

AD-A258 633



①

**Knowledge Acquisition for
Visually Oriented Planning**

Robert Leo'nard' Joseph

August 24, 1992

CMU-CS-92-188

DTIC
ELECTE
DEC 15 1992
S C D

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

*Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy*

DISTRIBUTION STATEMENT A

Approved for public release
Distribution Unlimited

© 1992 Robert L. Joseph

This research was sponsored in part by AT&T Bell Laboratories and in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of AT&T Bell Laboratories or the U.S. Government.

92-31344



92 12 14 013

1470

Keywords: Knowledge Acquisition, User Study, Artificial Intelligence, Planning Systems,
Graphical Input



School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

*Graphical Knowledge Acquisition for
Visually-Oriented Planning Domains*

ROBERT L. JOSEPH

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By <u>Per Lts.</u>	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

J. Harbort
THESIS COMMITTEE CHAIR

M. Harris
DEPARTMENT HEAD

8/27/92
DATE

9/28/92
DATE

APPROVED:

T. R. R. Y.
DEAN

9/28/92
DATE

Abstract

Many planning tasks can be represented using mental models in which an expert manipulates objects from one state to another (delivery route planning - trucks, buildings, packages, routes, etc.; part machining - parts, drill, mill, drill-bit, etc.). This suggests a highly graphical knowledge acquisition tool where the expert is able to capture the visual intuition of the problem solving to facilitate the encoding of a domain knowledge base. By exploring knowledge acquisition for object manipulation domains, insight will be gained in how knowledge is acquired and represented for such visually oriented tasks.

This thesis addresses graphical knowledge acquisition in visually oriented domains in the context of Prodigy, a general problem solving and planning architecture. The prototype system, called APPRENTICE, demonstrates the main ideas in the thesis. This system establishes the feasibility of a graphical interface to enhance the ability of the expert to develop factual domain knowledge (objects, relations, and operators) in multiple domains.

The system has been evaluated in four studies. In the first study, 32 AI students used the system to build their own domains. In the second study, domains developed by different types of users were completed faster using graphical input than using textual input. The third study was a learning study in which a subject developed several domains in APPRENTICE. Finally, the fourth study demonstrated the ability to develop a larger domain in the system. APPRENTICE and its techniques proved to be usable, flexible and extendable.

Acknowledgment

No man is an island. Although I assert that this research is my own work, I could not have done it by myself. There are a lot of special people who helped make this work a reality. Everybody who has touched my life has influenced this research and to all of you I owe a sincere thank you. In particular:

This thesis is dedicated to my parents, **Robert and Evelyn Joseph**, who through their guidance and support instilled in me a sense of drive to achieve my very best. They have always been my foundation - making sacrifices, providing encouragement, and giving me support. This work is as much a product of their support as it is of anything else. I owe them so much.

I want to thank my advisor, **Jaime G. Carbonell**, who has been with me through thick and thin. He has supported my research and helped me to grow as a person. Thanks to my committee members, **Herbert Simon, Dario Giuse, and Gary Kahn**. Herbert Simon's calm assurance that I was traveling down a good path for research helped me to press on. Dario Giuse's practical programming knowledge and insight into the field of User Interfaces and Knowledge Acquisition proved invaluable. Gary Kahn's industrial point of view has helped focus the research.

I would like to thank the initial users of APPRENTICE: **Jim Blythe, George Burroughs, Franne McNeal, William Scott Reilly, and Angelika Zobel**. Their patience was appreciated.

Several programmers helped with the implementation of the system: **Dan Kahn, Mike Miller, and David Rager**. I owe them my thanks.

I would also like to thank the 31 volunteer subjects from the 1991 Fall class of *Artificial Intelligence: Representation and Problem Solving*: **Dan Appelquist, Marcel Becker, Dina Berkowitz, Brian Bresnahan, Barry Brumitt, Jason Burgess, Jon Carlstrom, Colleen Duckett, Mike Gess, Jon Ghiloni, Melissa Goldman, Zach Hrabner, John Jeng, Maki Kato, Brian Kearney, Sanjay Khanna, Quan Le, Dave Leberknight, Mike Mantarro, Ben McCurtain, Adam Miller, Dan Morrow, Norman Murray, Dave Nespoli, Christine Page, Dave Park, David Poor, Kish Rao, Dan Schaffer, Jeremy Sechler, Anish Sirivasteva, and Andrew Weller**.

A thanks goes out to the volunteer subjects for Study 2: Sheila Anderson, Frank Berry, Ruben Carbonell, Alicia Pérez, Linda Schmandt, Charles H. Taylor, Manuela Veloso, José Garci, and Richard Goodwin

There were many people who proof read portions of this thesis and helped me to improve it. In addition to those previously mentioned I would like to also thank: Donna M. Auguste, Yolanda Gil, Jacquelyn E. Joseph, James Kistler, and Lisa A. Payne.

A special thanks to the CS community at CMU. Many individuals in the community have enriched my life.

And, finally, thanks to AT&T Bell Laboratories for sponsoring my education through the National Consortium for Graduate Degrees for Minorities in Engineering, Inc. (GEM) program and Cooperative Research Fellowship Program (CRFP).

Robert L. Joseph

Dedicated to my parents,
Robert and Evelyn Joseph.
They gave me life, a thirst for
knowledge and the will to
persevere.

Table of Contents

Chapter 1 - Motivation	1
1.1 Preview of Thesis	1
1.1.1 Example of a Domain Being Developed in APPRENTICE	2
1.2 Problem Outline	6
1.3 Significance of Research	7
1.4 Results	8
1.5 Reader's Guide	10
Chapter 2 - Related Work	11
Chapter 3 - Apprentice Architecture	17
3.1 Prodigy	17
3.1.1 Operation of the Prodigy Planner	18
3.1.2 Prodigy Format	20
3.1.2.1 Defining Object Types	20
3.1.2.2 Defining Predicates and Operators	20
3.1.2.3 Defining a State	22
3.1.3 Example Trace	22
3.2 Domain Builder	24
3.2.1 High Level Window Description	24
3.2.1.1 Model Window	25
3.2.1.2 Relation Window	26
3.2.1.3 Operator Window	29
3.2.1.4 State Window	30
3.2.1.5 Problem Window	32
3.2.1.6 Example Trace	33
3.2.2 Animation Algorithm	34
3.2.3 Primitive Elements for the Domain Builder	38
3.3 Framegraphics - Low Level Graphics	40
Chapter 4 - Empirical Analysis: User Studies	45
4.1 Study 1: Coverage and Usability	45
4.1.1. Study 1: Hypothesis	46
4.1.2 Study 1: Procedure	46
4.1.3 Study 1: Results	47
4.1.3.1 Eight Puzzle Domain	47
4.1.3.2 DNA Molecule Domain	51
4.1.3.3 Other Results	54
4.1.4 Study 1: Analysis	56
4.1.5 Study 1: Conclusion	58
4.2 Study 2: APPRENTICE Versus Emacs Study	58
4.2.1 Study 2: Hypothesis	58
4.2.2 Study 2: Procedure	58
4.2.2.1 Phase 1: Domain Building	59
4.2.2.1.1 Phase 1: Results	60
4.2.2.1.2 Phase 1: Analysis	61
4.2.2.2 Phase 2: Domain Understanding	62
4.2.2.2.1 Phase 2: Results	63
4.2.2.2.2 Phase 2: Analysis	64
4.2.3 Study 2: Analysis	65
4.2.4 Study 2: Conclusion	66

4.3 Learning Study	66
4.4 Study 4: Medium Size Domain Building	66
4.4.1 Study 4: Hypothesis	67
4.4.2 Study 4: Procedure	67
4.4.3 Study 4: Results.....	67
4.4.4 Study 4: Conclusion	69
Chapter 5 - Domain Characteristics.....	71
5.1 Positive Domain Characteristics	71
5.2 APPRENTICE Limitations.....	73
5.3 Techniques That Aid Large Domain Development	74
Chapter 6 - Conclusion.....	77
6.1 Summary of Findings	77
6.2 Contributions of This Research	77
6.3 Future Work.....	78
6.3.1 APPRENTICE-assisted Search Control Rule Development.....	79
6.3.2 Seamless Environment: Visual and Textual Representation.....	81
6.3.3 Spatial Multidimensional Relations	82
6.3.4 Apprentice Techniques for Non-visual Domains	82
Chapter 7 - References.....	83
Appendix A - Results Chart from Study 1	A-1
Appendix B - DNA Domain Code.....	B-1
Appendix C - Selected Domains from Study 1	C-1
Subject 1	C-2
Subject 12.....	C-3
Subject 22.....	C-5
Subject 24.....	C-6
Subject 29.....	C-8
Appendix D - Information Given to Subjects in Study 2 Phase 1	D-1
Prodigy and Domain Description.....	D-2
Emacs/Prodigy Example Domain.....	D-4
APPRENTICE Description	D-7
Strips World Description.....	D-10
Logistic World Description	D-11
Appendix E - Questions from Study 2 Phase 2	E-1
Appendix F - Code for Medium Size Domain.....	E-1
Appendix G - Learning Study Domains.....	E-1
Hiking World Description.....	E-3
Robot Picking Tulip Description	E-4

List of Figures

Figure 1.1 Several objects that are in a machining domain.....	2
Figure 1.2 Some relations between objects in the machining domain.....	2
Figure 1.3 State in the machining domain.....	3
Figure 1.4 The put-part-in-vise operator in the machining domain.....	3
Figure 1.5 Other operators in the machining domain.....	4
Figure 1.6 Sequence of operators in the machining domain to drill a hole in a part.....	5
Figure 1.7 Diagram of typical knowledge acquisition process.....	6
Figure 2.1 Table of differences among knowledge acquisition systems.....	15
Figure 3.1 Diagram of the APPRENTICE system.....	17
Figure 3.2 Functional view of Prodigy.....	19
Figure 3.3 IS-A Definitions for the blocks world.....	20
Figure 3.4 Operator format.....	21
Figure 3.5 Operators for the blocks world.....	22
Figure 3.6 Problem Definition for the blocks world.....	22
Figure 3.7 Partial solution trace in Prodigy for the blocks example domain.....	23
Figure 3.8 Model Window: editing the block-model object.....	26
Figure 3.9 Relation Window: developing the On-Table relation.....	27
Figure 3.10 Relation Window: showing a negated connection.....	28
Figure 3.11 Relation Window: showing a non-connection definition.....	28
Figure 3.12 Pickup operator before code is generated.....	30
Figure 3.13 Operator Window showing the Pickup operator with automatically generated Prodigy code.....	30
Figure 3.14 State Window: defining Initial-State-1.....	31
Figure 3.15 State Window: defining Goal-State-1.....	32

Figure 3.16 Problem Window: displaying the initial and goal state definitions.....	33
Figure 3.17 Visual animation showing blocks world problem being solved. These are several snapshot views of state changes with applied operator or backtrack command.....	34
Figure 3.18 Animation diagram.....	35
Figure 3.19 The Relation Window with the on relation.....	36
Figure 3.20 Adding the on relation to a state. The relative distance between the two objects in the relation is used to determine object placement.	36
Figure 3.21 Framegraphics diagram.....	41
Figure 4.1 Illustration of eight puzzle game: Initial and final state along with intermediate moves.....	48
Figure 4.2 Square and tile objects for the eight puzzle domain.....	48
Figure 4.3 Relations for the eight puzzle domain.....	48
Figure 4.4 APPRENTICE definition of the move operator for the eight puzzle domain.....	49
Figure 4.5 Automatically generated Prodigy code from the graphically.....	49
Figure 4.6 An initial state for the three-puzzle in APPRENTICE and in Prodigy.....	50
Figure 4.7 A goal state for the three-puzzle in APPRENTICE and in Prodigy.....	50
Figure 4.8 Steps for solving the 3-puzzle.	51
Figure 4.9 Objects in the DNA molecule domain.	51
Figure 4.10 Relations in the DNA molecule domain.....	52
Figure 4.11 Operators in the DNA molecule domain.....	53
Figure 4.12 Initial problem state in the DNA molecule domain.....	54
Figure 4.13 Goal problem state in the DNA molecule domain.	54
Figure 4.14 Statistics for 32 and 29 subjects in study 1.....	56
Figure 4.15 Plot of subjects versus domain development times from study 1.....	57
Figure 4.16 Comparison plot of Study 1 domain elements with the 29 subjects.....	57

Figure 4.17 Phase 1 domain building time. The chart shows faster development time for APPRENTICE than Emacs for all but the seasoned Prodigy expert.	61
Figure 4.18 Ratio of domain building time (EM/AP)	61
Figure 4.19 Graphical representations with a multiple-choice question.	62
Figure 4.20 Textual representations with a multiple-choice question.	63
Figure 4.21 Results of Phase 2, showing much better understanding (fewer errors) for APPRENTICE than Emacs.	64
Figure 4.22 Question 6 representing the move-block operator.	65
Figure 4.23 Table of subject 2's domain building time. The domains were built in order from left to right.	66
Figure 4.24 Table of the elements that are impacted when an element is changed.	68
Figure 5.1 Operator Window: machining domain organized at bottom of window.	76
Figure 6.1 Example of a STRIPS world type domain.	80
Figure 6.2 Example of possible control rule built with system assistance.	81
Figure 6.3 Currently APPRENTICE provides full data flow from the graphical interface to Emacs, but only limited data flow from Emacs to the graphical interface.	81
Table A.1 Study 1 Subjects	A-2

Chapter 1 - Motivation

1.1 Preview of Thesis

This thesis investigates the process of producing and exploiting graphically based specifications of visual domains for a general purpose planning system. The central thesis is that graphical specification helps both to reduce domain development time and to improve the accuracy of knowledge capture. The knowledge acquisition (KA) problem is presented and discussed in section 1.2.

This thesis sets out to:

- address the problems of ease, speed, and accuracy of knowledge acquisition for highly visual planning domains;
- develop techniques for providing knowledge acquisition access to people with domain expertise but with no knowledge acquisition skills;
- validate empirically these techniques across multiple domains and with different user populations;
- incorporate new graphical knowledge acquisition methods into a general purpose planning system without limiting its expressiveness.

A general paradigm has been developed for graphically defining a broad range of planning domains that lend themselves to a visual representation. A system, called APPRENTICE, has been developed, tested, and used to build many visual domains. Four studies were done with APPRENTICE to evaluate its performance with multiple users and numerous domains, as well as the evolution of the system's use over time. These studies suggest that the APPRENTICE techniques help the user to decrease development time, increase debugging efficiency, and comprehend existing domains faster.

1.1.1 Example of a Domain Being Developed in APPRENTICE

The APPRENTICE system tries to mimic some of the techniques that experts use to convey domain knowledge to a student or an apprentice. When teaching a new domain the expert first tries to systematically teach the student the objects, attributes, and relations in the domain, thus establishing a common vocabulary. The expert and student can then comfortably discuss this area of expertise. This technique can be used to develop many different domains; but to illustrate this interaction, I will describe the development of a portion of a machining task. This task is to develop a simple set of rules for drilling a hole in a part. In Chapter 4, this domain will be expanded into a larger one.

When experts describe a domain they first start by defining the objects that exist in that domain. Figure 1.1 shows different objects in the small machining domain. In this example the expert uses these objects to talk about the domain.

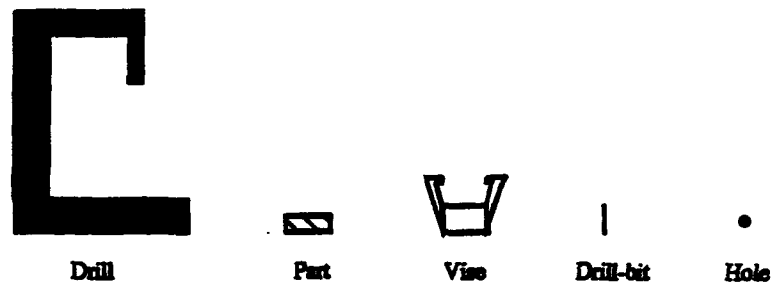


Figure 1.1: Several objects that are in a machining domain.

The pictures, along with the names of the objects, give the student a physical representation. The drill, drill bit, vise, part, and hole are now part of the common vocabulary between expert and student.

Once a subset of the objects is defined, the expert can describe how these objects relate to one another. In Figure 1.2 the relations between the objects in our machining domain are shown.

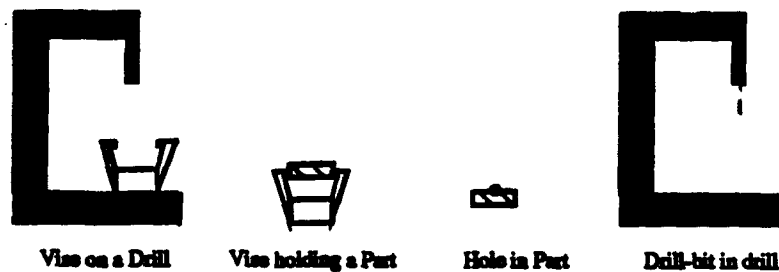


Figure 1.2: Some relations between objects in the machining domain.

These relations allow the expert to discuss how objects interact with one another. Again, the pictures and text allow the student to form a visual representation of relations. These are similar to the way the expert thinks about the object interactions.

With the objects and relations defined, the expert can then discuss conjunctive sets of relations or states. Figure 1.3 depicts a state in the machining domain. This figure represents the set of relations: vise on a drill, vise holding a part, and drill-bit in the drill.

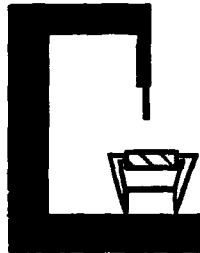


Figure 1.3: State in the machining domain.

This representation makes it relatively easy to visualize what is actually done in a machining shop. The expert is now able to discuss the machining domain based upon the expert's knowledge developed over time.

The expert can now discuss operations that, when applied, change the state. To describe these operators the expert first defines a pre-state (a state that has to exist before the operator can be applied), and then a post-state (a state that exists after the operator has been applied). In Figure 1.4 a simple operator, put-part-in-vise, is developed. This operator depicts the process of putting a part into a vise.

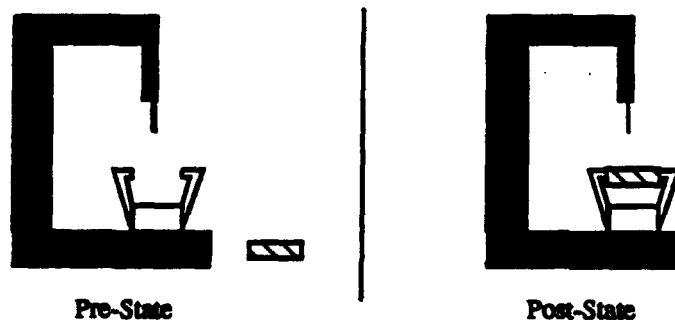


Figure 1.4: The put-part-in-vise operator in the machining domain.

The pre-state defines the conditions which are necessary before the operator can be applied: the vise must be on the drill and the drill bit must be in the drill. If these relations are true in the state then the operator can be applied and the state changed to reflect that the part is in the vise. These pre- and post-states represent the preconditions and the state transition for the put-part-in-vise operator. Figure 1.5 shows some other operators that are defined in a similar way (put-drill-bit-in-drill, put-vise-on-drill, drill-hole-in-part).

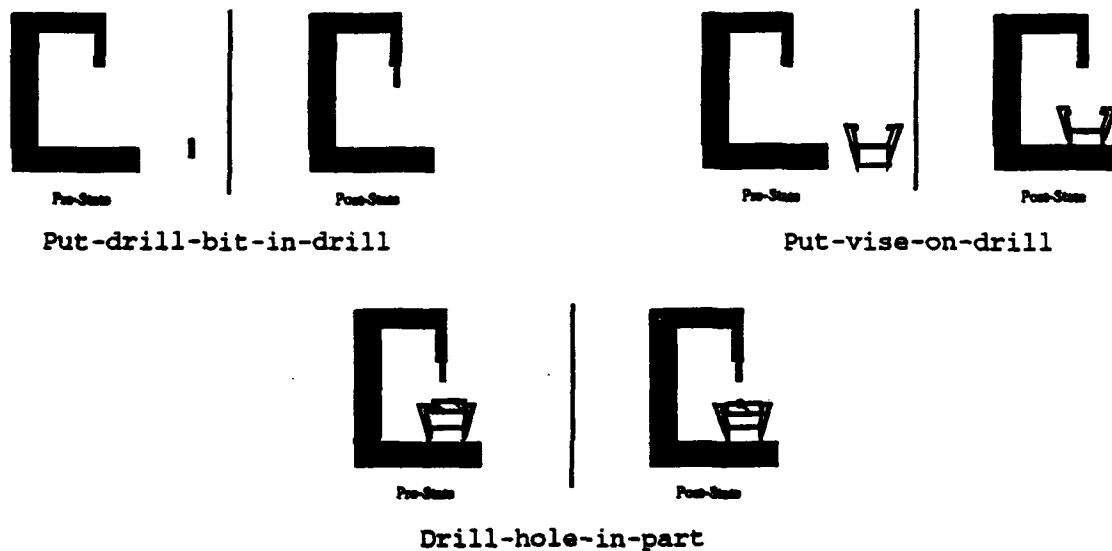


Figure 1.5: Other operators in the machining domain.

Finally, this information can be used by a planner to produce a sequence of operators that, when applied, will transform an initial state into a final state. The planner does this by first determining which operators achieve a goal and, if they are not applicable, which operators establish the pre-state of the previously related operator. This continues recursively until a sequence of operators is found that is both applicable to the initial state and achieves the goal. This process, called Means-Ends Analysis [Newell 72] may mean that the system has to try several alternative operator applications. Figure 1.6 illustrates one set of state transitions that the planning system took to solve the machining domain problem earlier defined.

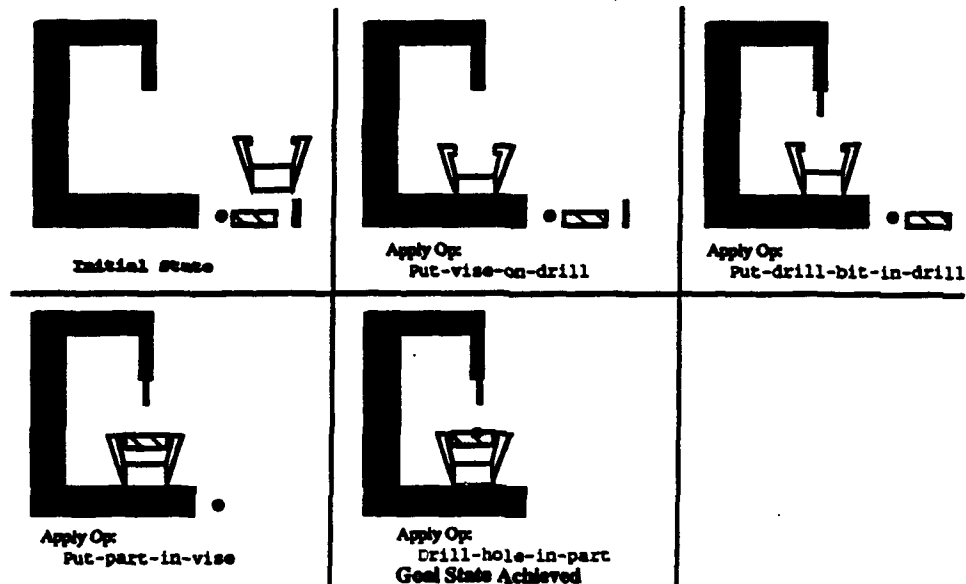


Figure 1.6: Sequence of operators in the machining domain to drill a hole in a part.

Each square represents the current state of the problem after the operator under the picture has been applied. Note that the initial state is composed of a part, a drill bit, a vise and the drill unassembled; the goal state is to have the part with a hole in it. The sequence of operators determined in Figure 1.6 to transform the initial state into the goal state is:

- 1) put-vise-on-drill
- 2) put-drill-bit-in-vise
- 3) put-part-in-vise
- 4) drill-hole-in-part

As the planner tries to find a solution, the expert can monitor the planner's progress. By monitoring the planner the expert can detect erroneous or incomplete knowledge. This helps with debugging domains, which is a significant part of domain development. The expert can also verify a sequence of operators by applying the operators one at a time.

APPRENTICE allows the expert to specify domain information similar to the domain description previously outlined. APPRENTICE, of course, handles significantly more complex domains as well. Chapter 3 describes this process in detail.

1.2 Problem Outline

Expert systems are increasingly being used in business and industry. The development of these systems requires the acquisition of knowledge from experts, a very time-consuming process for both domain experts and knowledge engineers. This knowledge acquisition (KA) process has been characterized as *"the transfer and transformation of problem-solving expertise from some knowledge source to a program"* [Buchanan 83]. By making the KA process faster and easier for experts, the domain development time and expense will be reduced.

Machine learning methods are helping to automate knowledge base development and improvement [Minton 88]. Nonetheless, initial domain development has heretofore been manual, and humans are still needed for domain refinements. Developing domains can be very difficult and time consuming, especially when the development tool does not reflect a simple mapping from the external domain to the system's user interface.

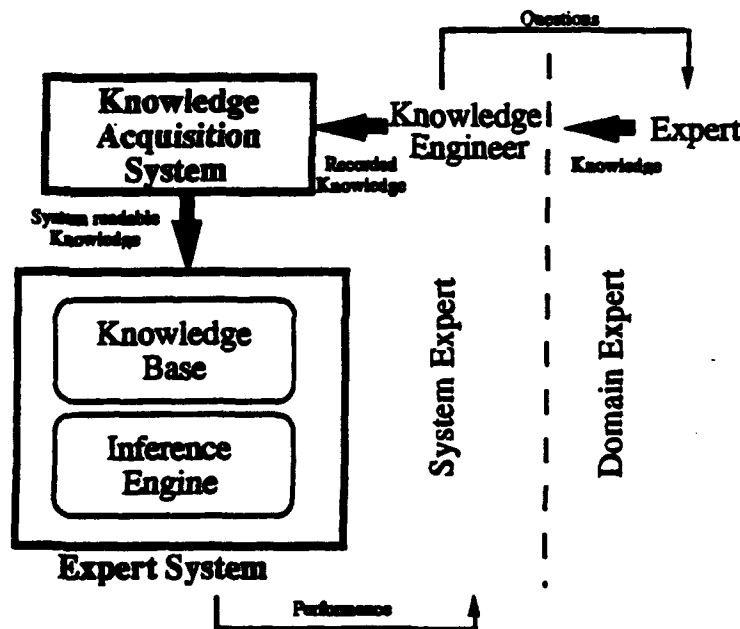


Figure 1.7: Diagram of typical knowledge acquisition process.

Knowledge base development is an interactive process. This is depicted in Figure 1.7. The role of the expert is to impart domain knowledge for the knowledge engineer to encode into the knowledge base. The knowledge engineer and the expert may be the same person or different people and they work towards developing the knowledge base. The knowledge

engineer has the responsibility of translating the expert knowledge into a representation for the expert system. If this representation is very different from the expert's representation then there are opportunities to introduce and propagate errors. These errors and misconceptions required diagnosis and correction, and thereby add to the development time of a domain.

In order to make this human development process more efficient, tools are needed to facilitate the initial domain specification, to aid in finding erroneous or incomplete information in a domain, and to help in correcting that domain information. This should be done within a paradigm that represents knowledge in a way experts can use and understand. This thesis examines a restricted type of knowledge acquisition that non-computer experts can use. I focus on the human interactive process involved when an expert develops a knowledge base for a planning system where the domain is composed of physical or mental objects being manipulated.

A graphical knowledge acquisition tool aids in domain development. The expert communicates with the computer in a way similar to the way an expert understands a domain, and the computer is responsible for transforming the information into a form that it can use. Also, having the computer communicate back to the expert in terms that the expert is more familiar with better enables the expert to debug and enhance the system's knowledge.

By focusing on the acquisition of knowledge for visual planning tasks in an expert/apprentice interaction, I show how knowledge base development can be facilitated graphically and improved for a range of users, including domain-knowledgeable but system-novice users.

1.3 Significance of Research

The research in this dissertation makes four significant contributions:

- a new knowledge acquisition methodology is developed for visual domains to decrease development time,
- the techniques can be applied by users without knowledge acquisition skills,
- the methodology is implemented and tightly integrated with a planning system (Prodigy),

- the methodology is tested by building multiple domains and using multiple subjects with varying skills.

The new graphical approach to knowledge acquisition allows the expert to reason about domain building in a way similar to how the expert thinks about the domain. These general techniques can be separated into basic elements for developing domains for a general expert system. With these techniques, the expert communicates with the system in a way that is comfortable to the expert; therefore, knowledge acquisition takes less time to complete. Also, the domain development can be done by experts that are not familiar with planning systems. This is radically different from other knowledge acquisition systems which provide detailed insight into the expert system, but in a way foreign to the actual domain. In such systems the expert or knowledge engineer is responsible for translating domain knowledge into the representation that the system expects.

The techniques developed in this thesis are used in building a working system that is tightly coupled with a planning system. This development allows proof of concept, testing of ideas, and enhancement of the methodology. Also, unlike knowledge acquisition tools that require a separate program to run the domain, APPRENTICE allows interactive domain development within a planning system. The system is tied directly with the planning system so that the results of a developed domain can be run on the system instantaneously. This is similar to using an interpretive language like LISP for development.

Once the system was developed, several experiments were designed and conducted to determine whether the ideas postulated in the thesis were true. These studies were done with different types of users doing different tasks. Results from these studies showed that different types of people were able to understand, build, and debug domains. The productivity of subjects was shown to be higher, and the subjects also found APPRENTICE more understandable than conventional methods.

1.4 Results

The primary contribution of these studies is to demonstrate the range of application and the efficiency of APPRENTICE. Most domains were acquired fully with the graphical interface, and acquisition speed proved faster for all types of users except Prodigy experts.

To investigate the claim of the thesis, four studies were performed. These studies tested the versatility, flexibility, and usability of the APPRENTICE system with multiple subjects in the development of several domains.

In the first study, APPRENTICE was used to build a diverse set of user-defined domains by 32 novice users. The users built domains of their own choosing. It took each user between one and 4.5 hours to construct a simple domain. All users succeeded in building functional domains, thereby proving the breadth, usability, and versatility of APPRENTICE.

In the second study, the APPRENTICE system yielded faster development time than conventional methods for several types of users. This was a comparison study with four sets of subjects with varying computer skills. Their skills ranged from novice computer users to advance planner experts. The development time of building a domain in APPRENTICE was compared with the development time using a text editor (Emacs). All but the most seasoned planner experts built domains significantly faster in APPRENTICE than in Emacs. Another phase of this study tested the ability of subjects to understand domains developed by others. This ability is important in domain development and system maintenance. The subjects understood the graphical representation of a domain more accurately (*i.e.*, they answered a set of questions about the domain more correctly) than they did for a comparable textual representation. This study proved the effectiveness of APPRENTICE in actual usage and validated the graphical KA paradigm for some important domains.

In the third study, APPRENTICE was used by one user to build a medium size domain that had 33 objects, 77 relations, and 34 operators in it. For this domain the graphical representation helped to enrich the development process. The pictures acted as visual cues in helping the subject develop the domain. For example, it was easy to recognize and correct the situation that a drill did not have a drill bit when creating the drill operator. Another advantage was that the graphics helped the user to remember the context in which he was working when he came back to the work after an interruption.

Finally, the last study was done with one user building several similar domains over time. In this study the user took longer to build the first domain but less time to build subsequent domains. This indicates that, as users build domains in APPRENTICE, their productivity increases.

These studies provide an indication of the merits of the APPRENTICE system. The system has been used:

- by multiple users
- to developed multiple domains
- over a period of time
- to developed a medium size domain.

Overall APPRENTICE was successfully used by 41 different people and 41 distinct domains were developed. These study results help to support the thesis of this dissertation.

1.5 Reader's Guide

Chapter 2 compares the research of this thesis with other, related work. Chapter 3 describes the implementation of the system. In Chapter 4 the empirical studies are explained and the results shown. Chapter 5 summarizes what I learned, and Chapter 6 draws some conclusions. The Appendices contain supporting material and are not needed in the understanding of the concepts reported in this thesis. Nevertheless, the information they contain may be of interest to some people, such as those attempting to replicate or expand upon the APPRENTICE system.

Chapter 2 - Related Work

The design and implementation of APPRENTICE draws upon ideas from several disciplines: user interfaces; visual programming and visual languages; machine learning; and, the most obvious, knowledge acquisition. The first three disciplines complement my work and have provided fertile ground from which to borrow many techniques. I will discuss some of the techniques that I used. The last discipline, knowledge acquisition has also provided many advantages to my work and helped me to explore new ideas. I will contrast my work with others in the area of knowledge acquisition.

User Interface

Techniques from the field of user interfaces were used to develop the human computer interaction for the APPRENTICE system. Direct manipulation was the basis for the interface design. This gives users, both novice and expert, an intuitive feel for how to use the system. User actions cause immediate reactions which are easy to relate to.

With the user interface several different types of interfaces were used. Techniques supported by MacDraw [MacDraw 89] were adapted to develop the user-defined objects. This provides users with an easily understood interface for creating simple graphical objects. Icon manipulation enables production of other domain elements with these objects as building blocks. This allows users shortcuts for developing relations between the objects represented as icons [Macintosh 87]. The interface also strives for consistency, user feedback, handling of user mistakes, a simple display and actions that have quick feedback. These are important qualities of a good interface design. These and other techniques are described in much of the literature [Goldberg 80] [Shneiderman 86] [Cardelli 88] [Myers 92].

Visual Programming and Visual Languages

Visual programming and visual languages are orthogonal fields of study that benefited APPRENTICE. The goals of Visual Programming are to create working programs [Cox 89], systems [Ichikawa 87] [Fischer 88] or investigate ideas [Henderson 86] [Gutfreund 87]. These are different from the goal of APPRENTICE, which is to develop domains for

a planning system. One difference in the way that APPRENTICE works is that most visual programming systems have a predefined set of icons, and therefore the users have to conform their thinking about the system in the way that the software designers had in mind. To aid in domain development, APPRENTICE provides the user with the ability to define the objects of the domain in a visual representation that corresponds closely to the physical world. This allows the user a comfortable environment to work in, where ideas representing physical relations do not have to be transformed into an unrelated representation.

The advantages of animation have been demonstrated by systems such as Balsa-I [Brown 84], Balsa-II [Brown 88], Animus [Duisberg 86], and STEAMER [Hollan 84]. Animation is also used in APPRENTICE to give the user feedback when the system is solving a problem.

Machine Learning

Machine learning is another field of study which has had influence on my work. Currently, machine learning research cannot create initial domain information without the aid of a human. Automated learning methods allow knowledge that has already been encoded to be refined. This can be achieved by decreasing the search space as with EBL [Minton 88], by statically compiling knowledge [Entz 90], by using abstraction techniques [Knoblock 90] or analogous plan usage [Veloso 92] or by adding information to an already existing knowledge base as with experimentation [Gil 92]. APPRENTICE is a tool that helps with initial domain formulation, and therefore could be a useful complement to machine learning systems.

Also, search control rules [Minton 89] (which can be learned by different methods [Minton 88] [Entz 90]) could be directly used in APPRENTICE to decrease search time. Currently, no matter how these search control rules are obtained they can be used with APPRENTICE.

Knowledge Acquisition

In the knowledge acquisition field several techniques have been used to try to elicit expert knowledge, and map that knowledge into expert system primitives.

Alexander et. al. [Alexander 87] have developed a technique called ontological analysis or SUPE-SPOONS which aids in knowledge-level analysis of a problem space. The analysis divides domain knowledge into three categories:

- 1) Static Ontology - physical objects or primitive objects in a problem space, their properties and relations.
- 2) Dynamic Ontology - state space of the problem-solving domain and the actions that transform the problem from one state to another state.
- 3) Epistemic Ontology - constraints and methods that control the use of knowledge applied to the static and dynamic ontologies.

Complex domain ontologies are constructed in a three-step process in the order that the categories are presented. The system ASTEK [Jacobson-90], based on the SUPE-SPOONS research, provides multiple paradigms for knowledge editing while maintaining a single consistent framework. This knowledge editing is in the form of natural language and graph editing techniques.

The ontologies created with the system are very similar to the workings of APPRENTICE. The static ontology parallels object and relation definitions. The dynamic ontology is like the operator definition. The epistemic ontology is similar to search control rules. Like the ontological analysis technique, APPRENTICE builds its knowledge base in similar stages. APPRENTICE attempts to capture and exploit both the visual and semantic domain knowledge.

Systems that take different approaches include: KREME [Abrett 87] a knowledge representation editing and modeling environment; KRITON [Diederich 90] a hybrid system that uses interview, protocol analysis, and incremental text analysis; and AQUINAS [Boose 87] a system that interviews experts to build tables of distinctions or repertory grids between elements. The systems fall into the following information gathering techniques:

- Visual and Semantic Acquisition (APPRENTICE)
- Interviewing and Protocol (KRITON)
- Clustering or Scaling (AQUINAS)
- Natural Language Systems (ASTECK , KRITON)
- Visual Graphs of Underlying Data Structure (KREME)

PROTEGE [Musen 88] has had success providing a graphical program for developing graphical knowledge acquisition and editing tools. These knowledge acquisition tools elicit knowledge for the domain independent planner e-ONCOCIN, which uses the method of skeletal refinement for problem solving. PROTEGE has been used to create KA tools for oncology protocols (p-OPAL) and hypertension protocols (HTN). With the KA tools, doctors, without the aid of knowledge engineers, can create expert systems to advise in the treatment of oncology and hypertension. The interface to these knowledge acquisition tools uses form filling and graphical flow chart building to elicit the domain knowledge. Unlike APPRENTICE, PROTEGE is not integrated with the target expert system. Also, PROTEGE's domain is more limited, although the authors are currently expanding the system into PROTEGE-II. Finally another important difference is that PROTEGE is not a system for developing planning domains.

QMR-KAT [Giuse 90] is another system for knowledge acquisition in medicine which lets experts build knowledge bases without a knowledge engineer to act as intermediary. The QMR knowledge base is one of the most comprehensive knowledge bases in use for medical decision support for diagnosis of internal medicine diseases. Knowledge base information consists of textual descriptions of a disease called *disease unit*. This representation is different than the visual approach of APPRENTICE, and of course the QMR system is not a planning system.

There are other knowledge acquisition systems that take a different approach than APPRENTICE. MORE [Kahn 85] is a system which automatically interviews experts for the information they use to solve diagnostic problem. ASK [Gruber 89] is a system that elicits justifications from experts and generates strategy rules. It is used with the system MUM, which is a knowledge system that concentrates on the strategic aspects of diagnosis of chest pain. DACRON [Mahling 89] is a visual knowledge acquisition tool for the POLYMER planning system. SALT [Marcus 86] is a system that assists in knowledge acquisition for incremental design tasks such as configuration. The system uses a propose-and-revise problem solving strategy. ROGET [Bennet 85] is a program to assist knowledge engineers and domain experts in developing EMYCIN-based expert systems. KNACK [Klinker 88] is a knowledge acquisition tool for building knowledge bases to generate reports. This system is interesting because it is sample-based: the expert provides a sample report, and the system analyzes and generalizes the sample.

To help explain some differences between the above systems, six system characteristics are outlined below. These characteristics are then shown in a chart for comparison.

Feedback Type - type of feedback the system gives (textual, animated, graphs).

Knowledge Developed - type of knowledge the system can acquire (static and/or dynamic knowledge).

Domain Type - types of domains the system is used for.

Generate-Test Cycle - development done in generate and test mode in conjunction with the target expert system.

Mode Type - system-directed acquisition or user-directed acquisition.

Knowledge Acquired - acquire incremental, factual and/or better performance knowledge.

	APPRENTICE	MORE	ASK	OPAL	DACRON	SALT	ROGET
Feedback Type	Animated	Text	Text	Form Fill Flowchart	Icons	Text	Text
Knowledge Developed	Static	Dynamic	Dynamic	Both	Both	Both	Both
Domain Type	Visual Planning	Diagnostics	Medical	Medical	Hierarchical	Constraint Satisfaction	Classification Problem Solving
Generate-Test Cycle	Yes	Yes	Yes	No	Yes	No	No
Mode Type	User	System	System	User*	User	System*	System
Knowledge Acquired	Factual	Incremental	Incremental	Factual	Incremental	Incremental	Incremental

	QMR-KAT	KNACK
Feedback Type	Text	Text
Knowledge Developed	Static	Static
Domain Type	Medical Diagnostics	Report Generation
Generate-Test Cycle	Yes	No
Mode Type	Both	Both
Knowledge Acquired	Incremental	Incremental

* No interaction with expert system

Figure 2.1: Table of differences among knowledge acquisition systems

Finally, knowledge base development environments such as KREME [Abrett 87], ONTOS [Nirenburg 88], and CYC [Lenat 86] elicit mostly static or factual knowledge. These systems investigate how to organize and maintain large knowledge based systems. Handling large knowledge bases is a future direction of the APPRENTICE system.

The major difference between APPRENTICE and other systems is that APPRENTICE enables users to model the physical world with their own defined representations. Also, the APPRENTICE system is interactive with the Prodigy planning system providing a closed loop for development and utilization of a knowledge base. Although the planner and acquisition tool are integrated the APPRENTICE techniques are also described independent of a planning system.

Chapter 3 - Apprentice Architecture

The APPRENTICE system can be subdivided into three components: the graphical interface component, Framegraphics; the domain building component, Domain Builder; and the planning system, Prodigy. I built the Framegraphics system and the Domain Builder for this research. Figure 3.1 depicts the relationships among the components in more detail. In this chapter, I will discuss in some detail the various components of the system.

