differences between abstractions and rules outweighed the benefit of having shorter solutions to large problems.

- The feature of allowing hideable object layers (Q1.5) was also considered as a method of decomposing the problem. This feature would allow separation of different abstract layers and was considered mostly because it seemed cool. Since it did not really simplify any of the solutions, it was not added.

Features that were invented and considered to fill specific needs often ended up in the final design. In contrast to abstract features derived directly from computer science, these features derived from working on the problems, either by doing walkthroughs or by experimenting with the prototype.

- The feature of automatically resizing identical objects (Q1.4a) was added simply so that objects of the same type can be modified with one action, making a more formal type mechanism unnecessary.
- The feature of inferring movement along paths from differences between the pattern and result of a rule (Q3.6c) made the feature of more explicitly defining movement unnecessary (Q3.6.a).
- The feature of inferring object replacement from rule differences between the pattern and result of a rule (Q4.6.1.2.1) was useful in some target problems without complicating the model of computation.
- The feature of selecting objects randomly during pattern matching (Q4.7.1b.2) was added initially to accommodate maze solutions and has been useful in solutions to other problems.
- The feature of associating probabilities with rules (Q4.7.2) is a very useful extension that is necessary for some target problems, but it requires additional explanation.
- A simple feature for counting (Q4.8) was found that solved the problems and did not complicate the model of computation.
- The grid creation interface (Q6.6) was added to reduce the tedious task of constructing a two dimensional array of containers.
- Allowing simulations to be imported (Q8) was added so that generally useful simulations can be reused.
- The possible feature of inferring placement of new objects based on new connections (Q4.6.1.2.2) is useful in some target problems without complicating the model of computation.
- The possible feature to allow user mouse clicks in a pattern (Q4.4) is a useful extension.

Features were added more often than subtracted as the system progressed. This creeping featurism was caused partly by a growing number of target

problems and partly by a lack of knowledge of generally useful simplifying computational approaches. Although features such as a counter and probabilities on rules depart from the basic concepts of the language, they are necessary in order to enable solutions.

The design search has not been completed. There are both related design alternatives that haven't been considered, and different combinations of old alternatives that haven't been considered. For example, the ShowTrains feature for creating rules by demonstration (Q4.1) was never re-introduced into later designs because of possible interferences with new additions to the computational model. Once a computational model is fixed for ChemTrains, it might be worthwhile to consider re-introducing automatic creation of rules through demonstration.

The working system is not complete. Even if the design of the language is considered complete, the working language and its supporting environment are not. Many of the following features should be considered in a future working system. The first three features have already been accepted in the final design, but have not been implemented.

- Allowing a click to be specified in pattern (Q4.4).
- Permitting more liberal modifications in a rule result, and supporting these actions with an intelligent rule compiler. (Q4.6.1.2.*)
- Allowing groups of probabilistic rules to be separated. (Q4.7.2b.1)

The following syntactic and environment features are at a slightly lower level than is described in the design space table.

- The current pictorial syntax is similar to MacDraw. The picture editing part of the system could be extended to handle more MacDraw-style features, such as object grouping, shading of objects, and two dimensional scrolling.
- Two dimensional scrolling is particularly important because some simulations may require a large or expandable space. For example a Turing Machine simulation may require a tape that grows as the simulation runs.
- Another possible environment feature would be to enable a simulation to be split into multiple windows. With this feature the Turing Machine could be separated into individual windows for the Turing tape, the input tape, the finite state machine, and the state transition table. The separation into different windows would allow the different aspects of a simulation to be scrolled, viewed, or hidden more independently. For example in the Turing simulation, the Turing tape head may shift to the left as the input tape head shifts to the right, requiring the simulation to scroll independently in both directions.
- The construction of individual rules is a little awkward in the current programming environment. An additional feature

should be considered for reducing the work of putting together the pattern and result pictures of a rule from existing components in the simulation display or another rule.

5.10 Design Process Summary

The methodology of Problem-Centered Design with Walkthrough Feedback as described in chapter two provided a framework for generating design alternatives and making design decisions. Here are the main features of the design methodology that were useful in the design of ChemTrains91.

The process of problem solving provided much guidance. One of the first steps in doing a programming walkthrough is producing a solution. This provided a first line of defense against unworkable features. Some features, such as being able to put a percentage on a rule to signify the percentage of data to accept (Q4.7.2a), did not require further walkthrough analysis. The fact that a working solution couldn't be found forced the designer to look for an alternative feature.

The process of writing walkthrough summaries forced the designer to isolate the most difficult aspects of the language. When a programming walkthrough is written, it appears as an almost sterile sequence of steps. The length of the steps gives a general idea of how long it might take to write the program, but it doesn't give a good idea of the stumbling blocks that a programmer may have along the way. Walkthrough summaries explicitly try to uncover these possible difficulties. In so doing, important design issues percolate out of the walkthroughs.

Although the design methodology did not provide explicitly for the representation of micro problems, design rationale often revolved around small pieces of problems. In the previous design of ChemTrains, the target problems were broken into what were called "micro problems" that focussed on specific requirements, which helped to focus the design discussions. In the design of ChemTrains91, the process of isolating design issues often corresponded directly with isolating micro problems of the target problems. Because micro problems played an integral role in making the design decisions, they should be more explicitly represented during the process.

The process of design became a game of erasing points of walkthrough summaries by making changes in the language. The individual cycles of the iterative design process were expected to be long and time consuming because there are so many steps involved. Although the first few iterations of design were long, the iterations became shorter as the walkthroughs stabilized. Many iterations of design were done simply by noting a particularly ugly part of a walkthrough summary, generating an alternative that possibly alleviates it, and exploring whether the feature makes sense. For example, the feature

that allows a rule to replace an object not inside another object (Q4.6.1.2.1) was added because the previous design required steps that stuck out like a sore thumb in the walkthrough summary of the house lighting problem.

Design decisions often hinged on subtle differences between walkthroughs. Especially in making close decisions, the designer had to look carefully at differences in isolated parts of walkthroughs and doctrine. For example, in deciding whether to add negation (Q4.3.5) or to supply advice for doing without negation, the alternative pieces of doctrine and the walkthrough steps on four different problems had to be isolated and compared. Some problems favored not adding negation because the use of negation led to longer solutions; other problems favored adding negation because it seemed a little more natural; and the comparison of doctrine also slightly favored adding negation.

Iterative design was used to explore and extend the design space. The process of doing programming walkthroughs and writing walkthrough summaries was useful not only in comparing designs but also in generating new alternatives and refinements that fed back into the design loop. Iteration was a key to the search process. For example, the design of the grid creation interface (Q6.*) evolved in several iterations:

1. Walkthrough analysis on the grasshopper problem showed that automatic generation of grids was necessary. This idea led directly to a set of design alternatives and an initial design.
2. Walkthrough analysis of this design on other problems pointed out additional features, which were added to the design space table.
3. Walkthrough analysis on five representative problems showed exactly which of the possible features were appropriate.

The design space table was a necessary resource. The organization of yes/no questions enabled the designer to find existing design alternatives quickly, to define any design in the space precisely, to compare designs easily, to record new design alternatives efficiently, and to sort out design issues simply. The format of the design space table was just rich enough to ease readability and just simple enough to ease extensibility. Design alternatives shown in a simpler format would be harder to look up. Design alternatives shown in a more complex format would be harder to add. The design space displayed in a much richer format, such as Questions, Options, and Criteria (MacLean, Young, Bellotti, & Moran, 1992) would help show more inter-dependencies between the issues. Since the design decisions are guided here by the walkthrough analysis rather than the design space table, a richer format is not needed. The purpose of the design space table was simply to grease the engine of design search, and it did.

6 User Testing

The purpose of doing user testing was to test the writability of ChemTrains programs in general and the newer features in specific. The testing was done on two types of users: computer scientists and non computer scientists who have a hard science or engineering background. The second group of subjects are the users that the system was intended and built for. Computer scientists were tested for the purpose of comparison.

The subjects were given a sequence of small tasks as part of one larger task. The individual tasks tested very specific issues, such as path creation, rule ordering, negation, and rule probabilities. Since the subjects had no previous knowledge of ChemTrains, and the testing was done in a single two hour session, the experiment ended up testing learnability as much as writability. In addition to this experiment, two subjects came back for another session to work on a larger problem that was not decomposed into smaller tasks.

This chapter first describes the main testing procedure and the task. Next the results of the experiment are described. Then the testing on the larger problem is described. Finally, the chapter makes some conclusions about the programmability of the language and makes further conclusions about the design methodology based on a comparison of user testing with walkthrough analysis.

6.1 The Grasshopper Experiment

For each subject the following testing procedure was followed:

1. The subject was asked to read and sign an informed consent form.
2. The subject filled out a background questionnaire on their training and experience in computer science, programming languages, and rule-based systems.
3. The subject was informed that the experiment was intended for evaluating the writability of the language rather than the usability of the environment, and that they would not use the environment but would have to issue instructions to the experimenter, acting as a medium.
4. A fifteen minute tutorial on ChemTrains was given that covered the terminology and the model of computation. These basics were demonstrated using the House Lighting simulation. The tutorial

covered almost all of the information in section "3.2 Programming Concepts," and was intended to describe the basic language abstractions without giving away much of the problem-solving knowledge (the doctrine). The following terms were defined during the demonstration:

simulation display, object, rectangle, oval, line segments, text string, icon, identical, inside, contains, container object, path, connects, directed path, non-directed path, path label, rule, pattern picture, result picture, variable, rule execution, rule pattern matching, parallel rules, rule ordering, rule priority, execute mode, and edit mode.

The simulation display terminology was introduced as a way to describe and talk about a simulation. The usage of these ideas was left up to the subject during problem solving, although some of the usage was clear from the demonstration of the house lighting simulation.

5. The subject was instructed to think aloud while working.
6. An audio and/or video recording was started.
7. Each subject was given an initial simulation to work from that contained a display but no rules. The subjects were given a sequence of tasks that started simple and ranged in difficulty. When a subject got stuck on a task and asked for advice, they were presented with a section of doctrine as shown in section "3.3 Programming Techniques." If a subject got stuck on applying the doctrine, the experimenter would sometimes help.
8. After completion of the tasks the subjects were asked to comment on the main difficulties they had with the language.
9. The recording was stopped.

The overall task is taken from the grasshopper target problem, and is divided into subtasks each of which tests a different feature or combination of features of the language. Figure 6-1 shows the initial display of the simulation as a subject would see it. Table 6-1 is the task sheet given to each subject.

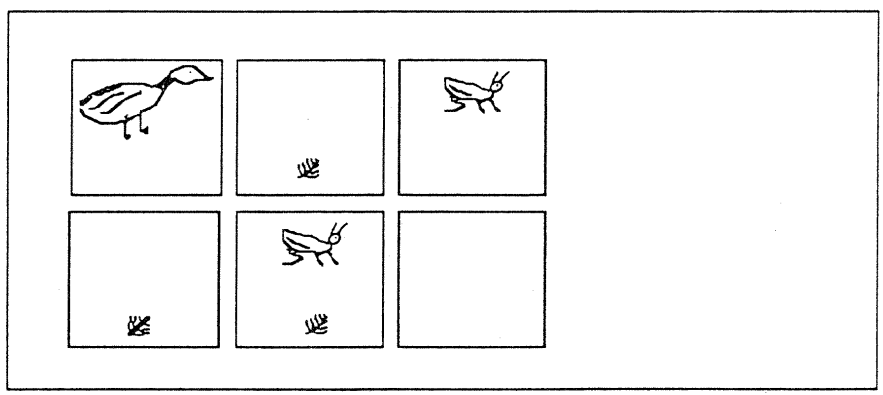


Figure 6-1: Initial simulation display of the grasshopper problem given to the subjects.

Table 6-1: Task sheet given to subjects.

Simulation of Grasshoppers in ChemTrains

Overall Task: Simulate a display that shows seasons changing, grasshoppers jumping around, avoiding ducks, and eating grass in the summer, and edible and poisonous grass growing in the spring.

This overall task can be solved by incrementally working on subtasks of the problem shown below. Solve the simulation tasks by writing rules. When necessary add new picture containers, objects, or paths to the simulation display to solve the tasks.

Task 1: Make the simulation have a grasshopper eat a piece of grass, when they are in the same rectangle.

Task 2: Make the simulation add a piece of grass for every rectangle in the picture.

Task 3: Make the simulation move the grasshoppers among the different rectangles, but only to adjacent rectangles that have a piece of grass and that don't have a duck.

- **Task 3a:** Allow the grasshoppers to leap around the rectangles.
- **Task 3b:** Force a leap only to adjacent rectangles.
- **Task 3c:** Force a leap only if food is available.
- **Task 3d:** Force a leap only if duck is not inside the destination rectangle.

Task 4a: Simulate the four seasons changing.

Task 4b: Modify the simulation so that grasshoppers only move and eat in the summer.

Task 5: Have the simulation produce poisonous grass 10% of the time and edible grass 90% of the time in the spring.

Task 6: Complete the overall task by having grasshoppers die when eating a poisonous grass.

6.2 Grasshopper Testing Results

Two groups of subjects were given the tasks:

- **Group A:** The computer science group consisted of five graduate students in computer science, all of whom had a lot of programming background in an assortment of languages, including C, Lisp, and Pascal. Two of the five subjects had programmed before in Prolog or a rule-based language. Two others were familiar with the idea of rule-based languages.

- **Group B:** The non computer science group consisted of nine people, mainly graduate students. This group included three electrical engineers, three physicists, a geologist, a software product manager, and a fifth grader fluent in MacDraw. The subjects had taken an average of one semester of computer programming, which was usually a college course in Fortran. Six of the nine subjects had not programmed a lot; the other three were experienced programmers without any computer science training. All nine subjects did not know much about rule-based or logic-based programming.