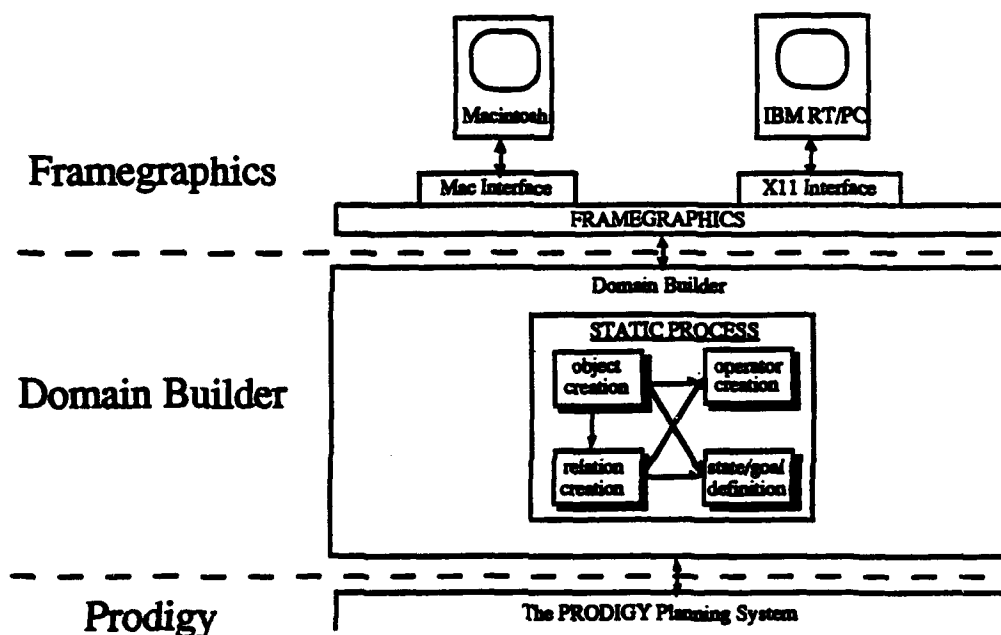


Figure 3.1: Diagram of the APPRENTICE system.

3.1 Prodigy

Prodigy [Minton 89] is a domain-independent problem-solver system used primarily as a testbed for research in planning, machine learning, and knowledge acquisition. It uses basic means-ends analysis in planning for high-level, symbolic domains. Since its inception, the Prodigy system has continued to develop. The first version of the system, Prodigy 2.0, was a linear planner (*i.e.*, it did not allow interleaving of goals) [Minton 89]. The next

version of Prodigy, Nolimit, was a non-linear planner which provided interleaving goals [Veloso 92]. The syntax of the two systems is slightly different, with the biggest difference being that Nolimit requires all objects used in a domain to be explicitly defined in the syntax (see section 3.1.2).

The basic design of APPRENTICE was able to accommodate both planners. APPRENTICE was first developed using Prodigy 2.0 but now works with Nolimit. Porting APPRENTICE to the different problem solvers required very little modification.

The basic functioning of APPRENTICE with a problem solver will be discussed later. This section will outline the Nolimit Prodigy planner. Then a simple trace of the planner solving a problem will be described; and finally, an outline of the input language will be presented. This is the planner which was used in the studies described later.

3.1.1 Operation of the Prodigy Planner

A domain theory in Prodigy consists of operators, inference rules, and search control rules. These rules are essentially If-Then rules. The left-hand sides of these rules are represented by a Prodigy Description Language (PDL). This language is a form of first-order predicate logic. PDL allows conjunctions, disjunctions, negations, and existential and universal quantifications. APPRENTICE automatically generates conjunctions and existential quantification from its pictorial representation. A disjunction can be modelled as multiple rules, and negations can be represented explicitly. Universal quantifications are not automatically generated by the APPRENTICE system but can be added manually to the generated code. Thus far, none of the domains that were built in APPRENTICE needed universal quantification. Therefore a lot of resources were not devoted to developing a simple way to generate and depict universal quantification graphically.

Operators and inference rules are factual knowledge that modifies the state as the planner is solving a problem. Although inference rules do not correspond to external actions, they can be modelled as operators. On the other hand, search control rules are control knowledge that directs the search for the solution. This thesis deals mainly with the creation of factual knowledge. Developing search control knowledge is discussed in section 6.3.

Operators have three parts: the parameter list, the preconditions, and the effects. The effects part of an operator either adds or deletes a predicate from the state. When given an initial and goal state along with a domain definition, the Prodigy planner tries to determine a

sequence of actions (instantiated operators) that when applied in a given order will change the initial state to achieve the goal state. The search tree initially begins with a single node representing a set of state predicates in the initial state and a set of goal predicates to be achieved. To move from the initial state to the goal state, the tree is expanded by repeating two phases: the decision phase and the expansion phase.

The first step during the decision phase is to choose a node. The second step is to choose a goal to focus on, which becomes the current goal. The third step is to choose an operator to achieve the current goal. The final step is to choose bindings for the variables in the operator (this is called instantiating the operator). The default search strategy for this decision process is depth-first search but can be guided by search control rules. These search control rules are used to prune the search tree [Minton 89].

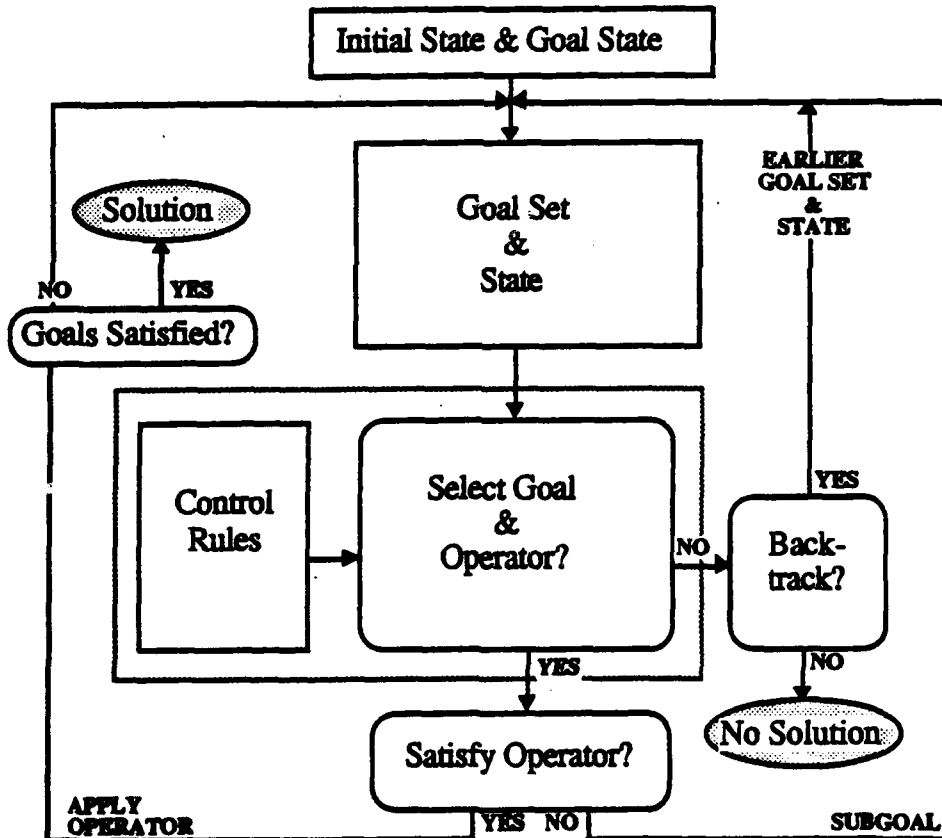


Figure 3.2: Functional view of Prodigy.

During the expansion phase a new node is created. If the instantiated operator's preconditions are satisfied by the state, then the operator is applied. Otherwise, the system

subgoals (changing the current goal) on an unmatched precondition: adding the old current goal to the set of goals that are pending; or subgoals on a previously unachieved goal.

When a state is achieved in which all of the top-level goals are satisfied, the problem is solved. Figure 3.2 shows a high level flowchart of the Prodigy system.

3.1.2 Prodigy Format

Writing a domain in Prodigy requires several steps. The user must create object prototype definitions, predicates representing relations among objects, operators, instances of objects, start states, and goal states. An example of a domain being built will be shown using a slight variation of the blocks world domain. The blocks world domain consists of a set of blocks that can be stacked on/unstacked off each other or put-down on/picked-up off the table. In this blocks world the table and the arm are represented explicitly.

3.1.2.1 Defining Object Types

Object definitions for a domain are defined with the "is-a" function. These definition types tell Prodigy the name of the objects in the domain. In the example blocks world, the user will need block objects, table objects, and arm objects. An object definition has the following format: (IS-A *object-name* TYPE). In this example, these objects are defined as follows:

```
(IS-A Block-model TYPE)
(IS-A Table-model TYPE)
(IS-A Arm-model TYPE)
```

Figure 3.3: IS-A Definitions for the blocks world

3.1.2.2 Defining Predicates and Operators

The operator defines legal transitions between states. Each operator has a precondition that must be satisfied before the operator can be applied and an effects-list that describes how the application of the operator changes the state. Specifically, the effects-list indicates the atomic formulas that are added to and/or deleted from the state when the operator is applied. These preconditions and effects are represented by predicates. Predicates represent relationships between objects. In order to allow an operator to work in a generic way, the predicates use variables. These variables match different objects in the state definition. "<>" surrounding a name represents a variable. An operator format is shown below:

```

(OPERATOR name-of-operator
  (PARAMS
    ((<var1> type1)
     (<var2> type2)
     (<var3> type3) ...))
  (PRECONDS
    (AND (pred1 <var1> <var2>)
          (pred2 <var2> <var3>)))
  (EFFECTS
    ((DEL (pred1 <var1> <var2>))
     (ADD (pred3 <var1> <var3>))) ... )

```

Figure 3.4: Operator format.

Operators in the blocks world are defined in Figure 3.5. Consider the operator Pick-up. In its PARAMS list the variable <ob> is of type block-model, <table> is of type table-model, and <arm> is of type arm-model. In the precondition, the predicate (clear <ob>) checks that the block <ob> has nothing on top of it; the predicate (on-table <table> <ob>) checks to see if the variable <ob> is on the <table>; and the predicate (empty <arm>) checks that the <arm> is empty. Finally, if all of these predicates are true for a particular block, table and arm in the current state, then the matching block is picked up off the table by the arm. Also the predicates in the effect list update the state by deleting/adding the relevant predicates.

```

(OPERATOR PICK-UP
  (params ((<ob> BLOCK-MODEL)
            (<table> TABLE-MODEL)
            (<arm> ARM-MODEL)))
  (preconds (and (clear <ob>)
                  (on-table <table> <ob>)
                  (empty <arm>)))
  (effects ((del (on-table <table> <ob>))
            (del (clear <ob>))
            (del (empty <arm>))
            (add (holding <arm> <ob>)))))

(OPERATOR UNSTACK
  (params ((<ob> BLOCK-MODEL)
            (<underob> BLOCK-MODEL)
            (<arm> ARM-MODEL)))
  (preconds (and (on <ob> <underob>)
                  (clear <ob>)
                  (empty <arm>)))
  (effects ((del (on <ob> <underob>))
            (del (clear <ob>))
            (add (holding <arm> <ob>))
            (add (clear <underob>)))))

(OPERATOR PUT-DOWN
  (params ((<ob> BLOCK-MODEL)
            (<arm> ARM-MODEL)
            (<table> TABLE-MODEL)))
  (preconds (holding <arm> <ob>))
  (effects ((del (holding <arm> <ob>))
            (add (clear <ob>))
            (add (empty <arm>))
            (add (on-table <table> <ob>)))))

(OPERATOR STACK
  (params ((<ob> BLOCK-MODEL)
            (<underob> BLOCK-MODEL)

```

```

      (<arm> ARM-MODEL))))
(preconds (and (clear <underob>)
               (holding <arm> <ob>)))
(effects ((del (holding <arm> <ob>))
          (del (clear <underob>))
          (add (clear <ob>))
          (add (on <ob> <underob>)))))

```

Figure 3.5: Operators for the blocks world.

3.1.2.3 Defining a State

After the domain is developed, the problems that the user wants to solve must be defined. A particular problem is defined by instances of objects, a start state, and a goal expression, which is a subset of the entire goal state. Below is a problem in the blocks world. The instances are A, B, C, A_Table, and An_Arm. The goal is to get A on A_Table, B on A, and C on B. The start state has Blocks A, B, and C on A_Table and An_Arm empty.

```

(HAS-INSTANCES Block-model A)
(HAS-INSTANCES Block-model B)
(HAS-INSTANCES Block-model C)
(HAS-INSTANCES Table-model A_Table)
(HAS-INSTANCES Arm-model An_Arm)

(GOAL (On B A)
      (On C B)
      (On-table A))

(STATE (AND (On-Table A_Table A)
            (On-Table A_Table B)
            (On-Table A_Table C)
            (Clear A)
            (Clear B)
            (Clear C)
            (Empty An_Arm))))

```

Figure 3.6: Problem Definition for the blocks world.

3.1.3 Example Trace

Using the domain and problem definition from the previous section, Figure 3.7 depicts a partial solution trace of the Prodigy planner. The planner uses means-ends analysis to solve the problem. The planner first determines which operators achieve the final goal. If the operators are not applicable, it then determines which operators establish the pre-state of the previously related operator. This continues recursively until a sequence of operators is found that is applicable to the initial state and achieves the goal.

The trace shows a segment of the solution path in which the system backtracks. Each line contains information about the step in the means-ends analysis being tried. The Tn# displays the node name of the solver's current position. The next part of the line is either a goal to be worked on, an instantiated operator to try and satisfy, or an operator to apply.

The applied operators are capitalized. If there are additional choices available at a particular node that information is displayed in the form of a list (e.g., goal-choices-left or ops-left).

The first part of the partial trace, starting with the *****, does not succeed because it cannot satisfy the goal (on b a) with c already stacked on b. This causes a condition where a state would be repeated and the solution search would end up in an infinite loop, therefore the system backtracks and reorders the top level goals. With this reordering, the second part of the trace leads to a solution. The solution found is listed at the end of the trace.

After several other attempted solutions during problem solving

```

*****
tn1 (done)
tn2 (*finish*)
| tn3 (on c b)
|   goal-choices-left: ((on b a))
| tn4 (stack b c an_arm)
| | tn5 (holding c an_arm)
| | tn6 (pickup an_arm c a_table)
| |   ops-left: ((unstack b c an_arm) (unstack c c an_arm) (unstack a c
an_arm))
| | TN7 (PICKUP AN_ARM C A_TABLE)
| TN8 (STACK B C AN_ARM)
| tn9 (on b a)
| tn10 (stack a b an_arm)
| tn11 (holding b an_arm)
| tn48 (unstack a b an_arm)
*** STATE LOOP ***

***** BACKTRACK TO NODE T2 *****

tn2 (*finish*)
| tn56 (on b a)
| tn57 (stack a b an_arm)
| | tn58 (holding b an_arm)
| | tn59 (pickup an_arm b a_table)
| |   ops-left: ((unstack b b an_arm) (unstack c b an_arm) (unstack a b
an_arm))
| | TN60 (PICKUP AN_ARM B A_TABLE)
| TN61 (STACK A B AN_ARM)
| tn62 (on c b)
| tn63 (stack b c an_arm)
| | tn64 (holding c an_arm)
| | tn65 (pickup an_arm c a_table)
| |   ops-left: ((unstack b c an_arm) (unstack c c an_arm) (unstack a c
an_arm))
| | TN66 (PICKUP AN_ARM C A_TABLE)
| TN67 (STACK B C AN_ARM)
TN68 (*FINISH*)

*****

This is the solution found:
(pickup arm b table)
(stack a b arm)
(pickup arm c table)
(stack b c arm)
(*finish*)

```

Figure 3.7: Partial solution trace in Prodigy for the blocks example domain.

3.2 Domain Builder

The domain builder allows an expert to graphically represent and create a domain. This graphical process resembles an expert's experience more closely than writing a domain in textual form as described in the previous section. Once a graphical domain is defined, the domain builder then automatically translates this graphical representation into PDL, which is loaded and run in the Prodigy planner. The graphical system is tightly integrated with the planner, providing intuitive domain development. This section outlines the implementation of graphical knowledge acquisition, then discusses the primitives that are needed for the process independent of the actual implementation.

3.2.1 High Level Window Description

In developing a domain using the domain builder, several things have to be defined: objects, relationships between the objects, operators to change sets of relationships, and initial and goal states for defining problems. In the domain builder, separate windows are used to develop each particular element. The windows are:

Model Window - Allows objects in a domain to be defined.

Relation Window - Allows relationships between objects to be defined.

Operator Window - Allows state changes to be defined.

State Window - Allows the definition of a start state and a goal state.

Problem Window - Allows the setup and running of a problem to be solved.

Apprentice Window - Allows the manipulation of a domain.

The four subcomponent windows (object, relation, operator, and state) are used to define a domain. The Problem Window is used to create a problem that will be run in the Prodigy planner. The Apprentice Window is used for domain manipulation: saving, deleting, and loading domains.

All the windows have a similar MacDraw™[MacDraw 89] style interface, which is mouse- and object-oriented. This similarity allows the user to interact with each window in a consistent manner. A single mouse "click" selects an object; "click and drag" moves an object; and "double-click" allows an object's name to be edited or a domain element to be made available for editing. Windows are reconfigurable. All objects in the window can be easily moved, and their locations saved, so that each window can be customized to a user's preference.

Each window has a work area in which a particular domain element type (objects, relations, operators, and states) can be developed. This work area allows elements to be developed one at a time. For example, in Figure 3.8 the Model Window's work area is used for building a block object. To build or modify a different object, the block object is iconified and another object can be manipulated. This iconifying process takes the sub-parts of the current element from the work area and represents all of the information in a single name that can be placed anywhere on the screen. An element can be edited again by double clicking on its iconified name. This action places the original element in the window, and the sub-parts of the double-clicked element are put into the work area ready to be modified.

3.2.1.1 Model Window

The Model Window is used to create prototype objects of the domain along with connection points (*i.e.*, location in which the objects will interact with one another). These objects are then used in defining relations, operators, and states. The work area allows one object at a time to be developed. The work area has a very simple object-oriented drawing package type interface. In this work area lines, text, instance names, and connection points can be added, deleted, or edited for an object. Each element in the work area can be manipulated. To create a particular element type, the user selects the corresponding box from the palette on the left of the work area. In Figure 3.8, the box element has been selected.

Once all the modifications to an object are made, then the results can be saved, thus incorporating the changes into the prototype object. Objects can also have attribute information attached to them. Attributes are added to objects by double clicking on the Attr: field. This brings up a popup editor which allows attribute information to be added. This attribute information is specified by an attribute name and the variables with type definitions that follow. In Figure 3.8 the attribute of the block is weight, and its value is some number. During the generation of a state or operator this attribute will translate into (weight <blockxxx> <lbs>). Each individual object can be edited to include or exclude attributes.

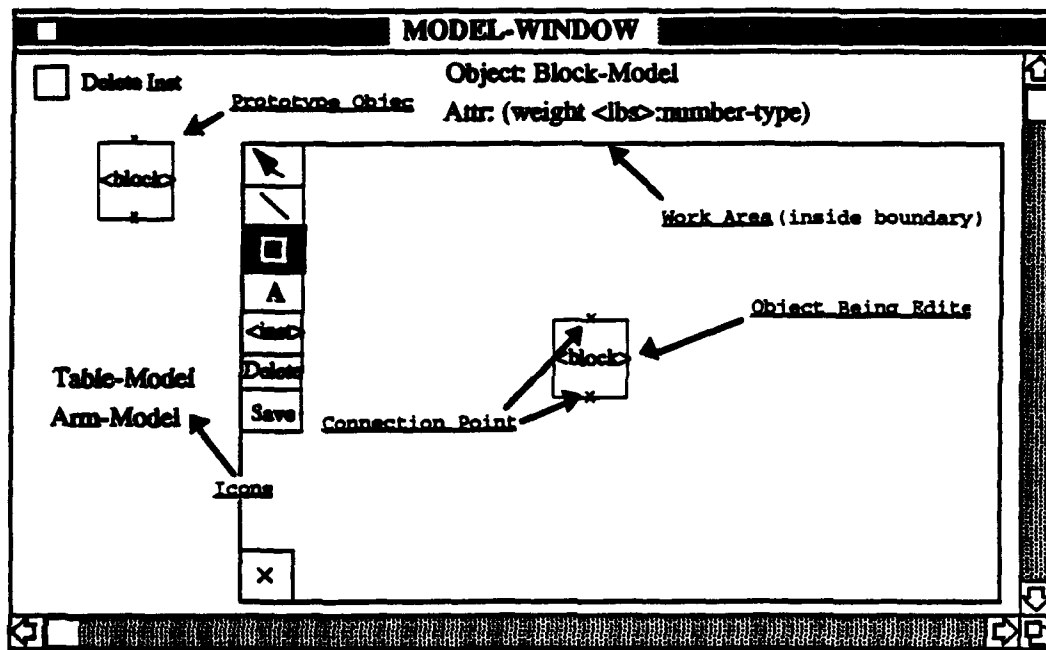


Figure 3.8: Model Window: editing the block-model object.

3.2.1.2 Relation Window

The Relation Window allows the user to build relations between objects and automatically generates a predicate from the objects. These relations are used in the automatic code generation during the operator and state definition. This allows users to define much of the rule and state knowledge graphically, and the system is able to check and flag inconsistencies in the definitions. Figure 3.9 is an example of the Relation Window.

In the Relation Window, object prototypes that the system knows about are pictured around the outside of the work area. These objects can be moved by the user to any position outside the work area. Relations are defined by dragging prototype objects, one at a time, into the work area. This produces instances of the objects. Instances are then connected together by their connection points to represent the relation. This visual picture translates into a textual list of the relation name and the objects in the relation. This textual representation is a predicate in PDL. These predicates are used in operator and state definitions.

Figure 3.9 represents the relation *on-Table* between a block and a table. The block and table are connected together at their connection points. This connection is designated by a line joining them. During the state and operator definitions, whenever a block and a table

are connected together in this fashion, the `On-Table` predicate will be automatically generated.

To the left of the work area are the object prototypes that were created in the Model Window. At the top of the window are two relations that have already been defined: `On` and `Holding`. The machine-generated PDL predicate (`on-table <block876> <table564>`) is in the window and can be directly edited by the user. The `Dep-List` in the window allows the user to define a dependency order among the objects that is used for the solution animation. The `Dep-List` will be discussed in section 3.2.2.

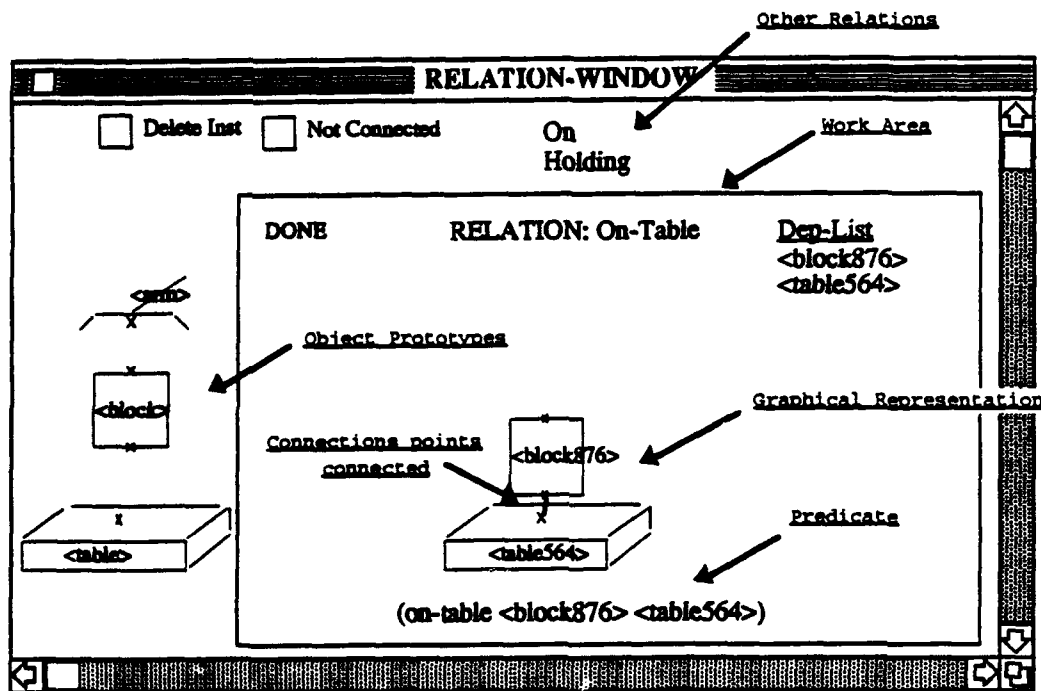


Figure 3.9: Relation Window: developing the On-Table relation.

The Relation Window also allows a user to define a negated connection existing between two objects (*i.e.*, arm and block). A connection is negated by selecting the connection point to be negated then clicking the Not Connected button. In Figure 3.10, when an arm is not holding a block, the empty relation holds. The square around the arm connection point signifies that the connection has been negated.

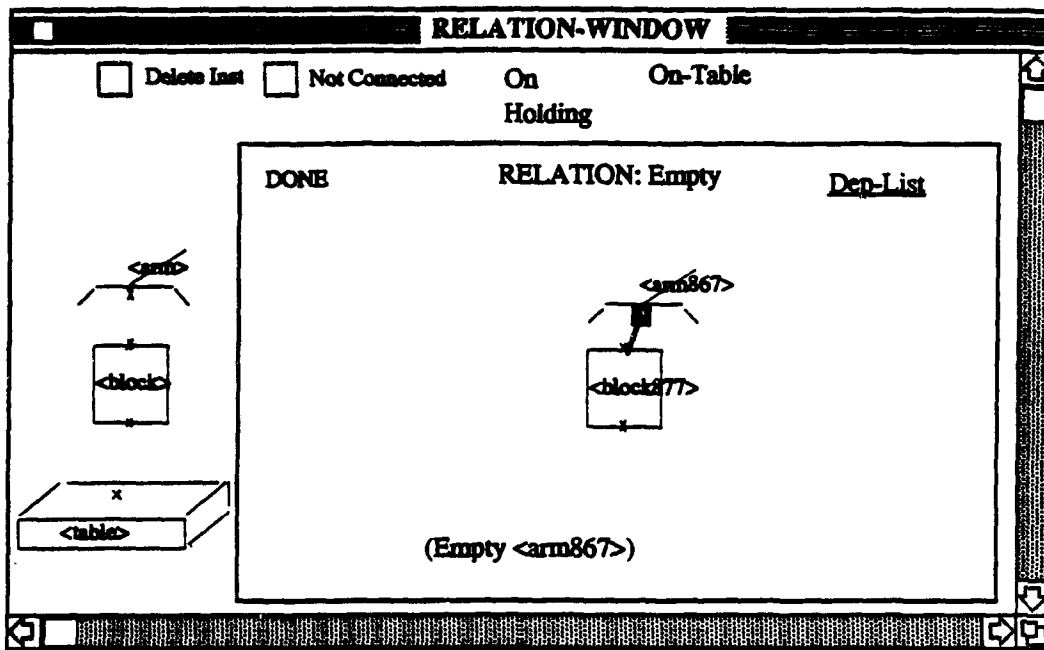


Figure 3.10: Relation Window: showing a negated connection.

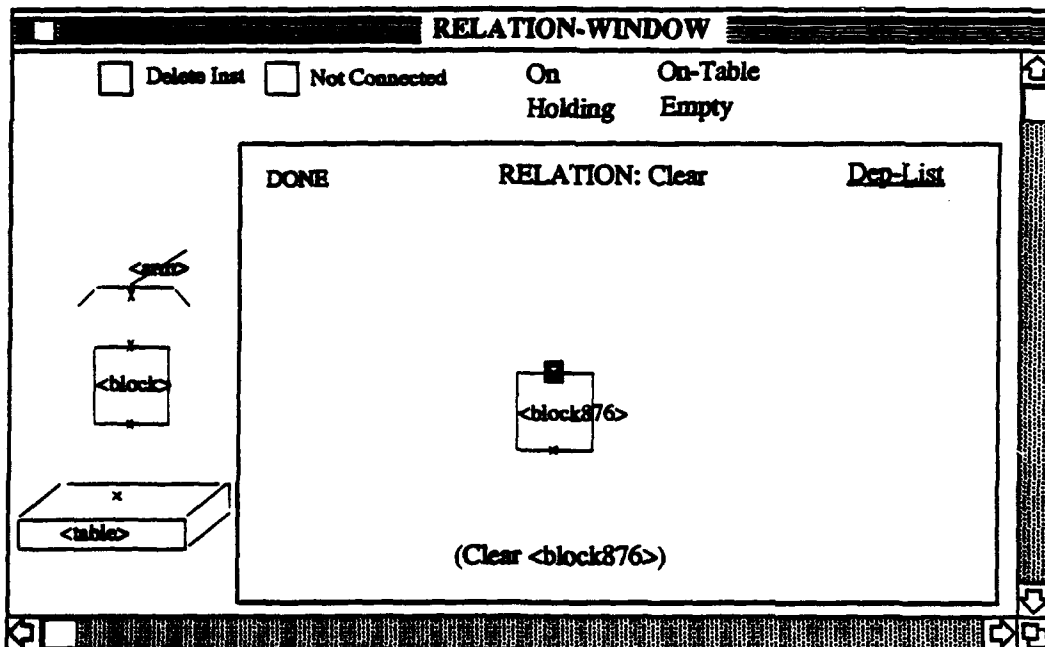


Figure 3.11: Relation Window: showing a non-connection definition.

Finally, there is a way to describe a relation when an object is not connected to any other object. This is done by selecting a connection point on an object, then clicking the Not Connected button. This signifies that when nothing is connected to that connection point, the relation holds. In Figure 3.11 the clear relation is shown. This relation is applicable

when there is nothing connected to the top of a block (*i.e.*, no block is on top of the block, and the block is not being held by an arm).

3.2.1.3 Operator Window

The Operator Window allows the graphical creation of operators or rules in the domain and automatically generates Prodigy operator code. In this window the work area is divided into two regions: before and after. When all the relations in the before region are true for a particular state, then the operator transforms the matched objects into the relations in the after region. Therefore, the before region defines the conditions that have to be true before the operator is applied, and the after region implicitly defines the transition between states when the operator is applied.

An operator is defined by building a picture of its before and after states. This creation process is similar to defining a relation. To create an operator, the relevant prototype objects are dragged into the before region of the work area and connected together to represent the before state. The user can then copy these objects to the after region by clicking the Copy Pre State button. This will also link the similar objects together. Thus, if the name of an object in the before state is changed, then the corresponding object's name in the after state is also changed, and vice versa. Objects can also be added directly to a region with the drag technique, but these objects are not linked with any other objects. To link two similar objects in the before and after region, the user should select the desired objects and click the Create Link button. Once the before and after states are defined, the system can automatically compute the preconditions and the effects for the operator in PDL. This computation is displayed so that the user can make modifications if needed. User modifications made to the generated code are recorded and used when the system automatically regenerates the operator due to graphical modifications the user makes.

In Figure 3.12 the operator Pickup is being defined. Similar to the relation window, the prototype objects are positioned around the work area. Note that names of objects in the operator have $\diamond$ around them. This represents a variable name. Figure 3.13 shows the generated code for the Pickup operator.

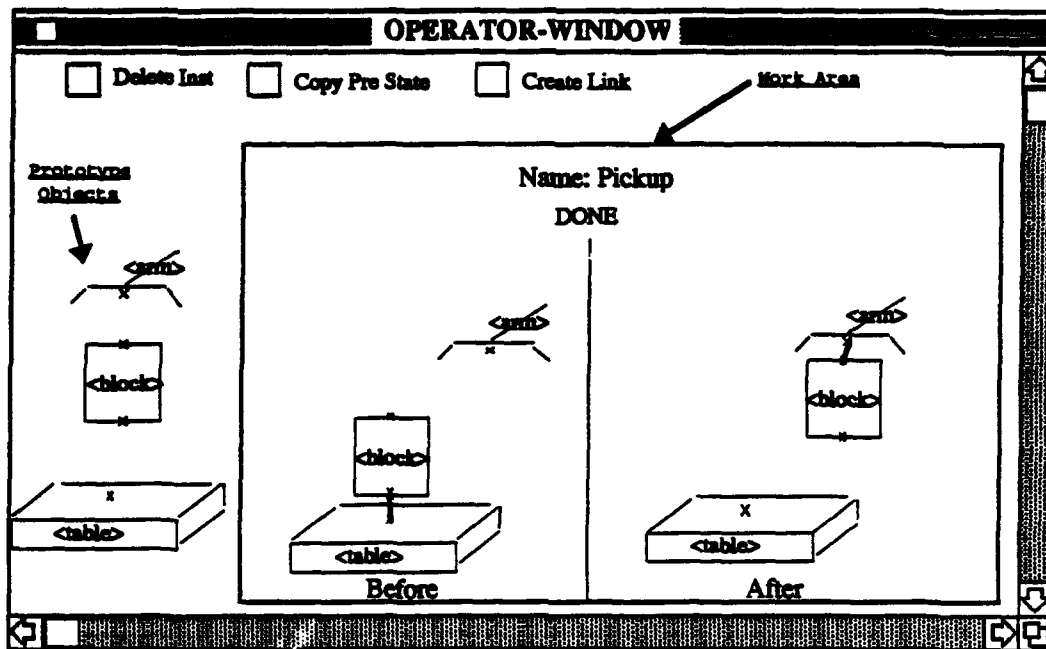


Figure 3.12: Pickup operator before code is generated.

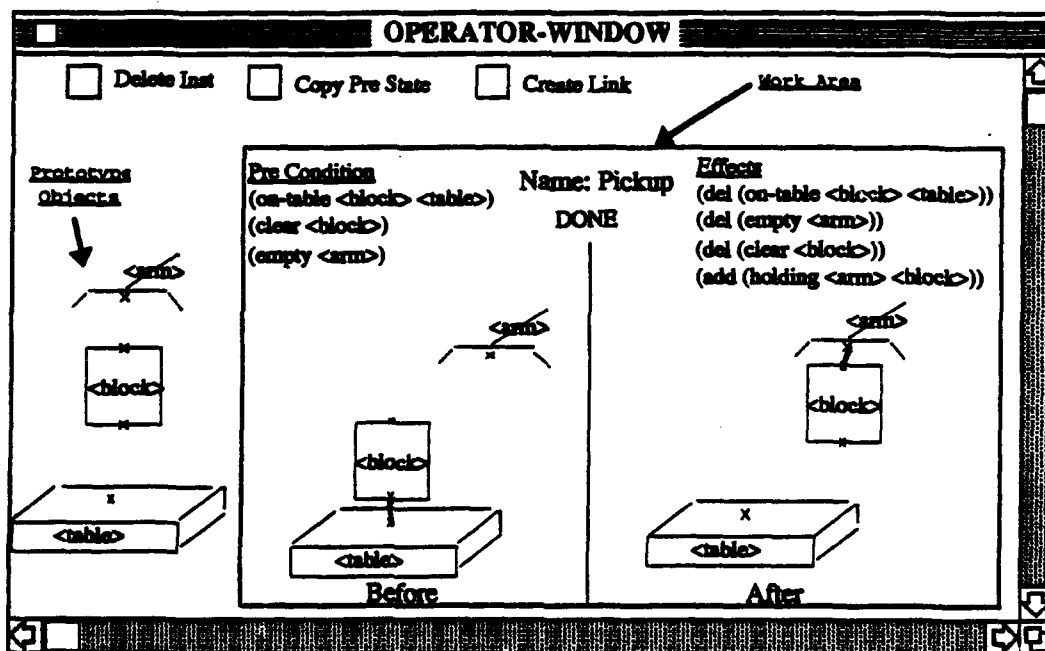


Figure 3.13: Operator Window: showing the Pickup operator with automatically generated Prodigy code.

3.2.1.4 State Window

The State Window allows the graphical description of a state and automatically generates the Prodigy state code to reflect the depiction. The initial state is a description of the state at

the beginning of planning. The definition procedure is similar to the other process. The state is defined by instances of objects and their connections. The user drags prototype objects into the work area and connects them together. From this visual representation the system generates an initial state code definition automatically. Figure 3.14 is an initial state that has three blocks on a table. The automatically generated state code is also part of the State Window.

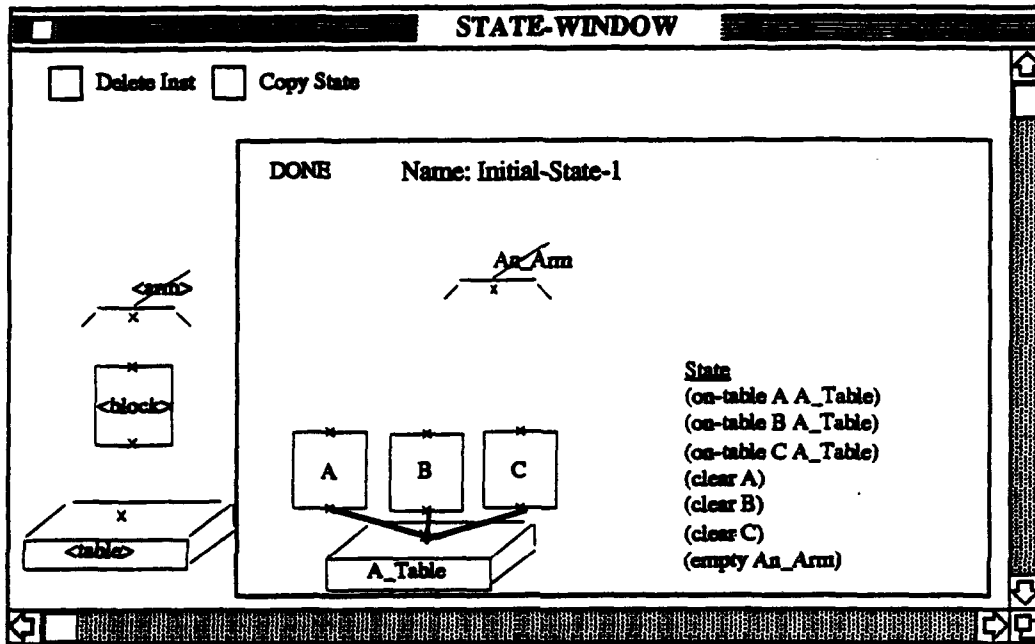


Figure 3.14: State Window: defining Initial-State-1.

The goal state is a little different from the initial state because the goal state can have a subset of the objects in the initial state. Figure 3.15 shows a goal state, again with the automatically generated state description. The state definition can also be directly edited by the user. Again, changes made to this definition are remembered and are applied when the system automatically regenerates the state description due to graphical changes.

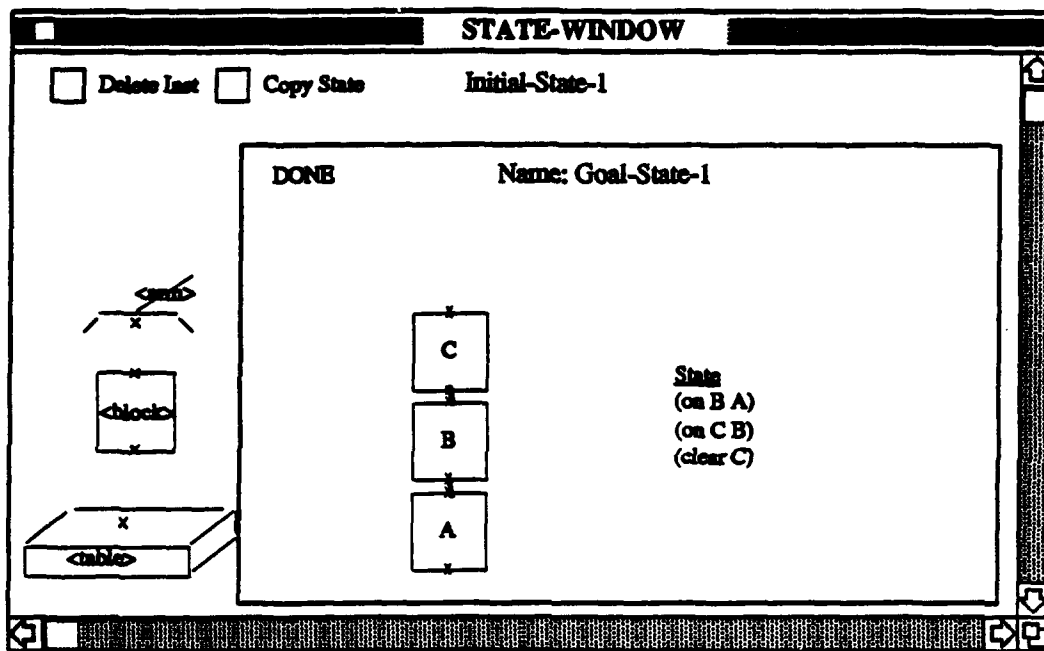


Figure 3.15: State Window: defining Goal-State-1.

3.2.1.5 Problem Window

In the Problem Window the user defines the problem that is to be solved and sees the *animation* as the system plans the solution. In order to solve a problem, Prodigy needs to be given: 1) the operators that are to be used, 2) the initial state for the problem and 3) the goal state for the problem. These are specified separately, providing flexible problem specification. For example, one may specify several problems with the same initial state but different goal states, or test only a subset of the available operators. Figure 3.16 shows the Problem Window based on the information we previously defined. By selecting the Execute Problem button, the current problem's code is loaded into Prodigy and executed. As the planner solves the problem, the solution is graphically animated, as shown in the next section.

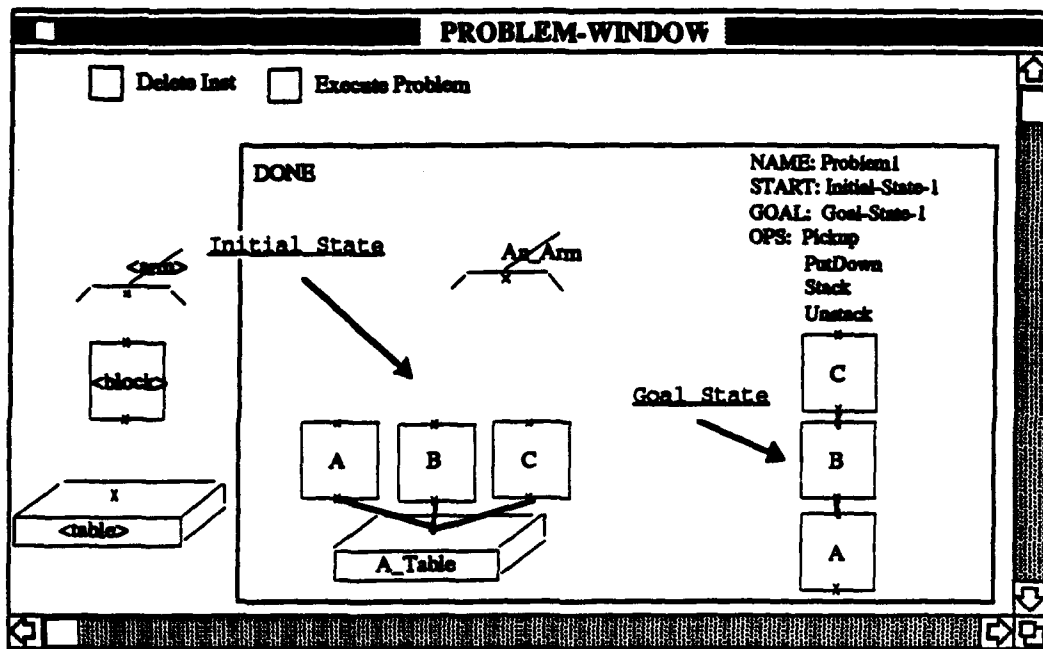


Figure 3.16: Problem Window: displaying the initial and goal state definitions.

3.2.1.6 Example Trace

Figure 3.17 is an example trace of the system solving the problem that we previously defined. The domain and problem definition are the same as the textual domain that we defined previously. The partial trace below shows the steps the system takes in order to achieve the goal state. The system starts out by stacking the blocks in the wrong order. Later on, the system backtracks and reorders the goals. The system then stacks the blocks in the correct order and solves the problem.

The solution for the defined problem is to:

```
(pickup arm b table)
(stack a b arm)
(pickup arm c table)
(stack b c arm)
```

Another debugging tool allows the user to apply instantiated operators one by one. When applying an operator, if preconditions are not satisfied in the particular state, a message is displayed containing the relations that were not matched in the precondition. This stepping process helps the user verify that an operator is performing as believed. Of course, all graphical animation is also updated when an operator is applied successfully.

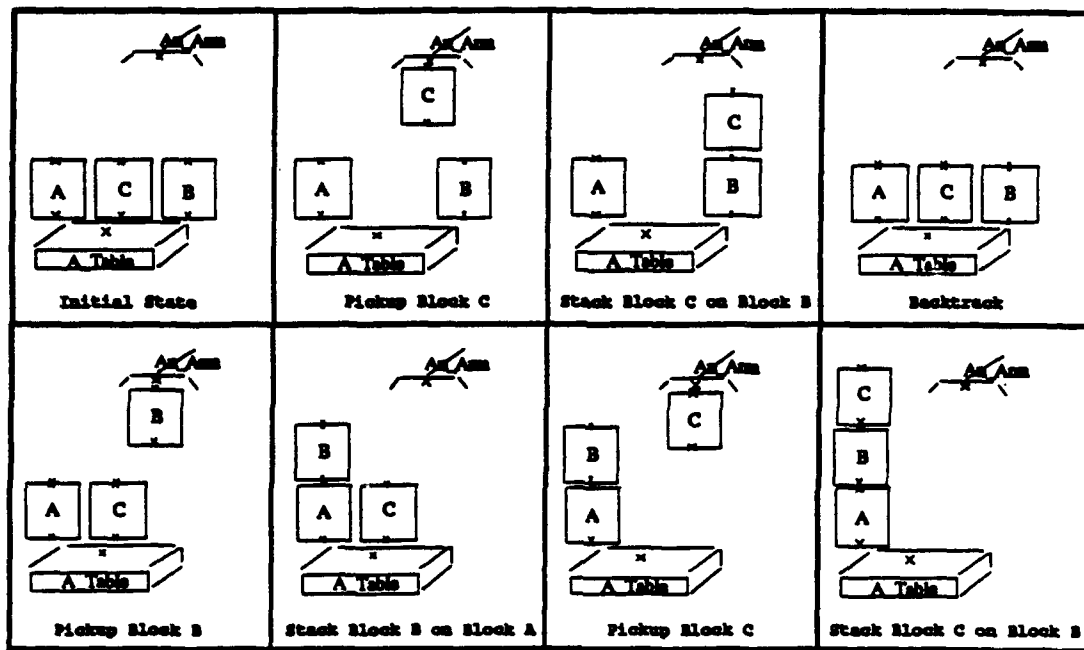


Figure 3.17: Visual animation showing blocks world problem being solved. These are several snapshot views of state changes with applied operator or backtrack command.

3.2.2 Animation Algorithm

The animation algorithm provides the ability to visually trace the progress of a solution during planning. This is done systematically, as shown by Figure 3.18. Once a problem is ready to be solved, then the animation routine begins. First the graphical object information is initialized; the graphical representation of each object is displayed in the problem window at a location based upon the initial state definition. A list of all the graphical objects with supporting information is compiled and stored for animation. The graphical dependencies and relative position for each object in all the relations are computed and stored. In the next step, the PDL definition for the domain is collected and prepared to be loaded into the Prodigy planner. Once the initialization is finished, the PDL code is loaded into the planner and planning begins. During the planning phase, the planner will update the graphical display as operators are applied. This update procedure will be explained later in more detail. If the planner backtracks, then the screen flashes and the new state being worked on is drawn.

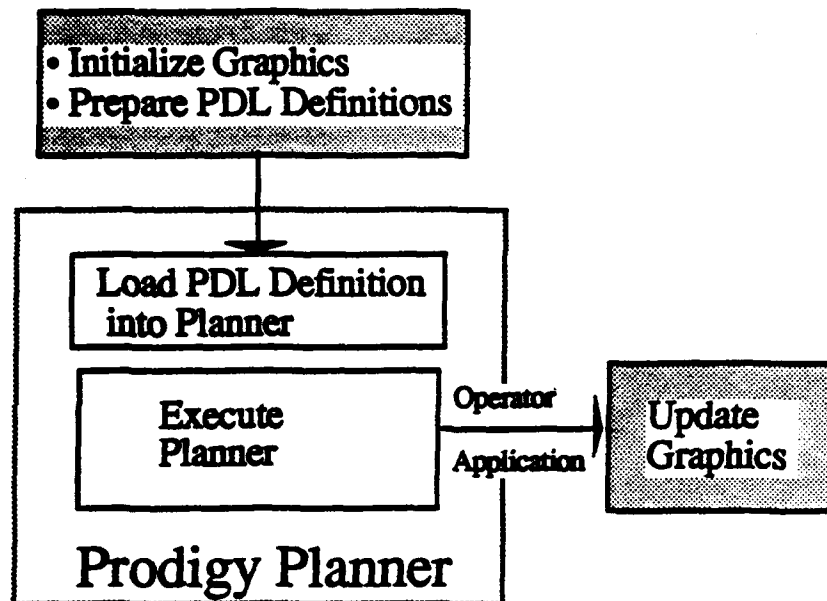


Figure 3.18: Animation diagram.

The key to this trace animation is getting the information about the object's interaction from the relation information. When an operator adds or deletes a set of relations in the current state, the relation information is used to update the graphical display. Therefore, the initial information needed from each relation is: 1) what are the object types in the relation; 2) what are the dependencies between objects; and 3) what are the relative distances between the objects. A short example illustrating the information needed and how it is used follows.

The objects in a relation are ordered according to the object's dependency with the other objects. This order is user-defined and represented in the Dep-List of the relation window. Figure 3.19 shows the relation window with the definition of the "on" relation. The bottom object in the Dep-List is the most stable object during movement. Thus in Figure 3.20 the top block is moved to a location relative to the bottom block. The graphical distance between the two objects is used when the objects are animated in the solution trace.

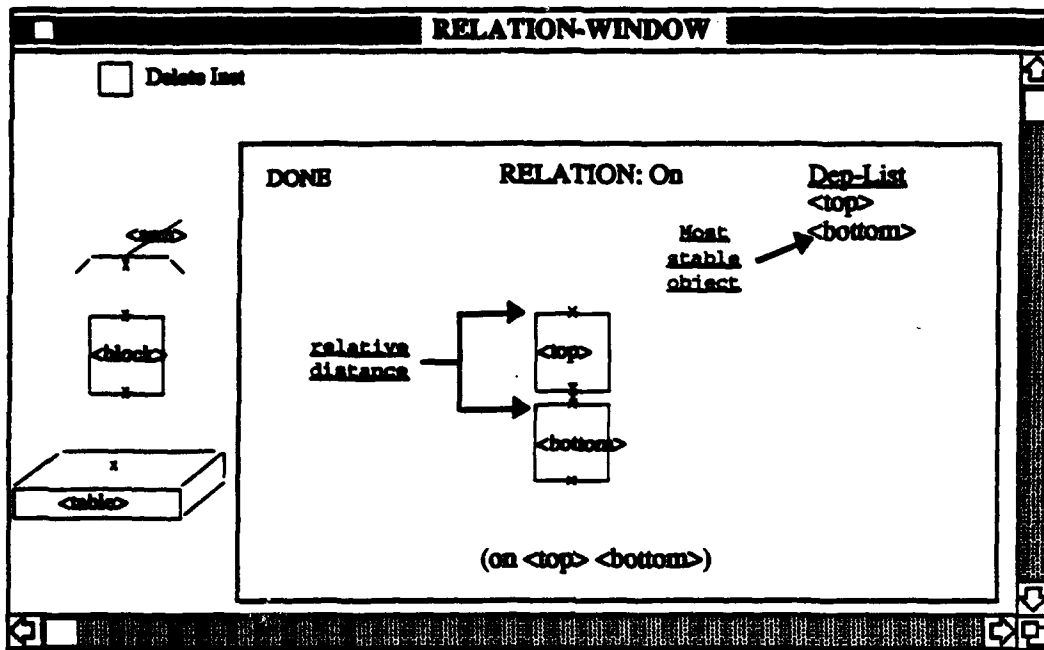


Figure 3.19: The Relation Window with the on relation

Figure 3.20 shows three snapshots of an animation trace. The first snapshot shows B right before an operator is applied to add (on B A). Snapshot two shows the result of that relation being added to the state. See how block B moved relative to the position of block A. Further in the solution trace, snapshot three shows the after effects of adding the (on C B) relation to the state. In this case block C moved relative to the position of block B.

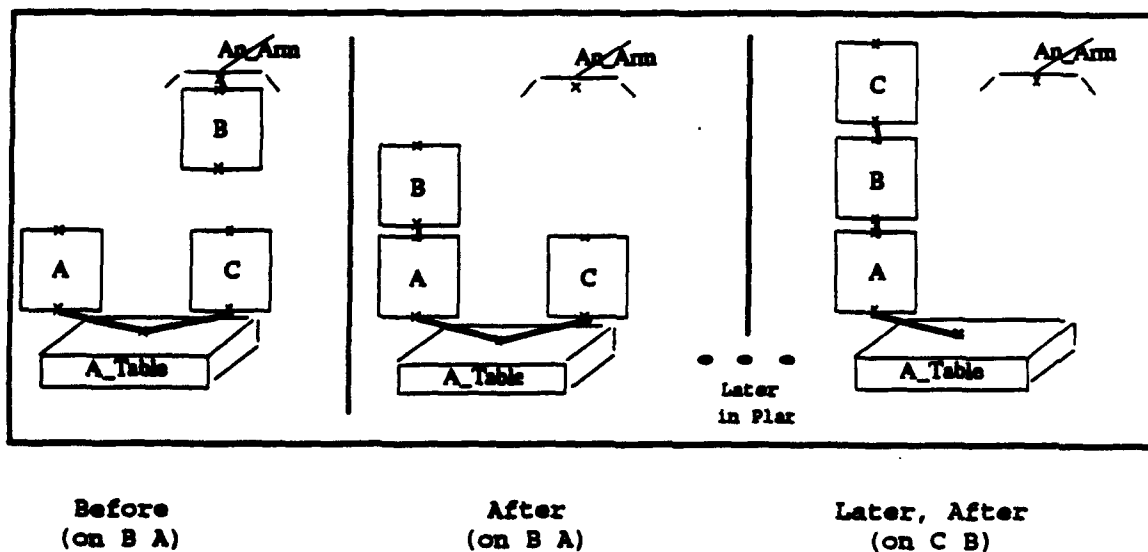


Figure 3.20: Adding the on relation to a state. The relative distance between the two objects in the relation is used to determine object placement.

The above description gives an overview of the animation process, but the movement of objects relative to each other is slightly more complicated. The previous dependencies of objects are also used to compute an object's new location. When an operator is applied, the internal state of the planner is updated and all the added and deleted relations are then used to generate the animation. Deleted relations remove object dependencies and erase objects on the screen; added relations create new dependencies among objects and move the objects to a new location. Below is a description and example of the relation effects.

For each deleted relation:

- Each object in the relation is erased from the screen.
- Each object's dependency, based upon the particular relationship, is deleted.

For example, deleting the (holding B An_Arm) relation erases the objects from the screen and breaks the dependencies of B and An_Arm.

For each added relation:

- New dependencies between objects are recorded based upon the relation. *For example, when the relation (on B A) is added, B is set to depend on A.*
- The order of object movement (i.e., move an object before another object) is set based upon the relation. *For example, in the previous on relation, A is moved before B.*
- A relative displacement function is computed for each object. *For example, the function (move-rel 0 -30 A B) is stored for the object B's move function relative to A.*
- A topological sort on the objects is done based upon its previous and present dependencies. *For example, in the states of Figure 3.20 sort all objects in the state (A, B, C, An_Arm, A_Table).*
- Move all the objects to their new locations in the order of the sort using the displacement function. This is done by applying the displacement functions to the objects in sorted order.
- Redraw all the objects in the domain.

The deletion of relations must be done before the addition of relations. Using the above algorithm in all the domains that were developed in APPRENTICE, the solution trace provided a realistic animation of the planner execution. In cases where there are circular

dependencies this animation may break down but this did not become a factor in the domains developed thus far.

3.2.3 Primitive Elements for the Domain Builder

APPRENTICE is just one implementation of domain developing techniques that aid in knowledge acquisition. This section removes the implementation details and describes the underlying set of primitives that are needed to develop a domain using the APPRENTICE technique. Understanding these primitives will make the incorporation of newly developed interface techniques easier. These primitives support the development of domain definitions that have objects, operators, states, and instances. The primitives are defined below, followed by examples from both the blocks world and a business organization chart domain.

Object Creation - The ability to define a visual representation of the domain objects. *For example, visually represent a block or a business division.*

Object Variable Definitions - The ability to refer to a class of objects. This is used in the definition of relations and operators. *For example, be able to make reference to any block or business division.*

Object Connection Points - The ability to define a set of physical locations, either explicitly or implicitly, on each object and allow the objects to connect with one another. This will form relations between the objects. *For example, form connections at the top and bottom of a block or connections at the top and bottom of a division icon.*

Object Attributes - The ability to associate attributes with individual objects. *For example, the weight of a block, the number of workers in a division.*

Relation Definition - The ability to define relationships between objects types or instances by connecting their connection points. *For example, block on another block, subdivision under a division.*

Relation Graphical Relative Position - The ability to define the graphical relative position of objects. This is important during animation to give a realistic portrayal of the solution. *For example, a block positioned on top of another block, a subdivision positioned underneath a division.*

Relation Dependency - The ability during the graphical trace for objects to be drawn relative to one another. The ability to define a dependency list defining the drawing precedence of objects is also important. This can be implicitly or explicitly controlled by the user. *For example, use the location of the bottom block to get the relative position of the top block, use the division location to get the position of its subdivision.*

Composition Element Creation - The ability to create a static description or state using a set of relations. *For example, block A on block B and block B on a table, a business organizational chart.*

Transition Element Creation - The ability to use the differences between composition elements to describe a state transition. *For example, pickup a block, reassigning a subdivision.*

Solution Tracing - The ability to inspect the problem solution as operators are applied. *For example, animate the robot arm moving or animate a subdivision changing the division it reports to during a reorganization.*

The above primitives can be combined together to create a system that allows users to develop domains faster and more accurately (see Chapter 4). The development paradigm is based on creating domain objects, relations, operators, and states. Below are the definitions of these elements.

Prototype and Instance Definition - A prototype is a class of physical objects defined. These prototypes are graphically defined, have ways of expressing connection with other physical objects, and can represent attributes about themselves. An instance is an actual physical object copied from the prototype definitions.

Relations Definition - Relations are definitions of how physical objects connect with one another. The definition should include the objects forming the relation, the physical orientation between the objects, and the interconnected dependency between objects. The latter two criteria are used for the animation of the domain.

State Definition - States are a set of objects and their relations with each other. States are defined by creating composition elements previously discussed.

Operator Definition - Operators are sets of objects and relations defining when the operator is applicable to a situation and a set of effects that occur when an operator has been applied. Therefore, the operator is a combination of the composition element for the before part and a transition element describing the difference between the before part and an after state for the effects part.

Solutions Animation - Problem solving can be graphically traced as the problem is being solved, providing visual feedback from the problem solver by using the relation definitions. An algorithm for dynamically tracing a solution was given in section 3.2.2.

Using APPRENTICE as the model, the domain elements described above are shown to be an effective model for doing knowledge acquisition of visual domains. It has also been demonstrated that these elements can be translated into code that a planner can use to solve a problem in the domain. With these primitives, other user interfaces can be built to be effective in providing users with an intuitive system for KA.

3.3 Framegraphics - Low Level Graphics

A flexible graphical tool was needed to build an interactive system like APPRENTICE. I developed a frame-based graphical system called Framegraphics for the development of APPRENTICE. Because of its ease of use, flexibility, and extensibility, Framegraphics has also been used as the interface for building a piano tutoring system [Joseph 91] and an ontological graphical editor [Nirenburg 88].

This object-oriented graphical toolkit was built using Framekit 2.0 [Nyberg 88], a frame-based knowledge representation language. Framegraphics provides a machine-independent graphical system for rapidly prototyping user interfaces in a Common Lisp environment. It is currently running on the Macintosh under Allegro 1.2, on the IBM RT/PC under CMULisp and X11, and on the Sparc workstation under Allegro and X11. Figure 3.21 shows a block diagram of Framegraphics organization.

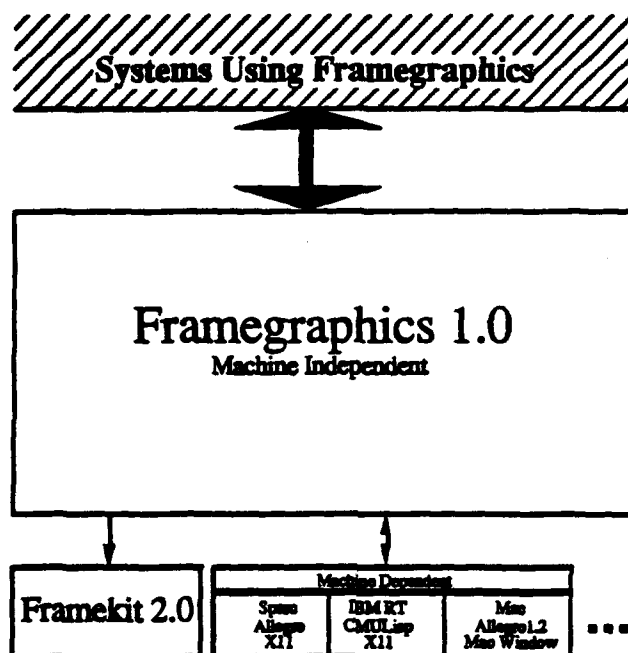


Figure 3.21: Framegraphics diagram.

In Framegraphics, graphical objects are represented by frames and stored in a is-a/instance-of hierarchy. The parent frame of all graphical objects is the frame "graphic-object," but only instance-of frames within the graphic-object hierarchy are displayable objects. The graphic-object frame stores the default behavior for graphical objects. This default behavior is in the form of functions, demons, and values stored in slots. This means that a graphical object can be created with very little work, yet have default behavior. This default behavior also helps users quickly learn how to use the application system by providing a consistent interface.

The system's default behavior is modelled after the Macintosh interface. Objects can be selected by a mouse click, they can be moved by click and drag, multiple selected objects can be moved, etc.

Most of this default behavior is implemented as function names stored as slot values. To run a particular function, first the slot value is retrieved using inheritance. Then the value is executed with the current frame information as its parameters. The `move-proc` slot is an example of this. When a screen item is moved, the `move-proc` slot is queried. If a value is not found locally, which is usually the case, the inheritance mechanism returns a value from the hierarchy. This value is then used as a function, and the arguments of the local frame are the parameters. This is a very powerful method to share code among objects.

This default behavior can also be easily modified or extended for an individual object or a class of objects. In APPRENTICE it was extremely easy to add a help facility. It took only an hour to develop a functional help system so that objects on the screen could provide information about themselves. This was accomplished by adding a `help-proc` slot with a help function to the high level "apprentice-graphic-object" frame, the top display frame for the apprentice interface. This function took the local frame name and retrieved information from its `help-msg` slot. These strings were then displayed along with the frame name. By adding a textual description to the selected `help-msg` slot, help information could be tailored for individual screen objects or classes of objects. Currently the help system is activated by an alt-click selection on a particular graphical object. Again, this activation was caused by adding a function to the `alt-click-proc` slot that called the `help-proc` slot function of a frame. This activation could be easily changed to work another way. For example, help could be activated by clicking a button that displays information about the selected frames. Also note that the help function can be made different for each object just by changing the `help-proc` slot function for the different frames.

In the same manner, I was able to add in a day the ability to generate a Postscript file from a display screen. This was developed after most of the APPRENTICE system was already built. I was able to modify the whole system by adding a function slot `ps-proc`. In this slot each frame or class of frames would have a function that wrote to a file the Postscript commands for displaying the object. This function was much like the drawing routine for the objects. To create a Postscript file of a window, all that was needed was to loop through the display list and have each frame run its `ps-proc` function sending the output to a file.

As with any object-oriented system, extending object types is a simple matter. The editor for the model window was developed and integrated into the system in about a day. It is selectable, moveable, and responds to help messages consistent with other objects, yet it

also allows the building of graphical objects. Another new object type developed was an object that displays graphs and allows interaction (selection, movement, and click procedures) with the nodes of that graph. The development and integration of these objects was effortless and straightforward.

Chapter 4 - Empirical Analysis: User Studies

The APPRENTICE system was designed as a knowledge acquisition tool to facilitate the construction of Prodigy domains. To be useful, the system needed to be usable by a wide range of people and provide more functionality and understandability than other systems. Four studies were done with APPRENTICE to evaluate its performance with multiple users and multiple domains, as well as the evolution of the system use over time. The four studies are comprised of a study of the system's coverage and usability, a study that compares APPRENTICE and Emacs, a study of system usage over time, and a study to build a medium size domain. This chapter will describe each of these studies in detail.

4.1 Study 1: Coverage and Usability

To test the coverage and usability of APPRENTICE, 32 students from an advanced AI class participated in developing their own domains using APPRENTICE. The objective of this study was to:

- *Have multiple subjects use APPRENTICE.* This provided insight on how users interacted with the system. It helped to determine what is easy and what is difficult for others to understand. It also showed if individual conceptual models can be incorporated into domain building in APPRENTICE.
- *Get different types of domains built using APPRENTICE.* This gave an indication of what types of domains can be built using this tool. It also investigated whether the domain building philosophy was sound.
- *Determine what additional functionality is needed.* This allowed me to more fully debug and enhance the system, because other users invariably tried things that I had not considered. It also pointed out additional functionality needed.
- *Measure ease of building a working domain from a concept using APPRENTICE.* This allowed me to quantify the time it takes a person unfamiliar with APPRENTICE to use the system productively.

To reach the above objectives, each subject built in APPRENTICE a domain that they specified. The time to build the domain and the final domain produced was recorded. What follows is a description of the building process, examples of two different subjects' domains, and the measured results and analysis for all subjects.

4.1.1. Study 1: Hypothesis

The majority of users in this study should be able to use APPRENTICE successfully to build new domains.

4.1.2 Study 1: Procedure

The participants were students enrolled in the class *Artificial Intelligence: Representation and Problem Solving*. The class description from the University catalogue is as follows:

Artificial Intelligence: Representation and Problem Solving

Intelligent computer programs can solve problems, understand natural language, reason about their actions, and learn from experience. They do these things by manipulating internal symbolic representation. The course will cover the main types of symbolic knowledge representation and the main techniques for planning and problem solving. LISP, a computer language designed for symbolic programming, will be taught during the course, and there will be a required programming project in LISP.

Each student was required to develop a simple domain on paper using a STRIPS type syntax [Fikes 71] as part of their normal course work. The students knew nothing about APPRENTICE while they were developing these domains. During the class after the students finished the above assignment, I gave a forty-five minute lecture on APPRENTICE. As an optional assignment, the students were asked to encode their domain in APPRENTICE. Thirty-two people volunteered to do the assignment. Each student individually did the following procedure:

Preliminary Setup

- 1) The student was given a three-page description of APPRENTICE to read. This description is listed in the Appendix D - Apprentice Description.
- 2) I gave a short introduction describing domain elements (objects, relations, and operators).
- 3) The student wrote down the objects, relations, and operators in their domain.
- 4) The student was given a brief demonstration of how to build a simple domain (the blocks world) in APPRENTICE.

This entire preliminary setup took approximately 40 minutes.

Test Procedure

5) The students independently built their domains using APPRENTICE.

During the study, the students never referenced their previous paper-based assignment.

With only two exceptions, all students were able to build their domains in APPRENTICE. One student (subject 3) needed an additional function for his domain that the system did not support at the time; another student's paper domain (subject 4) contained no search, so he built another domain.

The students' domains fall into several categories: recipes (making carrot cake, Kool-Aid, etc.), transportation (moving blocks outside the house, delivering pizza, etc.), games (eight puzzle), controlling an appliance (playing a CD player, washing clothes, etc.), and Biology (building DNA molecules). The students' domains ranged in complexity from the eight puzzle domain (containing 2 objects, 3 relations, and 1 operator) to a domain to compose a sub-part of a DNA molecule (containing 10 objects, 14 relations, and 11 operators). The development time for the domains ranged from 1 hour to 4.5 hours. The mean time for domain completion was 2 hours and 4 minutes. For a description of the entire subject population and domains built, see Appendix A.

4.1.3 Study 1: Results

The next section describes two representative domains (the eight puzzle and the DNA molecule domains) along with the students' visual representations and the system-generated code.

4.1.3.1 Eight Puzzle Domain

The eight puzzle domain consists of squares adjacent to one another and numbered tiles that sit on top of the squares. This was the simplest domain built as part of study 1. The playing board is composed of 3x3 squares and a set of tiles on all but one of the squares (there is always a square without a tile on it). The tiles have unique identifiers (e.g. numbers, letters, etc.). A tile can be moved to the empty square if the tiled square is adjacent to the empty square. The tiles start out in one configuration; the goal is to get to a final configuration through a series of tile moves. Figure 4.1 illustrates the eight puzzle game.

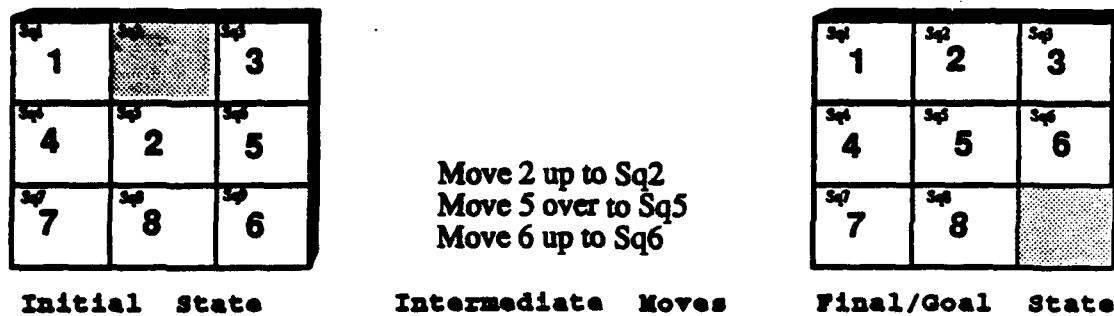


Figure 4.1: Illustration of eight puzzle game: Initial and final state along with intermediate moves.

Subject 8 encoded the eight puzzle domain in APPRENTICE as follows. The subject first built two objects (a tile and a square) with connection points. The tile and the square are pictured in Figure 4.2.

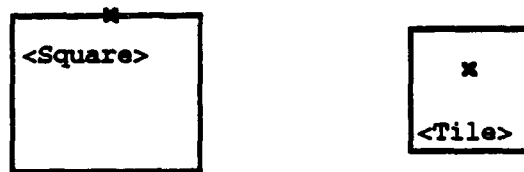


Figure 4.2: Square and tile objects for the eight puzzle domain.

The next step was to build relations. There are three relations. They are "adjacent," one square connected to another; "on," a square with a tile on it; and "square-empty," a square without a tile on it. The three relations are shown in Figure 4.3. Note: the box on the square connection point in the square-empty relation means that the relationship is true if the connection does not exist (there is no connection between the square and any tile).

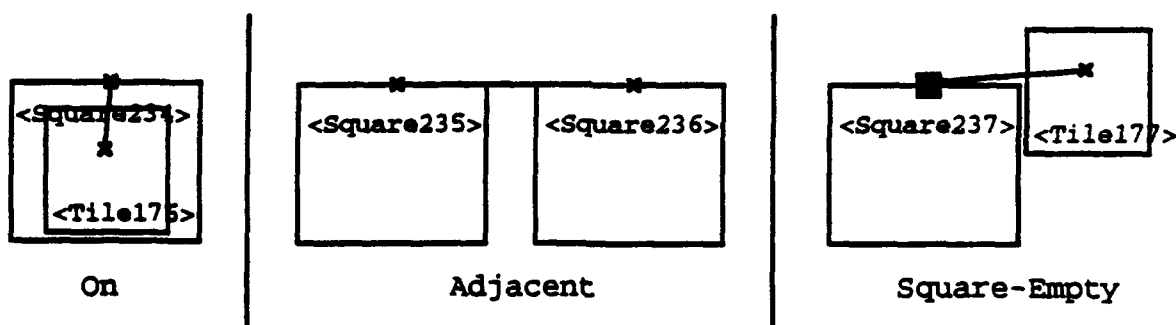


Figure 4.3: Relations for the eight puzzle domain.

The only operator is to move a tile. This operator is pictured in Figure 4.4. The automatically generated code that the system produced is in Figure 4.5. The MOVE

operator requires that there exist two squares that are adjacent to one another, and a tile is on one square, and the adjacent square is empty. When the operator fires, the tile moves from the initially tiled square to the other square, leaving the previously tiled square empty.

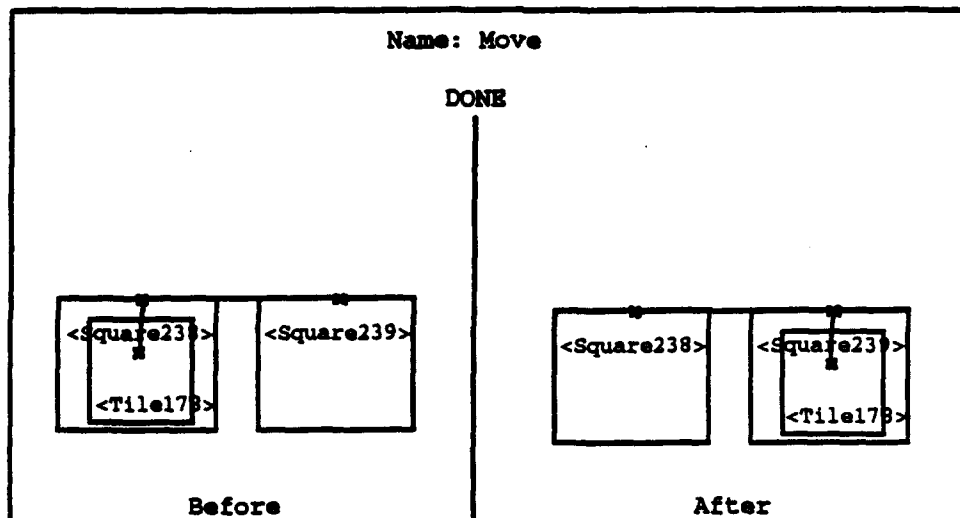


Figure 4.4: APPRENTICE definition of the move operator for the eight puzzle domain.

```
(Operator MOVE
  (prams ((<Square238> SQUARE)
          (<Square239> SQUARE)
          (<Tile178> TILE)))
  (preconds
    (and (On <Tile178> <Square238>)
          (Square-empty <Square239>)
          (Adjacent <Square238> <Square239>)
          (Adjacent <Square239> <Square238>)))
  (effects ((del (On <Tile178> <Square238>))
            (del (Square-empty <Square239>))
            (add (On <Tile178> <Square239>))
            (add (Square-empty <Square238>)))))
```

Figure 4.5: Automatically generated Prodigy code from the graphically defined MOVE operator.

Initial and goal states were also built. For brevity of explanation, a 2x2 puzzle will be used in the following description. Figure 4.6 shows an initial state that the user defined using a 2x2 puzzle (this problem is also known as the three-puzzle).

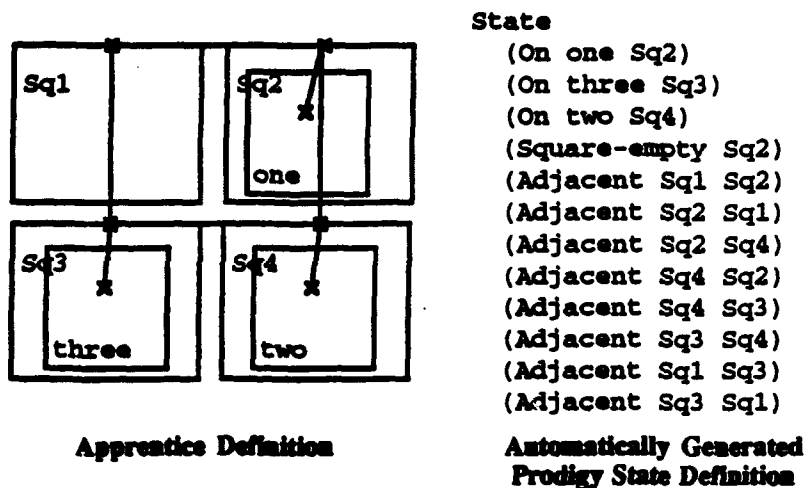


Figure 4.6: An initial state for the three-puzzle in APPRENTICE and in Prodigy.

A goal state for the three-puzzle is represented in Figure 4.7.

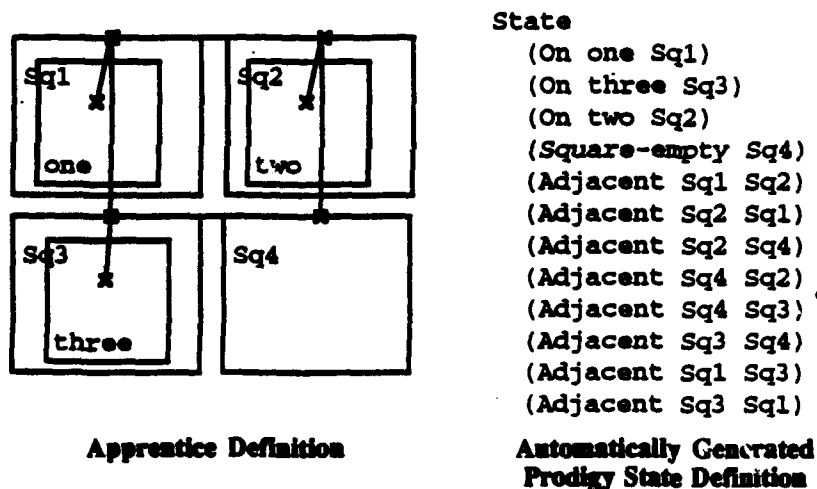


Figure 4.7: A goal state for the three-puzzle in APPRENTICE and in Prodigy.

By specifying the initial state, goal state, and operator(s), the problem can now be run with animation. Figure 4.8 shows a set of snapshots along the solution path. Since this was an easy problem, the solution was found right away.

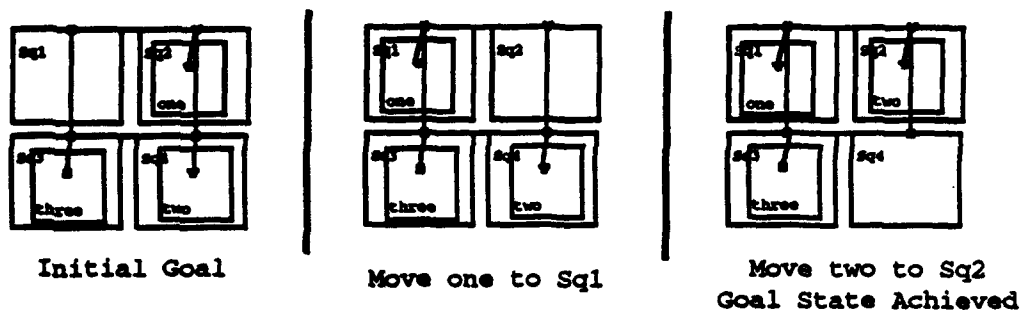


Figure 4.8: Steps for solving the 3-puzzle.

4.1.3.2 DNA Molecule Domain

Another domain that a student built was a procedure for building sub-parts of a DNA molecule with organic building blocks. This is one of the most complex domains build as a part of study 1. This domain is a student's view of a very complex process and should not be viewed as a valid description of a genetic process.

DNA is a double helical, long molecule structure composed of several building blocks. The connections between the two parallel sides consist of pairs of organic base elements: adenine, thymine, guanine, and cytosine. Adenine can only pair with thymine, and guanine can only pair with cytosine. This domain deals with the construction of these nucleotide pairs.

The basic elements in this domain are the organic building blocks (cytosine, guanine, thymine, adenine), a chemical mediator (enzyme), a transport element (ribosome), and nucleotide pairs (a-t-pairs and g-c-pairs). These objects are shown below.

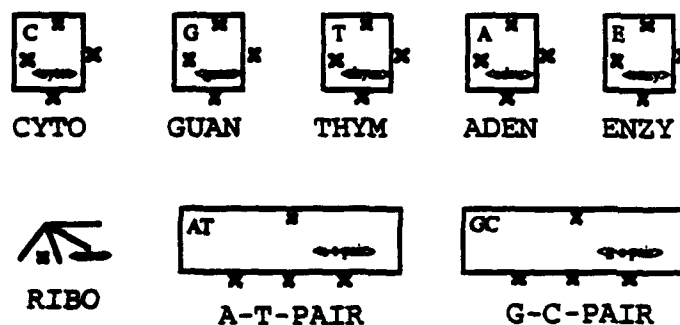


Figure 4.9: Objects in the DNA molecule domain.

The relations in this domain are depicted below and represent the interaction between the domain objects. These relationships are between the organic building blocks, the ribosome and the organic building blocks, the enzyme and the organic building blocks, and the composition elements of the nucleotide pairs.

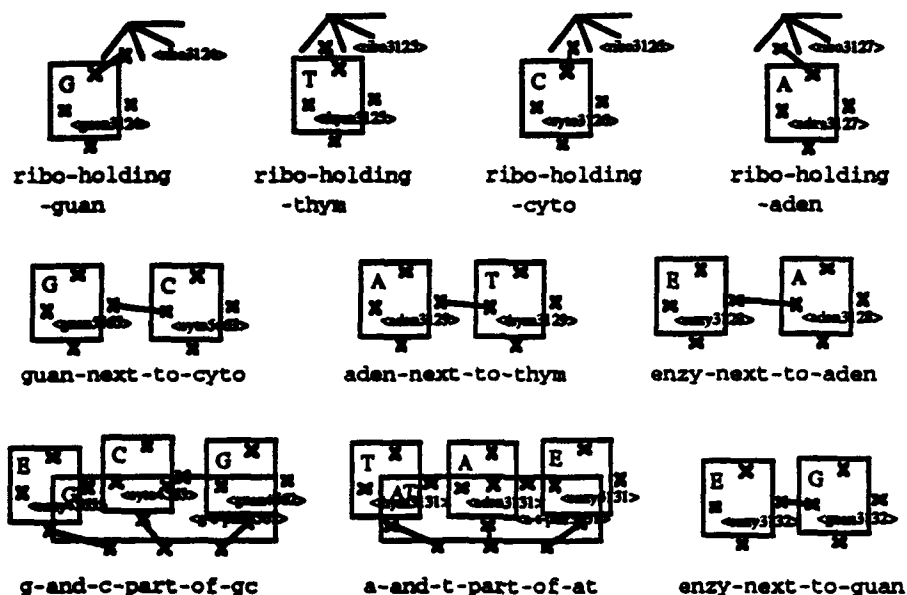


Figure 4.10: Relations in the DNA molecule domain.