As the subjects worked on the tasks, they encountered some difficulties. Four levels of problem solving difficulty were noted:

1. the subject solved the task with little or no difficulty;
2. the subject had some difficulties or was unsure of the solution, but did not need any advice in overcoming the difficulty;
3. the subject needed doctrine advice and applied it correctly; or
4. the subject needed doctrine advice and needed help applying it.

For each of the tasks, the results of testing are compared with the information given in the demonstration and the required doctrine. This section concludes with a summary of these results.

6.2.1 Grasshopper Eats

"**Task 1:** Make the simulation have a grasshopper eat a piece of grass, when they are in the same rectangle."

This task tests the ability to create a rule to simulate deletion, which is guided by **create rule** and **deletion action** doctrine. Figure 6-2 shows the expected solution.

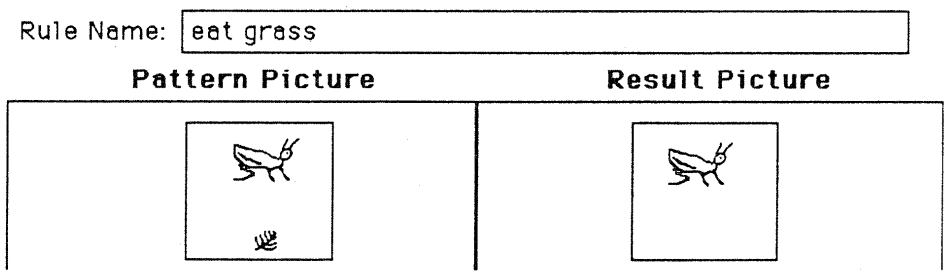


Figure 6-2: Rule describing a grasshopper eating grass.

All subjects were able to solve this task without difficulty, despite the fact that they had not been shown a rule that demonstrates deleting an object in a rule. The two rules of the house lighting solution only demonstrate replacement taking place within a rule. Several subjects noted that the eaten grass could be either deleted (as shown) or placed inside the grasshopper.

6.2.2 Grass Grows

"Task 2: Make the simulation add a piece of grass for every rectangle in the picture."

This task tests the ability to create a rule to simulate addition, which is guided by **create rule** and **addition action** doctrine. The task is confused by the issue of whether a rule may fire infinitely on the same data. Figure 6-3 shows the expected solution.

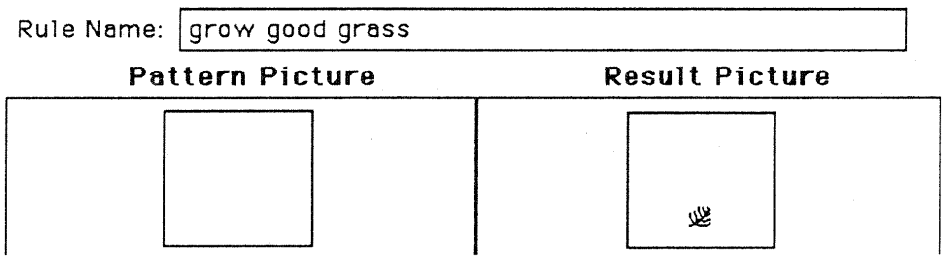


Figure 6-3: Grow grass rule.

All subjects solved the task without need for doctrine; however, one of the A group and four of the B group felt that multiple rules needed to be written, one rule for each of the cases of things possibly existing already in a rectangle. These subjects expected to need to write what one subject called a "very literal graphic representation" when writing a rule pattern. Four of these subjects wrote the correct rule first, thinking that it would only execute on empty rectangles. The other subject wrote a rule that had each of the rectangle conditions in one single rule, thinking that the conditions within one rule would be "or"ed. This subject realized the mistake when the rule was tested.

Since the pattern of the picture can always match something in the pattern, four of the A group and three of the B group were curious why the working rule did not execute infinitely. Three of the computer scientists were not satisfied until the principle of refraction was fully explained.

The fifth grader initially wrote a biologically more literal rule that grows more grass only if grass already exists. Since this rule would execute infinitely, she was asked to only grow one grass at a time even if there is no grass, and she appropriately generated the correct rule.

6.2.3 Grasshopper Leaps

"Task 3: Make the simulation move the grasshoppers among the different rectangles, but only to adjacent rectangles that have a piece of grass and that don't have a duck."

This task was broken down into 4 subtasks. Figure 6-4 shows the required representation and figure 6-5 shows the rule for the full solution. Subjects who attempted to solve this task all at once quickly gave up and worked on the individual subtasks.

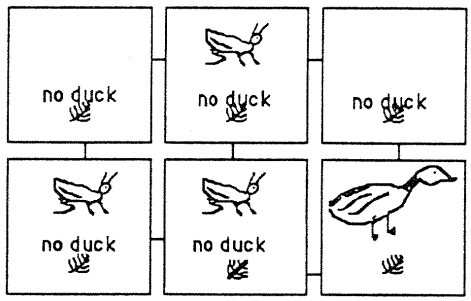


Figure 6-4: Simulation Display enhanced with "no duck" markers and non-directed paths necessary for task 3.

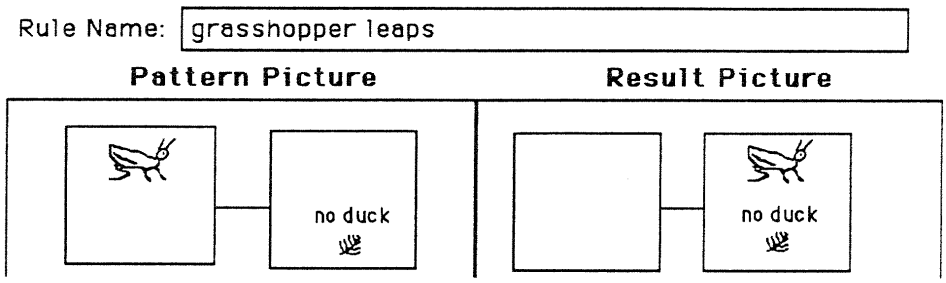


Figure 6-5: Complete grasshopper leap solution for task 3.

6.2.4 Grasshopper Leaps Randomly

"**Task 3a:** Allow the grasshoppers to leap around the rectangles." This task tests the ability to create a rule to simulate movement, which is guided by **create rule** and **movement action** doctrine.

All of the group A subjects and seven of the group B subjects solved this without difficulty. Of the two subjects who had difficulty, one of them needed the **movement action** advice. Another subject thought that all six rectangles should be in the pattern in order for a grasshopper to jump.

Three other subjects appropriately generated the right rule but weren't sure if it would work because they thought of the rule as an identity that wouldn't do anything. The only difference between the pattern and result pictures in the working rule is that the grasshopper is in a rectangle that is in a different relative position. They weren't sure if the system would understand the difference.

6.2.5 Grasshopper Leaps to Adjacent Rectangles

"**Task 3b:** Force a leap only to adjacent rectangles." This task tests the ability to modify the representation of the simulation display and modify a rule to utilize that representation. In particular, it tests the ability to use paths to represent adjacency, suggested by **draw non-directed path** doctrine.

All of the group A subjects and two of the group B subjects solved the problem without difficulty, even though this possible use of paths was not mentioned in the tutorial. Of the seven who had difficulties, three eventually solved the task without doctrine, and the other four needed and applied the **draw non-directed path** advice without help. All seven of the subjects thought that adjacency would or should matter, even though the tutorial included a written and spoken statement that "adjacency **does not** matter." Three of the subjects came up with attempts at solutions that had multiple rectangles to represent positions below and to the side of a rectangle with a grasshopper. Figure 6-6 shows a typical attempt. One of these subjects drew a pattern as shown in Figure 6-6 but drew a result with three grasshoppers in each of the other rectangles in the result, thinking that somehow the result would be "or"ed.

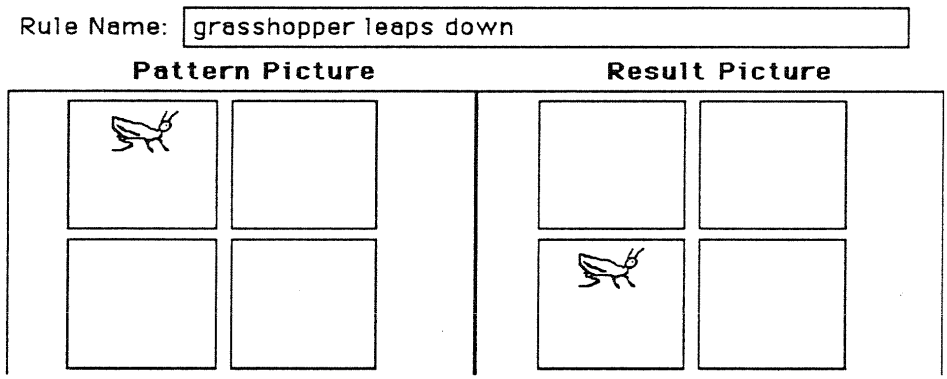


Figure 6-6: Common but unworkable attempt to leap to adjacent square.

One subject used larger rectangles as containers to represent adjacency (shown in figure 6-7). Four large rectangles are used to represent horizontal adjacency and three large rectangles are used to represent vertical adjacency. Since the horizontal and vertical large rectangles are different shapes, this rectangle needed to be variablized in the rule so that it can work in both cases. The subject suggested writing two rules. In order to make the simulation work using one rule, the subject needed the wildcard match advice and applied it appropriately.

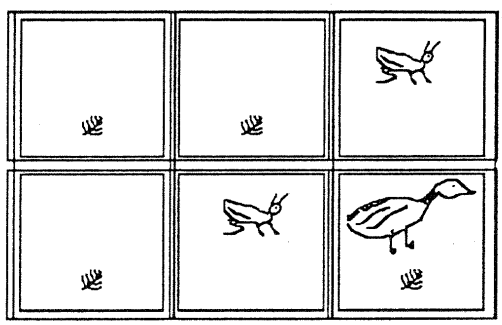


Figure 6-7: Representing adjacency with rectangular containers.

6.2.6 Grasshopper Chases Grass

"**Task 3c:** Force a leap only if food is available."

This task is a necessary subtask of the overall task. All subjects solved this without difficulty.

6.2.7 Grasshopper Avoids Ducks

"**Task 3d:** Force a leap only if duck is not inside the destination rectangle."

This task tests the ability to test for the absence of an object, which is guided by **draw empty container marker** and **absence testing doctrine**.

No one solved this problem using an empty marker solution. Before being subjected to the doctrine, they had several interesting attempts and ideas:

- Explicit statement of negation was considered by four of the group A subjects and one of the group B subjects. When a negation operator in a rule was suggested by the experimenter to other subjects, they seemed to understand its possible use.
- Three of the group A subjects and five of the group B subjects suggested the "no-op" solution. This solution involves writing an additional rule that does nothing if a grasshopper is adjacent to a duck (shown in figure 6-8). This solution does not work for two reasons. First, ChemTrains does not allow a rule to be created with no differences. Second, even if the rule could be written it would not prevent the eventual execution of a rule to move a grasshopper into a rectangle with a duck.

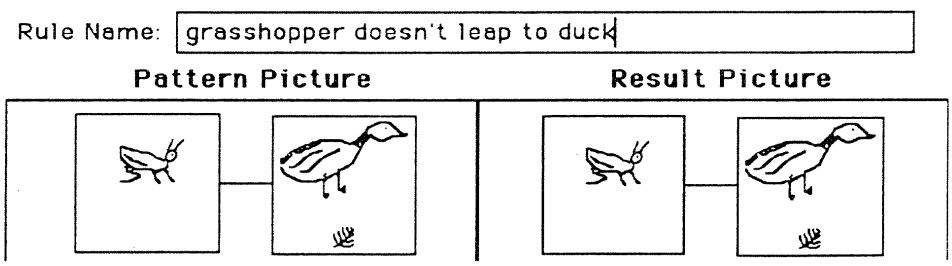


Figure 6-8: The "no-op" solution, a common but unworkable attempt to avoid ducks.

- Three of the subjects who came up with the "no-op" solution also came up with the solution of moving a grasshopper to a third adjacent rectangle when a duck is in an adjacent rectangle. This solution only works when there are not multiple ducks in the display.
- Several subjects suggested a jump out rule or a rule to have the duck eat the grasshopper anyway.

- One subject wrote a rule to delete all paths leading into rectangles that have a duck, which works, but at the expense of leaving a trail of broken places if a duck moves.
- Two subjects wrote a rule to have ducks eat grass, so that a grasshopper has no reason to jump into a rectangle with a duck.

Three subjects solved the task in a unique and interesting way that worked well enough (the last two solutions described). The remaining eleven subjects were shown the doctrine advice. Most subjects needed to read the advice more than once. Two subjects needed a lot of help applying the advice, and four needed some help. Because the example associated with the needed doctrine cannot be directly mapped onto this task, some people had trouble deciphering the relevant aspects of the example. One subject tried to apply all of the irrelevant details shown in the absence testing example. Computer scientists, in general, were more easily able to apply the advice.

6.2.8 Control of Jumping and Eating

At this point in the experiment, subjects usually had a simulation of grasshoppers jumping appropriately, but the grasshoppers were not eating because the jump rule had been created higher in the rule ordering list. Each subject realized the problem. They needed to solve it by reordering the rules, which is guided by the **reorder rules** doctrine.

Two subjects needed the doctrine. Two other subjects floundered a bit before they considered rule ordering. Three other subjects came up with interesting simulations by parallelizing the "eat" and "leap" rules. It was unclear whether this solution was considered because they were confused about rule ordering and rule priorities, or because they were trying to more accurately model nature. The rest of the subjects reordered rules without problems.

6.2.9 Representing and Changing Seasons

"**Task 4a:** Simulate the four seasons changing."

This task generally tests the ability to create a new picture that is separate from the field of six grasshopper positions, and specifically tests the ability to build a representation that serves the purpose of control. This task could be guided at a high level by the **draw control sequence structure** doctrine or could be guided more abstractly at a lower level by the **draw additional container, draw unique identifier,** and **create rule** doctrine. The representation of the control is shown in figure 6-9, and the control rule is shown in figure 6-10. The "now s" and "future s" are variables that aren't necessary for the rule to work.

Most subjects were able to solve the task without the use of doctrine advice, and solved the problem in approximately the same way. Four of the group B subjects had trouble thinking of the season picture as a separate picture which could be drawn independently from the grid of six grasshopper rectangles. They initially wanted to draw something to display the season in every

rectangle, and were gently prodded away from this type of solution by the experimenter.

Three wanted to represent the season with some sort of clock, and the other eleven subjects wanted to represent the current season as a single icon or label that appears on the display and changes. These subjects appropriately wrote a single rule to change from one specific season to the next specific season, and appropriately suggested writing four rules. These subjects were then told that we might later want to represent 52 seasons rather than 4, and that the task was to describe the behavior with one rule. Nine of these eleven subjects then appropriately designed a representation based on linking the seasons together with directed paths. Only one subject needed advice for drawing an additional container around a season label. Another subject needed the **reorder rules** advice in putting the "change season" rule last in ordering.

Six of the subjects slightly over-specified the season changing rule by putting a "season1" and "season2" variable in the pattern and result of the season containers (as shown in figure 6-10). The rule works, but the variables are unnecessary. Without the aid of doctrine help, all of these subjects volunteered putting in variables, thinking that since the rule has to execute for any two season labels, the labels should be variablized.

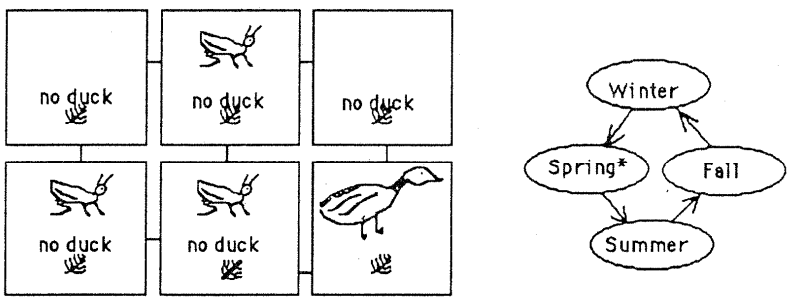


Figure 6-9: Representation of grasshoppers and seasons.

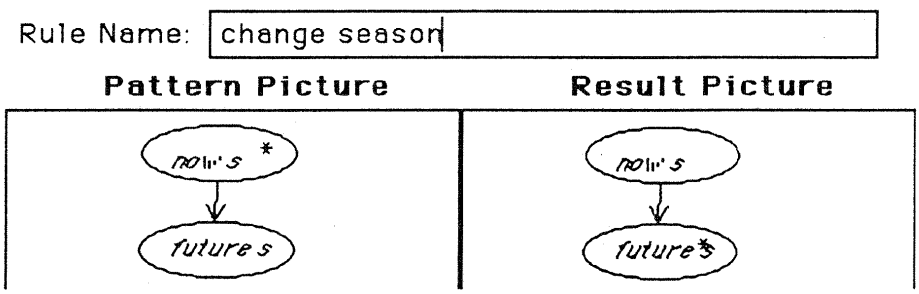


Figure 6-10: Rule to change season by moving a marker, the "*."

6.2.10 Using the Season for Control

"Task 4b: Modify the simulation so that grasshoppers only move and eat in the summer."

This task tests the ability to use the control structure to constrain the existing rules, and is guided at a high level by the **sequence structure condition** doctrine. The created rule for controlling the current season must also be placed last in rule ordering, and is guided by reorder rules doctrine.

Two subjects needed the advice, and each applied it easily. All other subjects solved the problem easily or with a little thought.

6.2.11 Poisonous and Good Grass Randomly Grow

"Task 5: Have the simulation produce poisonous grass 10% of the time and edible grass 90% of the time in the spring."

This task tests the ability to create rules that are parallel, which is guided by the **populate randomly** doctrine.

Thirteen of the fourteen subjects solved this problem immediately. The other subject wanted to specifically put one poisonous piece of grass for every nine pieces of good grass. Since the display only had six rectangles, he invented a solution of generating a circular sequence of 9 good grass icons and one poisonous that cycles through the grass in the same way that the seasons are cycled through.

Of the thirteen subjects who solved it immediately, four wanted to put a percentage on a rule without first creating the "grow poisonous grass" rule and parallelizing it with the "grow good grass" rule. These subjects subjects then remembered to parallelize the rules. The fact that subjects immediately solved the problem but without a good understanding of the underlying computational model, shows that they simply mapped the reference of percentages from the demonstration to the reference of percentages in the task. The subject who solved the problem with the circular structure admitted forgetting that part of the demonstration.

6.2.12 Grasshopper Dies

"Task 6: Complete the overall task by having grasshoppers die when eating a poisonous grass."

This task is simple, but is necessary for an interesting simulation. All subjects solved this without difficulty.

6.2.13 Summary of Results

Table 6-2 shows a simplified summary of how subjects fared on each of the tasks, breaking the subjects' performance on the tasks into the four levels. Each of the subjects is represented in this table as a letter. The letters "a" through "e" represent the five computer science subjects, and the underlined letters "f" through "n" represent the nine non-computer science subjects.

Table 6-2: Summary of subjects' performances on tasks.

	solved without doctrine		needed doctrine	
	little or no difficulty	some difficulty	applied w/o help	needed help
Task 1 deletion action	abcdefghijklmn			
Task 2 addition action	abcefhikl	dgjmn		
Task 3a movement action	abcdefhim	jkln	g	
Task 3b draw path	abcdefh	jkn	gilm	
Task 3c extra rule condition	abcdefghijklmn			
Task 3d absence testing		ehl	abcdfijk	gimn
Ordering Task reorder rules	abcdefijln	hk	gm	
Task 4a control sequence	acdefghi	klnm	bj	
Task 4b sequence condition	abcdefghijkn		lm	
Task 5 populate randomly	abcdefghijkn	m	l	
Task 6 deletion action	abcdefghijklmn			

6.2.14 Subjects' Overall Comments

After completing all of the tasks, each subject was asked what the biggest shortcomings in the language were.

- **Negation:** Every subject said that task 3d was the most difficult task. Many subjects seemed to think a negation feature would have been more straightforward.
- **Adjacency:** Many subjects recalled troubles in solving task 3b because they implicitly assumed adjacency. These subjects at the time of solving task 3b thought that adjacency should matter. After completing the tasks, they were not as convinced.
- **Asymmetry of positioning:** A few people noted that adjacency doesn't matter when pattern matching but does matter when interpreting the actions of a rule. Since they were told that "adjacency does not matter," they felt confused when they

discovered that relative positioning does affect the interpretation of a rule.

- **Rule Ordering:** Some subjects recalled feeling confused about how rules are chosen to execute. They often thought that the system would choose rules lower in rule ordering after a rule has been executed rather than choosing a rule from the beginning after each rule execution. These subject were often still confused at the end, even though the managed to solve the problem.
- **Specification of Condition:** A few subjects noted that they thought that in visual programming they would have to be very literal. They didn't consider that rules could be very loosely and very generally specified. One subject referred to writing ChemTrains patterns as "under specifying" the condition.
- **Interpreted Programming Environment:** A few subjects mentioned that they liked the interactive aspect of the environment, being able to test the simulation as they wrote and re-wrote rules.
- **Language Applicability:** Although all subjects seemed to enjoy learning about the language and working on the tasks, some of them mentioned that they couldn't see the applicability of the language in their own work. For example, the two physics subjects said that their work requires more quantitative models.

6.3 Testing of the Sliding Window Protocol

The first set of experiments unfortunately tested mainly the learnability of the language. To get a more accurate estimate of programmability, the language has to be tested on people with some training and experience using ChemTrains. Two people came in for a session to work on the sliding window network communication protocol problem. Like the first experiment, the experimenter acted as a medium between the subject and the computing environment. Each of the subjects were electrical engineers familiar with the protocol and with ChemTrains. One subject (subject #1) understood ChemTrains through previous demonstrations including the alternating bit protocol. This subject also had computer science training and knowledge of other visual languages. The other subject (subject #2) had done the previous experiment and had the typical problems learning the language (subject i in table 6-2). Figure 6-11, 12 & 13 show part of subject #2's solution. Each of the subjects were given the following problem description:

Sliding Window Protocol Problem: There are two window sizes: Ws (the sender's window) and Wr (the receiver's window). The sender's window size starts at 0 and can grow and shrink, but never grows past the maximum size Ws. The receiver's window starts out and remains at Wr. The sender and receiver each have two variables: Top and Bot (top and bottom of window, respectively) Initial values:

Sender: Bot = Top = 1

Receiver: Bot = 1, Top = Wr

We assume an infinite set of sequence numbers, although in real life, each sequence number has a fixed number of bits, so it will cycle. Rules of behavior:

- Send a message: If there is a new message to be sent, and there is room in the window (i.e., Top - Bot < Ws), then bundle and send the message and Top (i.e., (m,Top)) and increment Top.
- Resend a message: If there are messages in the window (Top > Bot), resend a message whose sequence number is between Bot and Top-1.
- Get an acknowledgement: If an acknowledgement is received, note the acknowledgement.
- Advance the window: If an acknowledgement has been noted whose value is equal to Bot, increment Bot.
- Receive a message: If a message is received whose sequence number is >= Bot and <= Top, record it. Otherwise, discard it.
- Acknowledge a message: If one of the noted messages has a sequence number equal to Bot, relay it to the user, send ack(Bot), increment Bot, increment Top.

In two hours both subjects were able to work most of the way through the problem without much difficulty and with very little need for guidance from the doctrine.

6.3.1 Results from Subject #1

Subject #1 worked from the existing solution of the alternating bit protocol. This subject had no difficulties with concepts that beginners had, such as expecting adjacency to matter and modifying the rule ordering. This subject had little difficulty enhancing the pictorial representation of the communication network so that it could more easily be used for the sliding window protocol. In the course of problem solving, the subject invented representations that:

- labeled message holders with numbers,
- defined a number type using containers,
- defined different kinds of markers to denote the top, bottom, and middle places in a window frame of a message, and
- drew containers to represent different parts of a single message.

The subject appropriately used these representations to write rules, and appropriately used variables in the rules. The subject wrote rules that mapped directly from the written specification of the protocol. The resulting simulation transmitted messages and acknowledgements correctly, except that the tail of the sliding window did not move up. The subject felt that he had to think about the problem alone rather than simply hacking on the rules in the interpreted environment.

6.3.2 Results from Subject #2

Since subject #2 had not seen the existing alternating bit protocol simulation, he started from scratch drawing on paper a representation of the entities of the network and the necessary components of those entities. After about fifteen minutes of work he came up with a representation that would work, including:

- network entities as boxes linked together with paths,

- a representation for loosing messages that used extra demon-like entities of the network that would take away messages meant for other entities, and
- a linked representation of the list of messages to be sent.

At this point the subject was given the representation from the alternating bit protocol. The major difference between the subject's drawn representation and the given representation was that the subject separated the network entities from a table description of the internal workings of each entity, and mapped the two representations with entity names.

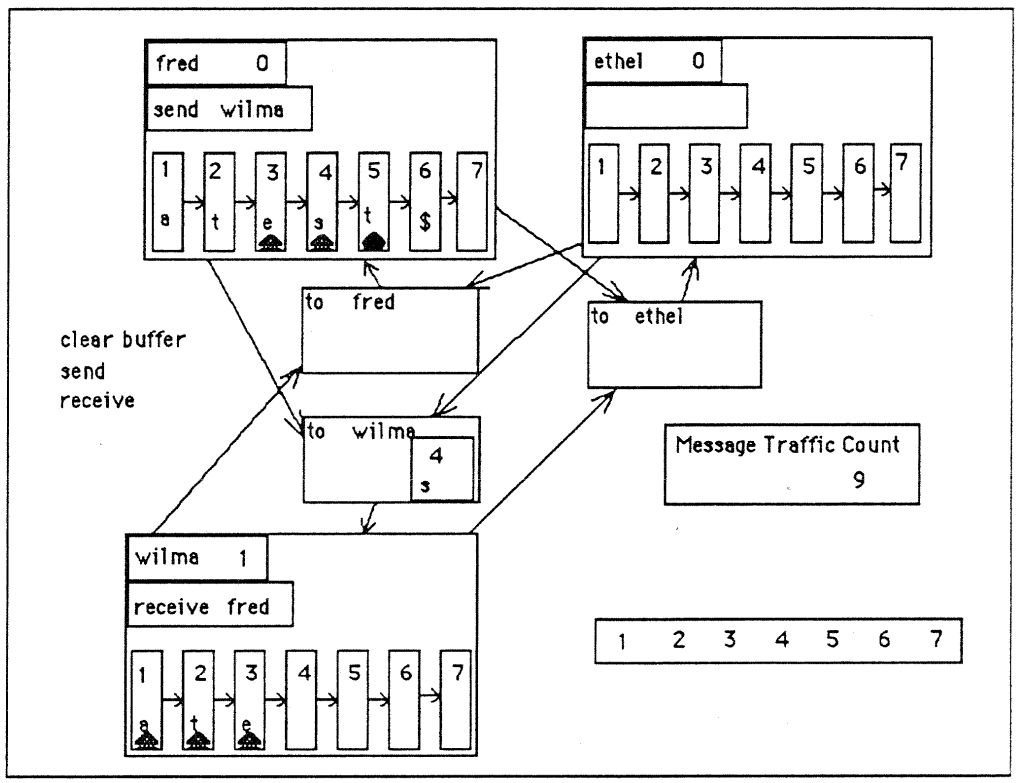


Figure 6-11: Simulation display from subject #2's solution.