Finally, the operators consists of the procedure to connect the organic building blocks into the appropriate nucleotide pair. This consists of using the ribosome to transport the building blocks to the correct location and allow the elements to be connected together. Below are the visual representations of the operators. The generated code will be included in Appendix B.

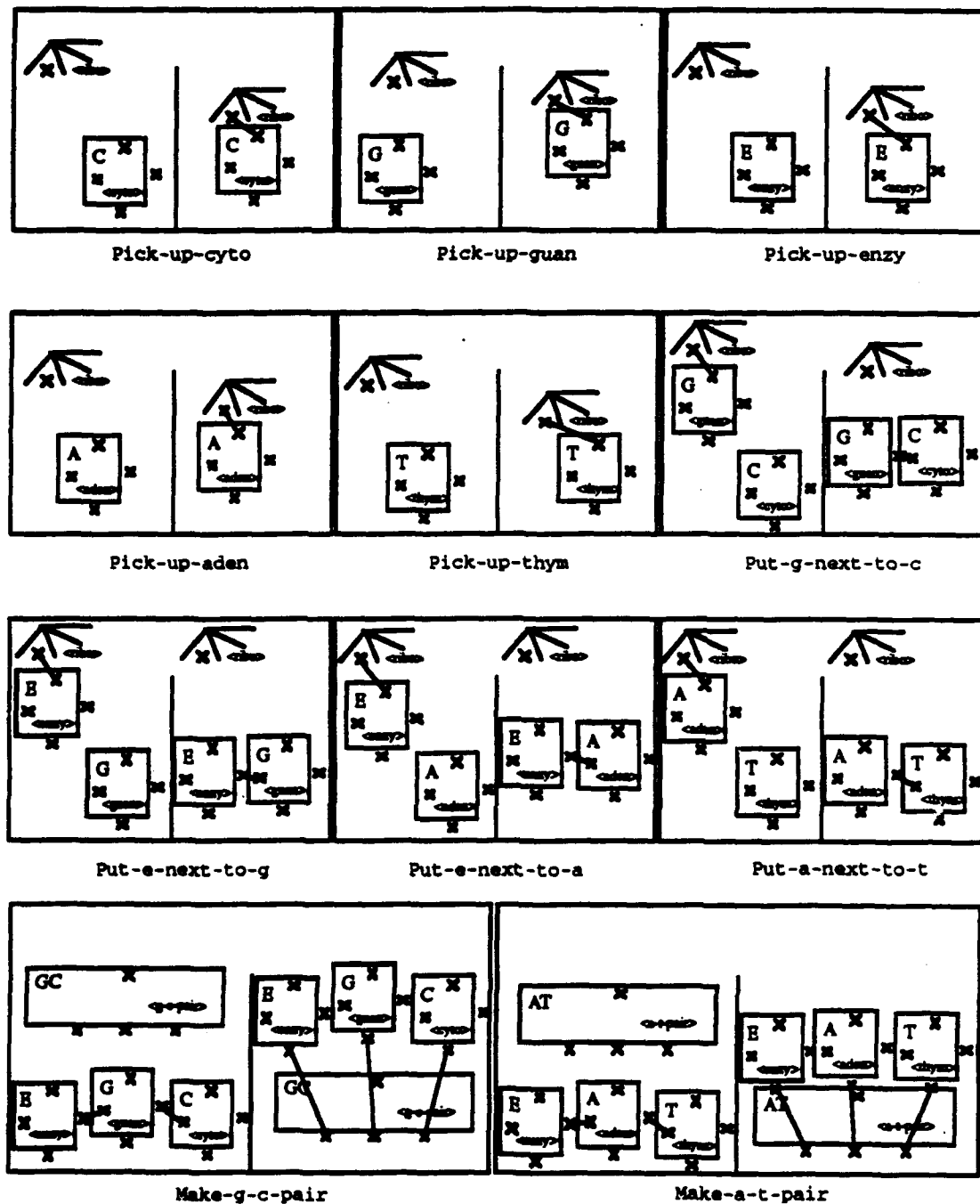


Figure 4.11: Operators in the DNA molecule domain.

Finally the initial and goal state of a problem is shown. The system then figures out the sequence of operators to apply to go from this initial state to the goal state.

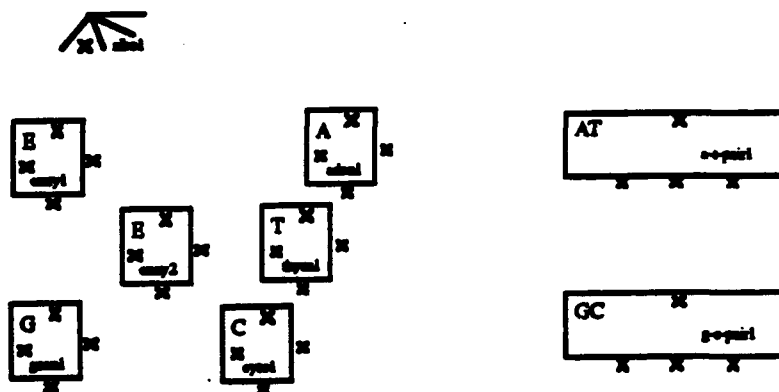


Figure 4.12: Initial problem state in the DNA molecule domain

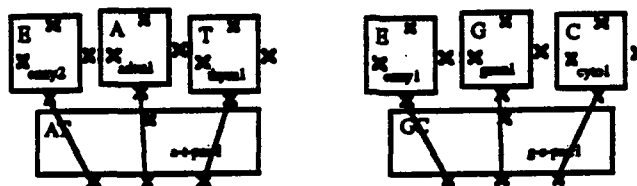


Figure 4.13: Goal problem state in the DNA molecule domain.

The solution for the above problem is:

```

pick-up-guan guan1 ribol
put-g-next-to-c guan1 ribol cyto1
pick-up-enz1 ribol enzy1
put-e-next-to-g ribol guan1 enzy1
make-g-c-pair g-c-pair1 cyto1 guan1 enzy1
pick-up-aden ribol aden1
put-a-next-to-t aden1 thym1 ribol
pick-up-enz2 ribol enzy2
put-e-next-to-a ribol enzy2 aden1
make-a-t-pair thym1 aden1 enzy2 a-t-pair1 thym1

```

4.1.3.3 Other Results

After students finished their domains, they were asked to complete a questionnaire. Their overall impressions were very favorable, and their comments helped me to improve the system. On the following page is a representative sample of these comments.

Excerpt of Quotes from Subjects for Study 1

I had a need for multiple subclasses but there was no easy way to do this.

The ability to specify objects being a subclass of another object is now available.*

Such a system seems necessary in order to deliver the power of planning engines to computer-laypeople.

Robot Taking Customer's Order - 1

The interactive nature of the Apprentice system made the development of the domain and the solution generation of the problems fun.

Bicycling - 8

For all the praise I offered in the above paragraph, I must now urge you to make the user interface even easier, and more conforming to various standards that have evolved in U.I. design. For example, a coherent and consistent way to navigate the program's states is needed. Sometimes, I would have to click on a word to return, and others I would click on a button. I really do feel like I am being nitpicky, however, because I was very pleased with all the less trivial aspects of Apprentice, in particular how it represents the domain with objects, operators and states. More consistency is added to how Apprentice works.*

I am skeptical about the usefulness of Apprentice (or even a more complex system of its nature) for a more complicated domain than the sort we have been looking at.

This problem is dealt with in the medium size domain.*

Robot Fetching - 10

I am not sure about the scalability issue — I think one of the bigger problems would be the graphical display of a lot of data and some automated methods for keeping the pictures cohesive and understandable.

This problem is dealt with in the medium size domain.*

+ Add the ability to animate the plan solution.

+ Have ability to save other plan solution

+ Have the intermediate steps displayed in an out put

Passing Blocks - 13

Providing for multiple objects of the same class, but with different attributes, or anything involving attributes was not as obvious. It needs to provide an easier method for dealing with attributes, perhaps labeling objects in relations and operators with the attributes.

The ability to handle attributes better will either be discussed or improved.*

Playing CD Player - 20

The graphical interface is very straightforward to use and simply requires a brief session of just playing around with the system to get used to it.

People Showering - 2

Certain functionality of the system and its use was underspecified in the system's help feature and in the hardcopy documentation.

A more thorough manual has been written and the help system has been updated.*

Killing Roach - 7

Actually drawing the pictures and making connections between them was easier than defining operators, states and assertions on paper.

Cat Getting Fish - 14

I built the domain I intended. I even ended up building more than I intended.

Moveworld with Blocks - 6

There was too much overlapping of graphics and text. It was at times difficult to tell what was on the screen. The fact that all user-defined objects and states automatically went to the same screen location (first position on the left side) was confusing. The arbitrariness of the location of various pieces of information was noticeable and added to the confusion of a novice.

This is a confusing point but is just a programming fix. It was useful enough for an experimental system.*

I think that the state information (which was not printed in novice mode) is quite helpful. However, once again, the screen clutter is a problem. Overall, I found it to be a great visual method for entering my domain, and I agree this is a superior method for knowledge acquisition.

Playing CD Player - 12

I built the domain I started out with although I added a few more operators and states to make it more complicated. It was generally easy to use, and it was kind of fun. I liked the fact that we had to draw our operators and states. I think this makes it more interesting and easier to follow.

Fixing Kool-Aid - 17

Learning whether to right click, left click, etc. at each point took some getting used to, but after time became easy to use.

Feeding the Cat - 21

* The Bold Text is my comments.

4.1.4 Study 1: Analysis

APPRENTICE was used successfully by a varied set of people, over several types of domains, in a relatively short time. All domains that the students tried to build in APPRENTICE successfully lead to a running system. The students were diverse, ranging from college sophomores to first year graduate students, and majors ranging from Computer Science to Music. The system supported the development of several types of domains like solving the eight puzzle, building a DNA molecule, cooking a carrot cake, and delivering pizza.

The students also helped debug and enhance the system. Some developments that resulted from this study were better consistency checking, increased functionality with class specification and hierarchy of objects, and increased system-directed help and user warnings. Finally, an intangible result was that the students enjoyed using the system. They found it "easy and fun."

A closer analysis of the study 1 data provides more characterization of the domain development process. The average time for domain development was two hours and four min. In Figure 4.14 statistics from the data are shown. Figure 4.15 is a plot of the subjects versus their development time. It also shows the development time median (120), quartiles (100, 140) and whiskers (60, 200). Notice that in the chart there are three subject development times that seemed to be inconsistent with the rest. A light gray box surrounds these times. Analysis was done on the data with and without these subjects.

<u>32 Subjects</u>	<u>29 Subjects</u>	
Median Time	120.0	120.0
Mean Time	124.4	114.5
Standard Deviation Time	40.7	24.9
Mean Objects	5.7	5.9
Mean Relations	8.8	8.4
Mean Operators	6.1	6.0

Figure 4.14: Statistics for 32 and 29 subjects in study 1.

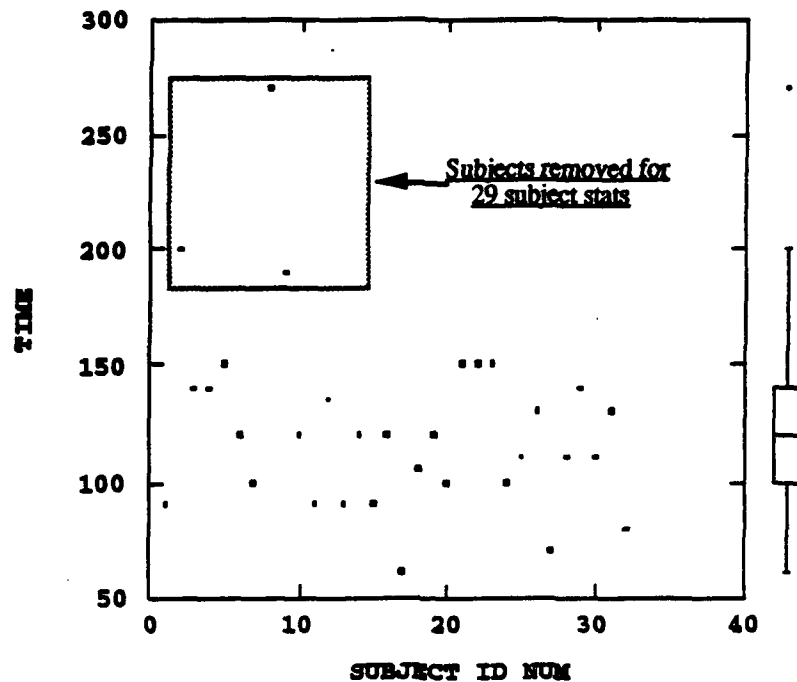


Figure 4.15: Plot of subjects versus domain development times from study 1.

I also analyzed the relationships between the different elements of the domains. Figure 4.16 shows the comparison plots. The object and relation elements are strongly correlated, where the operators are not correlated with either the objects or the relations.

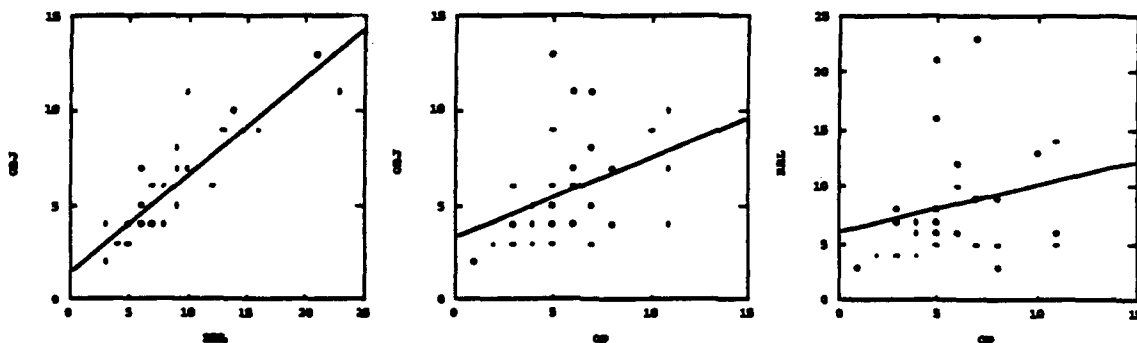


Figure 4.16: Comparison plot of Study 1 domain elements with the 29 subjects.

Using the data from the study I did a multiple regression on the 29 subjects. Time was the dependent variable and objects, relations, and operators were the independent variables. Also because of the close correlation between objects and relations, I did another multiple regression using just objects and operators as the independent variables. For each regression the squared multiple R was .2 which means that other factors were dominating

the time development (e.g., background of students, complexity of graphical objects, domain types being developed, etc.).

4.1.5 Study 1: Conclusion

The hypothesis was confirmed. All subjects were able to encode working domains using APPRENTICE and none of the domains was preselected for its visually oriented properties. APPRENTICE showed a wide applicability to many domains.

4.2 Study 2: APPRENTICE Versus Emacs Study

This study compares the use of APPRENTICE to build a domain versus the use of Emacs, a typical text editor commonly used by programmers, to build a domain. The objective of this study was to:

- *Test the productivity of users with APPRENTICE versus Emacs.* This provides quantitative results about users' productivity when building domains utilizing these two methods.
- *Test user's comprehension of new domains represented graphically versus domains represented textually.* This gives an indication of how well a domain is understood when it is represented by pictures or by simple textual "if-then" rules.
- *Test the usability of APPRENTICE with a variety of users from prodigy experts to non-programmer users.* This provides a diverse population of users for APPRENTICE. The system was tested and enhanced similar to study 1.

This is a two-phase study. Phase 1 compares the time it takes different types of users to build domains in APPRENTICE versus Emacs. Phase 2 measures each subject's ability to understand already built domains, using multiple choice questions.

4.2.1 Study 2: Hypothesis

Non-technical users will encode and understand domains faster and with more accuracy with APPRENTICE than Emacs.

4.2.2 Study 2: Procedure

In both phases four types of people were used:

PRODIGY EXPERTS are graduate students currently working on various projects that use Prodigy. They have an in-depth understanding of Prodigy and its use and have used Emacs as their primary tool for building domains.

AI INDUSTRY EMPLOYEES are members of Carnegie Group, Inc., an AI company located primarily in Pittsburgh. These employees are familiar with AI terminology and have worked with Emacs extensively. They knew nothing about Prodigy before the experiment.

CMU AI EXPERTS are subjects who are working in AI at Carnegie Mellon University. This group is very familiar with AI programming techniques and the use of Emacs but had no prior knowledge of Prodigy.

NON-TECHNICAL USERS are subjects who were unfamiliar with the field of AI and had only limited computer experience. Computer exposure for these individuals has been limited to word processing software, drawing packages, and simple database use.

4.2.2.1 Phase 1: Domain Building

For Phase 1 each subject's task was to build a domain in Apprentice and a different domain in Emacs. The specified domains were a package delivery domain and a robot path planning domain. The two domains were of equal complexity. They used a similar number of objects, relations, and operators. To gain experience developing domains in both systems, each subject used a small pizza-delivery domain for practice. A description of each domain is in Appendix D.

Emacs was chosen as the text editor for development comparison because it is currently used to develop domains in Prodigy and is similar to other methods used to build domains in typical planning systems. The Prodigy language syntax has an if-then syntax, indicative of the syntax of a lot of expert systems. The instruction that each user was given for how to use each system is in Appendix D. The instruction for using Emacs shows the PDL without universal quantifiers.

Phase 1: PROCEDURE

Each subject was given an introductory page about Prodigy and Expert Systems (see Appendix D). The page also presented the very simple practice domain, a pizza-delivery domain. Once subjects finished reading the introductory page, they used the following procedure with each system. For example, a subject would do the following procedure in Emacs, and then repeat the procedure using APPRENTICE.

Preliminary Setup

- 1) The subject read a two-page description of the system they were going to use.

- 2) The subject was given a demonstration building a subset of the blocks world domain in the current system.
- 3) The subject built the practice pizza-delivery domain in the current system.

Procedure

- 4) Finally, the subject built one of the experimental domains on their own using the appropriate system.

After the first system was used, the subjects went back to the Preliminary Setup and repeated the process with the other system.

While the subjects were building their domains, timing data was being recorded. Each subject built one domain using APPRENTICE and the other domain using Emacs. Each domain was built with each system being used by at least one person in a particular group. For example, in the AI Employee group, Subject 1 built the Strips domain using APPRENTICE and the Logistics domain using Emacs, whereas Subject 2 built the Strips domain using Emacs and the Logistics domain using APPRENTICE. This information is presented in Figure 4.17.

4.2.1.1.1 Phase I: Results

The charts in Figure 4.17 display the amount of time it took each subject to build a particular domain with a specific system. Subjects are grouped together according to the four types discussed earlier. Each subject's results are depicted in the boxes under S#. The domain that was being built is shown along the left side of the box (except for the non-technical users). The interior squares show the system that the subject used to build the particular domain and the amount of time it took that person to do it. The order in which the domains were built is represented from top to bottom. The shaded boxes are the domains that were built using APPRENTICE, and the bold face type shows the faster time for each subject. The XX represents an individual's inability to come close to task completion within a two-hour period.

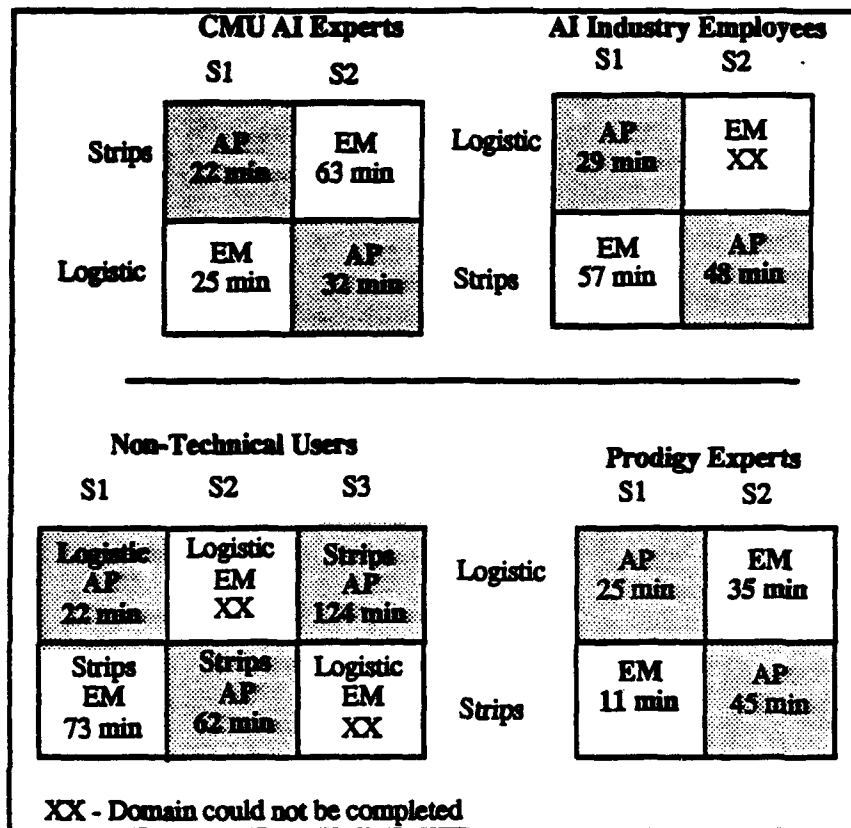


Figure 4.17: Phase 1 domain building time. The chart shows faster development time for APPRENTICE than Emacs for all but the seasoned Prodigy expert.

4.2.1.1.2 Phase 1: Analysis

Except for the Prodigy experts, all subjects were able to build domains faster by using APPRENTICE than by using Emacs. Several people were unable to even build domains using Emacs. As expected, the non-technical users had the biggest ratio difference between domains built using APPRENTICE and domains built using Emacs, as seen in Figure 4.18.

CMU AI Experts			AI Industry Employees	
S1	S2		S1	S2
1.14	1.97		1.97	XX
Non-Technical Users			Prodigy Experts	
S1	S2	S3	S1	S2
3.32	XX	XX	.44	.78

XX means that the Emacs domains were not successfully completed

Figure 4.18: Ratio of domain building time (EM/AP).

This strong indication that visual domains are built faster in APPRENTICE than in Emacs can be further analyzed. For this analysis let's ignore the Prodigy experts. They are well versed in Prodigy and had built many domains in Emacs similar to the domains in the study. Ignoring the Prodigy experts, all other subjects developed domains faster in APPRENTICE. The probability of all seven subjects randomly developing domains in APPRENTICE faster is $1/2^7$ or 0.78%. This gives a strong indication that people similar to the ones in this study (excluding the Prodigy experts) will develop visual domains faster in APPRENTICE than in Emacs.

4.2.2.2 Phase 2: Domain Understanding

Phase 2 tested the ability of a user to understand a new domain represented graphically versus a domain represented textually. Each subject was asked ten questions about a domain represented textually, and ten questions about a different domain represented graphically. The two domains used for this experiment were the monkey and banana domain and the blocks world domain. As with phase 1, each domain was studied using both systems for each subject type.

Phase 2: PROCEDURE

This process was also automated. The user was presented with an operator or a set of initial and goal states. The user was then asked several multiple-choice questions pertaining to the representation. Below are typical graphical and textual displays, along with a sample multiple-choice question. All questions for both representations are in Appendix E.

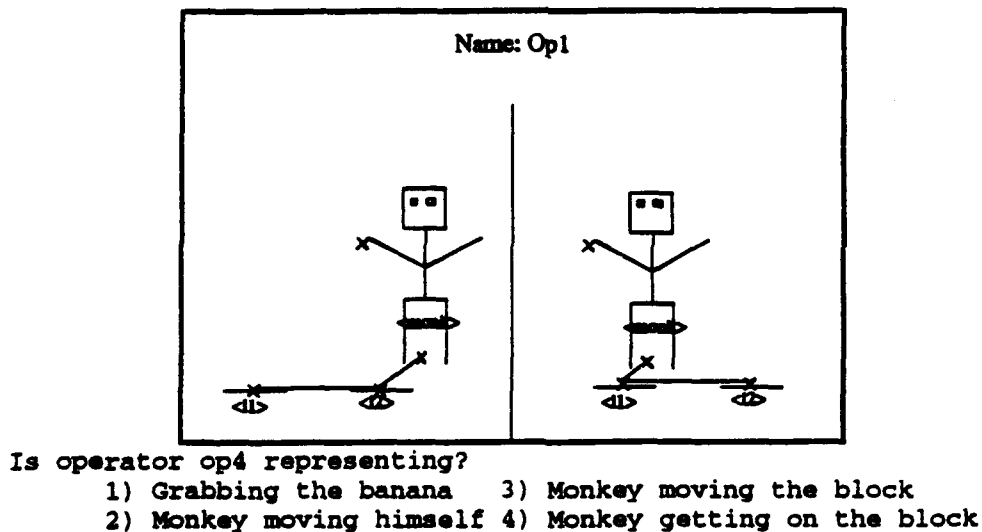


Figure 4.19: Graphical representations with a multiple-choice question.

```

Op1
  If
    (monkey-at-loc <l1> <monk>)
    (connected <l1> <l2>)
    (connected <l2> <l1>)))
  Then
    (del (monkey-at-loc <l1> <monk>))
    (add (monkey-at-loc <l2> <monk>))

```

Is operator op4 representing?

- 1) Grabbing the banana 3) Monkey moving the block
- 2) Monkey moving himself 4) Monkey getting on the block

Figure 4.20: Textual representations with a multiple-choice question.

The time to complete all the questions and the subject's answers were automatically recorded for each user.

4.2.1.2.1 Phase 2: Results

The results of the users' responses are shown in Figure 4.21. The highlighted areas depict answers about the graphical representation. Bold letters represent answers that were wrong. Inside the squares are the domain and representation the subject used, the time it took the subject to answer the questions, and the number of answers the subject got wrong. For a complete description of the questions and the wrong answers that were given see Appendix E.

CMU AI Experts			AI Industry Employees		
S1	S2		S1	S2	
Block-AP 2:25 0w	Block-EM 7:10 1w		Block-AP 4:16 0w	Block-EM 4:13 3w	
Monkey-EM 2:24 0w	Monkey-AP 4:59 0w		Monkey-EM 2:52 0w	Monkey-AP 2:36 0w	

Non-Technical Users			Prodigy Experts	
S1	S2	S3	S1	S2
Block-AP 4:32 0w	Block-EM 12:01 8w	Block-AP 7:03 0w	Block-AP 3:02 0w	Block-EM 2:24 1w
Monkey-EM 7:00 0w	Monkey-AP 3:17 1w	Monkey-EM 7:33 0w	Monkey-EM 2:23 0w	Monkey-AP 3:31 0w

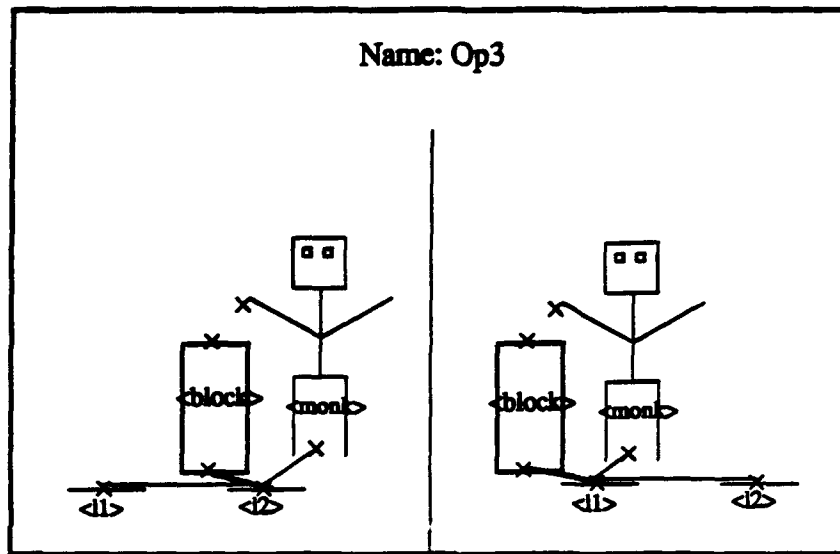
Figure 4.21: Results of Phase 2, showing much better understanding (fewer errors) for APPRENTICE than Emacs.

4.2.1.2.2 Phase 2: Analysis

The subjects understood domains presented graphically more accurately and more quickly than domains presented textually. The mean time for answering all ten questions for the graphic representation was 238 seconds with a standard deviation of 87 seconds, but for the textual representation the mean time was 320 seconds with a standard deviation of 200 seconds.

Also all the subjects got the answers correct for the graphical representation, except for one answer by a non-technical subject. In question 6 of the monkey and banana domain, the move block operator did not have the monkey connected to the block (see Figure 4.21). The subject didn't think the monkey was moving the block but just moving himself. Also

note that the same subject understood the graphical representation much better than the textual representation, as seen by the eight wrong answers about the textual representation.



Is operator op4 representing?

- | | |
|--------------------------|--------------------------------|
| 1) Grabbing the banana | 3) Monkey moving the block |
| 2) Monkey moving himself | 4) Monkey getting on the block |

Figure 4.22: Question 6 representing the move-block operator.

There were more wrong answers for the textual representation. Four subjects got a combined total of 13 wrong answers for the textual representative questions. Although the questions and the rules were very simple, the subjects still had difficulty quickly understanding a totally new domain.

4.2.3 Study 2: Analysis

The results of study 2 were:

- Domains were built faster using APPRENTICE than Emacs for all but the most seasoned Prodigy users. This study was done with users of different degrees of computer experience. Even users who were proficient in Emacs were able to use APPRENTICE faster.
- Users who had limited computer expertise were able to effectively build domains in APPRENTICE, yet some where unable to do so in Emacs.
- When given unknown domain definitions, users understood the domains better in APPRENTICE-type graphical descriptions than a textual STRIPS-type methodology.

- Again, users found the system to be "fun and easy."

4.2.4 Study 2: Conclusion

The hypothesis was confirmed. Non-technical (and all non-Prodigy) users were more productive with APPRENTICE than Emacs. Every user was able to build a successful domain using APPRENTICE while some non-technical users could not build a domain using Emacs.

4.3 Learning Study

In this study a subject built four domains over time. The goal of this study was to see the speed in which productivity increased. The subject used for this was subject 2 from the Prodigy expert group from Study 2. Three additional domains were built by the subject over the course of one day. These domains were hiking world, loading truck world, and robot picking tulip world. They can be found in Appendix G. All domains were of similar complexity to the strips domain that was previously built. The domains had three objects, four relations, and two to four operators in them. The results are as follows.

	Strips	Hiking	Load Truck	Picking Tulips
Subject 2	45 min	25 min	35 min	25 min

Figure 4.23: Table of subject 2's domain building time. The domains were built in order from left to right.

As Figure 4.23 shows, the user took less time to build the domains as experience increased. This experiment is not meant as a scientific study but as an indication that experience using APPRENTICE will improve development time.

4.4 Study 4: Medium Size Domain Building

A medium size domain with 34 operators was developed to test the scalability of APPRENTICE. The objectives of this study were to:

- *Determine the APPRENTICE techniques useful in developing larger domains. As domains get larger, how does the APPRENTICE system aid or hinder the development process?*
- *Explore additional techniques needed for development of larger domains. What are some of the requirements of domains as they get larger?*

- *Investigate issues of developing a domain over time.* In the other studies the domains were developed in one sitting. What are some of the things that affect the creation of a domain during multiple development sessions?

In this study, I took an already established large machining domain and developed a sub-portion of it in APPRENTICE over time. The domain is described in the next section, followed by what I learned from the experience.

4.4.1 Study 4: Hypothesis

APPRENTICE scales up to larger domains.

4.4.2 Study 4: Procedure

The domain I used had been previously developed and recorded in a technical report at CMU [Gil 90]. An excerpt from the abstract of that report follows:

Much research is being done on the automation of manufacturing processes. The planning component in the production stage is very significant, due to the variety of alternative processes, their complexity, and their interactions. This document describes a specification of some manufacturing processes, including the machining, joining, and finishing of parts. The aim of this specification is not to be comprehensive or detailed, but to present the AI community with a model of a complex and realistic application....[Gil 90]

Over a three-month period, working part time, I developed a sub-portion of the machining domain in APPRENTICE. This required understanding the machining domain, debugging and enhancing APPRENTICE, and understanding some subtle features about Nolimit. The developed domain consists of 33 objects, 77 relations and 34 operators. The operators consist of operations to drill, saw, plane, polish, and mill parts. The graphical and textual representations of the domain are included in Appendix F.

4.4.3 Study 4: Results

Larger domains are very hard to build using any system. In developing the machining domain there were three areas of results: 1) ways in which graphical techniques aided in the domain development; 2) additional programming algorithms that aided in domain development; and 3) techniques that were not programmed but could provide some support for domain development.

The graphical representation of the domain provided several aids to the domain development, the first being that visual images are very helpful in not forgetting relevant

information. When building the operator to saw a part it was obvious that the band saw had to have a blade in it and the part had to be on the band saw table. Also the graphical images helped me to remember what state I left the domain in from the last session. This was very useful because I was constantly going out of town during the development of this domain.

As the domain increased in size and complexity the knowledge base changed. These changes were in the form of object name changes, object appearance changes, connection points moved, relation definition changes, operator definition changes, and states definition changes. Some of the consistency checking was already a part of the system; but as the machining domain increased, more consistency checking and updating was needed. For example, when an object name changed, the relation, operator, and state that used that object had to be updated. Figure 4.24 shows the impact elements have on other elements when they change.

Impact Change	relations	operators	states	problem
objects	x	x	x	x
relations		x	x	x
operators				x
states				x

Figure 4.24: Table of the elements that are impacted when an element is changed.

Another feature in the system that proved to be useful was being able to test sub-parts of the domain. This was achieved by being able to define a problem with a subset of the defined operators and being able to individually test an operator with user-selected instantiation.

As the domain got bigger there were other features that I thought could aid in the domain development. One feature was the ability to copy and paste objects. The second was the ability to abstract the relation definition or at least automate the definition of relations for objects in the same hierarchy. Currently a relation definition involving a machine would also need to be explicitly defined for all different types machine (*e.g.*, put-vise-on-machine needs to be defined for the machine, drill, planner, and mill objects). It would be good if,

when a relation was defined for a parent object (*i.e.*, a machine object), the system defined the other relations automatically, allowing the user to change any that were incorrect. Finally, attributes need to be handled better. As the complexity of the domain increased attributes became more and more important. The size of the part and the hole position are some of these attributes. Allowing the attributes to be handled with a more graphical methodology would help increase the user's domain understanding.

4.4.4 Study 4: Conclusion

The hypothesis was confirmed. APPRENTICE does scales up to larger domains. For future work more investigation can be done on scaling up domains in APPRENTICE.

Chapter 5 - Domain Characteristics

This chapter discusses the domain characteristics that are conducive to efficient domain development in APPRENTICE. Since APPRENTICE provides the ability to textually edit any portion of a domain, any domain developed for Prodigy can be built using APPRENTICE. There are, however some characteristics that are better suited for APPRENTICE's graphical form of knowledge acquisition. In essence, APPRENTICE performs best with domains in which the central part of the domain is represented by objects, relations, and operators; with domains that are highly visual; and with domains that involve procedural tasks.

In Study 1, 30 of the 32 domains suggested by AI students prior to any knowledge of APPRENTICE were easily acquired via APPRENTICE. Therefore, visual-orientation is a very wide-ranging property for most domains.

It is my belief that knowledge acquisition should be done with multiple techniques, in which the KA system strives to closely resemble the way the expert thinks about and solves problems in the domain being developed. It is important for the designers of knowledge acquisition system to understand different techniques to help develop versatile "hybrid" KA systems. With such combined systems, the strengths of each technique can be utilized by selecting the most appropriate one for each situation.

To help identify the strengths of the APPRENTICE techniques, this chapter addresses the characteristics of a domain that makes the domain amenable to APPRENTICE. I will also discuss the ability of APPRENTICE to develop medium size domains, as well as some issues of expanding domain knowledge.

5.1 Positive Domain Characteristics

A set of domain characteristics emerged as multiple domains were built in APPRENTICE. APPRENTICE-like techniques are useful with domains having the primary characteristics of 1) objects central to the domain's representation; 2) visual images corresponding to the domain objects; and 3) modelling a procedural task. Other secondary domain

characteristics that APPRENTICE can handle well are domains in which the objects are grouped in a hierarchy and domains in which object attributes relate only to a particular object.

Domains in which objects are central to the representation can be described in terms of the objects, relations, and operators. This domain description parallels the development process for building domains in APPRENTICE. This allows the expert to focus on the domain information and not on the system representational language. This key characteristic has been a predominant characteristic in all the domains that have been built thus far in APPRENTICE.

Alternatively, in the absence of physical objects, the APPRENTICE approach works well if a mental description of the domain with a highly visual representation can be formulated (*e.g.* packet switching). This allows the expert to create visual images of a domain, similar to the mental representation that the expert already has, and eliminates some of the mental translation that the expert has to do in order to develop a domain.

Another characteristic that APPRENTICE supports well is domain modelling of a procedural task that has a structured approach to planning problem solutions, such as cooking a carrot cake or machining a metal part. These tasks follow the expert/apprentice paradigm in which the expert demonstrates to an apprentice how to solve a problem in the domain. The expert is first concerned about building a common language with the apprentice; thus, object and relation descriptions are needed. Then the expert is concerned about relaying to the apprentice the procedural or operational information in which the objects and relations are used to describe the operational information. For domains that are not solving a procedural task, such as text composition or unstructured troubleshooting, a different set of techniques are needed, and an APPRENTICE-like approach would not work well.

The other characteristics that have been shown to be supported by APPRENTICE are the hierarchy relationship between objects (*e.g.*, drill is-a machine), and the attribute description of individual objects (*e.g.*, weight of block, *etc.*).

5.2 APPRENTICE Limitations

There are also certain domain characteristics that do not seem to be conducive to the APPRENTICE techniques. Of course, domains that are not visual prove to be difficult to express with the system. Another limitation in the current system is the difficulty of expressing multidimensional relations. These types of relations represent the relationship between objects in two- or three- dimensional space. I briefly discuss the needs and specifications for allowing this type of relations definition in section 6.3.3.

Another difficulty with APPRENTICE derives from the planning system itself. Nolimit does not handle infinite type variables (*i.e.*, numbers, time, etc.) very cleanly. Variables of this type cannot be bound in an operator from the state definition during planning. This requires custom Lisp code to be written to generate the appropriate instances of the variables. This is handled by brute force in the APPRENTICE system, and none of the domains observed made extensive use of infinite type variables. Because the difficulty in dealing with infinite types is due to the underlying problem solver, APPRENTICE may deal with infinite types better using another planner.

For the largest domains that were built in APPRENTICE, the limited screen real estate did not cause a problem. But as domains get bigger and there is a high degree of interaction among objects, I expect that there will not be enough room on the screen for all the graphics. This limitation can be minimized by developing better techniques for allowing the user to store and display needed information, such as techniques for creating abstract object types that can graphically represent multiple objects as one. For example the strips domain could be extended to have multiple buildings with several rooms. A plan consists of moving packages from one location to another. This may mean moving packages between buildings. To encode the operator to move packages between buildings only the buildings need to be considered. To encode the operator to move packages between rooms only the rooms in a particular building need to be considered. By allowing abstract object definitions to handle this type of context sensitive usage screen real estate can be conserved and larger domains can be better organized.

Finally, some information can be expressed more concisely and with greater ease in a non-visual representation (*e.g.*, mathematical formulas, programming, etc.). As discussed previously, the expert should be allowed to use other methods to represented domain

information, thus allowing maximum flexibility for the expert. APPRENTICE shows some amount of this ability by allowing the generated code to be edited manually.

5.3 Techniques That Aid Large Domain Development

Large domains share certain characteristics that have to be addressed. With large domains there are many things to keep track of, multiple interactions to coordinate, and many details to incorporate into the knowledge base. Several APPRENTICE techniques aid in the development of these large domains.

The ability to have the visual representation closely match the physical domain aids in cueing the expert to needed information during domain development. For example, if the expert is developing the operator to drill a hole in a part, it is obviously easy to graphically identify the drill-bit not being in place. This error would be more difficult to find in a textual representation. As the domain increases in size, this ability to detect obvious inconsistencies becomes more and more valuable.

The structure of APPRENTICE gives an expert a consistent paradigm for developing a knowledge base, while also allowing the flexibility to build sub-parts one at a time. Because the domain closely resembles the knowledge being encoded and the development procedures are easily understood, the expert has a clear focus on what information needs to be added and where that information should go. Also, the expert can develop different parts of the domain as appropriate. This allows the expert to change focus and develop relevant information. For example, the expert can develop a portion of the domain (*e.g.*, relevant objects, relations, and operators) to drill a hole in a part, and later develop the portion of the domain to polish a part. This flexibility is very important in developing large domains in segments.