From this point he extended the alternating bit representation to handle the sliding window protocol in the following ways (also shown in figure 6-11):

- generating numbers to label the message holders, and
- generating a marker to show the boundaries of the window.

He needed to read the **draw set definition** and **set condition** doctrine, and applied this doctrine correctly in creating the number type checker. In contrast to subject #1, this subject worked from an abstract notion of the sliding window protocol and only used the written specification for reference. This subject was also less concerned with representing a variable size window and more concerned with an issue that subject #1 ignored, simulating the entities communicating at different speeds. Therefore, the subject simplified the protocol by only representing the window as a fixed size, and worked on representing multiple messages being sent at once rather than just one.

Figure 6-12 shows his rule to send two messages rather than one. Since the receiver only sends messages one at a time, the transmission rates are different. Figure 6-13 shows his rule to move the fixed sized sliding window, which stays at length 3. The subject needed one kind of marker (the gray marker) to represent messages within the sliding window and another kind of marker (the black marker) to represent the position of the last sent message within the sliding window. During the testing of the simulation, the rules had to be reordered a few times. The subject had trouble understanding the profound effect of rule ordering on execution, and did not understand how the system used rule ordering in conflict resolution. After an explanation, the subject more easily modified the rule ordering. The subject completed the task except for having the sender resending old messages and the receiver resending old acknowledgments. The subject was easily able to use variables in rules without help. In contrast to subject #1, the subject liked hacking on the rules and changing them in an interpreted environment.

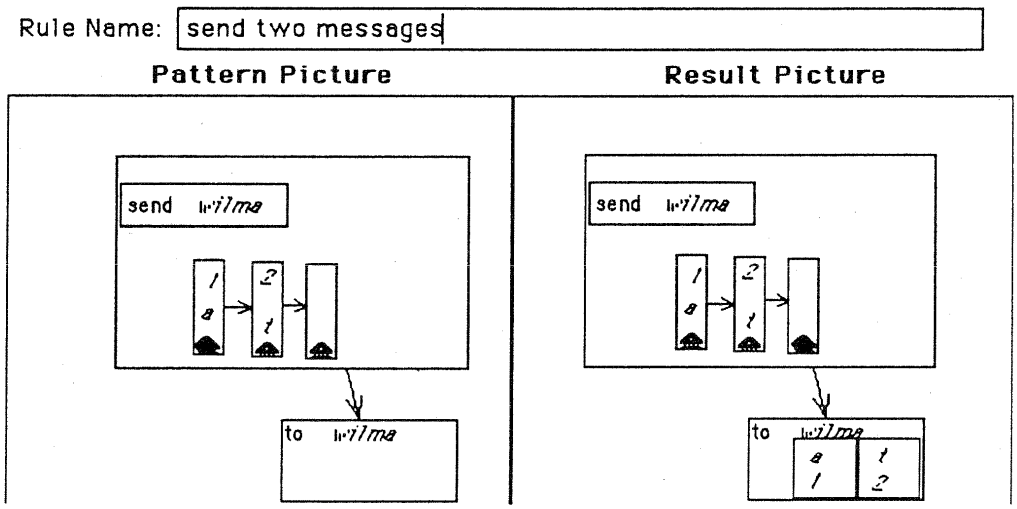


Figure 6-12: Subject #2's rule to send two messages, simulating messages being sent twice as fast as the acknowledgements from the receiver.

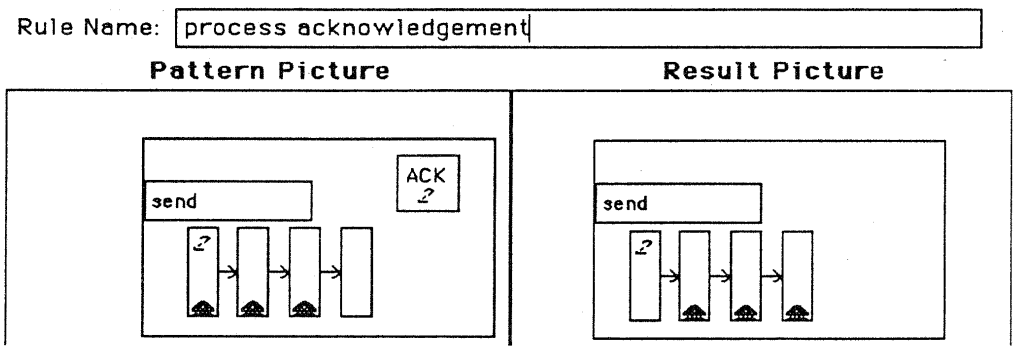


Figure 6-13: Subject #2's rule to slide window forward after an acknowledgement.

6.4 Conclusions

The following conclusions are based on:

- comparing the results of the grasshopper experiment with the results of the sliding window experiment and with the results of the bunsen burner experiment from a year ago (Bell, Rieman, & Lewis, 1991), and
- comparing the results of the experiments with the walkthrough analysis.

The language is easy to learn. Most programming languages take days or weeks to learn at a level sufficient to write interesting programs. The results of the sliding window protocol showed that subjects with minimal training can program in ChemTrains close to the level of the designer. Although the subject who took part in both experiments had previous traditional programming experience, he had the typical problems in working on the grasshoppers problem but came back and worked well on the more complicated and less structured problem, showing that he had learned much from the first experiment. The results within the grasshopper experiment also show the learnability of the language. In solving task 4a&b the subjects were able to invent and use a representation that required a fairly abstract use of containers and directed paths, even though they had not seen directed paths in use. Previous experience in using rectangular containers for grasshopper places, creating non-directed paths, and seeing directed paths as an option was sufficient for all but two of the subjects to invent the necessary structure, showing that the subjects had learned a lot about creating ChemTrains topological structures.

Geometric relationships such as adjacency are natural pictorial constraints. Many subjects expected adjacency to matter in the grasshopper and the bunsen burner experiment even when they had been warned that it didn't matter in pattern matching. After finally learning the lesson, subjects did not have the problem again. This shows that a language that uses these sorts of constraints, such as BITPICT and AgentSheets, may be more easily learnable.

Support for the language should be enhanced to encourage the use of paths when adjacency is important. A conclusion that could be drawn from the testing is that ChemTrains should be enhanced to match for adjacency when objects are adjacent in a pattern picture. Since almost all of the patterns in the solutions contain the same adjacency constraint that subjects expected to matter, adding such a feature would make these other solutions not work. For example, the "change season" rule shown in figure 6-10 might be interpreted to additionally mean that one oval is above the other oval. The freedom to place graphical objects in arbitrary relative positions gives ChemTrains much of its existing power. Since it does not make sense to

enhance the language, support for using paths when adjacency matters should be integrated into the manual and/or into the environment.

Negation should be added. Since all subjects had trouble implementing absence testing, and many specifically mentioned needing a feature to test for the absence, the negation feature should probably be added to ChemTrains. The only concern is the point made in the walkthrough analysis that the use empty marker solutions lead to simpler solutions in many problem solving situations. What the analysis failed to recognize was that the empty marker doctrine was very complex and hard to apply.

6.4.1 Methodological Lessons

The user testing confirmed much of the walkthrough analysis. The main thing that the walkthrough analysis showed was that building useful pictorial representations is the hardest aspect of writing ChemTrains programs. The users had more difficulty with tasks requiring extensions to the pictorial representation, tasks 3b, 3d, and 4a. Although the user testing generally fit the walkthrough analysis, there were some differences.

The complexity of walkthrough steps varies widely. In doing the programming walkthroughs, it was recognized that some steps of a walkthrough may be more complex than other steps. The process of writing walkthrough summaries was an attempt to evaluate the complexity of steps thoroughly. The user testing showed that the complexity of the steps varied more widely than anticipated. For example, although both the **draw empty container marker** doctrine and the **draw control sequence structure** doctrine were rated as being abstract and therefore complex, no subjects were able to invent the empty container marker solution in task 3d while all but two subjects invented the control sequence solution in task 4a. The testing also showed that there is a wide gap in the understandability of the doctrine. While almost every subject who was given the **draw empty container marker** advice needed to read it at least twice, the subjects who read the **draw control sequence structure** advice applied it immediately.

The length of doctrine is not an accurate measure of a language's writability. The discussion of the walkthrough method warned that design decisions should be based not only on the comparison of the length of doctrine but also on the complexity of doctrine. Although the design process adhered to the conventions of walkthrough analysis, the complexity of doctrine varied more than expected. Therefore, rather than using the length of the doctrine as an estimator, the designer should place more emphasis on subtleties in complexity that lie at the level of the walkthrough summary analysis, and more care should be taken in writing these summaries. Although the length of the doctrine should play a less important role, the construction of doctrine is still necessary in order to do walkthrough summary analysis.

Doctrine is not a useful guide in learning a language. The bunsen burner and grasshopper experiments differed in three important ways: the domains are different, the bunsen burner task was not pre-decomposed into subtasks for the subjects, and the initial tutorial for bunsen burner experiment was a presentation of doctrine rather than a demonstration of features. Even though the demonstration only covered a very small amount of the problem-solving doctrine, users were able to learn the system and solve problems with less frustration in the grasshopper experiment. Here are some examples where most of the subjects invented working representations without previous problem-solving advice or need for it:

- Even though the rule mechanism was stated as a way to replace one picture with another picture, and the description only demonstrated a replacement action (unlit house changes to lit house), the subjects easily generalized from the description to write rules that had deletion, addition, and movement actions.
- Even though paths were simply defined as a line segment connecting two objects and were not described as something for moving objects between containers, the subjects managed to invent the use of paths to represent grasshoppers moving among adjacent containers.
- Even though high-level composition advice was not described, people invented the complex structure of a control sequence from the containment and connection primitives, which they had just learned to use.
- The two subjects who attempted the sliding window problem managed to invent complicated high-level structures with little need for the programming advice.

These results show that a small well-chosen introduction to a language can go a long way in teaching how to write programs and may be more efficient than teaching specific points of doctrine. In teaching a language, the role of doctrine seems more appropriate in helping programmers when they are stuck.

7 Language Comparisons

This chapter compares the ChemTrains model of computation with two other production system languages, BITPICT and OPS5.

7.1 BITPICT

BITPICT is a system that at a high level has an identical description as ChemTrains: a system for constructing graphical simulations in which the behavior is specified by drawing graphical pattern/result rewrite rules. BITPICT rules and simulations are described at the pixel level, and specify exactly how a rectangular region of pixels should be replaced with a different pattern of pixels. The main difference with ChemTrains is that BITPICT doesn't have any higher level concepts, such as places, paths, or variables.

7.1.1 Translation of BITPICT programs to ChemTrains

Many programs in BITPICT can be written in ChemTrains with the same number of rules. A representation of the BITPICT pixel field can be constructed in ChemTrains so that BITPICT programs can be written. The pixels should be representing as a two dimensional grid of identical boxes, where adjacent boxes are connected by paths. Each box should contain either an icon representing "pixel off" or an icon representing "pixel on."

One possible solution to the bouncing arrows problem demonstrates the use of this representation. The solution shown in figure 7-2 is analogous to an identical solution in BITPICT, shown in figure 7-1. This simulation includes two rules to describe the behavior, a rule to move an arrow forward and a rule to bounce an arrow. Both in BITPICT and ChemTrains the rules will fire when the arrow is pointed right, left, up, or down, but the two systems enable this ability for different reasons. In BITPICT it is because the system explicitly matches rotations and reflections of a pattern. In ChemTrains, since position doesn't matter and connection does, the rules can ignore the position irrelevancies to match patterns of pixels based on their connections instead of their positional relation to each other. This representation will enable any BITPICT program to be written, because there is a straightforward mapping of BITPICT rules to ChemTrains rules.

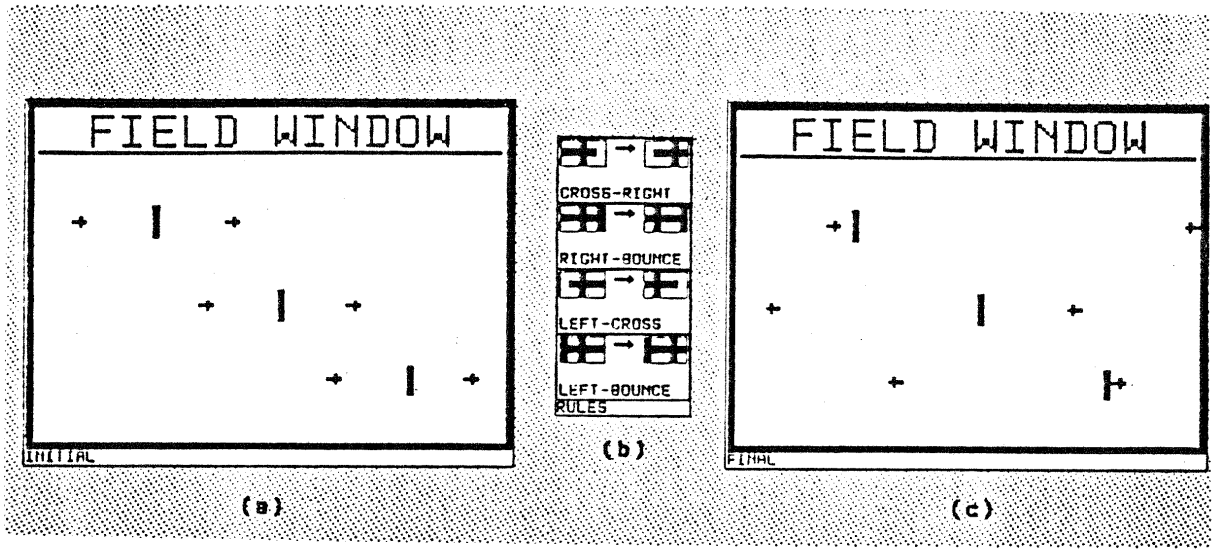


Figure 7-1: BITPICT bouncing arrows solution.

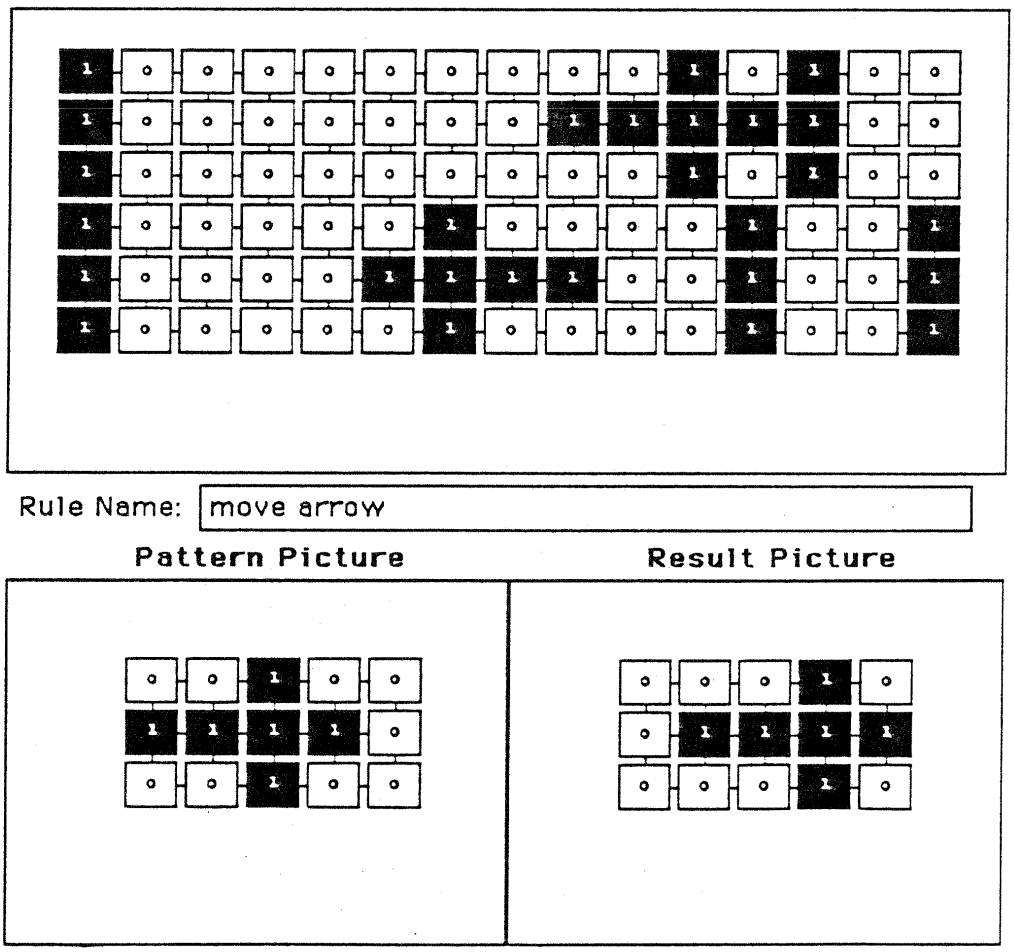


Figure 7-2: BITPICT Bouncing arrows in ChemTrains.

All rules in BITPICT cannot be mapped to the same rules in ChemTrains using this representation because BITPICT enables different freedoms of reflection and rotation to be turned off and on for each rule.

BITPICT has a built-in control structure, which allows rules to be grouped so that the execution of rules in one set of rules do not interfere with the execution of other rules. Although ChemTrains does not explicit support the grouping of rules, the solution to the grasshopper problem illustrates that groups of rules can be separated if an appropriate control structure diagram is drawn.

7.1.2 Translation of ChemTrains programs to BITPICT

ChemTrains programs cannot be easily written in BITPICT because of ChemTrains ability to match:

1. objects in different parts of the screen,
2. the "inside" constraint,
3. the "connected to" constraint,
4. multiple objects whose pictures are variable.

Since BITPICT must match a pattern exactly in one specific region of the simulation picture, ChemTrains rules that match on objects in different parts of the screen cannot be easily mapped to single BITPICT rules. Also, ChemTrains patterns that take advantage of constraints like "inside" and "connected to" need to be mapped to multiple BITPICT rules because these kinds of constraints describe many possible specific pictures in the simulation picture. Matching objects whose pictorial description is variable also needs to be mapped to multiple rules. If the choices for a variable can be enumerated by specific pictures at the time the rule is created, the single rule can be replaced by one rule for each possible instance. If the choices are not known at the time the rule is created, mapping a ChemTrains rule with variables to BITPICT rules is impossible.

7.1.3 Programmability Comparison

Aside from the issues of power, BITPICT has some distinct advantages over ChemTrains:

1. Because BITPICT has a limited computational model, the system and programs in it are simpler to learn and understand.
2. Because the domain (bitmaps) is limited, manipulation of the simulation picture and the rules is straightforward.

Just because any program in BITPICT can be written in ChemTrains with the same number of rules does not mean that you would want to write all graphical simulations in ChemTrains. The solutions of the bouncing arrows problem (shown in figure 7-1, 7-2, and 4-4) illustrate that simulations that are better suited to the pixel level, the domain that BITPICT understands. However, in programming at such a low level some hacking must be done to get interesting high-level behaviors of more complicated simulations. A comparison of solutions to the logic circuit problem illustrates these

difficulties. Figure 7-3 shows the BITPICT solution to the logic circuit problem, which can be compared with the ChemTrains solution shown in figure 4-7. Because BITPICT operates at the pixel level, the exact placement and setting of each pixel has to be arranged. In ChemTrains the gates, the wires, and the boolean values can each be manipulated as single entities, operating more directly in the domain that the end user of the simulation is working on.

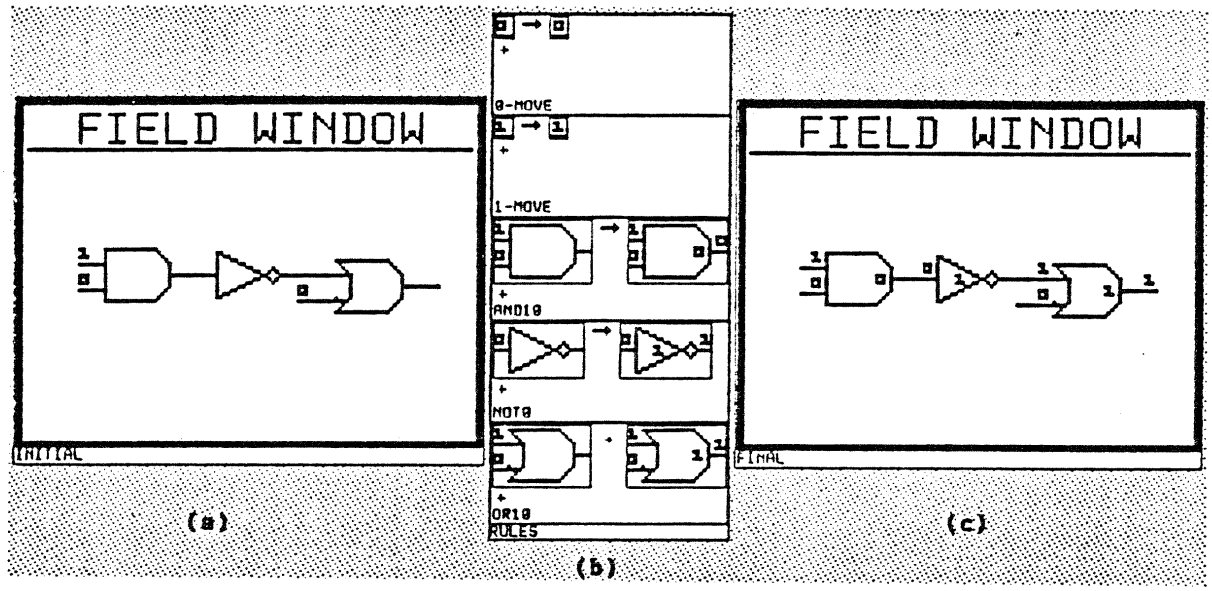


Figure 7-3: BITPICT solution to the logic circuit problem.

7.2 OPS5

Comparing OPS5 to ChemTrains is somewhat similar to comparing ChemTrains to BITPICT. OPS5 is clearly a more powerful programming language, but ChemTrains is easier to program for a limited class of problems. This section describes the similarities between the languages, by showing the ability to translate programs between the two languages.

7.2.1 Translation of ChemTrains programs to OPS5

ChemTrains programs can be directly translated into OPS5 programs. Here is an OPS5 representation, which allows any ChemTrains program to be translated into OPS5 in the same number of rules. The representation maps constraints of a ChemTrains picture onto individual working memory elements.

```
(literalize object
  unique-id
  type)

(literalize contains
  place
  containing)
```

```
(literalize connection
  unique-id
  object1
  object2)

(literalize directed-connection
  unique-id
  from-object
  to-object)
```

Because the OPS5 conflict resolution strategy executes rules based on recency of data rather than the ordering of rules, a data structure is needed to make sure that the rules are fired in order of their priority. The context working memory elements are used as a stack of goals to accomplish.

```
(literalize context
  name)
```

Here is an example of a straightforward translation of the rules of the ChemTrains maze program onto OPS5 rules. An additional rule is added, named "start-looking-for-cheese," to make sure that the rules execute with the right priority. This rule adds the three contexts in reverse order of priority because the last one created will be the most recently added working memory element, giving that context the highest priority.

```
(p start-looking-for-cheese
    (object ^type mouse)
  -->
    (make context ^name backtrack)
    (make context ^name look-for-cheese)
    (make context ^name eat-cheese))
```

The following rules each test the context as the first condition because the conflict resolution strategy of selecting most recent data looks at the recency of working memory elements in the order that they match the conditions.

```
(p eat-cheese
    (context ^name eat-cheese)
    (object ^type mouse ^unique-id <m>)
    (contains ^place <place-id> ^containing <m>)
    (object ^type cheese ^unique-id <c>)
    (contains ^place <place-id> ^containing <c>)
  -->
    (remove 4)
    (remove 5))

(p look-for-cheese
    (context ^name look-for-cheese)
    (object ^type mouse ^unique-id <m>)
    (contains ^place <from-id> ^containing <m>)
    (connection ^object1 <from-id> ^object2 <to-id>)
    (object ^type unseen-marker ^unique-id <u>)
    (contains ^place <to-id> ^containing <u>)
  -->
    (make object ^type string ^unique-id (new-symbol <s>))
```

```
    (modify 3 ^containing <s>)
    (modify 6 ^containing <m>)
    (remove 5))

(p backtrack
    (context ^name backtrack)
    (object ^type mouse ^unique-id <m>)
    (contains ^place <from-id> ^containing <m>)
    (connection ^object1 <from-id> ^object2 <to-id>)
    (object ^type string ^unique-id <s>)
    (contains ^place <to-id> ^containing <s>)
  -->
    (make object ^type seen-marker ^unique-id (new-symbol <sm>))
    (modify 3 ^containing <sm>)
    (modify 6 ^containing <m>)
    (remove 5))
```

ChemTrains rules can be directly mapped onto OPS5 rules as shown. The difference of executing on data based on recency rather than randomness will result in some difficulties in always mapping solutions directly from ChemTrains to OPS5.

7.2.2 Translation of OPS5 programs to ChemTrains

OPS5 uses a representation of records to define working memory elements. This representation can be mapped directly onto the ability in ChemTrains to draw attribute-value tables, described as advice in section 3.3.8. In addition, ChemTrains rules have almost the same amount of power as OPS5 rules. ChemTrains rules cannot match OPS5 patterns that contain the following features:

- the ability to compare the numeric values, and
- the ability to test for the absence of a pattern. (negation)

One claim of OPS5 is that the language can support decomposition necessary in writing large programs, such as XCON. The only things limiting the ability to write large programs in ChemTrains is the efficiency of the interpreter and the screen space available for drawing large simulations, both of which can be addressed in a future implementation of the language. The basic language features support both data and control decomposition in the following ways.

- The ability to label objects and paths allows components of a simulation to be further elaborated in other places on the screen. The ability to variablize components of rules enable programmers to take advantage of these decompositions.
- The ability to write finite state machines and to group rules based on current states of processing enables programmers to decompose large rule sets into more manageable chunks. The grasshopper simulation demonstrates a decomposition of rules into four sections. In the same way, a very large set of rules could be decomposed with the use of a goal or context tree.

8 ChemTrains Implementation

This chapter first gives an overview of the modules and the data structures in the implementation of the ChemTrains system. Next the algorithm for determining efficient patterns for search is described. Finally efficiency improvements to the search algorithm are proposed.