Large domains are developed over time during many sessions. The expert may go for long periods of time without using or reviewing certain information. The visualization helps the expert quickly recall the previous work. The graphics also help experts at the beginning of a new session determine where they ended from the last session.

With a shared visual representation, multiple developers can easily communicate to develop and maintain a large knowledge base. The demonstrated ease in domain understanding indicates the feasibility of this coordinated effort.

As a domain increases in size during development, some of the APPRENTICE techniques can be used to help make this development more efficient. APPRENTICE has several capabilities that aid in the expansion of a domain. These are consistency checking, inheritance of objects, unknown relations warnings, and a customizable interface.

Consistency checking allows changes made to parts of a domain to be propagated to the other relevant parts of the domain. For example, if the name or the appearance of an object is changed, then all domain elements using the object are modified to reflect the change. Changes to a relation also invoke the update of operators and states that use the relation. This provides flexibility in domain development, as described earlier.

The ability to define operators using objects high in the hierarchy allows for a more general operator definition, thereby consolidating information and making the domain better organized. This is demonstrated in the cooking carrot cake domain in Appendix C. The operators get and put-in-bowl are defined using the super class object ingr. This means that separate definitions are not needed for carrots, spices, sugar, oil, and flour. This makes the domain easier to understand and maintain.

As operators or states are being developed, sometimes the expert will describe object connections that have not yet been defined as a relation. When the system detects an undefined connection, a warning message is produced, giving the expert an opportunity to add the needed relation knowledge. An expert would create an undefined connection between objects because during operator development the expert has visual cues of the needed information for the operator. Thus when the expert thinks of a needed relation the connection is made. The system then uses this new information to solicit the missing relation definition from the expert. For example, in the machining domain, a relation between the drill machine and the drill-bit may not have been initially defined. During the building of the drill operator, the drill machine and drill-bit are connected together. It is visually obvious to have the drill holding a drill-bit in order to drill a hole in a part. The system recognizes that no relation definition exists between the drill machine and the drill-bit and warns the user of this. The user can then define the holding-tool relation between the drill machine and the drill-bit.

The final issue that aids in the development of large domains is to provide the expert a direct manipulation interface for organizing the workspace. Because the interface is easily

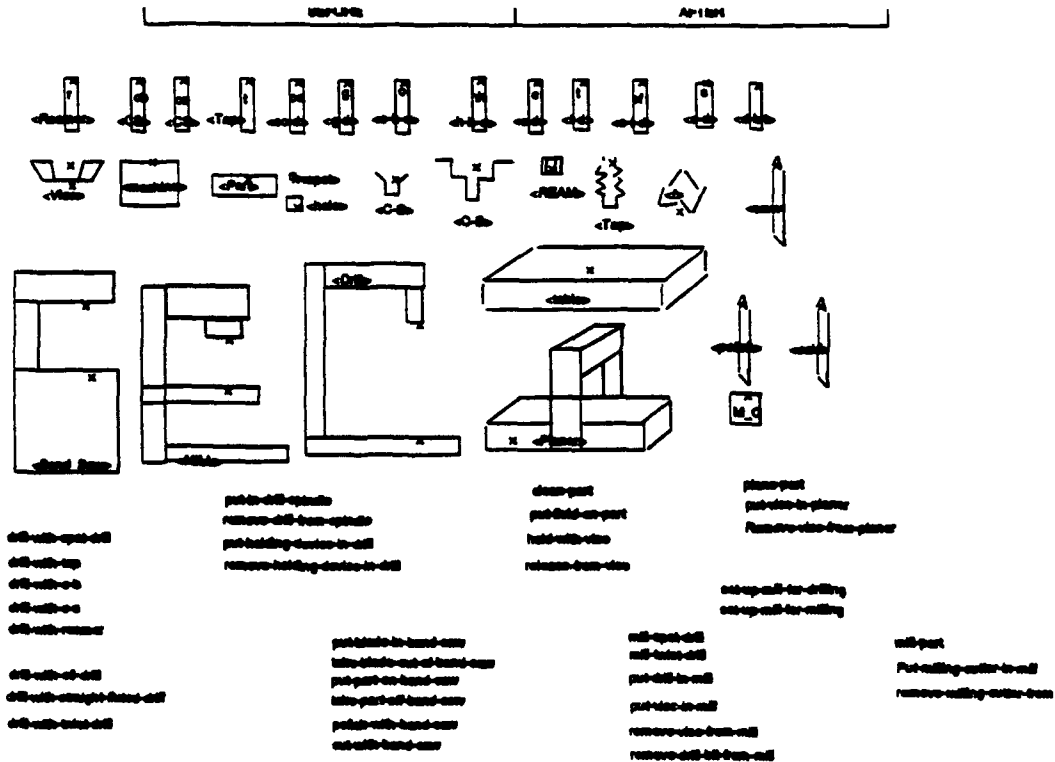


Figure 5.1: Operator Window: machining domain organized at bottom of window.

Chapter 6 - Conclusion

This chapter highlights the technical accomplishments of this dissertation and illuminates the possibilities for future research.

6.1 Summary of Findings

The ability to model real world information with metaphors similar to the visual aspects of the domain has been shown to aid domain development. This is because:

- Objects are based on the physical world. The objects in APPRENTICE occupy space on the screen similar to the space that objects occupy in the real world. Also similar to the physical world APPRENTICE objects are only at a single location at a time. These similarities help the expert understand and describe a domain using already developed intuitions.
- In using already learned intuitions to develop domains in APPRENTICE the experts do not have to mentally translate the physical domain into a foreign machine representation.
- When the system is developing a solution to a problem, the expert can quickly understand what the system is doing by observing the animation. This allows the expert confidence in the system and provides a faster indicator when some knowledge is incomplete.

6.2 Contributions of This Research

In this research, a method for graphical knowledge acquisition for visual planning domains was developed, described, implemented, and tested. A strong visual intuition for physical and conceptual domains maps very well with the conceptual representation of the domain. With these methods, the expert is able to express domain information similar to the way the information is thought about. These techniques were shown to increase the ease, speed, and accuracy of knowledge acquisition through a set of user studies.

The APPRENTICE system is tightly integrated into the Prodigy planning system. Although APPRENTICE is easy to use, it does not keep the expert from using the expressive power of the underlying planning system if needed. APPRENTICE allows the building of a domain using a graphical representation to create the information. Domain elements such as objects, relations, and operators are defined graphically. This allows the expert a straightforward mapping between the physical domain and the encoded representation.

Several studies were done with the APPRENTICE system to evaluate the ease, speed, and accuracy of the new techniques. Study 1 had 32 subjects developing their individual domains in APPRENTICE. This study showed the ease of use and the flexibility of the system with multiple subjects. Study 2 was a comparison between the productivity of developing domains in a text editor versus developing domains in APPRENTICE, using different types of users. The APPRENTICE system produced quicker development time and better domain understanding for all but the most seasoned Prodigy users. In Study 3, the system was used by a subject to develop four similar domains over time. This study indicates that domain development efficiency increases over time as the subject becomes more familiar with the system. Finally, in Study 4 the system was used to develop a machining domain. This study verified the ability of APPRENTICE to work with a larger domain.

An important contribution is that APPRENTICE demonstrated the soundness of the techniques even with users with very little prior computer knowledge. In Study 2 the non-technical computer users were able to develop working domains using the APPRENTICE system, but most of them were not able to develop similar domains using Emacs. Even with the non-technical subject who was able to build a domain in both systems, there was still about a 300% improvement in the development time for the APPRENTICE system.

6.3 Future Work

As I developed the core techniques for APPRENTICE, more ideas emerged than I had time to investigate. Some of these ideas would help make the system more usable, while others could be used to extend the system capabilities. I will describe the ideas in the following sections.

6.3.1 APPRENTICE-assisted Search Control Rule Development

In this dissertation I have only discussed the creation of factual knowledge for a domain. Search control knowledge is also important in encoding information to better guide the planning system. Future work should be done to aid the expert in developing these search control rules. This section outlines possible techniques that could be incorporated into the system.

The goal of a planning system is to find a set of operators that, when applied sequentially, will transform the initial state into the goal state. The search sequence is determined by the accuracy of the operators, the correct specification of the initial and goal states, and the set of defined search control rules. Thus, the efficiency of the planner is dependent on the selection of goals, operators, and objects in exploring the problem space. This selection can be directly controlled by search control rules. Search control rules can be developed by both the user and system, by directly observing and correcting search path mistakes.

There are three phases of development in creating search control rules. These phases are: 1) determining if the system is solving the problem correctly; 2) if the system is not solving the problem correctly, determining why and where the knowledge base is incomplete; and 3) creating search control rules to enhance the knowledge base.

To automate the development process the expert needs to determine where the planner made a wrong move and to show the planner the correct move that should have been made. The system could ask directed questions that would allow it to automatically formulate a control rule for current and future use.

To demonstrate a possible scenario I will use a STRIPS world type domain. The domain is as follows:

object: package, robot, and rooms

relation: in-room-package, package-on-robot, in-room-robot, rooms-connected

operator: pickup-package, put-down-package, move-robot

initial state: (in-room-package pk1 RoomA), (in-room-robot Robot1 RoomA),
(room-connected RoomA RoomB), (room-connected RoomB RoomA)

goal state: (in-room-robot Robot1 RoomB), (in-room-package pk1 RoomB)

Note: pickup-package removes the package from the room that the package is in, and put-down-package places the package in the same room as the robot.

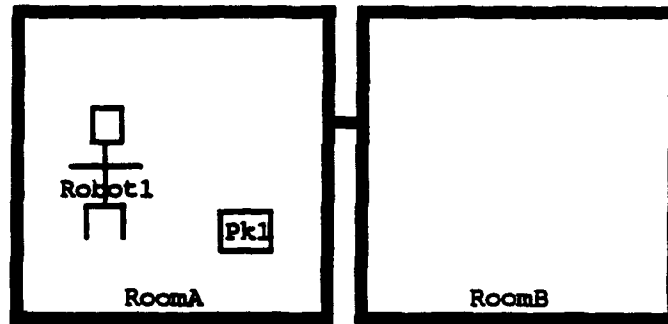


Figure 6.1: Example of a STRIPS world type domain.

When running the above problem the expert sees the robot move to RoomB, then come back to pick up pk1, and then move to RoomB with the package. This is because the first goal is to get (in-room-robot Robot1 RoomB). By watching this solution the expert notices that the robot should have picked up the pk1 first, then tried to move to the next room.

The expert then demonstrates to the system that the robot should pick up the package first before moving the robot. The system then notes that there is a discrepancy between what it did versus what it should have done. The system then sets about obtaining additional information it needs to avoid similar mistakes like this in the future.

Control rules take the form of selecting, deleting, or preferring some information. Much could be done to help the expert write these rules. The important information to determine is what kind of search control rule needs to be written and what state information is needed in order to use the control rule. The system could ask focus questions of the expert. These questions help develop control rules such as the following: if the goal of getting a package to a room has not yet been achieved and the robot is in the same room as the package, then prefer working on the in-room-package goal first. This control rule could be written as follows.

```

(REORDER-CANDIDATE-GOAL-RULE-1
  (lhs (and (current-node <n>)
            (candidate-goal <n> (in-room <Robby> <room>))
            (candidate-goal <n> (in-room-package <pk1> <room>))
            (known <n> (and (in-room-package <pk1> <RoomA>)
                          (not-equal <room> <RoomA>))))))
  (rhs (prefer goal (in-room-package <pk1> <room>)
                  (in-room <Robby> <room>))))

```

Figure 6.2: Example of possible control rule built with system assistance.

The above example has outlined a technique for possibly adding a search control rule to a knowledge base. This has been achieved by first noting that the planned solution was not optimal, next demonstrating to the system the correct sequence, and finally getting system-directed help to determine the relevant information that was needed to build a new search control rule. A deeper understanding of how a system can interactively aid an expert in developing search control rules will improve domain development.

6.3.2 Seamless Environment: Visual and Textual Representation

One of the things that Study 2 revealed was that Prodigy experts built domains faster in Emacs than in APPRENTICE. This suggests that the ability to build domains in Emacs should be incorporated into the APPRENTICE system. Providing a seamless bridge between the power of the graphical interface along with the ability to use Emacs to build domains could prove to be a dynamic combination.

Currently APPRENTICE supports full data flow from the graphical system to Emacs. APPRENTICE creates the Prodigy code automatically from the graphical description of the domain. This created code can be edited in Emacs. However, the Emacs-edited code does not automatically invoke a graphical representation in the graphical system. Thus, the graphical interface cannot be used to edit the Emacs code. Figure 6.3 shows this relationship.

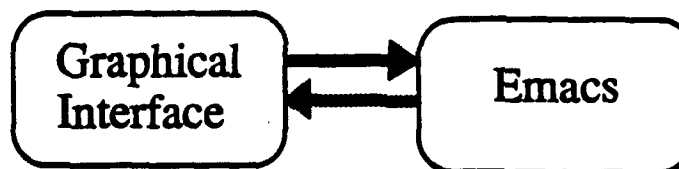


Figure 6.3: Currently APPRENTICE provides full data flow from the graphical interface to Emacs, but only limited data flow from Emacs to the graphical interface.

Further research could explore having text that corresponds to graphical relations developed with a text editor automatically update the graphical display. This would allow editing of the textual code with the graphical system.[Glinert 90].

6.3.3 Spatial Multidimensional Relations

During the course of exploring domains in APPRENTICE, it became apparent that for development of some domains two- and three-dimensional spatial information needs to be easily represented. This would be needed in a CAD/CAM domain, for example. In these types of domains, information is thought of as spatial positioning between objects. More work needs to be done into how to allow the expert an easy and intuitive way to represent and define this spatial positioning [Chang 90].

6.3.4 Apprentice Techniques for Non-visual Domains

The APPRENTICE techniques have proven to enhance the domain development process for visual planning domains. This is partially due to the ability of experts to understand the domain development process in respect to how they describe the domain. It may be possible to develop techniques similar to the ones used in APPRENTICE for non-visual domains that will also help enhance the development process.

Chapter 7 - References

- [1] Abrett, G. & Burstein, M. H. (1987). The KREME knowledge editing environment. *Knowledge Acquisition Tools for Expert Systems*. San Diego: Academic Press, pp. 1-24.
- [2] Adobe Systems, Inc. (1985). *Postscript Language Reference Manual*. Reading, Massachusetts: Addison-Wesley.
- [3] Alexander, J.H., Freiling, M. J., Shulman, S.J., Reh fuss, S. & Messick, S. L. (1987). Ontological analysis: an ongoing experiment. *Knowledge Acquisition Tools for Expert Systems*. San Diego: Academic Press, pp. 25-38.
- [4] Anzai, Y. & Simon, H. (1979). The theory of learning by doing. *Psychological Review*. 16(2), pp. 124-140.
- [5] Aoyama, M., Miyamoto, K., Murakami, N., Nagano, H. & Oki, Y. (1989). Design specification in Japan: Tree-structured charts. *IEEE Software*, 6(2), pp. 31-37.
- [6] Baker, B.R. (1986). Using images to generate speech. *Byte*, 11(3), pp. 160-168.
- [7] Barichella, E., Beretta, M., Dioguardi, N., Mussio, P., Padula, M., Pietrogrande, M. & Protti, M. (1990) A visual environment for liver simulation studies. In T. Ichikawa, E. Jungert & R. Korf hage (Eds.), *Visual Languages an Application*. New York: Plenum Press.,pp. 255-275.
- [8] Beach, R. & Stone, M. (1983). Graphical style: Towards high quality illustrations. *ACM Computer Graphics*. Proceedings of SIGGRAPH'83, Detroit, Michigan, 17(3), pp. 127-135.
- [9] Bennet, J.S. (1985) ROGET: A knowledge-based system for acquiring conceptual structure of a diagnostic expert system. *J Automated Reasoning*, 1(1).
- [10] Beretta, M., Mussio, P. & Protti. (1986). Icons: Interpretation and use. *Proceedings Workshop on Visual Languages, Dallas, Texas*. Los Alamitos, California: IEEE Computer Society Press, pp. 149-158.
- [11] Birmingham, W. & Klinker, G. (1989). *Building knowledge-acquisition tools*. Ann Arbor, Michigan: The University of Michigan.
- [12] Blattner, M., Sumikawa, D. & Greenberg, R. (1989). Earcons and icons: Their structure and common design principles. *Human-Computer Interaction*, 4(1), pp. 11-44.
- [13] Boose, J. & Bradshaw, J. (1987). Expertise transfer and complex problems: Using AQUINAS as a knowledge-acquisition workbench for knowledge-based systems. *Knowledge Acquisition Tools for Expert Systems*. San Diego: Academic Press, pp 39-64.
- [14] Borning, A. (1979). Thinglab: A constraint-oriented simulation laboratory, XEROX PARC, 79(3).
- [15] Borning, A. (1981). The programming language aspects of Thinglab: A constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4), pp. 33-387.
- [16] Bridgeland, D. (1990). Simulacrum: A system behavior example editor. In T. Ichikawa, E. Jungert & R. Korf hage (Eds.), *Visual Languages an Application*. New York: Plenum Press.,pp. 191-202.

- [17] Brown, M.H. & Sedgewick, R. (1984). A system for algorithm animation. *ACM Computer Graphics*. In *Proceedings of SIGGRAPH'84*, Minneapolis, Minnesota, 18(3), pp. 177-186.
- [18] Brown, M.H. & Sedgewick, R. (1985). Techniques for algorithm animation. *IEEE Software*, 2(1), pp. 28-39.
- [19] Brown, M.H. (1988). Exploring algorithms using Balsa-II. *IEEE Computer*, 21(5), pp.14-36.
- [20] Brown, M.H. (1988). Perspectives on algorithm animation. In *Conference Proceedings, CHI'88: Human Factors in Computing Systems*, Washington, D.C., New York: ACM Press, pp. 33-38.
- [21] Brown, M.L., Newsome, S.L. & Glinert, E.P. (1989). An experiment into the use of auditory cues to reduce visual workload. In *Conference Proceedings, CHI'89: Human Factors in Computing Systems*, Austin, Texas, New York: ACM Press, pp. 339-346.
- [22] Buchanan, G., Barstow, D., Bechtal, R. Bennett, J., Clancey, W., Kulikowski, C., Mitchell, T., & Waterman, D. (1983). Construction of an expert system. In F. Hayes-Roth, D. Waterman, & D. Lenat (Eds.), *Building Expert Systems*, Reading, Massachusetts: Addison-Wesley, pp 127 - 167.
- [23] Carbonell, J., Gil, Y., Joseph, R., Knoblock, C., Minton, S., & Veloso, M. (1990). Designing an integrated architecture: The PRODIGY view. In *Proceedings from The 12th Annual Conference of the Cognitive Science Society*.
- [24] Cardelli, L. (1988). Building user interfaces by direct manipulation. In *Proceedings UIST'88, ACM SIGGRAPH Symposium on User Interface Software and Technology*, Banff, Alberta, Canada, pp 152-166.
- [25] Chang, S. & Jungert, E. (1990). A spatial knowledge structure for visual information systems. In T. Ichikawa, E. Jungert & R. Korfhage (Eds.), *Visual Languages an Application*. New York: Plenum Press, pp. 277 - 304.
- [26] Chang, S., Tauber, M., Yu, B. & Yu, J. (1989). A visual language compiler. *IEEE Trans. on Software Engineering*, 15(5), pp. 506-525.
- [27] Citrin, W. (1991) *Visualization-based visual programming*. University of Colorado at Boulder, Boulder, CO, Technical Report CU-CS-535-91.
- [28] Clemons, E. & Greenfield, A. (1985). The SAGE system architecture: A system for the rapid development of graphics interfaces for decision support. *IEEE Computer Graphics and Applications*, 5(11), pp. 38-50.
- [29] Cohen, D. (1983). Symbolic execution of the Gist specification language, *Proceedings IJCAI-83, Karlsruhe, Germany*, pp 17 - 20.
- [30] Cohen, D. (1984). A forward inference engine to aid in understanding specifications, *Proceedings AAAI-84, Austin, Texas*, pp 56 - 60.
- [31] Cox, P., Giles, F. & Pietrzykowski, T. (1989). ProGraph: A step towards liberating programming from textual conditioning. *1989 IEEE Workshop on Visual Languages*, pp. 50-156.
- [32] Dannenberg, R. & Joseph, R. (1991). Human computer interaction in the Piano Tutor. In M. Blattner & R. Dannenberg (Eds.), *Multimedia Interface Design*. Reading, Massachusetts: Addison-Wesley.

- [33] Dannenberg, R., Sanchez, M., Joseph, A., Capell, P., Joseph, R. & Saul, R. (1990). A computer-based multi-media tutor for beginning piano students. *Interface*, 19, pp. 155-173.
- [34] Davis, R. (1979). Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, 12, pp 121-158.
- [35] Diederich, J., I. Ruhmann and M. May (1990). KRITON: a knowledge-acquisition tool for expert systems. In J.H. Boose and B. R. Gaines (Eds.), *The Foundation of Knowledge Acquisition*, London, San Diego: Academic Press.
- [36] Duisberg, R. (1986) *Constraint-based animation: the implementation of temporal constraints in the Animus system*. Dissertation, University of Washington, Seattle, published as Technical Report no. 86-09-01.
- [37] Duisberg, R. (1990). Visual programming of program visualizations: A gestural interface for animating algorithms. In T. Ichikawa, E. Jungert & R. Korfhage (Eds.), *Visual Languages an Application*. New York: Plenum Press., pp. 161 - 174.
- [38] Eshelman, L. & McDermott, J. (1986). MOLE: A knowledge acquisition tool that uses its head. *Proceedings Fifth National Conference on Artificial Intelligence, Philadelphia, Pennsylvania*.
- [39] Etzioni, O. (1990). *A structural theory of explanation-based learning*. Dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania, published as Technical Report no. CMU-CS-90-185.
- [40] Fikes, R. and N. Nilsson (1971) Strips: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence*, vol. 2.
- [41] Fischer, G. (1988) An overview of a graphical multilanguage applications environment. *IEEE Transactions on Software Engineering*, SE-14(6), pp. 774-786.
- [42] Foley, J. & McMath, C. (1986). Dynamic process visualization. *IEEE Computer Graphics and Applications*, 6(3), pp. 16-25.
- [43] Forsythe, D. & Buchanan, B. (1989). Knowledge acquisition for expert systems: Some pitfalls and suggestions. *IEEE Transactions on Systems, Man, and Cybernetics*, 19(3), pp. 435-442.
- [44] Gaver, W. (1989). The SonicFinder: An interface that uses auditory icons. *Human-Computer Interaction*, 4(1), pp. 67-94.
- [45] Gil, Y. (1992). *Acquiring domain knowledge for planning by experimentation*. Dissertation, Carnegie Mellon University, Pittsburgh, PA, published as Technical Report no. CMU-CS-92-175.
- [46] Giuse, D. (1989). KR: Constraint-based knowledge representation. Carnegie Mellon University. Technical Report.
- [47] Giuse, D., Giuse, N., & Miller, R. (1990). Towards computer-assisted maintenance of medical knowledge bases. *Artificial Intelligence in Medicine* (Elsevier Science Publishers B.V.), 2, pp. 21-33.
- [48] Giuse, D., Giuse, N., Bankowitz, R., and Miller, R. (1991). Heuristic determination of quantitative data for knowledge acquisition in medicine. *Computers and Biomedical Research* (Academic Press), 24, pp. 261-272.
- [49] Glinert, E. & Gonczarowski, J. (1987). A (Formal) model for (Iconic) programming environments." In *INTERACT '87, Proceedings of the Second IFIP Conference on Human-Computer Interaction, Stuttgart, West Germany*, pp. 283-290.

- [50] Glinert, E. (1987). Out of Flatland: Towards three-dimensional visual programming. In *Proceedings of the Second Fall Joint Computer Conference, Dallas, Texas*. Los Alamitos, California: IEEE Computer Society Press, pp. 292-299.
- [51] Glinert, Ephraim P. (Ed.) (1990) *Visual Programming Environments: Applications and Issues*. IEEE Computer Society Press, Los Alamitos, CA.
- [52] Goldberg, A. (1984). *SMALLTALK-80 The Interactive Programming Environment*. Reading, Massachusetts: Addison-Wesley.
- [53] Golin, E. & Reiss, S. (1989). The specification of visual language syntax. In *Proceedings Workshop on Visual Languages, Rome, Italy*, Los Alamitos, California: IEEE Computer Society Press, pp. 105-110.
- [54] Golin, E. (1991). *A method for the specification and parsing of visual languages*. Unpublished dissertation, Brown University, Providence, Rhode Island.
- [55] Gordon, R., MacNair, E., Gordon, K., & Kurose, J. (1987). *A visual programming approach to manufacturing modeling*. Alameda: IBM Research Division.
- [56] Graf, M. (1990). Visual programming and visual languages: Lessons learned in the trenches. In *Visual Programming Environments Applications and Issues*, pp 452-454.
- [57] Gruber, T. (1989). A method of acquiring strategic knowledge. *Knowledge Acquisition*, 1(3), pp 255-278.
- [58] Gutfreund, S. (1987). Maniplicons in ThinkerToy. *Proceedings OOPSLA '87, Orlando, Florida*. New York: ACM Press, pp. 307-317.
- [59] Gyssens, M., Paredaens, J., Van den Bussche, J. & Van Gucht, D. (1991). A graph-oriented object database model. Indiana University, Bloomington, Indiana, Technical Report No. 327.
- [60] Habermann, A. & Notkin, D. (1982). The Gandalf software development environment. In *The Second Compendium of Gandalf Documentation*, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [61] Haerberli, P. (1988). ConMan: A visual programming language for interactive graphics. *ACM Computer Graphics, Proceedings SIGGRAPH '88, Atlanta, Georgia*, 22(4), pp. 103-111.
- [62] Hartfield, B., Winograd T., & Bennett, J. *Learning HCI design: Mentoring project groups in a course on human-computer interaction*. Center for the Study of Language and Information, Technical Report CSLI-91-161 PCD-3.
- [63] Helttula, E., Hyrskykari, A., & Raiha, K. (1990). Principles of ALADDIN and other algorithm animation systems. In T. Ichikawa, E. Jungert & R. Korfhage (Eds.), *Visual Languages an Application*. New York: Plenum Press, pp. 175 - 187.
- [64] Henderson Jr., D. & Card, S. (1986). ROOMS: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. on Graphics*, 5(3), pp. 211-243.
- [65] Henderson Jr., D. (1986). The TRILLUM user interface design environment. In *Conference Proceedings, CHI'86: Human Factors in Computing Systems, Boston, Massachusetts*, New York: ACM Press, pp 221-227.
- [66] Hollan, J., Hutchins, E., & Weitzman, L. (1984). STEAMER: An interactive inspectable simulation-based training system. *AI Magazine*, Summer, pp. 15-17.

- [67] Ichikawa, T. & Hirakawa, M. (1987). Visual programming: Toward realization of user-friendly programming environments. In *Proceeding Second Fall Joint Computer Conference, Dallas, Texas*, Los Alamitos, California: IEEE Computer Society Press, pp. 129-137.
- [68] Jacobson, C. and M. J. Freiling (1990). ASTEK: a multi-paradigm knowledge acquisition tool for complex structural knowledge. In J.H. Boose and B. R. Gaines (Eds.), *The Foundation of Knowledge Acquisition*, London, San Diego: Academic Press.
- [69] Jarvenpaa, S. & Dickson, G. (1988). Graphics and managerial decision making: Research based guidelines. *CACM*, 31(6), pp. 764-774.
- [70] Joseph, R. (1984). *An expert system for completing partially routed printed circuit boards*. Master's thesis, Electrical Engineering and Computer Science Department, Massachusetts Institute of Technology.
- [71] Joseph, R. (1989). *FrameGraphics: A framebased graphic system*. Unpublished paper, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [72] Joseph, R. (1991). Multimedia presentation used in a computer based Piano Tutor system. *Proceedings for The Multimedia Workshop for AAAI-91*.
- [73] Joseph, R., Ensor, J., Dickinson, A., & Blumenthal, R. (1986). Describe: An explanation facility for an object-based expert system. *Proceedings from the Second Annual Artificial Intelligence & Advanced Computer Technology Conference*. pp. 69-73.
- [74] Kahn, G., Breaux, E., Joseph, R. & DeKlerk, P. (1987). An intelligent mixed-initiative workbench for knowledge acquisition. *Knowledge Acquisition Tools for Expert Systems*. San Diego, California: Academic Press, pp 161-174.
- [75] Kahn, G., Nowlan, S. & McDermott, J. (1985). MORE: An intelligent knowledge acquisition tool. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California*.
- [76] Kangassalo, H. (1988). CONCEPT D: A graphical language for conceptual modeling and data base use. In *Proceedings Workshop on Visual Languages*, Pittsburgh, Pennsylvania, Los Alamitos, California: IEEE Computer Society Press, pp. 2-11.
- [77] Klinker, G. (1988). Knack: Sample-driven knowledge acquisition for reporting systems. In S. Marcus (Ed.), *Automating Knowledge Acquisition for Expert Systems*, Boston: Kluwer Academic Publishers.
- [78] Kodratoff, Y. & Vrain C. (1991). *Acquiring first-order knowledge about air traffic control*. Laboratory of Research in Information, University of Paris-South, Center of Orsay, Orsay, France.
- [79] Kurlander, D. & Feiner, S. (1988). Editable graphical histories. In *Proceedings Workshop on Visual Languages, Pittsburgh, Pennsylvania*, Los Alamitos, California: IEEE Computer Society Press, pp. 127-134.
- [80] LabVIEW (1986) LabVIEW: Laboratory virtual instrument engineering workbench. *Byte*, pp. 84-92.
- [81] Ladner, R. (1988). Public Law 99-506, Section 508: Electronic equipment accessibility for disabled workers. Panel position paper in *Conference Proceedings, CHI'88: Human Factors in Computing Systems, Washington, DC*, New York: ACM Press, pp. 219-222.

- [82] Laird, J., Rosenbloom, P. & Newell, A. (1986). Chunking in SOAR: The anatomy of a general learning mechanism, *Machine Learning*, 1(1).
- [83] Larkin, Jill & Simon, H. (1987). Why a diagram is (sometimes) worth ten thousand words, *Cognitive Science*, 11, pp. 65-99.
- [84] Lenat, D., Prakash, M. & Shepherd, M. (1986). CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *AI Magazine*, 6, pp. 65-85.
- [85] Lieberman, H. (1989) A three-dimensional representation for program execution. In *Proceedings Workshop on Visual Languages, Rome, Italy*, Los Alamitos, California: IEEE Computer Society press, pp. 111-116.
- [86] Lodding, K. (1983). Iconic interfacing. *IEEE Computer Graphics and Applications*, 3(2), pp. 11-20.
- [87] MacDraw Manual. (1989). Claris Software, 5201 Patrick Henry Drive, Santa Clara, CA, 95052, 408-727-8227.
- [88] Macintosh (1985). *Inside Macintosh Volume I*. Reading, Massachusetts: Addison-Wesley.
- [89] MacKinlay, J. & Genesereth, M. (1984). Expressiveness of languages. *Proceedings AAAI-84, Austin, Texas*.
- [90] MacKinlay, J. (1986). Automating the design of graphical presentations of relational information. *ACM Trans. on Graphics*, 5(2), pp. 110-141.
- [91] MacroMind Director Manual. (1990). 410 Townsend Suite 408, San Francisco, CA 94107, Phone 415-442-0200, Director.
- [92] Macus, S. (1986). Taking backtracking with a grain of SALT. *Proceedings of the First AAAI Knowledge Acquisition for Knowledge-based Systems Workshop, Banff, Canada*.
- [93] Mahadevan, S. (1990). An apprentice-based approach to learning problem-solving knowledge. The State University of New Jersey, Rutgers, New Brunswick.
- [94] Mahling, D. E. (1989). A visual language for the aquisition and display of plans, In *Proceedings of 1989 IEEE Workshop on Visual Languages, Washington, D.C.*: IEEE Computer Society Press, pp. 50 - 55.
- [95] Malone, T.W. (1980). *What makes things fun to learn? A study of intrinsically motivating computer games* (extended excerpt). Unpublished dissertation, Stanford University, Stanford, California.
- [96] McCleary Jr., G. *An effective graphic 'vocabulary.'* Los Alamitos, California: IEEE Computer Society Press.
- [97] McDermott, J. (1982). R1: A rule-based configurer of expert systems. *Artificial Intelligence*, 28.
- [98] Miller, G. (1956). The magic number seven plus or minus two: Some limits on our capacity for information processing. *Psychological Review*, 63(2), pp. 81-96.
- [99] Minton, S. (1988). *Learning effective search control knowledge: An explanation-based approach*. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [100] Minton, S., Knoblock, C., Kuokka, D., Gil, Y., Joseph, R. & Carbonell, J. (1989). *PRODIGY 2.0: The manual and tutorial*. Unpublished paper, Carnegie Mellon University, Pittsburgh, Pennsylvania.

- [101] Moher, T. (1988). PROVIDE: A process visualization and debugging environment. *IEEE Trans. on Software Engineering*, SE-14(6), pp. 849-857.
- [102] Musen, M. (1988). *Generation of model-based knowledge-acquisition tools for clinical-trial advice systems*. Doctoral dissertation, Stanford University, Stanford, California.
- [103] Myers, B. (1987). *Creating user interfaces by demonstration*. Doctoral dissertation, University of Toronto, Canada.
- [104] Myers, B. (1988). *The state of the art in visual programming and program visualization*. Carnegie Mellon University, Pittsburgh, Pennsylvania, Technical Report CMU-CS-88-114.
- [105] Myers, B. (1992). State of the art in user interface software tools. Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [106] Newell, A. & Simon, H. (1972). *Human problem solving*, Englewood Cliffs, New Jersey: Prentice-Hall.
- [107] Newell, A. (1981). The knowledge level. *AI Magazine*, 2, pp 1 - 20.
- [108] NeXT Inc. (1991) 900 Chesapeake Drive, Redwood City, CA 94063, NeXTStep and the NeXT Interface Builder.
- [109] Nirenburg, S., Monarch, I. Kaufmann, T. & Carbonell J. (1988). Acquisition of very large knowledge bases: methodology, tools and applications. *Third AAAI-sponsored Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada.
- [110] Novak Jr., G. (1977). Representation of knowledge in a program for solving physics problems. *IJCAI*, pp 286-291.
- [111] Nyberg, E. (1988). The FRAMEKIT user's guide version 2.0. Unpublished paper, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [112] Pace, L. & Fabrocini, F. (1988). Using classification to guide knowledge acquisition and refinement in real-world domains. Submitted to IJCAI-89.
- [113] Papert, S. (1980). *Mindstorms children, computers, and powerful ideas*. New York: Basic books.
- [114] Park, H. (1990). Abstract object types equal abstract data types plus abstract knowledge types plus abstract connector types. The University of Iowa, Iowa City, Iowa.
- [115] Pepper, J., Joseph, R. & Hayes, P. (1986). GWW: A structured environment for building natural language interfaces. *Computer*, pp 85-88.
- [116] Perlin, M. (1989). Call-Graph caching: Transforming programs into networks. *Proceedings IJCAI-89*.
- [117] Potosnak, K. (1988). Do icons make user interfaces easier to use? *IEEE Software*, 5(3), pp. 97-99.
- [118] Richer, M. & Clancey, W. (19??). *Guidon-Watch: A graphics interface for viewing a knowledge base*.
- [119] Rogers, G. (1990). Visual programming using graphics, relations, and classes. Department of Computer Science, University of Illinois at Urbana-Champaign.
- [120] Roman, G. & Cox, K. (1989). A declarative approach to visualizing concurrent computations. *IEEE Computer*, 22(10), pp. 25-36.

- [121] Schaffner, S. & Borkan, M. (1988). SEGUE: Support for distributed graphical interfaces. *IEEE Computer*, 21(12), pp. 42-55.
- [122] Scott, A., Clayton, J. & Gibson, E. (1991). A practical guide to knowledge acquisition. Reading, Massachusetts: Addison-Wesley.
- [123] Shaw, M. (1986). An input-output model for interactive systems. In *Conference Proceedings, CHI'86: Human Factors in Computing Systems, Boston Massachusetts*. New York: ACM Press, pp. 261-273.
- [124] Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *IEEE Computer*, 16(8), pp. 57-69.
- [125] Shneiderman, B. (1987). *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, Massachusetts: Addison-Wesley.
- [126] Shortliffe, E. (1976) *Computer-based Medical Consultations: MYCIN*. American Elsevier.
- [127] Shu, N. (1988). A visual programming language designed for automatic programming. In *Proceedings of the 21st Hawaii International Conference on System Sciences (HICSS-21), Kailua Kona Hawaii, Volume 2: Software Track*, Los Alamitos, California: IEEE Computer Society Press, pp. 662-671.
- [128] Simon, H. & Paige, J. (1966). Cognitive processes in solving algebra word problems. In H. Simon, *Models in Thought*, pp 201-229.
- [129] Smith, D. (1988). The interface construction set, *Workshop on Visual Languages, Pittsburgh, Pennsylvania*, pp. 109-120.
- [130] Stallman, R. (1979). *Emacs: The extensible, customizable, self documenting display editor*, Technical report 519, MIT Artificial Intelligence Lab.
- [131] Steele, G. (1984). *Common Lisp: The language*. Billerica, Massachusetts: Digital Press.
- [132] Sutherland, I. (1963). SketchPad: A man-machine graphical communication system. *AFIPS Spring Joint Computer Conference*, pp 329-346.
- [133] Swartout, B. (1982). GIST English generator. *Proceedings AAAI-82, Pittsburgh, Pennsylvania*, pp 404-409.
- [134] Swartout, B. (1983). The GIST behavior explainer. *Proceedings AAAI-83, Washington D.C.*
- [135] Tanimoto, S. & Glinert, E. (1986). Designing iconic programming systems: Representation and learnability. In *Proceedings Workshop on Visual Languages, Dallas, Texas*. Los Alamitos, California: IEEE Computer Society Press, pp. 54-60.
- [136] Tanimoto, S. (1979). Stylization as a means of compacting pictorial databases. *Journal of Policy Analysis and Information Systems*, 3(2), pp. 67-89.
- [137] Tanimoto, S. (1987). Visual representation in the game of adumbration. In *Proceedings Workshop on Visual Languages, Linkoping, Sweden*, pp. 17-28.
- [138] Tortora, G. & Leoncino, P. (1988). A model for the specification and interpretation of visual languages. In *Proceedings Workshop on Visual Languages, Pittsburgh, Pennsylvania*. Los Alamitos, California: IEEE Computer Society Press, pp. 52-60.

- [139] Van Nes, F., Juola, J. & Moonen, R. (1987). Attraction and distraction by text colors on displays. In *INTERACT '87, Proceedings of the Second IFIP Conference on Human-Computer Interactions*, Stuttgart, West Germany, pp. 513-518.
- [140] Veloso, M. (1992). *Learning by analogical reasoning in general problem solving*. Doctoral dissertation, Carnegie Mellon University, Pittsburgh, Pennsylvania., published as Technical Report no. CMU-CS-92-174.
- [141] Veloso, M. M., D. Borrajo, and M. A. Perez (1992) *NoLimit - The Nonlinear problem solver for PRODIGY: User's and programmer's manual*, School of Computer Science, Carnegie Mellon, Technical Report forthcoming.
- [142] Vener, A. & Glinert, E. (1988). MAGNEX: A text editor for the visually impaired. In *Proceedings 16th Annual ACM Computer Science Conference, Atlanta, Georgia*. New York: ACM Press, pp. 402-407.
- [143] Waterman, D. & Newell, A. (1971). Protocol analysis as a task for artificial intelligence. *Artificial Intelligence*, 2(3-4).
- [144] Winston, P. (1977). *Artificial Intelligence*. Reading, Massachusetts: Addison-Wesley.
- [145] Wood, W. & Wood, S. (1987). Icons in everyday life. In G. Salvendy (Ed.), *Social, Ergonomic and Stress Aspects of Work with Computers. Proceedings of the Second International Conference on Human-Computer Interaction, Honolulu, Hawaii*, 1, pp. 97-104.
- [146] York, B. & Karshmer, A. (1989). Tools to support blind programmers. In *Proceedings 17th Annual ACM Computer Science Conference, Louisville, Kentucky*. New York: ACM Press, pp. 5-11.
- [147] Yoshida, N., Kikuno, T., Miyao, J. & Hirakawa, N. Advanced functions in a form system based on a formal form model. *Visual Programming Environments Applications and Issues*, pp 198-205.
- [148] Zloof, M. (1980). A language for office and business automation. In *Office Automation Conference Digest*. Reston, Virginia: AFIPS Press, pp. 249-260.