8.1 Overall Structure of ChemTrains

The system was written in Common Lisp (Steele, 1984) and built in the Macintosh Allegro Common Lisp programming environment, version 1.3 (Apple Computer, 1989.) The code is not portable to computers other than computers that run the Macintosh system 6 operating system, because the code uses and depends on facilities specific to the MACL 1.3 dialect of lisp, which is only compatible with system 6. Here are the facilities that were used in the production of ChemTrains that are specific to MACL 1.3:

- an **object-oriented facility** called Object Lisp,
- a **user interface toolbox**, which supplied high-level constructs for creating and operating on windows, menus, and dialog items, such as buttons, check boxes, and scrollable lists, and
- a **library of low-level drawing primitives** for drawing ovals, lines, and text strings, which is needed to draw ChemTrains displays and rules.

The system could be ported to other computing environments if a subset of each of these facilities is supplied.

The ChemTrains implementation is separated into the following main components:

1. a **simulation programming environment**, which enables a programmer to edit the main display, create and edit rules, and execute and trace the simulation,
2. a **picture editor**, for drawing the simulation and rules,
3. a **rule compiler**, which parses the pattern and result of a picture, creating a data structure for efficiently matching and executing rules, and
4. a **rule inference engine**, which executes a recognize-act cycle, using the rule compilations to do pattern matching and to make modifications on the main display.

8.1.1 The Simulation Environment

This is the high-level module of the system that maintains the appropriate positions of the interface dialog items, and makes procedure calls to the following lower level modules:

- to the picture editor when picture editing commands are executed,
- to the rule compiler when a rule is saved,
- to the inference engine when the environment is put in execute mode, and
- to smaller modules for manipulating the rule ordering and saving and restoring a simulation on a file.

The simulation environment also supports the ability to show or hide the rule editor, so that the environment can be used for programming or simply for executing a simulation.

8.1.2 The Picture Editor

The picture editor provides a way to draw, move, cut, copy, or paste objects of a display, which may be either the main simulation display, a rule's pattern, or rule's result. The picture editor allows objects to be dragged between the three different display windows, so that a rule can easily be constructed from objects in the main display.

Graphical objects and paths are created as instances of the following hierarchy of object classes. All classes of graphical objects or paths inherit the attributes of **ct-visible-thing**, which specifies the object's position and size on a window, whether it is hidden, and whether it is a component of a grid. The point data structure has two components for representing a length in the horizontal and vertical dimensions.

```
ct-visible-thing =
  window: window
  hidden?: boolean
  position: point
  size: point
  grid: grid or nil
```

Any graphical object also inherits the attributes of the **ct-object** class, which specifies whether the object is a variable (only applicable if the object is inside a rule) and specifies its connections and its placement within the topology of the other objects on the picture.

```
ct-object =
    (inherits from ct-visible-thing)
  variable?: boolean
  containers: list of ct-object
  contents: list of ct-object
  connections: list of ct-path
```

Although the current interface does not provide a way to specify the thickness or shading of an object, the **ct-outline-object** class has attributes that could be used to specify this.

```
ct-outline-object =
    (inherits from ct-object)
  thickness: integer
  shade: bit-array
```

The **ct-box-object**, **ct-ellipse-object**, and **ct-polyline-object** classes each inherit from ct-outline-object. Boxes, ovals, and lines shown in a ChemTrains display are instances of these classes. The ct-polyline-object class additionally has an attribute for representing the connected points.

```
ct-box-object =
    (inherits from ct-outline-object)

ct-ellipse-object =
    (inherits from ct-outline-object)

ct-polyline-object =
    (inherits from ct-outline-object)
  points: list of point
```

A **ct-text-object** is a ct-object with a text string. The font specification attribute cannot currently be changed through the interface. If the text object labels a path, the labeled-path attribute refers to this path.

```
ct-text-object =
    (inherits from ct-object)
  text: string
  font: font-specification
  labeled-path: ct-path or nil
```

A **ct-icon-object** is a ct-object that refers to the name of an icon in an icon library. The interface provides for access and manipulation of icons in the icon library.

```
ct-icon-object =
    (inherits from ct-object)
  icon-name: string
  icon: bit-array
```

A **ct-path** is a ct-visible-thing that connects two objects. This class has attributes which specify the placement of the path, the connected objects, the destination points of the path within these objects, the line segments of a possible arrowhead, and a text object that labels the path.

```
ct-path =
    (inherits from ct-visible-thing)
  points: list of point
  ob1: ct-object
  ob2: ct-object
  ob1-point: point
  ob2-point: point
  arrow-points: list of point or nil
  label: ct-text-object or nil
```

The **grid** class is used to specify grids of connected boxes. The attributes specify the exact placement and size of the grid cells and the way in which the cells connect horizontally and vertically.

```
grid =
```

```
name: string
hidden?: boolean
contents: 2d array of ct-box-object
size: point
start-position: point
cell-size: point
cell-spacing: point
horizontal-paths: (:both :right :left :undirected nil)
horizontal-path-labels: list of string
vertical-paths: (:both :up :down :undirected nil)
vertical-path-labels: list of string
```

The picture editor represents a picture as a list of grids, a list of objects, and a list of paths.

8.1.3 The Rule Compiler

The rule compiler uses the pattern and result displays of a rule to generate a representation of the rule that includes a specification of:

- the pattern to match, and
- the actions to take when the rule is executed.

The pattern is represented as a list of object specifications, one for each object in the pattern picture. Each individual object specification is a list describing the kind of object to match and the constraints on this object with respect to previously matched objects. The actions of a rule are represented as a list of objects and paths to remove and a list of objects and paths to add.

Figure 8-1 illustrates many of the constraints and actions that can occur in a ChemTrains rule.

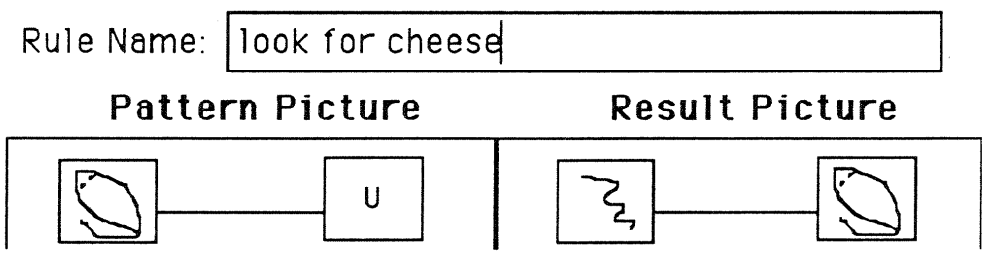


Figure 8-1: Rule to move a mouse to an unseen place in the maze.

For this rule the compiler generates a pattern specification that consists of four elements, one for each object shown in the pattern. The first element of each pattern is the object type specification and the rest of the elements of the list consist of constraints. For example, the pattern labeled "0" matches a mouse, and the pattern labeled "1" matches a box of a specific size that contains the previously matched mouse.

```
0. ((ct-icon-object "mouse"))
1. ((ct-box-object 2031652) (contains 0))
2. ((ct-box-object 2031652) (connected 1))
3. ((ct-text-object "U") (inside 2))
```

The compiler generates a rule's actions by computing the differences between the pattern specification shown above with a specification of the rule's result represented in the same syntax. In making the comparison, the compiler generates a listing of objects to add and remove. For the rule in figure 8-1, the compiler recognizes that the mouse and the "U" text object should be removed, and represents this as a list of positions in the pattern:

```
(0 3)
```

The compiler also recognizes that a string and a mouse should be added, and represents this with the following list structure:

```
(((ct-icon-object "mouse") in 2 at 262150)
 ((ct-icon-object "string") in 1 at 393225))
```

The first element states that a mouse should be created at a certain position inside the object matched by pattern in position #2.

In determining the meaning of this rule, the compiler must choose between two interpretations that arise from the fact there are multiple identical boxes in the pattern and result. In addition to the meaning already shown, the actions of the rule could mean that the matched mouse is to stay in its current position, and the "U" text string is to be replaced by a "string" icon. In choosing among the possibilities, the rule compiler first creates a mapping from objects in the pattern to objects in the result and then chooses a single mapping by employing a branch-and-bound algorithm to optimize for the smallest difference in positioning between the two pictures.

8.1.4 The Rule Inference Engine

The inference engine uses a recognize-act cycle to execute the simulation. In the recognize part of the cycle, the engine loops through the ordered list of rules until a rule's pattern specification can match the simulation display. If no rules match, the inference engine halts. While iterating through the list of rules, when the engine comes across a rule that is in a parallelized group of rules, it randomly picks between them based on the priority weights specified in the program. If a randomly picked rule cannot match the simulation, the engine continues to select among the rest of the parallelized group of rules in the same manner in which the first was chosen.

The engine matches a single rule's pattern by using a depth-first search algorithm. In order for the pattern to be matched, each object in a pattern must match a different object in the simulation and each of the constraints must be met. During search when a single pattern object and its constraints cannot appropriately match an object in the simulation, the search is pruned. The following section describes more specifically how the pattern matcher reduces the search space.

When a rule is finally selected and matched, its actions are executed. Before executing the additions and removals, the engine checks to see if any object to

be removed is identical to an object to be added. If any of these movement actions are movements between connected containers, the engine drags the object along the path before placing it at the destination. For example, when the rule shown in figure 8-1 is executed, the mouse will travel along the paths of the maze. The inference engine rather than the rule compiler is responsible for detecting movement actions because a movement action can result from an object matching both a variable object in a pattern and a constant in a result, which cannot be determined at the time of rule compilation.

8.2 Pattern Matching and the Pattern Compiler

The pattern matcher is exponential. If there are *o* objects in the drawing, *r* rules, and the most complex rule pattern has *p* object descriptions, the size of the pattern matching search space is approximately O(*r*o^p*). For each of the *r* rules, *p* objects must be picked from a selection of *o* objects on the drawing. So if 4 objects exist in a pattern and there are 10 objects on the screen, the size of the search space is 10 * 9 * 8 * 7 (5,040) because each of the 4 objects must pick between 10 objects and may not select a previously chosen object. The pattern matcher takes advantage of other factors in the rule pattern and the simulation picture to drastically limit the search. It uses four techniques described below:

1. using an object's type to limit search,
2. using the constraints in a pattern to limit the search,
3. matching the most constraining objects earliest in the search, and
4. matching mutually constraining objects together in the search.

The first two ideas are also used in the well known RETE algorithm for efficiently executing rules of a production system (Forgy, 1982.) ChemTrains does not use the RETE algorithm, which reduces work by remembering which data objects need to be tested and which objects don't need to be tested during rule execution. Instead of implementing the RETE algorithm, I have explored an orthogonal idea which has also been incorporated in more recent production system interpreters: the idea of guessing an optimal order for matching pattern objects (Scales, 1986.) This is especially appropriate in graphical rule-based languages, because order of patterns in a single rule in no way influences conflict resolution strategy, as it does in OPS5.

8.2.1 Reduce Search based on Type

The first thing it uses to reduce search is to only search possibilities in which each of the pattern objects is bound to an identical object on the screen. So if the simulation picture contains 5 squares, 3 triangles, and 2 circles, and a rule's pattern contains 2 squares, 1 triangle, and 1 circle, the pattern matcher will only consider matching objects of the appropriate type, and the size of the search space will be reduced to 5 * 5 * 3 * 2 (150).

8.2.2 Reduce Search based on Constraints

Another thing that the pattern matcher considers during search is the constraints described by the pattern. The constraints considered in ChemTrains are:

1. whether an object is inside another object;
2. whether an object is connected to another object;
3. whether an object is the label of a connection; and
4. whether two variables in a pattern are identical, forcing their matched objects to be identical.

Suppose that in the previous example the two squares are connected and a circle is inside one of the squares, as shown in figure 8-2. Also, suppose that in the simulation picture each of the 5 squares is connected to 2 other squares, and two circles are never together inside the same square, as shown in figure 8-3.

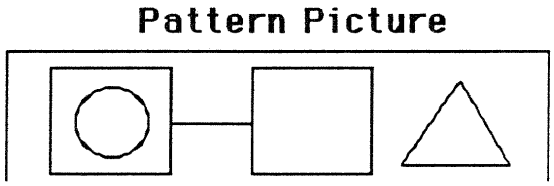


Figure 8-2: example pattern with four objects and one path.

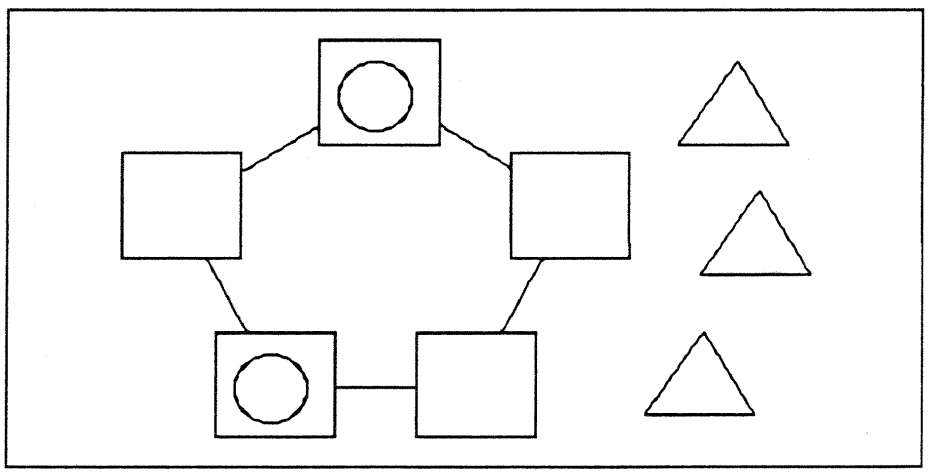


Figure 8-3: example simulation display.

If the pattern matcher takes advantage of the constraints during search, the algorithm will select among 5 choices for picking the first square; then it will pick from only 2 choices for the second square because every square is only connected to two other squares; then it will pick from the usual 3 choices for the triangle; and finally it will pick from only 1 choice for circle because it is forced to choose a circle inside a square. The size of the search for this pattern will be reduced to 5 * 2 * 3 * 1 (30).

8.2.3 Reduce Search by Ordering the Pattern

ChemTrains employs one other technique for reducing the search space. When a pictorial rule is compiled into a set of pattern specifications and a set of constraints, the compiler arranges the order of the pattern objects so that the most constraining objects are chosen first. The rule compiler estimates how constraining an object is based on:

1. the population of the object in the drawing at the time the rule is compiled, and
2. the number of constraints that tie to objects already placed in the pattern ordering.

The rule compiler generates a pattern ordering by successively selecting the most constrained pattern object of the pattern objects that have not been previously selected. In evaluating how constraining an object is, the algorithm roughly divides the object population by the number of constraints plus 1. In choosing a pattern object, the algorithm picks the object with the lowest number. The rule compiler generates the following ordering for the pattern shown figure 8-2:

```
0. ((ct-ellipse-object 1703963))
1. ((ct-box-object 2555948) (contains 0))
2. ((ct-box-object 2555948) (connected 1))
3. ((ct-polyline-object pline0))
```

With the new reordering the size of the search for this pattern will be reduced to 2 * 1 * 2 * 3 (12). The algorithm picks the circle first because it is believed to be most constraining based on its current population; it then picks the square containing the circle because it is constrained by the circle; it then picks the other square because it is constrained by the previously picked square; and finally it picks the triangle.

The rule compiler employs a branch and bound strategy for determining which of multiple equally constraining objects should be first in the ordering. In the simple example described above this is not a concern, and does not affect the outcome of producing an appropriate pattern ordering. Figure 8-4 shows a pattern that requires such a search to find a good ordering.

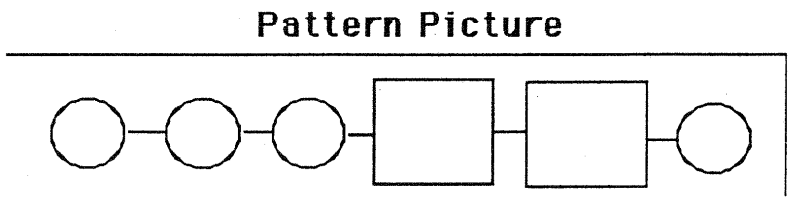


Figure 8-4: another example pattern.

In this pattern, suppose that circles were believed to be the more constraining than squares because the simulation picture contains many more squares than circles. When the algorithm makes its first choice, since it believes that the four circles are equally constraining and are each more constraining than

the squares, it tries building a pattern ordering trying each of the circles first. Through search the algorithm will correctly first pick either of the three circles on the left of the pattern rather than the circle on the right because these objects are connected to more constraining objects earlier in the search.

8.2.4 Using Object Population to Determine Ordering

One problem with the current algorithm is that the proportions of the various types of objects on the screen at the time that a rule is compiled may not be a good estimate of their proportions at the time a rule is executed. This is easily demonstrated in the tic tac toe interface built in ChemTrains. If the rules are compiled before a game is played when the game board is empty, the system plays quickly in the beginning of the game, plays slower with each move, and finishes playing much slower. If the rules are compiled when the game board is full, the system plays slowly in the beginning and plays faster with each move.

The current system does not address this problem. Here are a few ways to attack the problem:

1. reorder the rule patterns during rule execution;
2. keep running statistics on the object populations, and compile the rules after these statistics have been collected; or
3. keep multiple pattern orderings for each rule, that are based on different simulation situations.

The disadvantage with 1 is that the effort of rule recompilation will offset the benefits. Rule compilation in many cases can be slower than pattern matching. The disadvantage with 2 and 3 is that the system will need some guidance by the programmer to decide when to recompile a rule. A hard problem in implementing 3 is that the system has to make some judgement as to what differences in rule ordering are important.

8.3 Search Complexity

In terms of *r* (rules), *o* (objects), and *p* (pattern objects), the complexity of the search is O(*r***o*^*p*), and it is possible to write a set of rules that is this complex. For example, consider a simulation picture that had 10 identical circles and a single box. The simulation contains one rule, that matches 4 circles and puts an X in the box. The meaning of this rule to ChemTrains is: "For each permutation of 4 circles in the picture, place an X in the box." Therefore, in the picture 10 * 9 * 8 * 7 (5040) X's will be placed in the box. The order of complexity of the pattern matching O(*o*^*p*) is appropriate for the task at hand.

The patterns of rules in most problems posed to ChemTrains contain enough constraints that the generalized O(*r***o*^*p*) is not remotely useful as an estimate of how complex the search space is. For example, the ChemTrains simulation of a Turing Machine has ~150 objects in the simulation picture, 5 rules, and each rule has ~20 pattern objects. The rough estimate of the search

space disregarding object types and constraints is 5*150^20 (~10^44). The ChemTrains simulation of a Tic Tac Toe player has ~50 objects in the simulation picture, 20 rules, and each rule has ~12 pattern objects. The rough estimate of this search space is 20*50^12 (~10^15). Not only are these estimates not close to the space actually searched, but the Turing Machine simulation runs faster than the Tic Tac Toe simulation because the search space of the Turing Machine although possibly much larger is much more constrained.

The general form of the size of the search space for a simulation is:

```
SimulationSearchSize =
  Sum for rule in SimulationRules()
    RuleSearchSize(rule,SimulationPictureObjects());

RuleSearchSize(rule,pictureobs) =
  PatternSearchSize(pictureobs, RulePatternObjects(rule),
                    RuleConstraints(rule), null);

PatternSearchSize(pictureobs,patternobs,constraints,bindings) =
  If (null patternobs) then 1
    else Sum for ob in GetConstrainedObjects(pictureobs,
                                             patternob,
                                             constraints,
                                             bindings)
                 PatternSearchSize(pictureobs,
                                   rest(patternobs),
                                   constraints,
                                   cons(ob,bindings));
```

GetConstrainedObjects(pictureobs, patternob, constraints, bindings) returns the set of all objects that are in the pictureobs list, are of the same type as patternob, meet the constraints, and don't already exist in the binding list.

8.4 Efficiency Improvements

The efficiency of the current inference engine can be greatly improved by employing previously used algorithms. The Rete algorithm (Forgy, 1982) uses a complex data structure to implement the following techniques:

1. The pattern matcher only makes minor tests in any given cycle, because previous pattern matching information is saved.
2. It also takes takes advantage of the fact that different rules have common patterns.

In addition to these enhancements, the efficiency of the system could be improved with data structures for making range searching in a two dimensional space more efficient, as suggested by Bentley (1979.) Because the displays are represented as linear lists of objects in the current system, when an object is added to the simulation display, the system has to execute a linear algorithm to place the object in the topology. A B-tree or a two dimensional array would greatly improve the performance of the low-level operation of adding and deleting objects.

9 Conclusion

This thesis attempted to show that:

1. The current version of ChemTrains is a language that can be learned and used by people with no computer science training.
2. "Problem-Centered Design with Walkthrough Feedback" was an effective design methodology.

The degree of success of each of these is elaborated below.

9.1 ChemTrains Conclusions

As computers with bitmap displays become more abundant, people without computer science training or even without any programming training will want and need to program. ChemTrains was designed to fill this need.

9.1.1 Design of ChemTrains

The current design is the third completed design and prototype in the last three years. The top design goals remain the same: to create an expressive and facile language for building graphical simulations that can be described with qualitative rather than quantitative models. The first design and implementation allowed objects to participate in reactions only when they are in the same place and allowed objects to travel along paths when they match filters on these paths. The second design and implementation permitted a more flexible description of rules and added variables for reasons of expressiveness. This thesis described the third design main design and implementation, exploring design alternatives needed in solving slightly larger problems that may have additional quantitative requirements. The language has evolved to become much more expressive, while still maintaining a fairly simple model of computation. The important general features of the current language are:

- matching the topology rather than geometry enables rules to be written at a high level;
- matching multiple configurations of objects enables individual rules to match conditions from multiple independent processes;
- inferring a rule's actions from differences in the pattern and result pictures enables a wide range of rules to be written;
- arbitrary nesting of objects enables simulation pictures to be described in detailed layers;

- variables enable rules to be generalized and enable rules to make use of attribute-value tables that are needed in decomposing complicated simulations;
- the lack of explicit typing enables simulations to be quickly created without a lot of setup work; and
- allowing a user to manipulate the simulation during the
- execution of rules enables a wide class of interactive behaviors to be described.

Together these features yield a language that is powerful enough to enable solutions to complex problems, yet is simple enough to learn quickly. During the design process the design space was expanded and organized into 111 design decisions. Many of the more detailed features were added for reasons of facility and/or expressiveness:

- object resizing enables a class of objects to be changed when any instance is changed;
- inferring object movement along paths enables the programmer to simply state a rule, even if the rule actions involve object movement;
- object replacement enables an object to replace another without needing an additional container;
- choosing data randomly rather than based on recency enables many simulations to work in a natural manner;
- rule probabilities enable other types of random behaviors to be described;
- conflict resolution based on rule ordering enabled the control of rules to be easily manipulated while not hindering the ability to control large sets of rules;
- the numeric operators enable objects or occurrences of events to be counted;
- automatic grid creation enables arbitrarily sized and connected fields of containers to be created; and
- importing simulations enables previous work to be reused.

The walkthrough analysis of the final design showed that there are only a few difficulties in solving every target problem. The walkthrough analysis also showed that the construction of the picture representation is generally harder than the construction of the rules. The user testing confirmed this and also showed that the many of the basics of the final language could be used effectively with very little training.

Although the language was designed for scientists and engineers, and these types of people performed well in the user testing, they showed some reservations in thinking how they could use the language in there own work. Many people have noted that ChemTrains may be more useful as a tool used by teachers for creating instructional interfaces for students than as a scientist's automated laboratory.

9.1.2 Future of ChemTrains

The current ChemTrains91 design could be improved along many dimensions. The direction of future efforts depends on whether it is to be used by students, teachers, scientists, user interface designers, or experienced programmers. Here are the main areas in which ChemTrains may be improved, regardless of its use.

- **Language features.** Sections 5.9 and 6.4 list a few things that should have been put into the final language, such as allowing a mouse click to be specified in a pattern, permitting more liberal modifications in a rule result, and allowing negation in a pattern.
- **Low-level language features.** Section 5.9 also lists a few lower level features that should be considered, including scrolling, multiple windows, and more graphical features, such as shading. These features were not considered in the current design of ChemTrains because the programming walkthrough analysis did not reach this level of detail, and also because the target problems did not emphasize some of the issues of scale. Lower level walkthroughs on a few select problems could help determine what these features should specifically look like.
- **Interface design.** The interface to the language was designed with little thought put into issues like what kind of command menu to use and how to separate drawing operations from rule operations. Since the user testing only tested the language, more research is needed to determine the actual usability of the language together with its interface.
- **Tutorial support.** ChemTrains like other languages for building interfaces has an advantage of possibly being able to teach the language to a new user with a tutorial simulation written in itself. When learning ChemTrains, the tutorial could guide the new programmer from using a simulation to learning its rules to modifying the rules to writing new rules.
- **Problem-solving support.** The interface could incorporate the problem-solving advice shown in section 3.3 into an organized help system. The advice could also be tied to working examples that demonstrate specific points of advice.
- **Efficiency improvement.** The efficiency of the inference engine could be improved by making changes suggested in section 8.4.
- **Name change.** The language was named "ChemTrains" when it was first being designed. Since the evolution of the language steered away from the original ideas that the language was based on, the language should be renamed. The computational model of the final system is no longer like Chemical reactions because patterns can match objects in multiple places in the display rather than objects together in the same place, and although paths still play an important role in the language, the fact that objects may move like trains along paths is not a significant feature.

9.1.3 Future Language Design

As computers become faster, the burden of program efficiency becomes less of an issue to programmers in many domains, freeing language designers to build languages which emphasize writability rather than efficiency. Most popular programming languages used today fit the underlying von Neumann architecture of most computers for reasons of efficiency. For example, the C programming language became popular partially because it allowed programmers to write code that mapped very directly onto machine instructions, making it very efficient while still having high-level programming constructs. Since writability was the main concern during the design of ChemTrains, efficiency did not affect the design of the language features. The fact that ChemTrains uses a model of computation that doesn't fit the underlying model of today's computer systems is irrelevant when considering writability.

The combination of computers becoming faster and cheaper should bring at least some demand for languages that can be used by people without programming training. Languages that fill this demand may not fit well with current computer architectures, forcing more ingenuity from the language compiler writers in the short term, and creating inspiration for computers whose architecture fits these languages better in the long term.

9.2 Design Methodology Conclusions

The methodology of "Problem-Centered Design with Walkthrough Feedback" was formalized and tested in the design of ChemTrains91 for two important reasons, to test the method of programming walkthrough evaluation more thoroughly and to test the usefulness of the programming walkthrough method in guiding and generating design alternatives.

9.2.1 Programming Walkthrough Analysis

The programming walkthrough method of analysis can be used to evaluate the writability of a single language design or to compare languages that are very close in nature. During the design of ChemTrains91, the method was used in the first capacity to find shortcomings with any given design and was used in the second capacity to compare isolated sections of possible designs.

The procedure as described in chapter two extends previous descriptions of the procedure by adding an additional step at the end, called the "walkthrough summary." This step was added because design decisions often hinged on subtle differences that lay below the surface of the walkthrough steps. The evaluation of the writability of a language can be likened to the "seashellability" of a beach. To get a quick but good estimate of whether the beach is worth shelling, someone has to actually pick up some shells and inspect whether they are good, rather than just counting the shell-like objects from a distance. In the same way, to get a good estimate of the writability of a

language, the evaluator has to look closely at the steps of the walkthrough, rather than just simply counting the walkthrough steps and counting the problem-solving advice. As with beach "seashellability," both the lengths and the close-up inspections need to be considered in an evaluation.