Appendix A - Results Chart from Study 1

Student Descriptions	Domain Descriptions	Time (hrs:min)	Objects	Relations	Operators
1) Master, HSS, PHI	Taking customer's order	1:30	5	6	5
2) Junior, SCS, M-C	People showering	3:20	4	11	5
3) Master, CMU, INI	Packet switching	2:20	4	7	4
4) Doct, SIA, IA	House building	2:20	6	8	3
5) Junior, SCS, M-C	Macaroni & Cheese	2:30	8	9	7
6) Doct, HSS, SDS	Move world with blocks	2:00	3	5	7
7) Junior, SCS, M-C	Person killing a roach	1:40	3	5	5
8) Programmer	Person bicycling	4:30	6	14	6
9) 5 Sen, MCS, MTH	Square game	3:10	2	12	10
10) 5 Sen, MCS, MTH	Robot fetching a cigarette	2:00	5	9	7
11) Junior, SCS, M-C	Hitting freshman with bat	1:30	3	4	3
12) Soph, HSS, HOO	Playing a CD in a player	2:15	4	5	8
13) Doct, CMU, RI	Robot passing blocks	1:30	3	4	4
14) Senior, CFA, MUS	Cat getting fish	2:00	7	10	6
15) Junior, SCS, M-C	Eight puzzle	1:30	2	3	1
16) Senior, SCS, M-C	People in room, sitting	2:00	4	8	5
17) Senior, SCS, M-C	Fixing Kool-Aid	1:00	7	9	8
18) Doct, CMU, RI	Cooking and eating Spam	1:45	9	13	10
19) Senior, SCS, M-C	Building DNA molecule	2:00	10	14	11
20) Senior, MCS, BSC	Playing a CD player	1:40	4	3	8
21) Senior, SCS, M-C	Feeding cat	2:30	11	23	7
22) Senior, SCS, M-C	Traveler to Mecca	2:30	11	10	6
23) Senior, SCS, M-C	Automobile starting	2:30	6	7	5
24) Senior, SCS, M-C	Washing clothes	1:40	4	5	11
25) Junior, IM, IMC	Automotive trip	1:50	9	16	5
26) Senior, SCS, M-C	Making sandwich	2:10	6	12	6
27) Senior, SCS, M-C	Delivering pizza to a TA	1:10	5	6	4
28) Senior, SCS, M-C	Moving blocks outside	1:50	4	6	6
29) 5 Sen, HSS, PSY	Cooking carrot cake	2:20	13	21	5
30) Senior, SCS, M-C	Taking out the trash	1:50	4	7	3
31) Junior, SCS, M-C	Three location blocks world	2:10	3	4	2
32) Junior, SCS, M-C	Getting ready for school	1:20	7	6	11

Table A.1: Study 1 Subjects

Chart 1 is used in conjunction with Table 1 to help decipher the student description column. This information is broken down by each student's year, department, and major.

Department	Major
CFA - College of Fine Arts	BSC - Biological Sciences
CMU - Carnegie Mellon University	HOO - General HSS
HSS - Humanities and Social Science	IA - Industrial Management
IM - Industrial Management	INI - Information Networking Institute
MCS - Mellon College of Science	IMC - IM-CIT/MCS Track
SCS - School of Computer Science	PHI - Philosophy
SIA - Graduate School of Industrial Administration	PSY - Psychology
CMU - Carnegie Mellon University	MTH - Math
HSS - Humanities and Social Science	MUS - Music
IM - Industrial Management	M-C - Math and Computer Science
MCS - Mellon College of Science	RI - Robotics
SCS - School of Computer Science	SDS - Social and Decision Science
SIA - Graduate School of Industrial Administration	

Table A.2: Student Categories

Appendix B - DNA Domain Code

DNA Domain Definition

```
(is-a domain-object-model type)
(is-a logic-state type)
(is-a dna type)
(is-a g-o-pair type)
(is-a a-t-pair type)
(is-a eny type)
(is-a adn type)
(is-a thy type)
(is-a gun type)
(is-a cyto type)
(is-a ribo type)
```

```
(operator pick-up-cyto
  (params ((<ribo> ribo) (<cyto> cyto)))
  (postconds nil)
  (effects ((add (ribo-holding-cyto <cyto> <ribo>))))))
```

```
(operator pick-up-gun
  (params ((<ribo> ribo) (<gun> gun)))
  (postconds nil)
  (effects ((add (ribo-holding-gun <ribo> <gun>))))))
```

```
(operator pick-up-eny
  (params ((<eny> eny) (<ribo> ribo)))
  (postconds nil)
  (effects ((add (ribo-holding-eny <eny> <ribo>))))))
```

```
(operator put-g-nest-to-c
  (params ((<cyto> cyto) (<ribo> ribo) (<gun> gun)))
  (postconds (ribo-holding-gun <ribo> <gun>))
  (effects
    ((del (ribo-holding-gun <ribo> <gun>))
     (add (gun-nest-to-cyto <cyto> <gun>))))))
```

```
(operator put-e-nest-to-g
  (params ((<eny> eny) (<gun> gun) (<ribo> ribo)))
  (postconds (ribo-holding-eny <eny> <ribo>))
  (effects
    ((del (ribo-holding-eny <eny> <ribo>))
     (add (eny-nest-to-gun <gun> <eny>))))))
```

```
(operator make-g-o-pair
  (params
    ((<eny> eny) (<gun> gun)
     (<cyto> cyto) (<g-o-pair> g-o-pair)))
  (postconds
    (and (eny-nest-to-gun <gun> <eny>)
         (gun-nest-to-cyto <cyto> <gun>))))
```

```
(effects
  ((del (eny-nest-to-gun <gun> <eny>)
    (del (gun-nest-to-cyto <cyto> <gun>))
    (add (g-and-o-part-of-gc <eny> <cyto> <gun>
      g-o-pair))))))
```

```
(operator pick-up-adn
  (params ((<adn> adn) (<ribo> ribo)))
  (postconds nil)
  (effects ((add (ribo-holding-adn <ribo> <adn>))))))
```

```
(operator pick-up-thy
  (params ((<thy> thy) (<ribo> ribo)))
  (postconds nil)
  (effects ((add (ribo-holding-thy <ribo> <thy>))))))
```

```
(operator make-a-t-pair
  (params
    ((<thy> thy) (<a-t-pair> a-t-pair)
     (<eny> eny) (<adn> adn)
     (<thy> thy)))
  (postconds
    (and (adn-nest-to-thy <thy> <adn>)
         (eny-nest-to-adn <adn> <eny>)))
  (effects
    ((del (adn-nest-to-thy <thy> <adn>)
     (del (eny-nest-to-adn <adn> <eny>))
     (add (a-and-t-part-of-at <eny> <adn> <a-t-pair>
      <thy>))))))
```

```
(operator put-e-nest-to-a
  (params ((<adn> adn) (<eny> eny) (<ribo> ribo)))
  (postconds (ribo-holding-eny <eny> <ribo>))
  (effects
    ((del (ribo-holding-eny <eny> <ribo>))
     (add (eny-nest-to-adn <adn> <eny>))))))
```

```
(operator put-e-nest-to-t
  (params ((<ribo> ribo) (<thy> thy) (<adn> adn)))
  (postconds (ribo-holding-adn <ribo> <adn>))
  (effects
    ((del (ribo-holding-adn <ribo> <adn>)
     (add (adn-nest-to-thy <thy> <adn>))))))
```

Problem Domain Definitions

```
(has-instance GUN gun3156)
(has-instance CYTO cyto3156)
(has-instance ENY eny3156)
(has-instance G-C-PAIR g-o-pair3156)
(has-instance RIBO ribo3156)
(has-instance A-T-PAIR a-t-pair2)
(has-instance ENY eny2)
(has-instance ADN adn2)
(has-instance THY thy2)
(has-instance RIBO-SOURCE ribo-source0088)
```

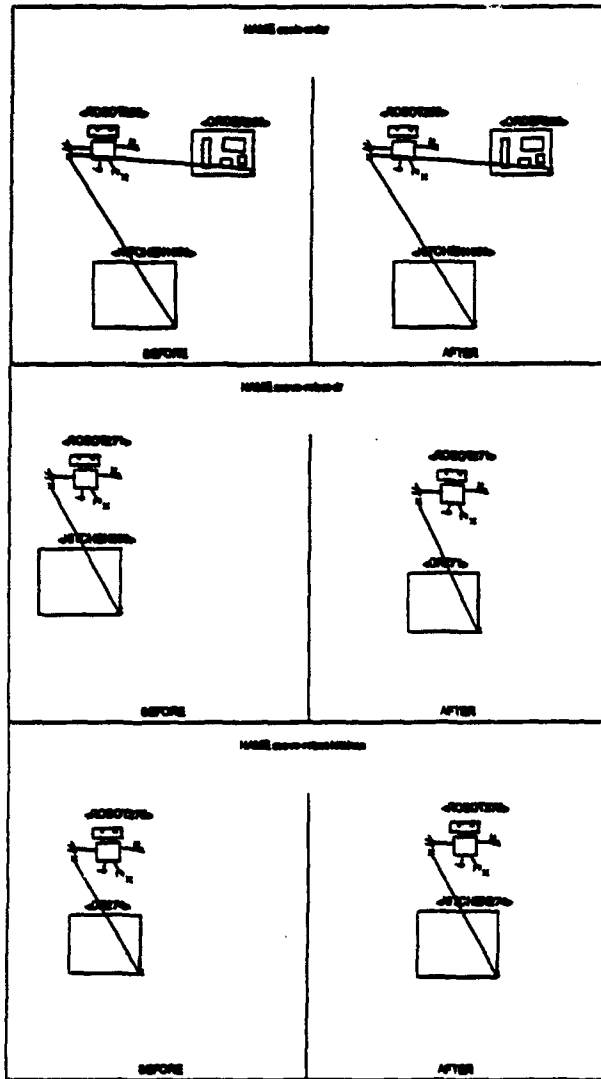
```
(CONL (NO (A-ND-T-PART-OF-AT ENY? ADN? A-T-PAIR THY?)
  (G-ND-O-PART-OF-GC ENY3156 CYTO3156 GUN3156 G-C-PAIR3156)))
```

```
(SOURCE (NO (RIBO-SOURCE RIBO-SOURCE0088)))
```

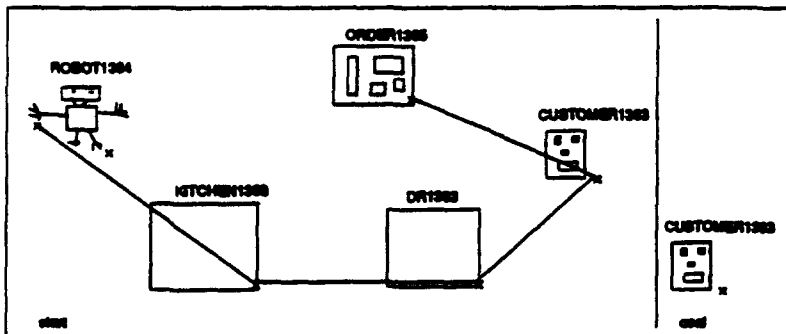
Appendix C - Selected Domains from Study 1

Subject 1
Subject 12
Subject 22
Subject 24
Subject 29

Customer Ordering Domain Built By Subject 1



Start and Goal State in APPRENTICE for Subject 1



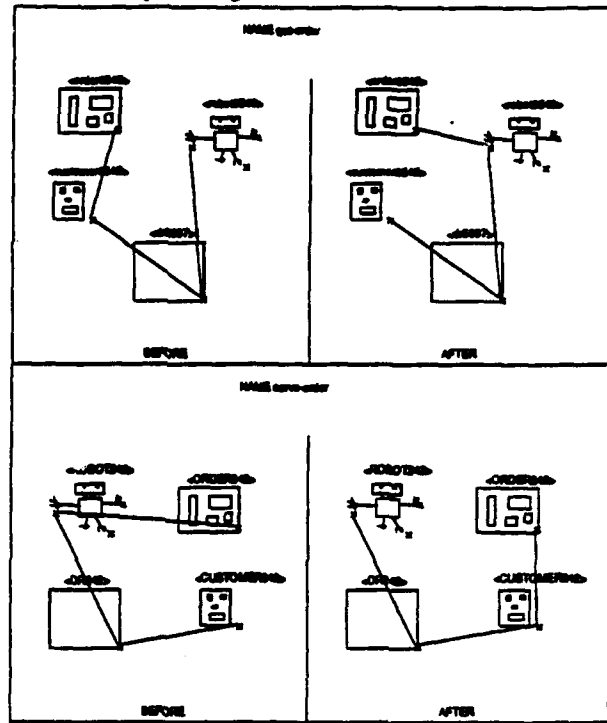
Automatically Generated Start & Goal State Description for Subject 1

```
(has-instance kitchen kitchen1363)
(has-instance dr dr1363)
(has-instance customer customer1363)
(has-instance robot robot1364)
(has-instance order order1365)

(goal (cust-state customer1363 happy))

(state (and (order-condition order1365 uncooked)
  (cust-state customer1363 unhappy)
  (space kitchen1363 dr1363)
  (order-food order1365 customer1363)
  (cust-in-dr dr1363 customer1363)
  (robot-in-kitchen kitchen1363 robot1364)))
```

APPRENTICE
OPERATORS
FOR
SUBJECT
1



Automatically Generated
Prodigy Domain Code for Subject 1

```
(is-a order type)
(is-a dr type)
(is-a kitchen type)
(is-a customer type)
(is-a robot type)

(operator serve-order
  (params ((<order242> order) (<robot242> robot)
    (<customer242> customer) (<dr242> dr)))
  (preconds
    (and (cust-state <customer242> unhappy)
      (order-condition <order242> cooked)
      (robot-with-order <order242> <robot242>)
      (cust-in-dr <dr242> <customer242>)
      (robot-in-dr <robot242> <dr242>)))
  (effects ((del (cust-state <customer242> unhappy))
    (del (robot-with-order <order242> <robot242>))
    (add (cust-state <customer242> happy))
    (add (order-food <order242> <customer242>)))))

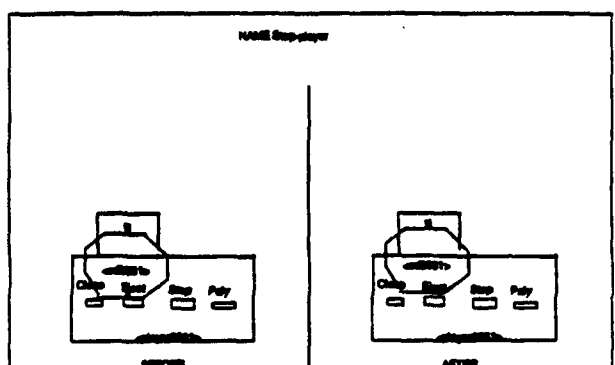
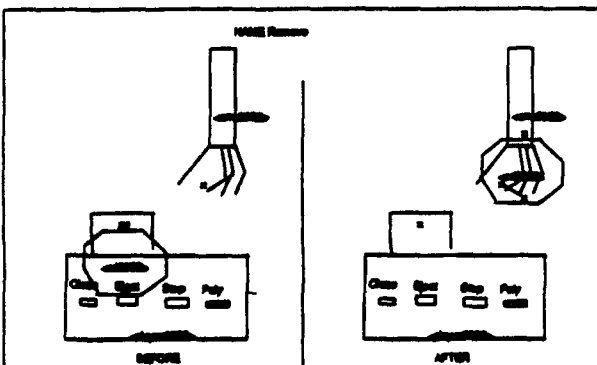
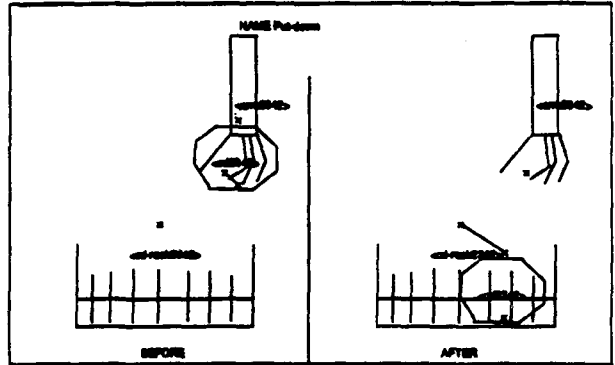
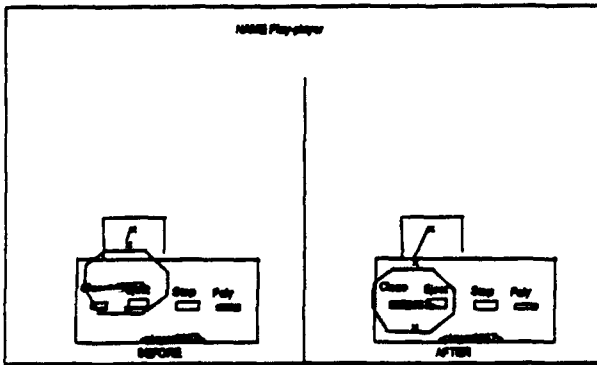
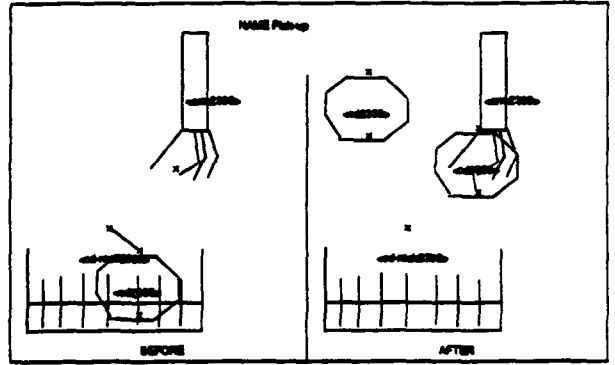
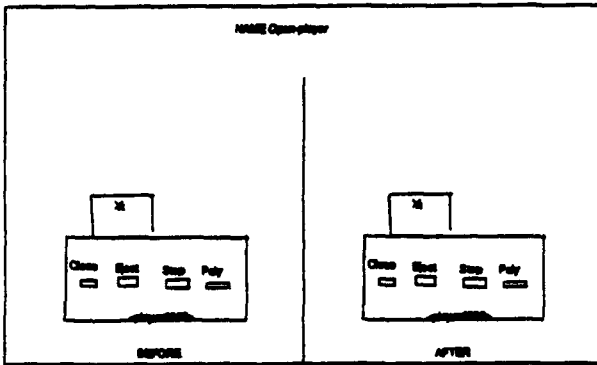
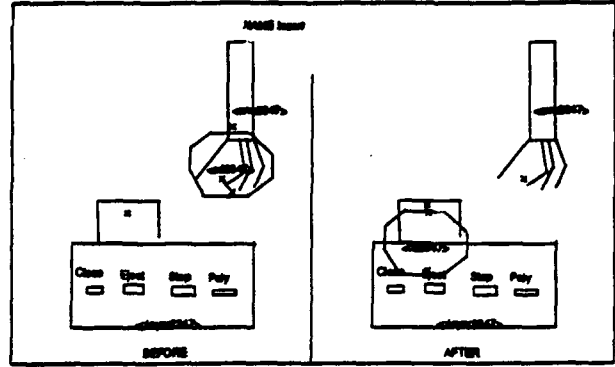
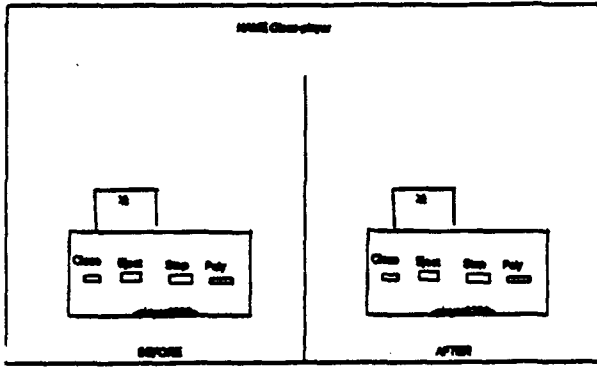
(operator cook-order
  (params ((<robot255> robot) (<order255> order) (<kitchen1350> kitchen)))
  (preconds (and (order-condition <order255> uncooked)
    (robot-with-order <order255> <robot255>)
    (robot-in-kitchen <kitchen1350> <robot255>)))
  (effects ((del (order-condition <order255> uncooked))
    (add (order-condition <order255> cooked)))))

(operator move-robot-dr
  (params ((<dr271> dr) (<kitchen269> kitchen) (<robot271> robot)))
  (preconds (robot-in-kitchen <kitchen269> <robot271>))
  (effects ((del (robot-in-kitchen <kitchen269> <robot271>))
    (add (robot-in-dr <robot271> <dr271>)))))

(operator move-robot-kitchen
  (params ((<kitchen274> kitchen) (<robot275> robot) (<dr274> dr)))
  (preconds (robot-in-dr <robot275> <dr274>))
  (effects ((del (robot-in-dr <robot275> <dr274>))
    (add (robot-in-kitchen <kitchen274> <robot275>)))))

(operator get-order
  (params ((<order3545> order) (<customer3545> customer)
    (<robot3548> robot) (<dr5007> dr)))
  (preconds (and (cust-state <customer3545> unhappy)
    (order-condition <order3545> uncooked)
    (order-food <order3545> <customer3545>)
    (cust-in-dr <dr5007> <customer3545>)
    (robot-in-dr <robot3548> <dr5007>)))
  (effects ((del (order-food <order3545> <customer3545>))
    (add (robot-with-order <order3545> <robot3548>)))))
```

Playing CD Player By Subject 12



A
P
P
R
E
N
T
I
C
E

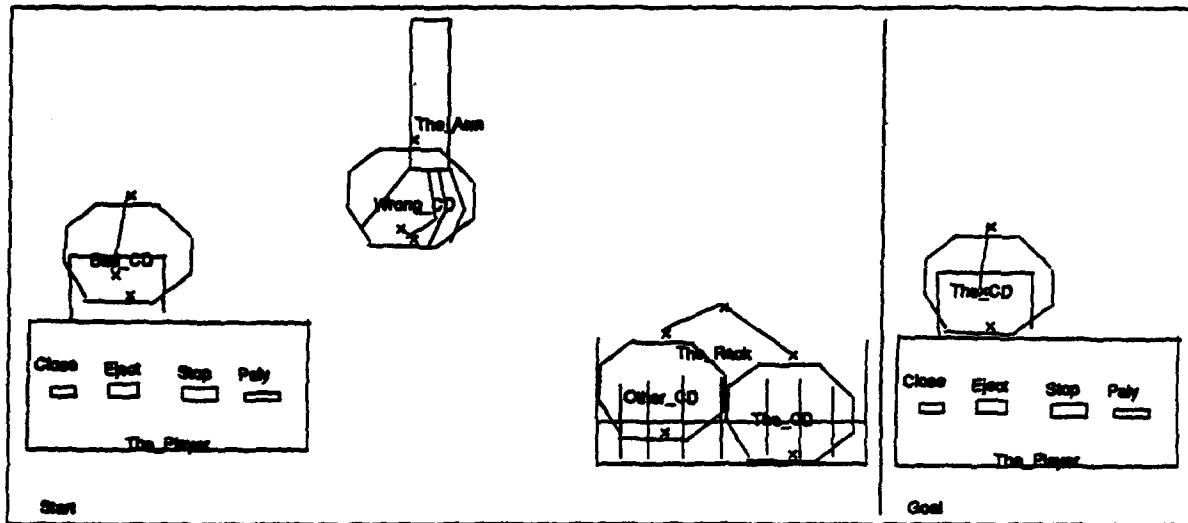
O
P
E
R
A
T
O
R
S

F
O
R

S
U
B
J
E
C
T

1
2

Start and Goal State in APPRENTICE for Subject 12



Automatically Generated Start & Goal State Description for Subject 12

```
(has-instance player the_player)
(has-instance cd wrong_cd)
(has-instance cd the_cd)
(has-instance cd-rack the_rack)
(has-instance cd other_cd)
(has-instance am the_am)
(has-instance cd bad_cd)

(goal (and (player-state the_player closed)
           (player-running the_player playing)
           (in the_cd the_player)))

(state (and (player-state the_player closed)
            (player-running the_player playing)
            (on-rack the_cd the_rack)
            (on-rack other_cd the_rack)
            (in bad_cd the_player)
            (holding wrong_cd the_am)))
```

Automatically Generated Prodigy Domain Code for Subject 12

```
(is-a domain-object-model type)
(infinite-type number-type #*makeup)
(is-a am type)
(is-a player type)
(is-a cd type)
(is-a cd-rack type)

(operator pick-up
  (purpose ((<cd-rack2336> cd-rack) (<cd2336> cd) (<am2336> am)))
  (preconditions (and (empty-am <am2336>))
                 (on-rack <cd2336> <cd-rack2336>)))
  (effects (del (on-rack <cd2336> <cd-rack2336>))
           (add (holding <cd2336> <am2336>))))

(operator put-down
  (purpose ((<cd-rack2342> cd-rack) (<cd2342> cd) (<am2342> am)))
  (preconditions (holding <cd2342> <am2342>))
  (effects (del (holding <cd2342> <am2342>))
           (add (empty-am <am2342>))
           (add (on-rack <cd2342> <cd-rack2342>))))

(operator insert
  (purpose ((<player2347> player) (<am2347> am) (<cd2347> cd)))
  (preconditions (and (player-state <player2347> open)
                     (player-running <player2347> stopped)
                     (empty-player <player2347>))
                 (holding <cd2347> <am2347>))
  (effects (del (empty-player <player2347>))
           (del (holding <cd2347> <am2347>))
           (add (empty-am <am2347>))
           (add (in <cd2347> <player2347>))))

(operator remove
  (purpose ((<player2352> player) (<cd2352> cd) (<am2352> am)))
  (preconditions (and (player-state <player2352> open)
                     (player-running <player2352> stopped)
                     (empty-am <am2352>))
                 (in <cd2352> <player2352>))
  (effects (del (in <cd2352> <player2352>))
           (add (empty-am <am2352>))
           (add (holding <cd2352> <am2352>))))

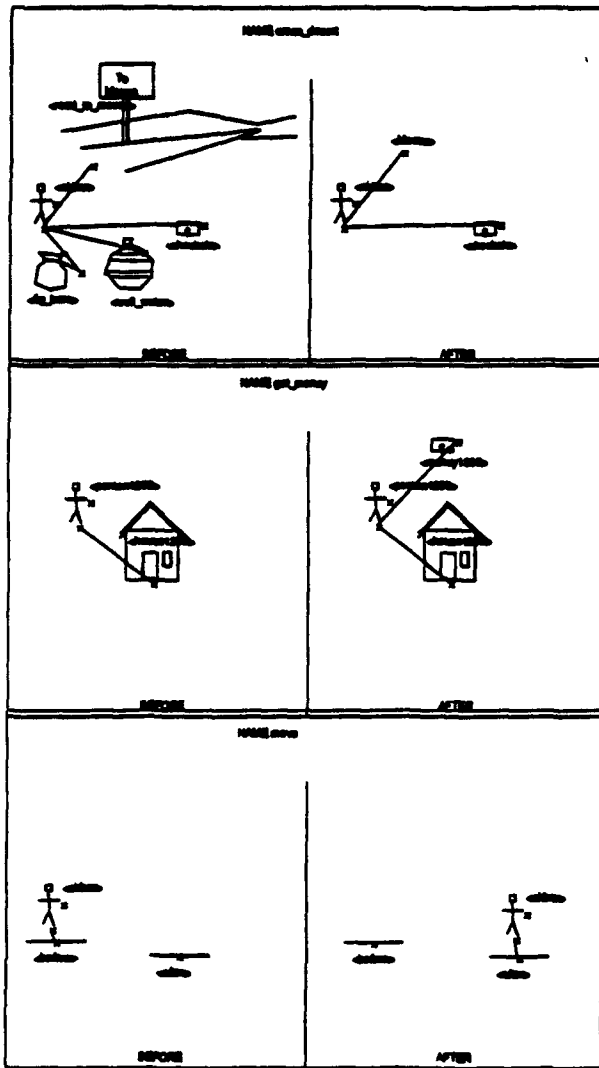
(operator play-player
  (purpose ((<player2357> player) (<cd2357> cd)))
  (preconditions (and (player-state <player2357> stopped)
                     (in <cd2357> <player2357>))
                 (player-state <player2357> closed))
  (effects (del (player-running <player2357> stopped))
           (add (player-running <player2357> playing))))

(operator stop-player
  (purpose ((<player2361> player) (<cd2361> cd)))
  (preconditions (and (in <cd2361> <player2361>))
                 (player-state <player2361> closed)
                 (player-running <player2361> playing))
  (effects (del (player-running <player2361> playing))
           (add (player-running <player2361> stopped))))

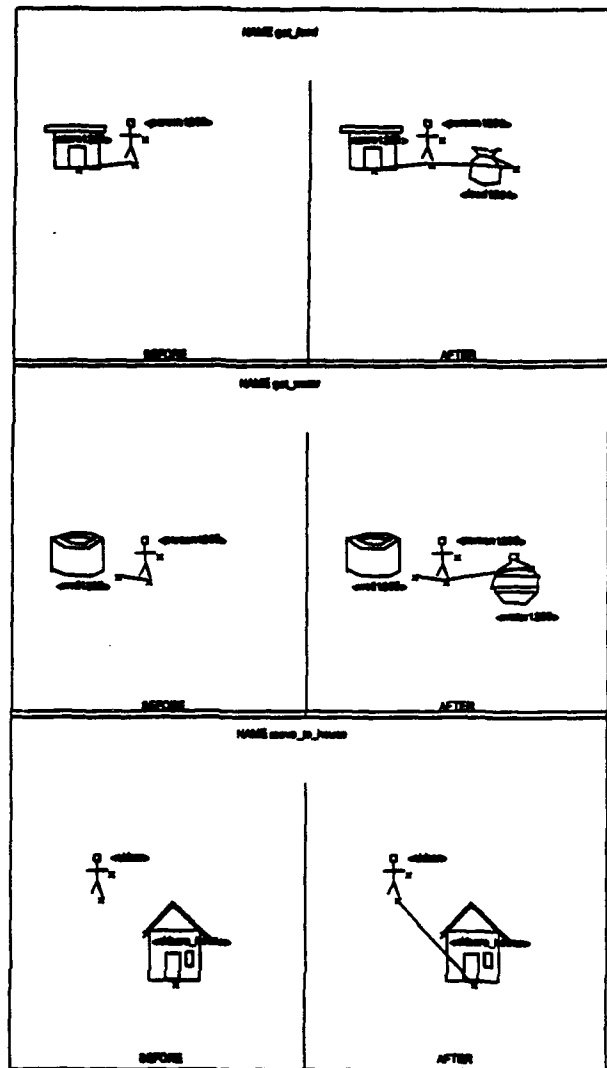
(operator open-player
  (purpose ((<player2365> player)))
  (preconditions (and (player-state <player2365> closed)
                     (player-running <player2365> stopped))
                 (effects ((del (player-state <player2365> closed))
                          (add (player-state <player2365> open))))))

(operator close-player
  (purpose ((<player2366> player)))
  (preconditions (and (player-state <player2366> open)
                     (player-running <player2366> stopped))
                 (effects ((del (player-state <player2366> open))
                          (add (player-state <player2366> closed))))))
```

Trip to Mecca By Subject 22



APPRENTICE OPERATORS FOR SUBJECT 22



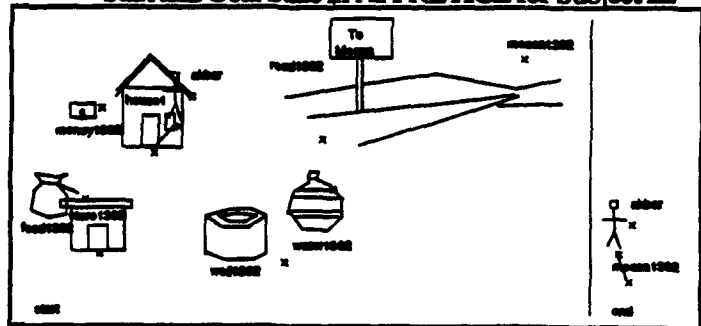
Start and Goal State in APPRENTICE for Subject 22

Automatically Generated Start & Goal State Description for Subject 22

```
(has-instance person albar)
(has-instance house house1)
(has-instance money money1382)
(has-instance wall wall1382)
(has-instance store store1382)
(has-instance road road1382)
(has-instance food food1382)
(has-instance water water1382)
(has-instance money money1382)

(goal (at-money albar money1382))

(escape (and (at-loc albar house1 house)))
```



Automatically Generated Prodigy Domain Code for Subject 22

```
(is-a money type)
(is-a possession type)
(is-a location type)
(is-a person type)

(operator move_to_house
  (params ((<albar> person) (<albar_house> house)))
  (preconditions nil)
  (effects ((add (at-loc <albar> <albar_house>))))))

(operator move
  (params ((<albar> person) (<after> location) (<before> location)))
  (preconditions (at-loc <albar> <before>))
  (effects ((del (at-loc <albar> <before>))
            (add (at-loc <albar> <after>)))))

(operator get_water
  (params ((<water1296> water) (<person1296> person) (<wall1296> wall)))
  (preconditions (at-loc <person1296> <wall1296>))
  (effects ((add (has-possession <water1296> <person1296>))))))
```

```
(operator get_food
  (params ((<food1296> food) (<person1296> person) (<store1291> store)))
  (preconditions (at-loc <person1296> <store1291>))
  (effects ((add (has-possession <food1296> <person1296>)))))

(operator get_money
  (params ((<money1306> money) (<person1296> person) (<house1296> house)))
  (preconditions (at-loc <person1296> <house1296>))
  (effects ((add (has-possession <money1306> <person1296>)))))

(operator cross_street
  (params ((<money> money) (<albar> person) (<road_to_store> road)
            (<fig_hare> food) (<wall_water> water) (<checkbox1> money)))
  (preconditions (and (has-possession <wall_water> <albar>)
                    (has-possession <fig_hare> <albar>)
                    (has-possession <checkbox1> <albar>)
                    (at-loc <albar> <road_to_store>)))
  (effects ((del (has-possession <wall_water> <albar>))
            (del (has-possession <fig_hare> <albar>))
            (del (at-loc <albar> <road_to_store>))
            (add (at-money <albar> <money>)))))
```

Washing Clothes By Subject 24

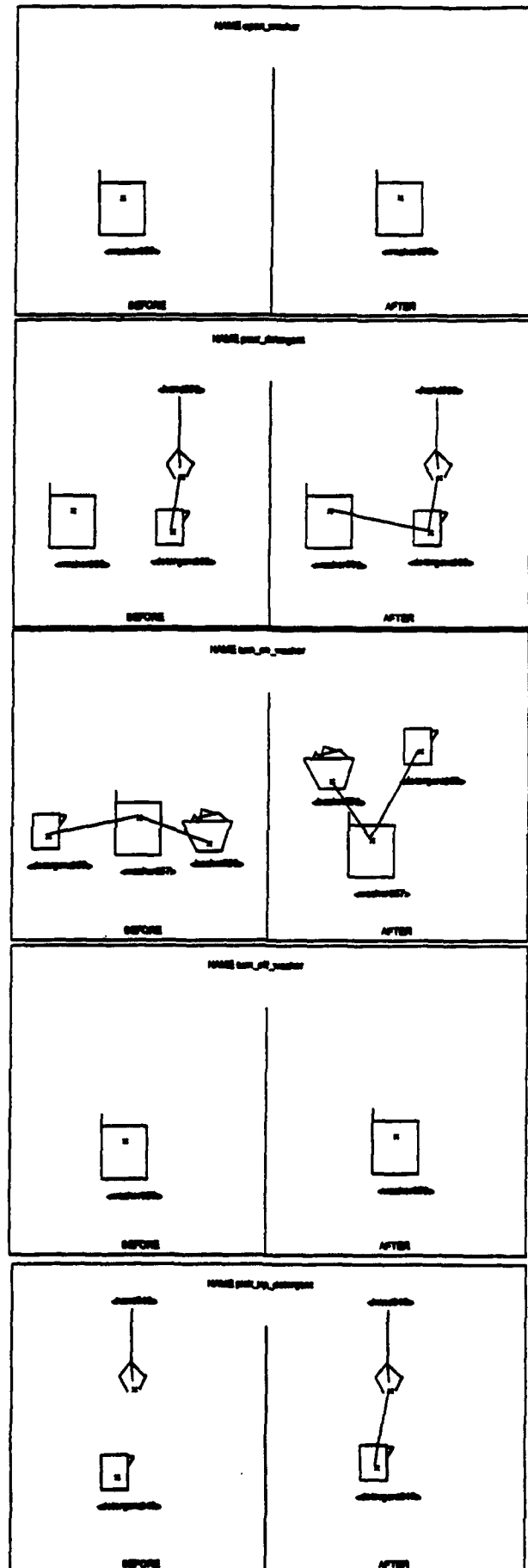
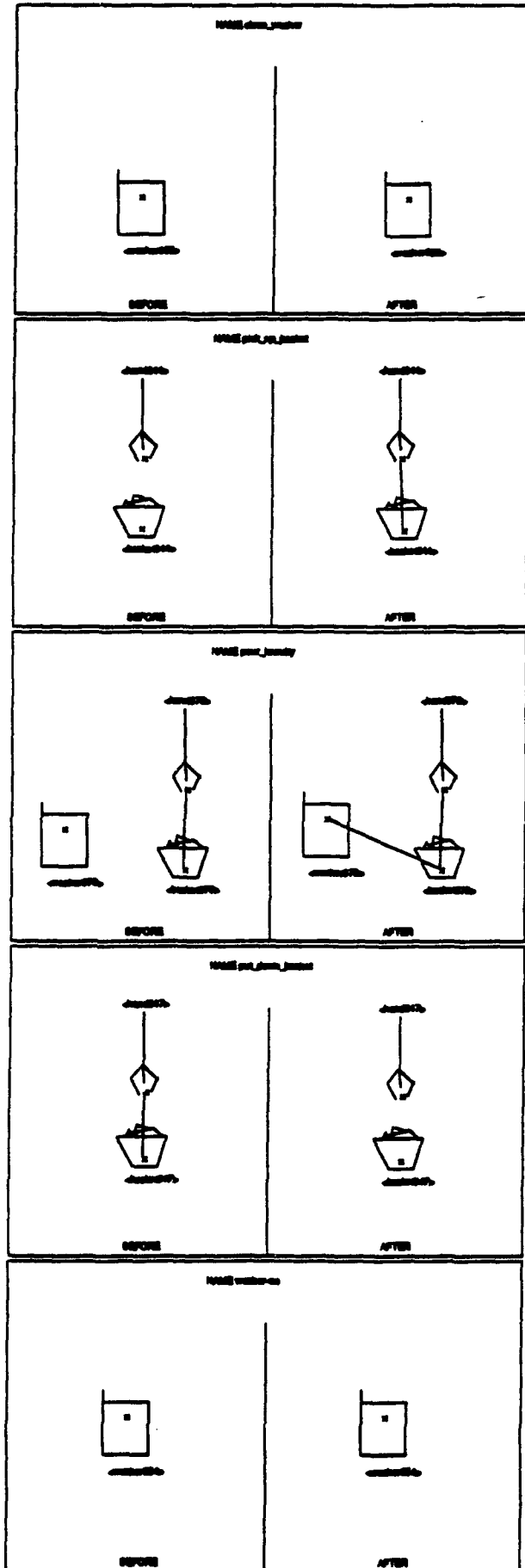
A
P
P
R
E
N
T
I
C
E

O
P
E
R
A
T
O
R
S

F
O
R

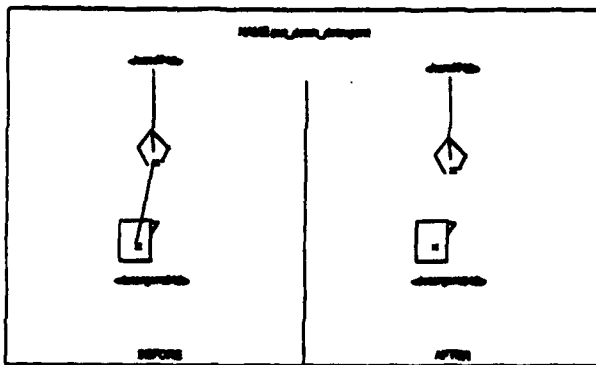
S
U
B
J
E
C
T

2
4



Operator (cont)

Automatically Generated Start & Goal State Description for Subject 24

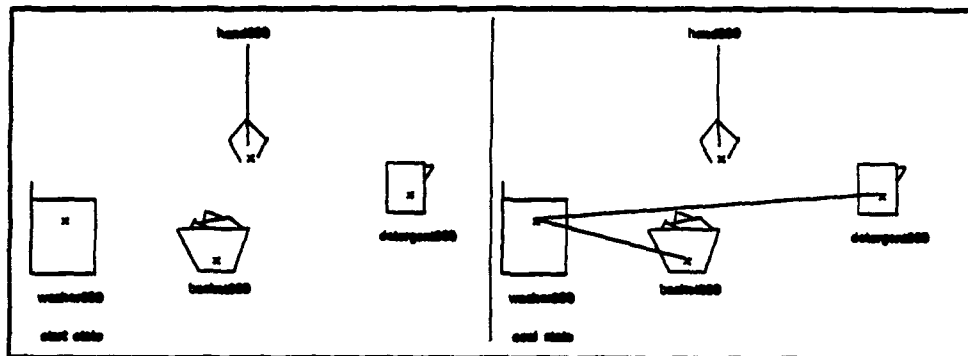


```
(has-instance washer washer000)
(has-instance basket basket000)
(has-instance detergent detergent000)
(has-instance hand hand000)

(goal (and (wash-state washer000 on)
  (door-state washer000 closed)
  (clothes_in_washer basket000 washer000)
  (detergent_in_washer washer000 detergent000)
  (handempty hand000)))

(state (and (wash-state washer000 off)
  (door-state washer000 closed)
  (handempty hand000)))
```

Start and Goal State in APPRENTICE for Subject 24



Automatically Generated Prodigy Domain Code for Subject 24

(is-a hand type)
(is-a detergent type)
(is-a washer type)
(is-a basket type)

```
(operator pick-up-detergent
  (params ((<hand040> hand) (<detergent040> detergent)))
  (preconditions (handempty <hand040>))
  (effects ((del (handempty <hand040>))
    (add (hand_holding_detergent <detergent040> <hand040>)))))
```

```
(operator put-down-detergent
  (params ((<hand040> hand) (<detergent040> detergent)))
  (preconditions (hand_holding_detergent <detergent040> <hand040>))
  (effects ((del (hand_holding_detergent <detergent040> <hand040>))
    (add (handempty <hand040>)))))
```

```
(operator pick-up-basket
  (params ((<hand040> hand) (<basket040> basket)))
  (preconditions (handempty <hand040>))
  (effects ((del (handempty <hand040>))
    (add (hand_holding_basket <basket040> <hand040>)))))
```

```
(operator put-down-basket
  (params ((<hand040> hand) (<basket040> basket)))
  (preconditions (hand_holding_basket <basket040> <hand040>))
  (effects ((del (hand_holding_basket <basket040> <hand040>))
    (add (handempty <hand040>)))))
```

```
(operator washer-on
  (params ((<washer050> washer)))
  (preconditions (and (wash-state <washer050> off)
    (door-state <washer050> closed)))
  (effects ((del (wash-state <washer050> off))
    (add (wash-state <washer050> on)))))
```

```
(operator close-washer
  (params ((<washer050> washer)))
  (preconditions (and (wash-state <washer050> on)
    (door-state <washer050> open)))
  (effects ((del (door-state <washer050> open))
    (add (door-state <washer050> closed)))))
```

```
(operator open-washer
  (params ((<washer050> washer)))
  (preconditions (and (wash-state <washer050> off)
    (door-state <washer050> closed)))
  (effects ((del (door-state <washer050> closed))
    (add (door-state <washer050> open)))))
```

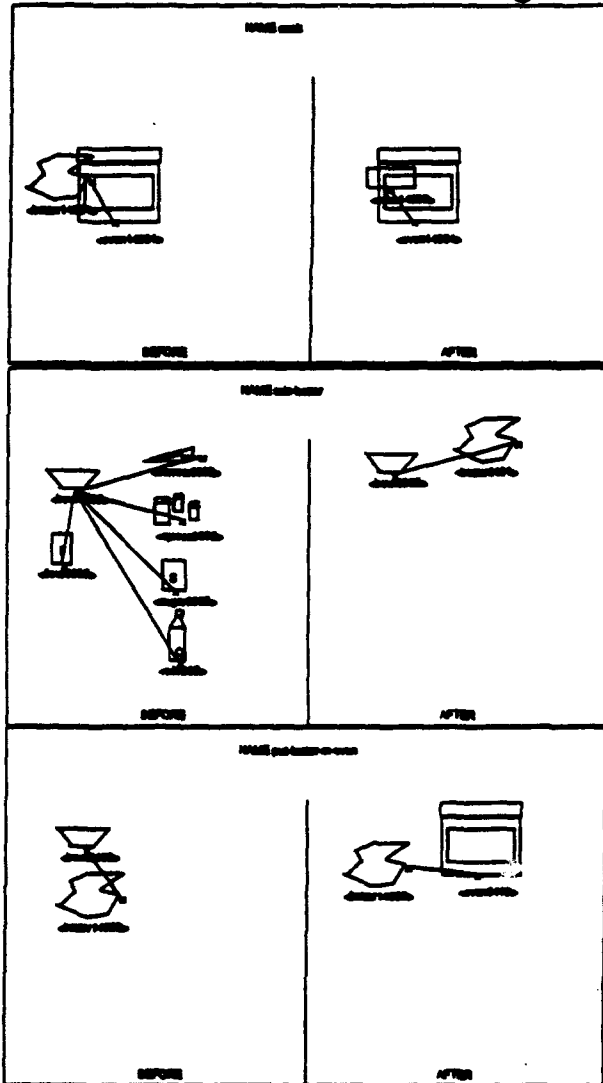
```
(operator turn-on-washer
  (params ((<washer050> washer) (<detergent050> detergent)
    (<basket050> basket)))
  (preconditions
    (and (wash-state <washer050> off)
      (door-state <washer050> closed)
      (clothes_in_washer <basket050> <washer050>)
      (detergent_in_washer <washer050> <detergent050>)))
  (effects ((del (wash-state <washer050> off))
    (add (wash-state <washer050> on)))))
```

```
(operator turn-off-washer
  (params ((<washer050> washer)))
  (preconditions (and (wash-state <washer050> on)
    (door-state <washer050> closed)))
  (effects ((del (wash-state <washer050> on))
    (add (wash-state <washer050> off)))))
```

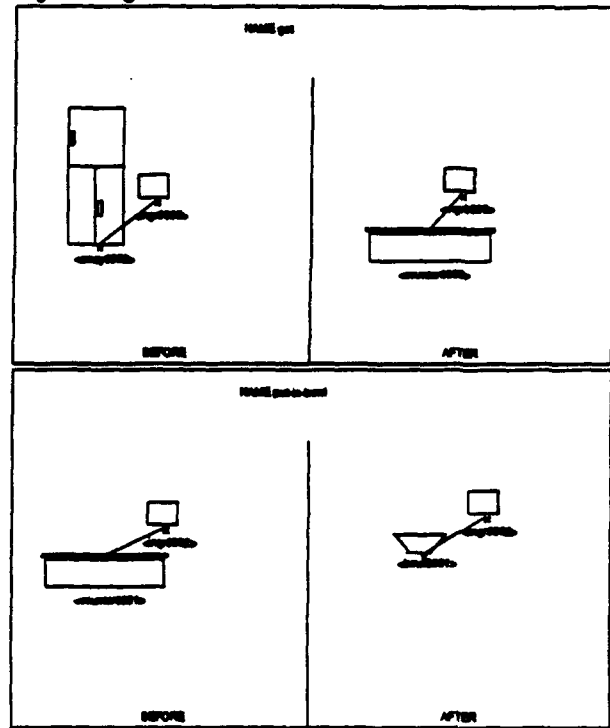
```
(operator pour-detergent
  (params ((<hand060> hand) (<detergent060> detergent)
    (<washer060> washer)))
  (preconditions
    (and (wash-state <washer060> off)
      (door-state <washer060> open)
      (hand_holding_detergent <detergent060> <hand060>)))
  (effects ((del (door-state <washer060> open))
    (add (door-state <washer060> closed))
    (add (detergent_in_washer <washer060> <detergent060>)))))
```

```
(operator pour-laundry
  (params ((<hand070> hand) (<basket070> basket) (<washer070> washer)))
  (preconditions
    (and (wash-state <washer070> off)
      (door-state <washer070> open)
      (hand_holding_basket <basket070> <hand070>)))
  (effects ((add (clothes_in_washer <basket070> <washer070>)))))
```

Cooking Carrot Cake By Subject 29



APPRENTICE OPERATORS FOR SUBJECT 29



Automatically Generated Prodigy Domain Code for Subject 29

```
(is-a cake type)
(is-a low type)
(is-a ingr type)
(is-a batter type)

(operator get
  (params ((<counter338> counter) (<may338> may) (<ingr338> ingr)))
  (preconds (may-ing <ingr338> <may338>))
  (effects ((del (may-ing <ingr338> <may338>))
    (add (on-counter <ingr338> <counter338>)))))

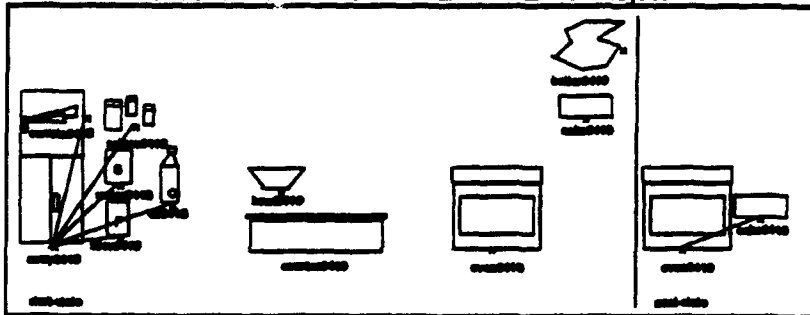
(operator put-in-bowl
  (params ((<bowl339> bowl) (<counter339> counter) (<ingr339> ingr)))
  (preconds (on-counter <ingr339> <counter339>))
  (effects ((del (on-counter <ingr339> <counter339>))
    (add (in-bowl <ingr339> <bowl339>)))))

(operator mix-batter
  (params
    ((<batter340> batter) (<bowl339> bowl) (<carrots339> carrots)
      (<apleas339> aples) (<sugar339> sugar) (<oil339> oil)
      (<flour339> flour)))
  (preconds
    (and (in-bowl <flour339> <bowl339>)
      (in-bowl <oil339> <bowl339>)
      (in-bowl <sugar339> <bowl339>)
      (in-bowl <carrots339> <bowl339>)
      (in-bowl <apleas339> <bowl339>)))
  (effects
    ((del (in-bowl <flour339> <bowl339>))
      (del (in-bowl <oil339> <bowl339>))
      (del (in-bowl <sugar339> <bowl339>))
      (del (in-bowl <carrots339> <bowl339>))
      (del (in-bowl <apleas339> <bowl339>))
      (add (in-bowl <batter340> <bowl339>)))))

(operator put-batter-in-oven
  (params ((<oven341> oven) (<bowl340> bowl) (<batter340> batter)))
  (preconds (in-bowl <batter340> <bowl340>))
  (effects ((del (in-bowl <batter340> <bowl340>))
    (add (in-oven <batter340> <oven341>)))))

(operator cook
  (params ((<cake342> cake) (<oven341> oven)
    (<batter340> batter)))
  (preconds (in-oven <batter340> <oven341>))
  (effects ((del (in-oven <batter340> <oven341>))
    (add (in-oven <cake342> <oven341>)))))
```

Start and Goal State in APPRENTICE for Subject 29



Automatically Generated Start & Goal State Description for Subject 29

```
(has-instance may may3413)
(has-instance carrots carrots3413)
(has-instance aples aples3413)
(has-instance sugar sugar3413)
(has-instance oil oil3413)
(has-instance flour flour3413)
(has-instance bowl bowl3413)
(has-instance counter counter3413)
(has-instance oven oven3413)
(has-instance batter batter3413)
(has-instance cake cake3413)

(goal (in-oven cake3413 oven3413))

(state (and (may-ing flour3413 may3413)
  (may-ing oil3413 may3413)
  (may-ing sugar3413 may3413)
  (may-ing aples3413 may3413)
  (may-ing carrots3413 may3413)))
```

Appendix D - Information Given to Subjects in Study 2 Phase 1

Prodigy and Domain Description

In the field of Artificial Intelligence one way of emulating human problem solving techniques is with planning systems. Planning systems allow a user to define a situation and problem systematically and have the system plan a solution. PRODIGY is such a general purpose planning system.

In PRODIGY the actions that the planner takes are defined by operators (actions a person would use to solve a problem). The problem is a coarse representation of relevant facts describing the initial situation before a plan is executed; and the final situation desired after a set of operators are executed. Let's take a very simple example like the process of delivering a pizza. This can be a very complicated process, but in this case we only want a very coarse grain model of the "domain" or situation.

Domain: You have a car and work for a pizza place. When the pizza is ready you load it into your car, which is at the pizza place, and deliver it to someone's home. For this domain you can only deliver one pizza at a time. There are 3 operators that you will be concerned with: load-pizza-in-car, drive-to-location and unload-pizza-from-car. The load-pizza-in-car states that if the pizza is ready and the car is at the same location as the pizza and the car is empty then the pizza can be put inside the car. The next operator, drive-to-location, states that if a car is in at location1 then it can drive to a different location, location2. Finally the unload-pizza-from-car states that if a pizza is in the car then the pizza can be unloaded from the car and the pizza is now in the same location that the car is and the car is empty.

load-pizza-in-car if pizza at location1 car at location1 then put pizza in car	drive-car if car at location1 then car at location2	unload-pizza-from-car if car at location pizza in car then unload pizza to location
---	---	--

Lets describe two problems:

PROBLEM 1: The initial situation is that the pizza and the car are at the store. The goal is to deliver the pizza to Larry's house. The solution is to load-pizza-in-car at the store, drive-car to Larry's house and then unload the pizza at Larry's house.

PROBLEM 2: Here is a slightly more complicated situation. The initial situation is that the car is now at Larry's house and the pizza is at the pizza place. Again the goal is to deliver the pizza to Larry's house. Here the solution is to drive-car to the store then load-pizza-in-car at the store, drive-car to Larry's house and then unload the pizza at Larry's house.

How do you encode this information so a computer can use it? That is what planning systems allow you to do.

The pizza delivery domain is right now a very simple domain, but by adding more delivery locations and route length information the complexity of the problem increases and the system can actually be useful in planning pizza delivery routes.

The task is to build this simple pizza delivery route domain as an exercise in Prodigy. You will use Apprentice, a visual domain building tool, and Emacs, a textual editor.

Emacs/Prodigy Example Domain

1. Introduction

Writing a domain in Prodigy requires several steps. You must create object type definitions used in the domain, operators, instances of objects, start states, and goals states. Below is an example of all these steps for the pizza domain. We will use the pizza domain to define a domain in Prodigy using Emacs.

2. Defining Object Types

First you must define object definitions for a domain. These type definitions tell Prodigy the name of the objects you will be using in the domain. For the pizza domain you will need pizza objects, car objects and location objects. An object definition has the following skeleton (IS-A *object-name* TYPE). For the pizza domain these objects are defined as below.

```
(IS-A Location-Model TYPE)
(IS-A Car-model TYPE)
(IS-A Pizza-model TYPE)
```

3. Defining Predicates and Operators

Next you must define operators. The operators define how to change states in the domain. This change is defined by giving the preconditions of a situation and then the effects of the change. These pre-conditions and effects are represented by predicates. Predicates represent relations between objects. In order to allow an operator to work in a generic way the predicates use variables. These variables match different objects in the state definition and are represented by <> surrounding the name of the variable. An operator skeleton is as shown below:

```
(OPERATOR name-of-operator
  (PARAMS
    ((<var1> type1)
     (<var2> type2)
     (<var3> type3) ...))
  (PRECONDS
    (AND (pred1 <var1> <var2>)
         (pred2 <var2> <var3>))...)
  (EFFECTS
    ((DEL (pred1 <var1> <var2>))
     (ADD (pred3 <var1> <var3>))) ... )
```

Real operators in the pizza domain are defined below. Look at the operator LOAD-PIZZA. In the PARAMS list the variable <pizza> is of type pizza-model, etc. In the precond list the condition Car-at-loc says that the variable <car> has to be at variable location <loc> and the condition pizza-at-loc says that the variable <pizza> has to be at the same location <loc> as the variable <car>. Finally if these predicates are true in the state, then the variable <pizza> is moved inside the <car> variable and the <pizza> is no longer at the location <loc>. See if you can determine what the other operators mean.

```
(OPERATOR LOAD-PIZZA
  (PARAMS
    ((<pizza> PIZZA-MODEL) (<loc> LOCATION-MODEL) (<car> CAR-MODEL)))
  (PRECONDS
    (AND (CAR-AT-LOC <car> <loc>) (PIZZA-AT-LOC <pizza> <loc>)))
  (EFFECTS
    ((DEL (PIZZA-AT-LOC <pizza> <loc>))
     (ADD (PIZZA-IN-CAR <pizza> <car>))))
```

```

(OPERATOR MOVE-CAR
  (PARAMS
    ((<doc1> LOCATION-MODEL) (<doc2> LOCATION-MODEL) (<car> CAR-MODEL)))
  (PRECONDS (CAR-AT-LOC <car> <doc1>))
  (EFFECTS
    ((DEL (CAR-AT-LOC <car> <doc1>))
     (ADD (CAR-AT-LOC <car> <doc2>))))))

(OPERATOR UNLOAD-PIZZA
  (PARAMS
    ((<doc> LOCATION-MODEL) (<pizza> PIZZA-MODEL) (<car> CAR-MODEL)))
  (PRECONDS
    (AND (PIZZA-IN-CAR <pizza> <car>) (CAR-AT-LOC <car> <doc>)))
  (EFFECTS
    ((DEL (PIZZA-IN-CAR <pizza> <car>))
     (ADD (PIZZA-AT-LOC <pizza> <doc>))))))

```

4. Defining a State

After the domain is described then the problems that you want to solve need to be defined. The problem is defined by a start state (STATE, below) and a goal state (GOAL, below) in the domain. Below the State says that the Pepperoni and Domino-Car are at the Pizza-place and the goal is to get the Pepperoni to Larry's place. When defining problems for a domain you must specify what the instances are in the problem. For example the word Pepperoni is an actual pizza and Domino-Car is an actual car.

```

(HAS-INSTANCES Location-Model Pizza-Place)
(HAS-INSTANCES Location-Model Larry)
(HAS-INSTANCES Car-model Domino-Car)
(HAS-INSTANCES Pizza-model Pepperoni)

(GOAL (PIZZA-AT-LOC PEPPERONI LARRY))

(STATE (AND (PIZZA-AT-LOC PEPPERONI PIZZA-PLACE) (CAR-AT-LOC DOMINO-CAR PIZZA-PLACE)))

```

5. Emacs and Prodigy Usage Description

You will have three windows available. The first window will be the domain window, the second window will be the start and goal window and the third window will be the lisp window.

Text is entered in each window by typing. The text is placed at the point of the blinking cursor █ in the active window. A window becomes active by clicking the mouse in it. The cursor can be moved within a window by pointing and clicking the mouse button at the desired location or with the use of control key + text keys. The cursor movements for different keys are listed below. ^<key> means hold the control key down with the respective <key>.

^a - move cursor to the front of the line	^e - move cursor to the end of the line
^f - move cursor forward one space	^b - move cursor backward one space
^n - move cursor down one line	^p - move cursor up one line
^g - abort whatever you did	^x^o - move to the above window

6. Starting Prodigy

Once you have created the objects and the operators for a domain then ^c^l will load that information. When you have typed in a problem, ^c^p will load the problem and run Prodigy. The results of the run will be printed out in the lisp window. You can also test each operator separately against the current system state. To start testing individual

operators use the ^c^f command. Then use the step-operator function in the lisp window to run operators. Below is the syntax for the step-operator:

```
(step-operator <op-name> (<var1> VAR1-NAME) (<var2> . VAR2-NAME))
```

Example of step-operator from pizza domain above:

```
(step-operator load-pizza (<pizza> . Pepperoni) (<car> . Domino-Car) (<loc> Pizza-Place))
```

To view the current state type: (print-state)

7. Steps for Defining the Pizza Domain and Problem

- 1) Define the object types (car, pizza, and location)
- 2) Define needed relations (pizza-in-car, car-at-loc, and pizza-at-loc)
- 3) Define the operators (load-pizza, unload-pizza, and drive-car)
- 4) Define the problem with instances (Pepperoni, Domino-car, Pizza-Place, and Larry-Place)
- 5) Define the start state and the goal state
- 6) Debug domain (fix syntax errors and step through individual operators)

APPRENTICE Description

1. Introduction

APPRENTICE allows an expert to create each part of a domain in Prodigy using pictures that the expert develops. There is an APPRENTICE development window for each aspect of a domain. To make domain development easier, the main input device is a mouse. In each window there are buttons that perform actions and textual items that display information. Each window has a development box that the expert works in to develop a particular piece of knowledge.

Users familiar with the Macintosh interface should be very comfortable using APPRENTICE. Every item other than the development box is movable by moving the mouse to a position over the object and holding the left mouse button down while moving the object to the desired location.

2. APPRENTICE Windows

Model Window - Allows objects in a domain to be defined (e.g., Truck, Pizza, Location)

Relation Window - Allows relationships between objects to be defined (e.g., inside-truck, at-location).

Operator Window - Allows state changes to be defined (e.g., load-pizza, move-car, unload-pizza).

State Window - Allows the definition of a start state and a goal state.

Problem Window - Allows the setup and running of a problem to be solved.

Common Apprentice Window Facts

- Hold down button over an object move it.
- Hold down button in blank area and drag rectangle select object inside rectangle.
- Shift click button select multiple object.
- Click on object allows editing information.
- Click return in work window allows work on another thing.
- Shift click on anything give help message about thing clicked.
- Alt Click on object allows editing attributes of the object.



- Selected items appear highlighted.
- Shift right click on an item gives a help message on the item.
- The name text represents the name of a item (i.e., object, relation, operator, state or problem).
- Multiple items selection: 1) hold the shift key down; 2) drag a box around a set of items.
- Text can be edited by clicking the right button.
- Object names can be changed by right clicking on the object.
- Buttons can be moved with the mouse by there text part are activated by the box part.

2.1. Model Window

The model window allows the expert to develop the objects of a domain. The objects have an appearance as well as connection spots. The development box allows you to work on a particular object. Window components:

NAME: name of the particular object type

GRAPHICAL EDITOR: An area used to build the visual representation of objects

DELETE INST BUTTON - Deletes selected model instances

2.1.1. Graphical Editor

Models or objects are made up of simple drawing elements. The editor lets you draw these simple elements to build an object. Each element in the editor can be manipulated with mouse movement.

The following drawing objects are available:

-  Arrow allows selection, movement, and alteration of elements.
-  Allows you to draw a line element.
-  Allows you to add a text element.
-  Allows you to draw a box element.
-  Allows the instance name of the object to be displayed.
-  DEL deletes the selected elements.
-  Saves the elements in the editor to the actual object. If no object is currently being worked on, then a new object is created.
-  Allows you to place connection point element on the object.

2.2. Relation Window

Drag the objects into the work area that defines a relationship between the objects. Connect the objects to one another with their connection points by clicking on each relevant connection point. When you are finished with a relation then right click on DONE.

2.3. Operator State

Operators are defined by dragging objects and connecting them together in the pre state and then in post state. The Copy prestate button copies the information in the pre state side into the post state side. The Link Obj buttons allows an object selected in the pre state to be linked with an object selected in the post state. When you are finished with an operator then right click on DONE.

2.4. State Window

To define a state, drag the appropriate objects into the work area window and connect the objects up as they would be in a state. Copy state button will copy a selected state into the work area. When you are finished with a state then right click on DONE.

2.5. Problem Window

The problem window allows you to define a problem that consists of a start state, goal state, and a set of operators. The problem elements are defined by right clicking on the START-STATE, GOAL-STATE, and OPS text. Once a problem is defined, to have the system try and plan a path click the RUN DOMAIN button. If you want to test on operator at a time click the STEP FORWARD button.

3. Steps for Defining the Pizza Domain and Problem

- 1) Draw the object types with connection (car-model, pizza-model, and location-model)
- 2) Define needed relations (pizza-in-car, car-at-loc, and pizza-at-loc)
- 3) Define the operators by pre and post state (load-pizza, unload-pizza, and drive-car)
- 4) Define the start state and the goal state graphically. The instances should have the names:
Pepperoni, Domino-car, Pizza-Place, and Larry-Place
- 5) Define the problem by combining operator, start state, and goal state
- 6) Debug domain (fix syntax errors and step through individual operators)

Strips World Description

Problem Description

The strips domain is a fictitious world comprised of the following objects: robots, boxes, and rooms. Relations between the objects are as follows: 1) robots can hold a box; 2) two rooms can be connected together; 3) robots can be in a room; and 4) boxes can be in a room. The actions that are performed in this domain are: 1) robots can pick up boxes; 2) robots can put down boxes; 3) robots can move between connected rooms with a box; and 4) robots can move between connected rooms without a box. The goal of this domain is to plan paths of movement for boxes to get from an initial location to a final destination.

Task: Define an initial state, goal state, and domain. Have the system find the plan needed to go from initial to goal state of the following problems.

Problem 1: For the initial state create two rooms (Room1, Room2), a robot, and a box. Room1 and Room2 are connected together. Have the robot and box in Room1. The goal is to have the box placed in Room2.

Problem 2: Use the work from the previous problem. For the initial state create three rooms (Room1, Room2, Room3), a robot, and a box. Room1 is connected to Room2 and Room2 is connected to Room3. Have the robot in Room1 and the box in Room2. This time the goal state is to have the box be put in Room3.

Logistic World Description

Problem Description

The logistic world is comprised of the following objects: boxes and hubs. Hubs are buildings to which boxes are brought for sorting and distribution. Relations between objects are as follows: 1) boxes can be at the indoor of a hub 2) boxes can be at the outdoor of a hub and 3) hubs are connected together from the outdoor of one hub to the indoor of another hub. The actions in the domain are that 1) boxes can move between the outdoor of one hub to the indoor of a connected hub and 2) a box at a hub's indoor must be sorted to get to the hub's outdoor. The goal of this domain is to plan paths of movement for boxes to get from an initial location to a final destination.

Task: *Define an initial state, goal state, and domain. Have the system find the plan needed to go from initial to goal state of the following problems.*

Problem 1: The initial state has three hubs (hub1, hub2, and hub3). Hub1's outdoor is connected to Hub2's indoor. Hub2's outdoor is connected to Hub3's indoor. There is initially a box (Box1) at the inside door of Hub1. The goal is to have the box delivered to the inside door of Hub3.

Problem 2: Use the work from the previous problem. In this problem the initial state has four hubs (hub1, hub2, hub3 and hub4). Hub1's outdoor is connected to Hub2's and Hub3's indoor. Hub3's outdoor is connected to Hub4's indoor. A box (Box1) starts at the inside door of Hub1. The goal is to have the box delivered to the inside door of Hub4.

Appendix E - Questions from Study 2

Phase 2

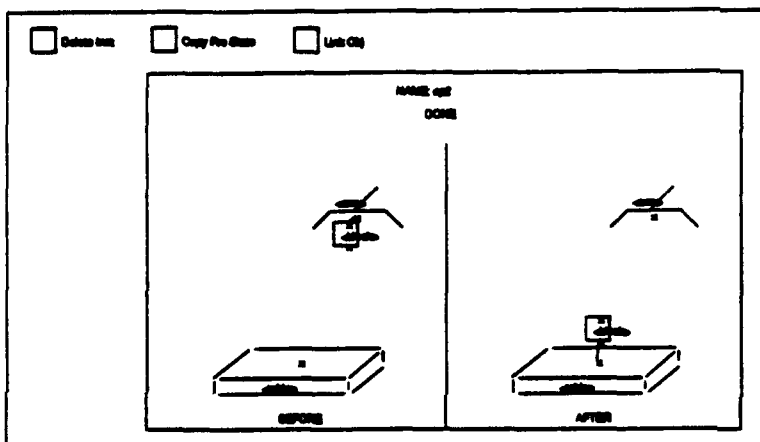
Questions from Study 2 of Phase 2

The following section contains the domains, questions, and results from study 2 of phase 2. For this phase two domains were used: the monkey and banana domain and the blocks domain. The questions are about operators or states in these domains.

Each question section has a graphical representation of an operator or state along side the textual representation of the same information. The textual representation was automatically generated with APPRENTICE from the graphical representation. The multiple-choice question is below these representations. The correct answer to the question has been made bold for your convenience. To the right of the question is the data showing the subjects that answered this question incorrectly. Only one incorrect answer was given during the graphical representation testing. The incorrect question for the textual representation will be duly noted.

Blocks Domain

Question 1:



OPERATOR NAME: op2

params

(<bottom> block-model)

(<top> block-model)

(<arm> arm-model)

preconds

(on <top> <bottom>)

(empty-arm <arm>)

(clear <top>)

effects

(del (on <top> <bottom>))

(del (empty-arm <arm>))

(del (clear <top>))

(add (holding <top> <arm>))

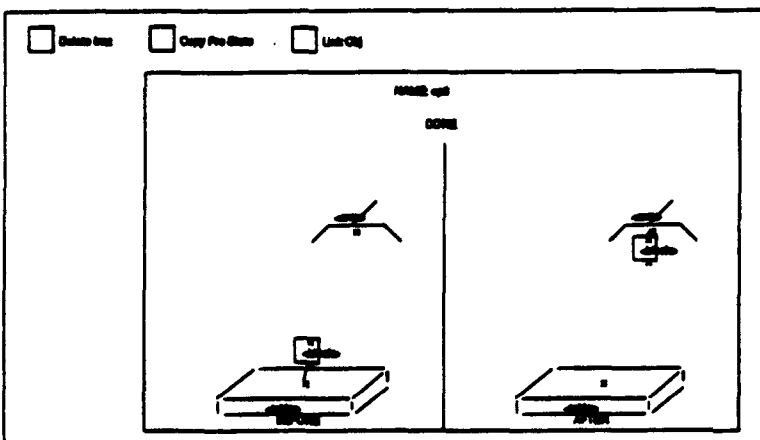
(add (clear <bottom>))

If op2 has fired is the arm empty or holding a block?

- 1) Holding
- 2) **Empty**
- 3) Unknown

AI Employee S2: 1
Non-Tech S2: 3

Question 2:



OPERATOR NAME: op3

params

(<arm> arm-model)

(<block> block-model)

(<table> table-model)

preconds

(on-table <block> <table>)

(empty-arm <arm>) (clear <block>)

effects

(del (on-table <block> <table>))

(del (empty-arm <arm>))

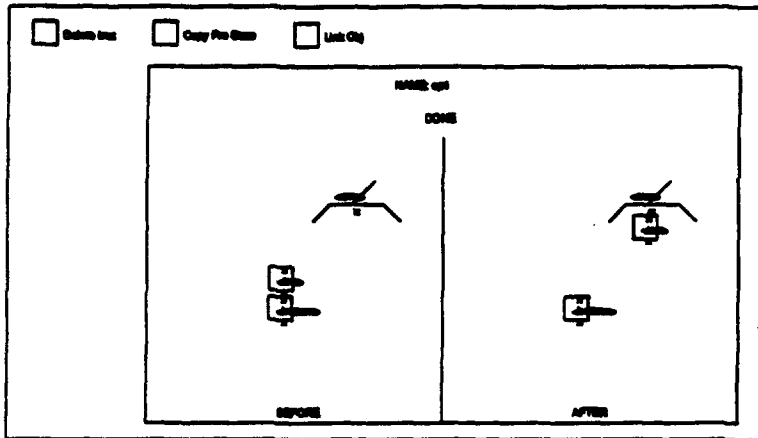
(del (clear <block>))

(add (holding <block> <arm>))

If op3 has fired is the arm empty or holding a block?

- 1) **Holding**
- 2) Empty
- 3) Unknown

Question 3:



OPERATOR NAME: op1

params
 (<table> table-model)
 (<block> block-model)
 (<arm> arm-model)

preconds
 (holding <block> <arm>)

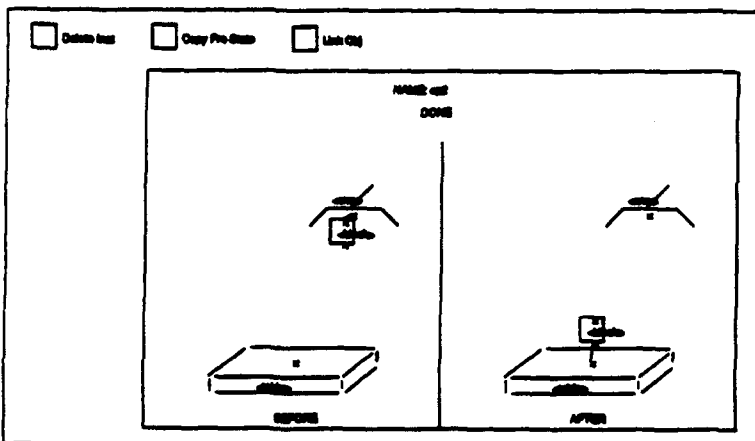
effects
 (del (holding <block> <arm>))
 (add (on-table <block> <table>))
 (add (empty-arm <arm>))
 (add (clear <block>))

Is operator op1 representing?

- 1) Picking up a block off the table
- 2) Putting down a block on the table
- 3) Stacking a block on another
- 4) Unstacking a block off another

AI Expert S2: 1
 AI Employee S2: 1
 Non-Tech S2: 1
 Prodigy S2: 1

Question 4:



OPERATOR NAME: op2

params
 (<bottom> block-model)
 (<top> block-model)
 (<arm> arm-model)

preconds
 (on <top> <bottom>)
 (empty-arm <arm>)
 (clear <top>)

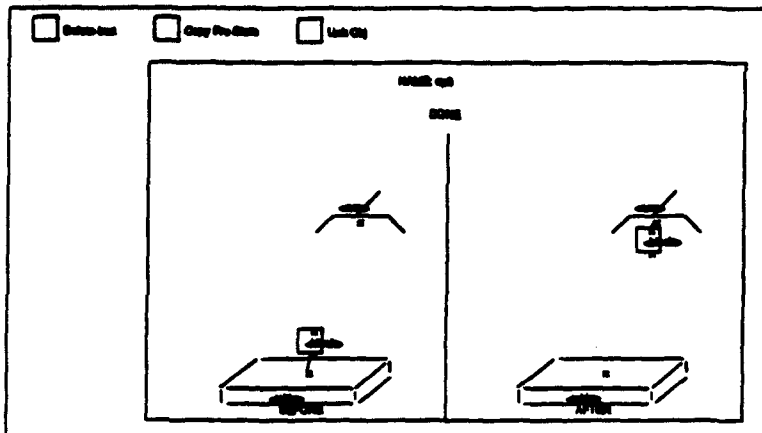
effects
 (del (on <top> <bottom>))
 (del (empty-arm <arm>))
 (del (clear <top>))
 (add (holding <top> <arm>))
 (add (clear <bottom>))

Is operator op2 representing?

- 1) Picking up a block off the table
- 2) Putting down a block on the table
- 3) Stacking a block on another
- 4) Unstacking a block off another

Non-Tech S2: 1

Question 5:



OPERATOR NAME: op3

```

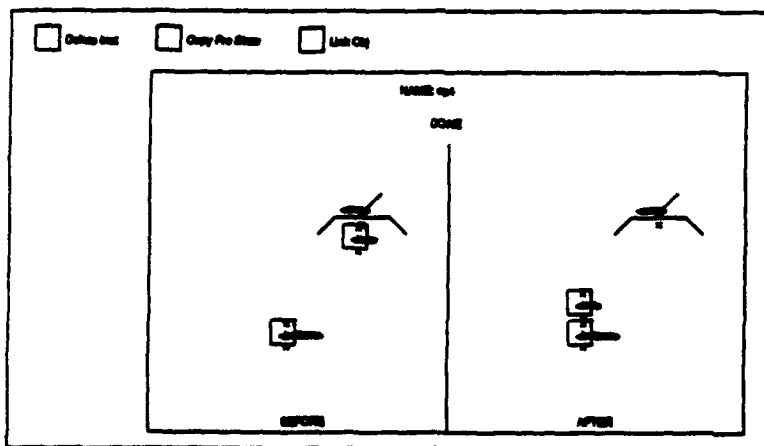
params
  (<arm> arm-model)
  (<block> block-model)
  (<table> table-model)
preconds
  (on-table <block> <table>)
  (empty-arm <arm>) (clear <block>)
effects
  (del (on-table <block> <table>))
  (del (empty-arm <arm>))
  (del (clear <block>))
  (add (holding <block> <arm>))
  
```

Is operator op3 representing?

- 1) Picking up a block off the table
- 2) Putting down a block on the table
- 3) Stacking a block on another
- 4) Unstacking a block off another

Non-Tech S2: 2

Question 6:



OPERATOR NAME: op4

```

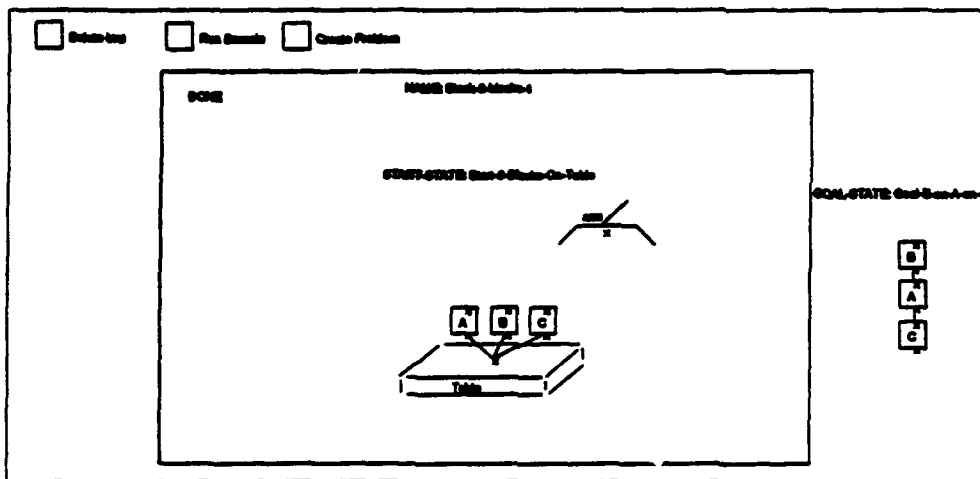
params
  (<bottom> block-model)
  (<top> block-model)
  (<arm> arm-model)
preconds
  (holding <top> <arm>)
  (clear <bottom>)
effects
  (del (holding <top> <arm>))
  (del (clear <bottom>))
  (add (on <top> <bottom>))
  (add (empty-arm <arm>))
  (add (clear <top>))
  
```

Is operator op4 representing?

- 1) Picking up a block off the table
- 2) Putting down a block on the table
- 3) Stacking a block on another
- 4) Unstacking a block off another

AI-Employee S2: 4
Non-Tech S2: 4

Question 7:



START STATE

(clear c)
(clear b)
(clear a)
(empty-arm arm)
(on-table c table)
(on-table b table)
(on-table a table)

GOAL STATE

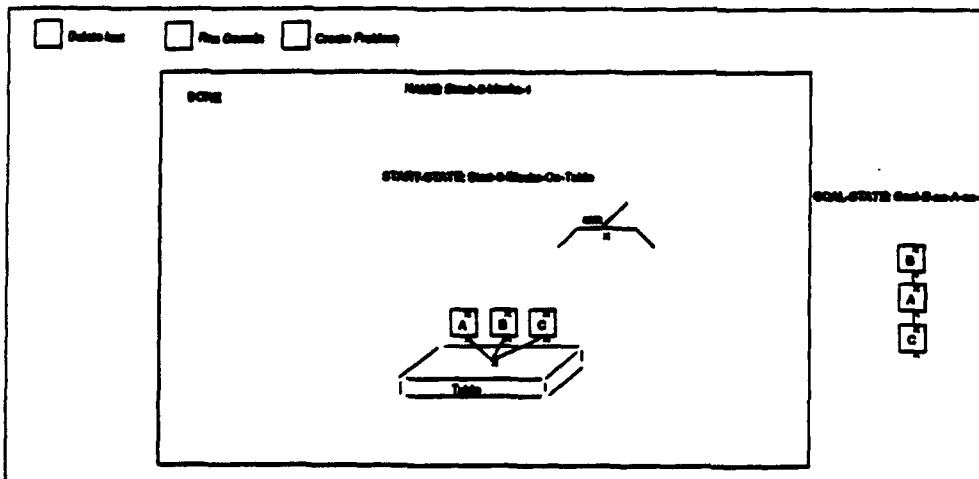
(clear b)
(on a c)
(on b a)

In the goal state of problem stack-3-block-1 what is blockB on top of?

- 1) BlockC
- 2) BlockA
- 3) The Table
- 4) None of the above

Non-Tech S2: 4

Question 8:



START STATE

(clear c)
(clear b)
(clear a)
(empty-arm arm)
(on-table c table)
(on-table b table)
(on-table a table)

GOAL STATE

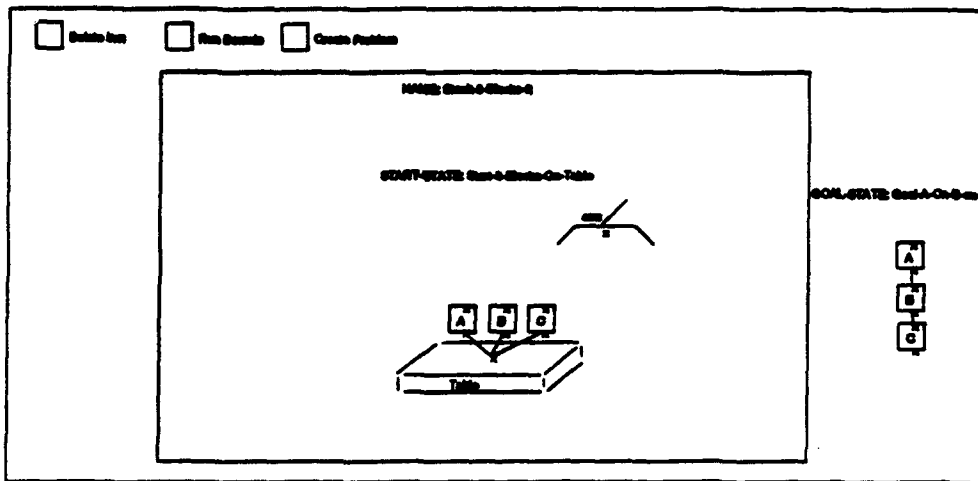
(clear b)
(on a c)
(on b a)

In the start state of problem stack-3-block-1 what is blockB on top of?

- 1) BlockC
- 2) BlockA
- 3) The Table
- 4) None of the above

Non-Tech S2: 4

Question 9:



START STATE

(clear c)
(clear b)
(clear a)
(empty-arm arm)
(on-table c table)
(on-table b table)
(on-table a table)

GOAL STATE

(clear a)
(on b c)
(on a b)

In the goal state of problem stack-3-block-2 what is blockB on top of?