The user testing confirmed that the complexity of walkthrough steps varies widely by showing that the subjects' performances on different tasks guided by doctrine of similar sizes varied widely. Even though there was a conscious effort to consider details below the level of the length of walkthrough steps in the analysis, the results of user testing showed that the importance of the doctrine length and walkthrough length was overestimated, while the importance of the complexity estimates in the walkthrough summaries was underestimated.

In addition to varying widely, the complexity rating of the walkthrough steps did not always map directly to the subjects performances, showing that the practice of writing walkthrough summaries is fallible.

9.2.2 Problem-Centered Design with Walkthrough Feedback

Although the walkthrough evaluations did not perfectly predict user's difficulties, the evaluations were good enough to point out many shortcomings and to provide a basis for making the design decisions. The fact that the process uncovered many design alternatives and the final language was generally usable shows that the methodology was successful. Here is a summary of the methodological benefits, which have been more fully explained in section 5-10:

- The process forced the designer to solve problems as a first step in any design, quickly weeding out unworkable designs.
- The process of writing walkthrough summaries forced the designer to isolate and focus on the most difficult aspects of the language.
- The process of trying to simplify complex parts of walkthrough summaries led directly to the generation of new design features.
- The iterative design process was used to refine designs and to jump around the design space.
- The reuse of old walkthroughs made each design iteration quick.
- The design space table was a valuable resource in keeping track of the design alternatives.

9.2.3 The Failure of Automated Design Support

In formalizing the design methodology, an automated support tool was proposed as a possible way to lighten the drudgery of doing lots of walkthroughs and to organize and more easily deal with the connections between whole designs, design features, doctrine, target problems, and programming walkthroughs. A minimal automated tool for manipulating

these parts within the framework of iterative design was built. The tool allowed the designer:

- to generate versions of each of the types of design objects,
- to connect doctrine to features by name, and
- to connect walkthrough steps to doctrine by name.

From previous walkthroughs, previous doctrine, and the connections made between these objects and the features, the automated design tool was able to generate an initial doctrine and initial walkthroughs each with holes, given a specification of a new design as a set of answers to design questions. The design tool was used for the first two iterations of the design of ChemTrains91 but was then given up for the following reasons:

- The required mapping between the pieces of doctrine and the design alternatives was very fuzzy, requiring a lot of thought.
- The questions and the organization of the design space table were frequently modified, forcing many connections to be refigured out and changed.
- The connections between objects were all done with labels rather than with "hard links," making any link invalid when a name was changed.

A successful automated design tool would need to have hypertext-like links.

9.2.4 Future Design Methodologies using Walkthroughs

Every aspect of the current methodology was executed by a single person. This has the advantage that the progress of every design issue at any given time can easily be assessed, but has the disadvantage that the evaluation feedback is not very diverse, and the ideas for new design features is limited. As in the previous design of ChemTrains where three designers used informal discussion to unearth design shortcomings and alternatives, the addition of more designers into the process of iteratively analyzing programming walkthroughs should bring out more design ideas and a better overall understanding of a design's shortcomings.

Introducing more designers into the process raises the difficulty of communicating about the design. Good communication about design at least depends on maintaining:

- a consistent understanding of the current state of design, and
- some acknowledgement of the shortcomings of a design.

In the design of ChemTrains91, the design space table provided a concise way to keep track of the current design and the design alternatives, and walkthrough summaries provided an easy way to find shortcomings in the writability of the current design. In doing group design work, these two structures would help concisely and efficiently communicate the state of design.

References

Apple Computer, Inc. (1989). *Macintosh Allegro Common Lisp 1.3 Reference Manual.*

Baecker, R. (1980). Human-Computer Interactive Systems: A State-of-the-Art Review. In Kolers, P.A., Wrolstad, M.E. & Bouma, H. (Eds.) *Processing of Visible Language 2*, New York: Plenum Press, 423-443.

Balzer, R., Cheatham, T., & Green, C. (1983). Software Technology in the 1990's Using a New Paradigm, *IEEE Computer*, November 1983, 39-45.

Barry, D. (1985). *Bad Habits*, Henry Holt Co., NY, 169.

Bell, B. (1987). BOOGIE: An Object-Oriented Graphical Network Editor. Technical Report CU-CS-353-87, University of Colorado, Boulder.

Bell, B., Citrin, W., Lewis, C., Rieman, J., Weaver, R., Wilde, N., & Zorn, B. (1991) The Programming Walkthrough: A Structured Method for Assessing the Writability of Programming Languages. Technical Report CU-CS-xxx-91, Department of Computer Science, University of Colorado, Boulder. Submitted to SIGPLAN Conference on Programming Language Design and Implementation.

Bell, B., Rieman, J. & Lewis, C. (1991). Usability Testing of a Graphical Programming System: Things We Missed in a Programming Walkthrough. *Proceedings of CHI'91*, 7-13.

Bentley, J. & Friedman, J. (1979) Data Structures for Range Searching. *Computing Surveys 11*, 4, December 1979, 397-409.

Borning, A. (1979). ThingLab -- A Constraint Oriented Simulation Laboratory. Technical Report SSL-79-3, XEROX Palo Alto Research Center.

Buxton, W. & Sniderman, R. (1980). Iteration and the Design of the Human-Computer Interface. *Proceedings of the 13th Annual Meeting of the Human Factors Association of Canada*, 72-81.

Christensen, C. (1968) An example of the manipulation of directed graphs in the AMBIT/G programming language. *Interactive systems for experimental applied mathematics*, Klerer and Reinfelds, eds. New York, Academic Press, 423-435.

Christensen, C. (1971) An introduction to AMBIT/L, A diagramatic language for list processing. Proceedings of the 2d symposium on symbolic and algebraic manipulation. 248-260.

Clocksin, W. & Mellish, C. (1981). *Programming in Prolog*, Springer-Verlag, Berlin.

Citrin. W. (1990). Design Considerations for a Visual Language for Communications Protocol Specifications. *1991 IEEE Workshop on Visual Languages.*

Cox, P. & Pietrzykowski, T. (1988). Using a Pictorial Representation to Combine Dataflow and Object-Orientation in a Language Independent Programming Mechanism. *Proceedings International Computer Science Conference.* 695-704.

Edel, M. (1986). The Tinkertoy Graphical Programming Environment. *IEEE Proceedings COMPSAC.* 466-471.

Foley, J., & Wallace, V.L. (1974). The Art of Natural Graphic Man-Machine Conversation. *Proceedings of the IEEE 62*(4), 462-471.

Forgy, C. (1981). OPS5 User's Manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University.

Forgy, C. (1982). Rete: A Fast Algorithm for Many Pattern / Many Object Pattern Match Problem. *Artificial Intelligence 19*, 17-37.

Furnas, G. (1990). Formal Models for Imaginal Deduction. *Proceedings of the Twelfth Annual Conference of the Cognitive Science Society*, July 25-28, 1990, Cambridge, Mass. Hillsdale, NJ: Lawrence Erlbaum, 662-669.

Furnas, G. (1991). New Graphical Reasoning Models for Understanding Graphical Interfaces. *Proceedings of CHI'91*, 71-78.

Glinert, E. & Tanimoto, S. (1984). Pict: An Interactive Programming Environment. *IEEE Computer*, November, 1984. 7-25.

Gould, J., & Lewis, C. (1985). Designing for Usability: Key Principles and What Designers Think. *Communications of the ACM 28*(3), 300-311.

Grudin, J., Ehrlich, S., & Shriner, R. (1987). Positioning Human Factors in the User Interface Development Chain. *Proceedings of CHI+GI'87*, 125-131.

Hayes-Roth, F., Waterman, D., and Lenat, D., (eds.) (1983). *Building Expert Systems*. Addison-Wesley: Reading, Mass.

Ichikawa, T., & Hirakawa, M. (1987). Visual Programming - Toward Realization of a User Friendly Programming Environment. *Proceedings 2nd Fall Joint Computer Conference,* IEEE. 129-137.

Kahn, K. & Saraswat, V. (1990). Complete Visualizations of Concurrent Programs and their Executions. ? of IEEE. 7-15.

Kirkpatrick, S., Gelatt, C. & Vecci, M. (1983). Optimization by Simulated Annealing. *Science 220*, 671-680.

Laird, J., Newell, A., & Rosenbloom, P. (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence, 33*, 1-64.

Lewis, C. (1982). Using the 'thinking-aloud' method in cognitive interface design, IBM Research Report RC 9265 (#40713).

Lewis, C., Polson, P., Rieman, J. & Wharton, C. (1990). Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. *Proceedings of CHI'90*, 235-242.

Lewis, C., Rieman, J. & Bell, B. (1991, in press). Problem-Centered Design for Expressiveness and Facility in a Graphical Programming System. *Human-Computer Interaction 6*, 319-355.

Kennedy, K. & Schwartz, J. (1974). An Introduction to Set Theoretical Language SETL.

MacLean, A., Young, R., Bellotti, V., & Moran, (1991, in press). Questions, Options, and Criteria: Elements of a Design Rationale for User Interfaces. *Human-Computer Interaction 6*.

MacLean, A., Young, R., & Moran, T. (1989). Design Rationale: The Argument behind the Artifact. *Proceedings of CHI'89*, 247-252.

Markov, A. (1954). A Theory of Algorithms. USSR: National Academy of Sciences.

Maulsby, D. & Witten, I. (1989). Inducing programs in a direct-manipulation environment. *Proceedings of CHI'89*, 57-62.

McDermott, J. (1982). R1: A Rule-based Configurer of Computer Systems. *Artificial Intelligence 19,* 39-88.

McDermott, J. & Forgy, C. (1978). Production System Conflict Resolution Strategies. In D. Waterman & F. Hayes-Roth (Eds.) *Pattern-Directed Inference Systems.* New York: Academic Press.

Mosier & Smith (1986). Application of Guidelines for Designing User Interface Software. *Behaviour and Information Technology 5(1),* 39-46.

Myers, B. (1988). The State of the Art in Visual Programming and Program Visualization. Technical Report CMU-CS-88-114, Computer Science Department, Carnegie Mellon University.

Nassi, I. & Schneiderman, B. (1973). Sigplan Notices 8, 8, August, 1973.

Newell, A. (1973) Production Systems: Models of Control Structures, in Chase, W. (ed.) *Visual Information Processing.* New York: Academic Press, 463-526.

Osterweil, L. (1986). Software Process Interpretation and Software Environments. Technical Report CU-CS-324-86, Department of Computer Science, University of Colorado, Boulder.

Post, E., (1943). Formal Reductions of the General Combinatorial Problem, *American Journal of Mathematics, 65*: 197-268.

Repenning, A. (1991). Creating User Interfaces with AgentSheets. Technical Report CU-CS-517-91, Department of Computer Science, University of Colorado, Boulder.

Rieman, J., Bell, B., & Lewis, C. (1990). Design Supplement to ChemTrains Design Study. Technical Report CU-CS-480-90, Department of Computer Science, University of Colorado, Boulder.

Ringwood, G. (1988). Predicates and Pixels. *New Generation Computing, 7,* 1989. 59-70.

Sauers, R. & Farrell, R. (1982). GRAPES User's Manual, Technical Report ONR-82-3, Carnegie Mellon University.

Sauers, R. & Walsh, W. (1983). On the Requirements of Future Expert Systems. *Proceedings of IJCAI-83,* 110-115.

Scales, D. (1986). Efficient matching algorithms for the Soar/OPS5 production system (Technical Report KSL-86-47). Palo Alto, CA: Stanford University, Knowledge Systems Laboratory, Department of Computer Science.

Shu, N. (1988). *Visual Programming.* Van Nostrand Reinhold.

Spencer, R. (1985). *Computer Usability Testing and Evaluation.* Prentice-Hall, Englewood Cliffs, New Jersey.

Steele, G. (1984). *Common LISP, The Language.* Digital Press.

Swartout, W. & Balzer, B. (1982). On the Inevitable Intertwining of Specification and Implementation. *Communications of the ACM 25(7),* July 1982, 438-440.

Tambe, M. and Newell, A. (1988). Some chunks are expensive. *Proceedings of the fifth International Conference on Machine Learning.* 451-458.

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

Ungar, D. & Smith, R. (1987). Self: The Power of Simplicity. *Communications of the ACM,* October, 1987, 227-242.

Varley, C., Gradwell, D., & Hassell, M. (1973). *Insect Population Ecology: an analytic approach.* Berkeley, University of California Press.

Wasserman, A., Pircher, P., Shewmake, D. & Kersten, M. (1986). Developing Interactive Information Systems with the User Software Engineering Methodology. *IEEE Transactions on Software Engineering SE-12(2),* February 1986, 147-156.

Waterman, D., and Hayes-Roth, F. (eds.) (1978). *Pattern-directed Inference Systems.* New York: Academic Press.

Weaver, R. & Lewis, C. (1990). Examining the Usability of Parallel Language Constructs from the Programmer's Perspective. Technical Report CU-CS-492-90, Department of Computer Science, University of Colorado, Boulder.

Wilde, N. & Lewis, C. (1990). Spreadsheet-based interactive graphics: From prototype to tool. *Proceedings of CHI'90,* 153-159.

Young, R., Green, T., & Simon, T., (1989). Programmable User Models for Predictive Evaluation of Interface Designs. *Proceedings of CHI'89,* 15-19.

Yourdan, E. (1989). *Structured Walkthroughs*. Prentice-Hall, Englewood Cliffs, NJ.

Doctrine and Target Problem Index