- 1) BlockC
- 2) BlockA
- 3) The Table
- 4) None of the above



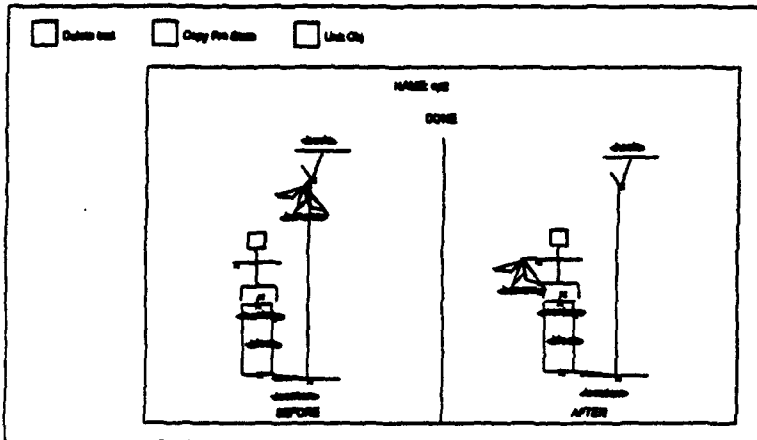
Question 10:

This domain is for?

- 1) Stacking Blocks
- 2) Moving Packages
- 3) Getting bananas

Monkey and Banana Domain

Question 1:



OPERATOR NAME: op2

params

```
(hook hook-model)
(location location-model)
(monkey monkey-model)
(block block-model)
(banana banana-model))
```

preconds

```
(on-block monkey block)
(under-hook location hook)
(block-at-location block location)
(on-ceiling banana hook))
```

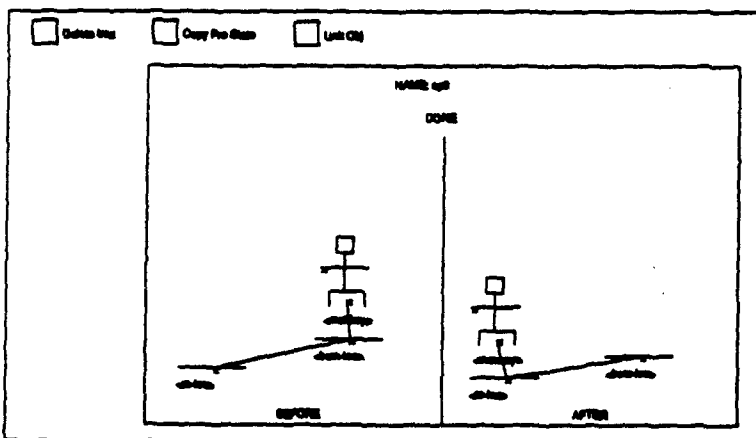
effects

```
(del (on-ceiling banana hook))
(add (holding-banana banana monkey))
```

If op2 has fired is the monkey's hand empty or holding the bananas?

- 1) Holding the bananas
- 2) Empty
- 3) Unknown

Question 2:



OPERATOR NAME: op3

params

```
(<to-loc> location-model)
(<from-loc> location-model)
(monkey monkey-model)
```

preconds

```
(connected to-loc from-loc)
(connected from-loc to-loc)
(monkey-at-location monkey from-loc))
```

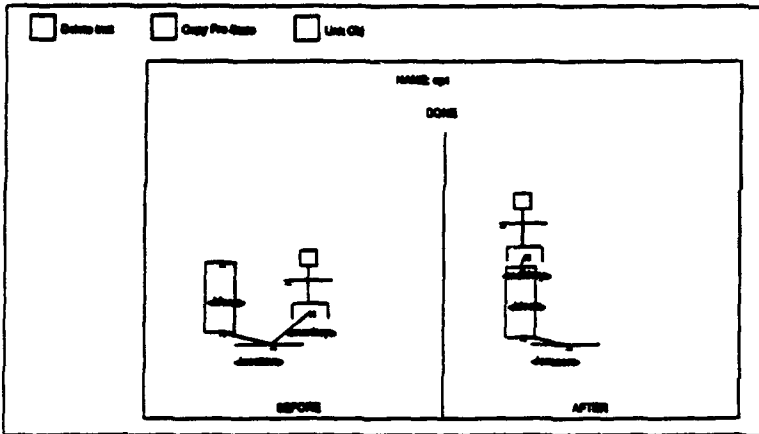
effects

```
(del (monkey-at-location monkey from-loc))
(add (monkey-at-location monkey to-loc))
```

If op3 has fired is the monkey in the same location?

- 1) Same location
- 2) Different location
- 3) Unknown

Question 3:



OPERATOR NAME: op1

params

(<block> block-model)
(<location> location-model)
(<monkey> monkey-model)

preconds

(monkey-at-location <monkey> <location>)
(block-at-location <block> <location>)

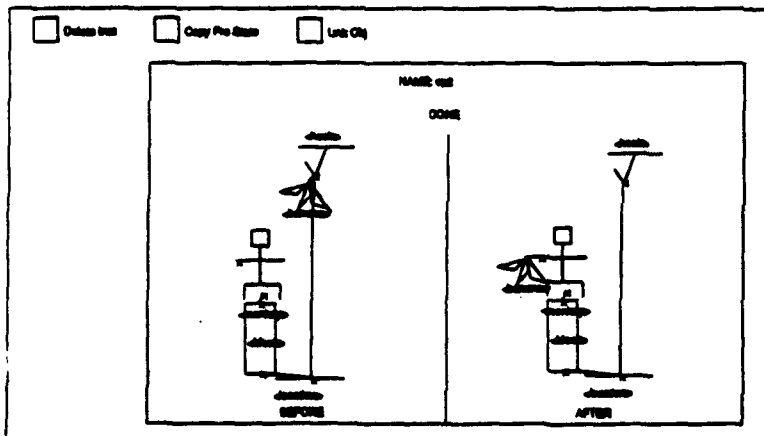
effects

(del (monkey-at-location <monkey> <location>))
(add (on-block <monkey> <block>))

Is operator op1 representing?

- 1) Grabbing the bananas
- 2) Monkey moving himself
- 3) Monkey moving the Block
- 4) **Monkey getting on Block**

Question 4:



OPERATOR NAME: op2

params

(<hook> hook-model)
(<location> location-model)
(<monkey> monkey-model)
(<block> block-model)
(<banana> banana-model))

preconds

(on-block <monkey> <block>)
(under-hook <location> <hook>)
(block-at-location <block> <location>)
(on-ceiling <banana> <hook>))

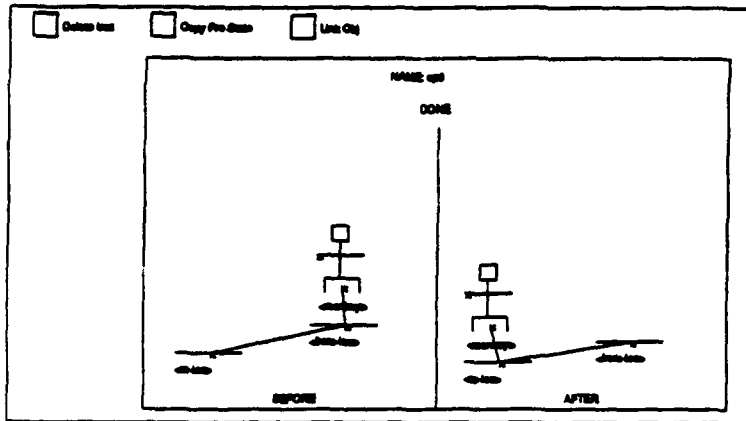
effects

(del (on-ceiling <banana> <hook>))
(add (holding-banana <banana> <monkey>))

Is operator op2 representing?

- 1) **Grabbing the bananas**
- 2) Monkey moving himself
- 3) Monkey moving the Block
- 4) Monkey getting on Block

Question 5:



OPERATOR NAME: op3

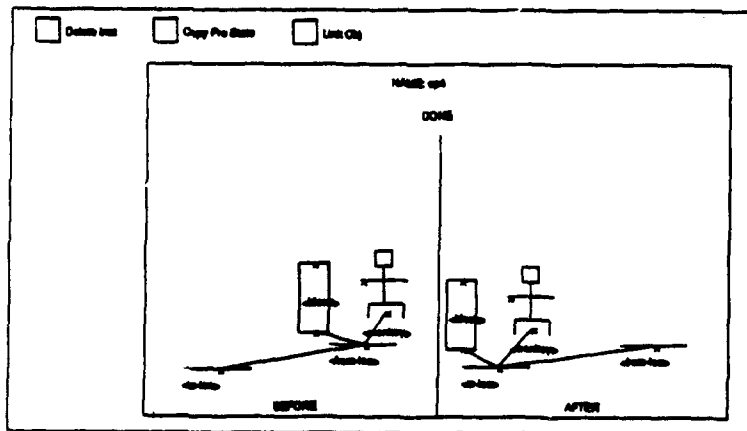
```

params
  (<to-loc> location-model)
  (<from-loc> location-model)
  (<monkey> monkey-model)
preconds
  (connected <to-loc> <from-loc>)
  (connected <from-loc> <to-loc>)
  (monkey-at-location <monkey> <from-loc>)))
effects
  (del (monkey-at-location <monkey> <from-loc>))
  (add (monkey-at-location <monkey> <to-loc>))
  
```

Is operator op3 representing?

- 1) Grabbing the bananas
- 2) **Monkey moving himself**
- 3) Monkey moving the Block
- 4) Monkey getting on Block

Question 6:



OPERATOR NAME: op4

```

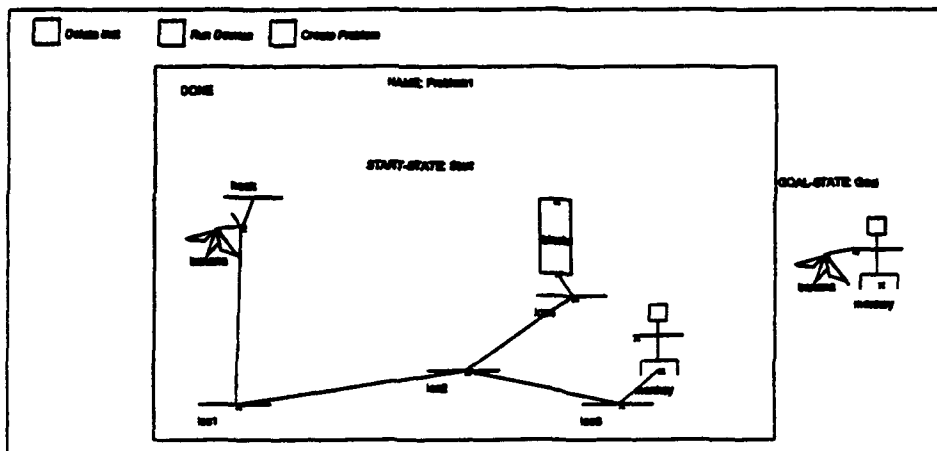
params
  (<to-loc> location-model)
  (<from-loc> location-model)
  (<monkey> monkey-model)
  (<block> block-model)
preconds
  (connected <to-loc> <from-loc>)
  (connected <from-loc> <to-loc>)
  (monkey-at-location <monkey> <from-loc>)
  (block-at-location <block> <from-loc>)))
effects
  (del (monkey-at-location <monkey> <from-loc>))
  (del (block-at-location <block> <from-loc>))
  (add (monkey-at-location <monkey> <to-loc>))
  (add (block-at-location <block> <to-loc>))
  
```

Is operator op4 representing?

- 1) Grabbing the bananas
- 2) Monkey moving himself
- 3) **Monkey moving the Block**
- 4) Monkey getting on Block

Non-Tech S2: 2
Using Graphical
Representation

Question 7:



START STATE

(on-ceiling banana hook)
 (block-at-location block loc4)
 (monkey-at-location monkey loc3)
 (connected loc2 loc4)
 (connected loc2 loc3)
 (connected loc1 loc2)
 (connected loc4 loc2)
 (connected loc3 loc2)
 (connected loc2 loc1)
 (under-hook loc1 hook)

GOAL STATE

(holding-banana banana monkey)

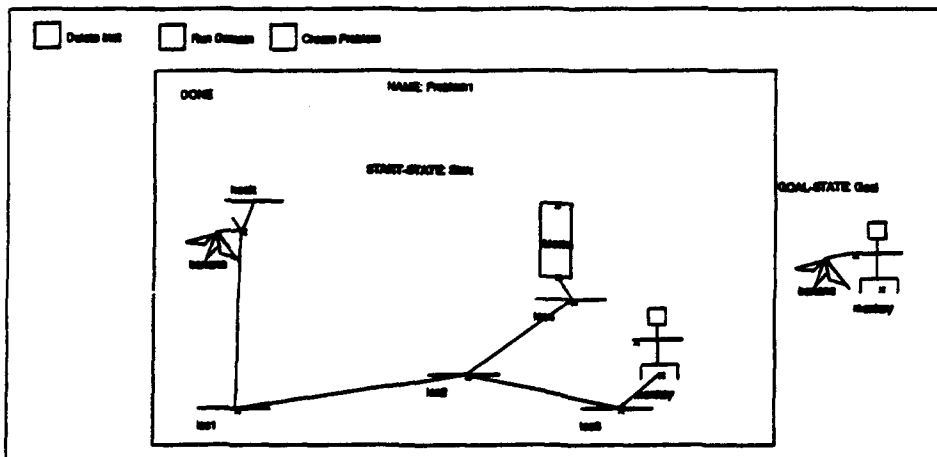
What is the goal of problem1?

- 1) To have the monkey on the block
- 2) To have the monkey at location 3
- 3) To have the monkey holding the bananas

nanas

- 4) None of the above

Question 8:



START STATE

(on-ceiling banana hook)
 (block-at-location block loc4)
 (monkey-at-location monkey loc3)
 (connected loc2 loc4)
 (connected loc2 loc3)
 (connected loc1 loc2)
 (connected loc4 loc2)
 (connected loc3 loc2)
 (connected loc2 loc1)
 (under-hook loc1 hook)

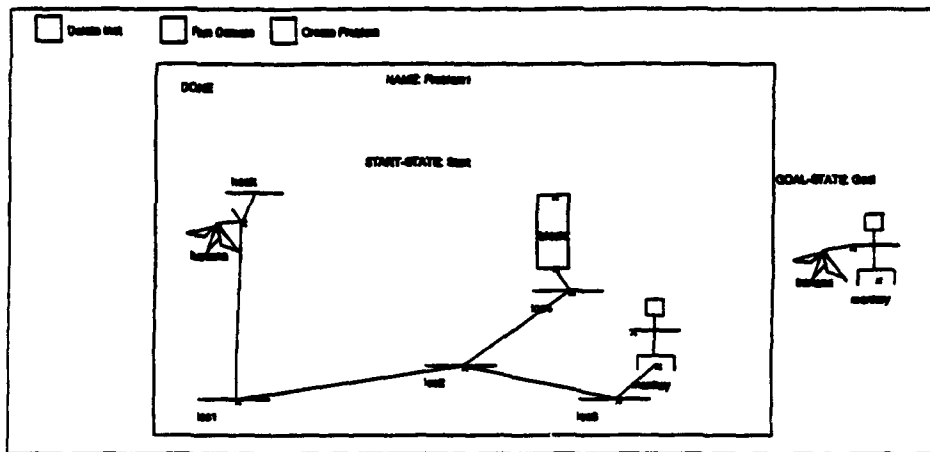
GOAL STATE

(holding-banana banana monkey)

In the start state of problem1 where is the block?

- 1) Loc1
- 2) Loc2
- 3) Loc3
- 4) None of the above

Question 9:



START STATE

(on-ceiling banana hook)
 (block-at-location block loc4)
 (monkey-at-location monkey loc3)
 (connected loc2 loc4)
 (connected loc2 loc3)
 (connected loc1 loc2)
 (connected loc4 loc2)
 (connected loc3 loc2)
 (connected loc2 loc1)
 (under-hook loc1 hook)

GOAL STATE

(holding-banana banana monkey)

In the start state of problem1 where is the monkey?

- 1) Loc1
- 2) Loc2
- 3) **Loc3**
- 4) Loc4

Question 10:

This domain is for?

- 1) Stacking Blocks
- 2) Moving Packages
- 3) **Monkey getting bananas**

Appendix F - Code for Medium Size Domain

```

; DOMAIN TYPE DEFINITIONS
; IS-A domain-object-kind TYPE
; (instance-type number-type #numberp)
; IS-A PART-SIZE TYPE
; IS-A PART-SIZE TYPE
; IS-A HOLE-DRILL TYPE
; IS-A HOLE-SURFACE-CORRECTION TYPE
; IS-A MILLING-CUTTER TYPE
; IS-A FLUTE TYPE
; IS-A SPIN TYPE
; IS-A TAP-SOLE TYPE
; IS-A COUNTER-BORE TYPE
; IS-A COUNTER-SINK TYPE
; IS-A THREA TYPE
; IS-A OIL-HOLE-DRILL TYPE
; IS-A MACHINING TYPE
; IS-A FLAMER MACHINING
; IS-A BOND-SAW MACHINING
; IS-A MILLING-MACHINE MACHINING
; IS-A DRILL MACHINING
; IS-A VISE MACHINING
; IS-A DRILL-BIT TYPE
; IS-A SPIN DRILL-BIT
; IS-A COUNTERSINK DRILL-BIT
; IS-A COUNTERSINK DRILL-BIT
; IS-A TAP DRILL-BIT
; IS-A COUNTER-DRILL DRILL-BIT
; IS-A GUN-DRILL DRILL-BIT
; IS-A HOLE-HELI-DRILL DRILL-BIT
; IS-A STRAIGHT-FLUTED-DRILL DRILL-BIT
; IS-A COARSE-DRILL DRILL-BIT
; IS-A SPOT-DRILL DRILL-BIT
; IS-A TWIST-DRILL DRILL-BIT
; IS-A PART TYPE
; IS-A SPOT TYPE
; IS-A HOLE TYPE
; IS-A BOND-SAW BLADE TYPE
; IS-A BOND-SAW BOND-SAW-BLADE
; IS-A BOND-FILE BOND-SAW-BLADE
; IS-A COUNTER-SINK TYPE

```

; OPERATIONS IN DOMAIN

```

(operator put-in-drill-spindle
  (params ((<drill175> drill) (<bit756> drill-bit)
            (<table404> table)))
  (preconditions
    (and (is-available-tool-holder <drill175>)
          (is-available-tool <bit756>)
          (on-table <bit756> <table404>)))
  (effects
    ((del (is-available-tool-holder <drill175>))
     (del (is-available-tool <bit756>))
     (del (on-table <bit756> <table404>))
     (add (holding-tool <bit756> <drill175>)))))

(operator put-holding-device-in-drill
  (params ((<drill181> drill) (<vise813> vise)
            (<table404> table)))
  (preconditions
    (and (is-available-table <drill181>)
          (is-empty-holding-vise <vise813>)
          (on-table <vise813> <table404>)))
  (effects
    ((del (is-available-table <drill181>))
     (del (on-table <vise813> <table404>))
     (add (has-device <vise813> <drill181>)))))

(operator clean-part
  (params ((<part1237> part)))
  (preconditions (is-available-part <part1237>))
  (effects ((add (clean <part1237>)))))

(operator hold-with-vise
  (params ((<vise123> vise) (<part123> part)
            (<table258> table)))
  (preconditions
    (and (clean <part123>)
          (on-table <part123> <table258>)
          (is-available-part <part123>)
          (is-empty-holding-vise <vise123>)))
  (effects
    ((del (on-table <part123> <table258>))
     (del (is-available-part <part123>))
     (del (is-empty-holding-vise <vise123>))
     (add (holding <part123> <vise123>)))))

(operator remove-drill-from-spindle
  (params ((<drill1404> drill) (<bit404> drill-bit)

```

```

            (<table404> table)))
  (preconditions (holding-tool <bit404> <drill1404>))
  (effects
    ((del (holding-tool <bit404> <drill1404>))
     (add (is-available-tool-holder <drill1404>))
     (add (is-available-tool <bit404>))
     (add (on-table <bit404> <table404>)))))

(operator remove-holding-device-in-drill
  (params ((<drill1425> drill) (<table425> table)
            (<vise425> vise)))
  (preconditions (and (has-device <vise425> <drill1425>)
                      (is-empty-holding-vise <vise425>)))
  (effects
    ((del (has-device <vise425> <drill1425>))
     (add (is-available-table <drill1425>))
     (add (on-table <vise425> <table425>)))))

(operator release-from-vise
  (params ((<vise4312> vise) (<part4312> part)
            (<table4312> table)))
  (preconditions (holding <part4312> <vise4312>))
  (effects
    ((del (holding <part4312> <vise4312>))
     (add (on-table <part4312> <table4312>))
     (add (is-available-part <part4312>))
     (add (is-empty-holding-vise <vise4312>)))))

(operator drill-with-spot-drill
  (params
    ((<side> part-side) (<loc> number-type) (<locy> number-type)
     (<spot12647> spot) (<drill19515> drill) (<vise9515> vise)
     (<part9515> part) (<drill19515> spot-drill)))
  (preconditions
    (and (clean <part9515>)
          (holding <part9515> <vise9515>)
          (has-device <vise9515> <drill19515>)
          (holding-tool <drill19515> <drill19515>)))
  (effects
    ((del (clean <part9515>))
     (add (spot-location <spot12647> <side> <loc> <locy>))
     (add (has-burns <part9515>))
     (add (has-spot <spot12647> <part9515>)))))

(operator drill-with-straight-fluted-drill
  (params
    ((<depth> number-type) (<hole2480> hole) (<drill12376> drill)
     (<ch> number-type) (<ch-f-2376> straight-fluted-drill)
     (<vise2402> vise) (<part2405> part) (<side> part-side)
     (<loc> number-type) (<locy> number-type) (<spot2483> spot)))
  (preconditions
    (and (spot-location <spot2483> <side> <loc> <locy>)
          (clean <part2405>)
          (material-of <part2405> brass)
          (diameter <ch-f-2376> <ch>)
          (holding <part2405> <vise2402>)
          (has-device <vise2402> <drill12376>)
          (has-spot <spot2483> <part2405>)
          (holding-tool <ch-f-2376> <drill12376>)))
  (effects
    ((del (spot-location <spot2483> <side> <loc> <locy>))
     (del (clean <part2405>))
     (del (has-spot <spot2483> <part2405>))
     (add (hole-location <hole2480> <side> <depth> <ch> <loc> <locy>))
     (add (has-burns <part2405>))
     (add (has-hole <hole2480> <part2405>)))))

(operator drill-with-twist-drill
  (params
    ((<depth> number-type) (<hole> hole) (<drill> drill)
     (<vise> vise) (<part> part) (<side> part-side)
     (<loc> number-type) (<locy> number-type) (<spot> spot)
     (<ch> number-type) (<ch> twist-drill)))
  (preconditions
    (and (diameter <ch> <ch>)
          (spot-location <spot> <side> <loc> <locy>)
          (clean <part>)
          (holding <part> <vise>)
          (has-device <vise> <drill>)
          (has-spot <spot> <part>)
          (holding-tool <ch> <drill>)))
  (effects
    ((del (spot-location <spot> <side> <loc> <locy>))
     (del (clean <part>))
     (del (has-spot <spot> <part>))
     (add (hole-location <hole> <side> <depth> <ch> <loc> <locy>))
     (add (has-burns <part>)) (add (has-hole <hole> <part>)))))

```

```

(operator drill-with-cap
  (params
    ((<cap4973> tap-hole) (<drill4944> drill) (<cap4944> tap)
      (<vise4944> vise) (<side> part-side) (<depth> number-type)
      (<ch> number-type) (<close> number-type) (<loop> number-type)
      (<hole> hole) (<part4944> part)))
  (preconds
    (and (clean <part4944>)
      (hole-location <hole> <side> <depth> <ch> <close> <loop>)
      (diameter <cap4944> <ch>)
      (holding-tool <cap4944> <drill4944>)
      (holding <part4944> <vise4944>)
      (has-device <vise4944> <drill4944>)
      (has-hole <hole> <part4944>)))
  (effects
    ((del (clean <part4944>))
      (del (hole-location <hole> <side> <depth> <ch> <close> <loop>))
      (del (has-hole <hole> <part4944>))
      (add (is-capped <cap4973> <hole> <side> <depth> <ch> <close> <loop>))
      (add (has-burrs <part4944>))
      (add (has-cap <cap4973> <part4944>))))))

```

```

(operator drill-with-o-b
  (params
    ((<angle> number-type) (<ch> <is5044> counter-bore)
      (<drill5027> drill) (<part5043> part)
      (<vise5043> vise) (<ch> <is5044> counter-bore) (<side> part-side)
      (<depth> number-type) (<ch> number-type) (<close> number-type)
      (<loop> number-type) (<hole> hole)))
  (preconds
    (and (hole-location <hole> <side> <depth> <ch> <close> <loop>)
      (diameter <ch> <is5044> <ch>)
      (clean <part5043>)
      (holding-tool <ch> <is5044> <drill5027>)
      (holding <part5043> <vise5043>)
      (has-device <vise5043> <drill5027>)
      (has-hole <hole> <part5043>)))
  (effects
    ((del (hole-location <hole> <side> <depth> <ch> <close> <loop>))
      (del (clean <part5043>))
      (del (has-hole <hole> <part5043>))
      (add (is-counter-bore <ch> <hole> <side> <depth> <ch> <close> <loop> <angle>))
      (add (has-burrs <part5043>))
      (add (has-o-b <ch> <is5044> <part5043>))))))

```

```

(operator drill-with-o-s
  (params
    ((<angle> number-type) (<ch> <is5071> counter-sink)
      (<drill5061> drill) (<vise5061> vise)
      (<is5061> counter-sink) (<part5063> part) (<side> part-side)
      (<depth> number-type) (<ch> number-type) (<close> number-type)
      (<loop> number-type) (<hole> hole)))
  (preconds
    (and (hole-location <hole> <side> <depth> <ch> <close> <loop>)
      (clean <part5063>)
      (diameter <ch> <is5061> <ch>)
      (holding-tool <ch> <is5061> <drill5061>)
      (holding <part5063> <vise5061>)
      (has-device <vise5061> <drill5061>)
      (has-hole <hole> <part5063>)))
  (effects
    ((del (hole-location <hole> <side> <depth> <ch> <close> <loop>))
      (del (clean <part5063>))
      (del (has-hole <hole> <part5063>))
      (add (is-counter-sink <ch> <hole> <side> <depth> <ch> <close> <loop> <angle>))
      (add (has-burrs <part5063>))
      (add (has-o-s <ch> <is5071> <part5063>))))))

```

```

(operator put-fluid-on-part
  (params
    ((<part5088> part) (<fluid5088> fluid) (<vise5088> vise)
      (<machine5088> machine) (<table4154> table)))
  (preconds
    (and (clean <part5088>)
      (fluid-on-table <fluid5088> <table4154>)
      (has-device <vise5088> <machine5088>)
      (holding <part5088> <vise5088>)))
  (effects
    ((del (fluid-on-table <fluid5088> <table4154>))
      (add (has-fluid <fluid5088> <part5088>))))))

```

```

(operator drill-with-runner
  (params
    ((<ream5103> ream) (<drill5091> drill) (<vise5091> vise)
      (<part5091> part) (<runner5091> runner) (<fluid5091> fluid)
      (<side> part-side) (<depth> number-type) (<ch> number-type)
      (<close> number-type) (<loop> number-type) (<hole> hole)))
  (preconds
    (and (hole-location <hole> <side> <depth> <ch> <close> <loop>)
      (diameter <ream5091> <ch>)
      (clean <part5091>)))

```

```

      (has-fluid <fluid5091> <part5091>)
      (holding-tool <ream5091> <drill5091>)
      (holding <part5091> <vise5091>)
      (has-device <vise5091> <drill5091>)
      (has-hole <hole> <part5091>)))
  (effects
    ((del (hole-location <hole> <side> <depth> <ch> <close> <loop>))
      (del (clean <part5091>))
      (del (has-fluid <fluid5091> <part5091>))
      (del (has-hole <hole> <part5091>))
      (add (is-reamed <ream5103> <hole> <side> <depth> <ch> <close> <loop>))
      (add (has-burrs <part5091>))
      (add (has-ream <ream5103> <part5091>))))))

```

```

(operator drill-with-oil-drill
  (params
    ((<depth> number-type) (<ch> number-type) (<hole6845> hole)
      (<drill6838> drill) (<ch> <is6838> oil-hole-drill)
      (<side> part-side) (<close> number-type) (<loop> number-type)
      (<spot6838> spot) (<vise6838> vise) (<part6838> part)
      (<fluid6838> fluid)))
  (preconds
    (and (clean <part6838>)
      (spot-location <spot6838> <side> <close> <loop>)
      (has-fluid <fluid6838> <part6838>)
      (holding-tool <ch> <is6838> <drill6838>)
      (holding <part6838> <vise6838>)
      (has-device <vise6838> <drill6838>)
      (has-spot <spot6838> <part6838>)))
  (effects
    ((del (clean <part6838>))
      (del (spot-location <spot6838> <side> <close> <loop>))
      (del (has-fluid <fluid6838> <part6838>))
      (del (has-spot <spot6838> <part6838>))
      (add (hole-location <hole6845> <side> <depth> <ch> <close> <loop>))
      (add (has-burrs <part6838>))
      (add (has-hole <hole6845> <part6838>))))))

```

```

(operator put-blade-in-band-saw
  (params ((<band_saw7007> band_saw) (<table7007> table)
    (<is7007> band-saw-blade)))
  (preconds
    (and (band-saw-available-for-blade <band_saw7007>)
      (is-available-band-saw <is7007>)
      (on-table-blade-saw <is7007> <table7007>)))
  (effects
    ((del (band-saw-available-for-blade <band_saw7007>))
      (del (is-available-band-saw <is7007>))
      (del (on-table-blade-saw <is7007> <table7007>))
      (add (holding-band-saw <is7007> <band_saw7007>))))))

```

```

(operator take-blade-out-of-band-saw
  (params ((<band_saw7014> band_saw) (<is7014> band-saw-blade)
    (<table7015> table)))
  (preconds (holding-band-saw <is7014> <band_saw7014>))
  (effects
    ((del (holding-band-saw <is7014> <band_saw7014>))
      (add (band-saw-available-for-blade <band_saw7014>))
      (add (is-available-band-saw <is7014>))
      (add (on-table-blade-saw <is7014> <table7015>))))))

```

```

(operator put-part-on-band-saw
  (params
    ((<band_saw7019> band_saw) (<is7019> band-saw-blade)
      (<table7020> table) (<part7020> part)))
  (preconds
    (and (holding-band-saw <is7019> <band_saw7019>)
      (is-available-part <part7020>)
      (on-table <part7020> <table7020>)))
  (effects
    ((del (on-table <part7020> <table7020>))
      (add (part-on-band-saw <part7020> <band_saw7019>))))))

```

```

(operator take-part-off-band-saw
  (params
    ((<band_saw7030> band_saw) (<is7030> band-saw-blade)
      (<part7031> part) (<table7032> table)))
  (preconds
    (and (part-on-band-saw <part7031> <band_saw7030>)
      (holding-band-saw <is7030> <band_saw7030>)
      (is-available-part <part7031>)))
  (effects
    ((del (part-on-band-saw <part7031> <band_saw7030>))
      (add (on-table <part7031> <table7032>))))))

```

```

(operator cut-with-band-saw
  (params
    ((<is-size> part-size) (<band_saw7040> band_saw)
      (<is> part-side) (<condi> part-surface-condition)

```

```

(set-up-for <mill13940> <->)
(holding-drill-bit-in-mill <-d13943> <mill13940>)
(holding-vice-in-mill <visual3943> <mill13940>)
(holding <part13943> <visual3943>)
(hse-spot <spot13948> <part13943>)
(is-available-tool <-d13943>)))

(effects
  (del (spot-location <spot13948> <side> <down> <down>))
  (del (clean <part13943>))
  (del (hse-spot <spot13948> <part13943>))
  (add (hole-location <hole13948> <side> <up> <-> <down> <down>))
  (add (hse-burr <part13943>))
  (add (hse-hole <hole13948> <part13943>))))))

(operator mill-spot-drill
  (params
    ((<side> part-side) (<down> number-type) (<down> number-type)
     (<spot13978> spot) (<-> mill-drill)
     (<mill13960> milling_machine) (<-d13960> spot-drill)
     (<part13960> part) (<visual3960> vise)))
  (preconds
    (and (clean <part13960>)
          (set-up-for <mill13960> <->)
          (holding-drill-bit-in-mill <-d13960> <mill13960>)
          (holding-vice-in-mill <visual3960> <mill13960>)
          (holding <part13960> <visual3960>)
          (is-available-tool <-d13960>)))
  (effects
    (del (clean <part13960>))
    (add (spot-location <spot13978> <side> <down> <down>))
    (add (hse-burr <part13960>))
    (add (hse-spot <spot13978> <part13960>))))))

(operator put-milling-cutter-in-mill
  (params ((<table14009> table) (<a_c14010> milling_cutter)
           (<mill14010> milling_machine)))
  (preconds
    (and (set-up-for <mill14010> milling)
          (milling-cutter-on-table <a_c14010> <table14009>)
          (not-holding-drill-bit-in-mill <mill14010>)))
  (effects
    (del (milling-cutter-on-table <a_c14010> <table14009>))
    (del (not-holding-drill-bit-in-mill <mill14010>))
    (add (milling-cutter-on-mill <a_c14010> <mill14010>))))))

(operator remove-milling-cutter-from-mill
  (params ((<table14018> table) (<mill14018> milling_machine)
           (<a_c14018> milling_cutter)))
  (preconds
    (and (milling-cutter-on-mill <a_c14018> <mill14018>)
          (not-holding-vice-in-mill <mill14018>)))
  (effects
    (del (milling-cutter-on-mill <a_c14018> <mill14018>))
    (add (milling-cutter-on-table <a_c14018> <table14018>))
    (add (not-holding-drill-bit-in-mill <mill14018>))))))

(operator mill-part
  (params
    ((<size> part-size) (<-> mill-drill)
     (<mill14023> milling_machine) (<visual4023> vise)
     (<co> part-side) (<cond1> part-surface-condition)
     (<size> dimension-size) (<size1> part-size)
     (<part14023> part) (<a_c14023> milling_cutter)))
  (preconds
    (and (surface-condition <part14023> <co> <cond1>)
          (dimensions <part14023> <size> <size1>)
          (set-up-for <mill14023> <->)
          (milling-cutter-on-mill <a_c14023> <mill14023>)
          (holding-vice-in-mill <visual4023> <mill14023>)
          (holding <part14023> <visual4023>)))
  (effects
    (del (surface-condition <part14023> <co> <cond1>))
    (del (dimensions <part14023> <size> <size1>))
    (add (surface-condition <part14023> <co> <rough>))
    (add (dimensions <part14023> <size> <size2>))))))

(operator put-vice-in-planer
  (params ((<table14104> table) (<visual4105> vise)
           (<planer14105> planer)))
  (preconds (and (on-table <visual4105> <table14104>)
                 (is-empty-holding-vice <visual4105>)))
  (effects
    (del (on-table <visual4105> <table14104>))
    (add (holding-vice-in-planer <visual4105> <planer14105>))))))

(operator remove-vice-from-planer
  (params ((<table14109> table) (<planer14109> planer)
           (<visual4109> vise)))
  (preconds
    (and (holding-vice-in-planer <visual4109> <planer14109>))

```

```

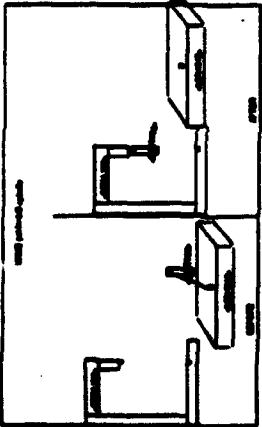
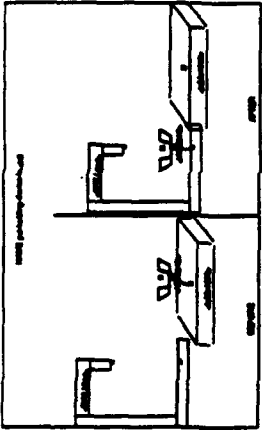
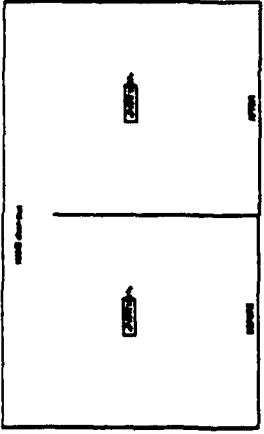
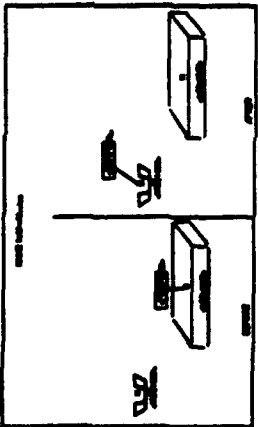
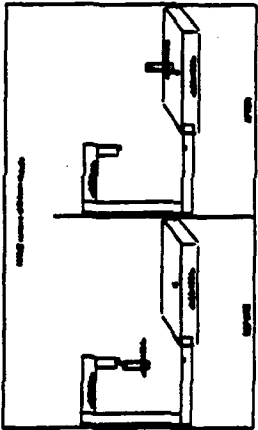
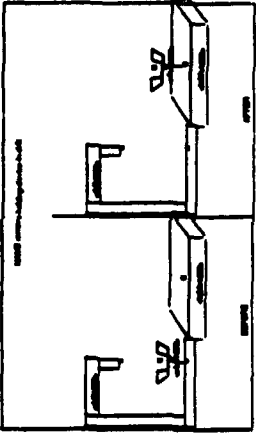
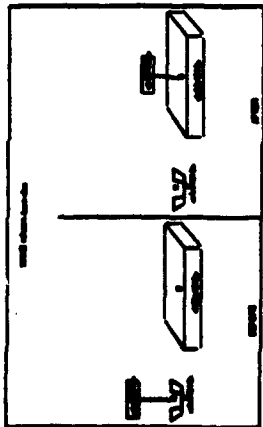
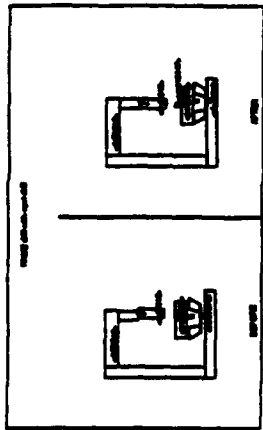
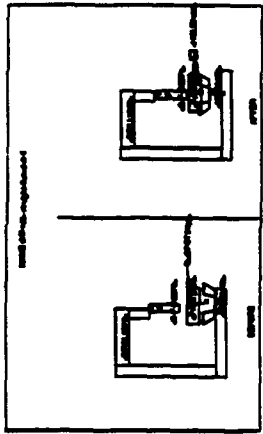
(is-empty-holding-vise <vise14109>))
(effects
  (del (holding-vise-in-planer <vise14109> <planer14109>))
  (add (on-table <vise14109> <table14109>))))))

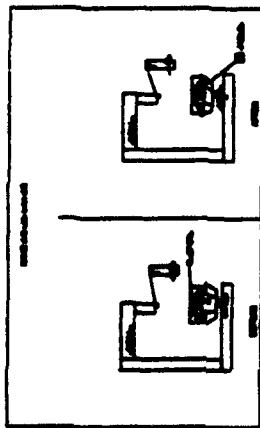
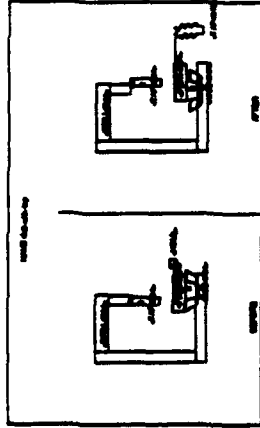
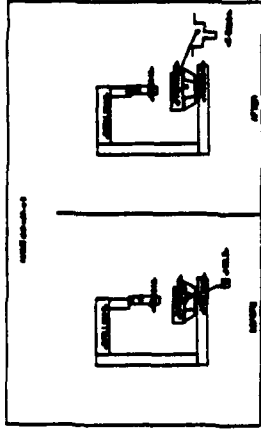
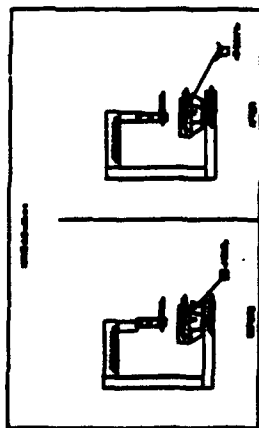
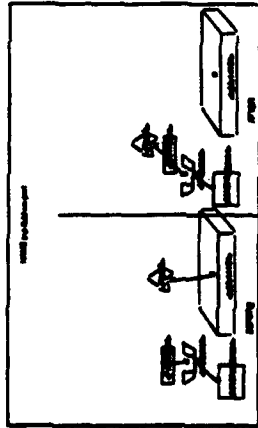
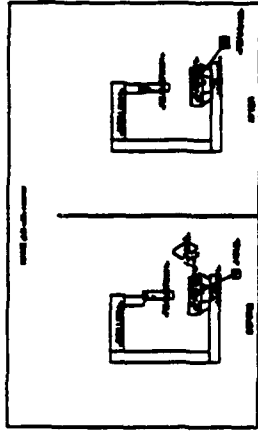
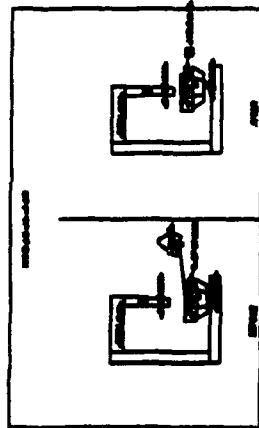
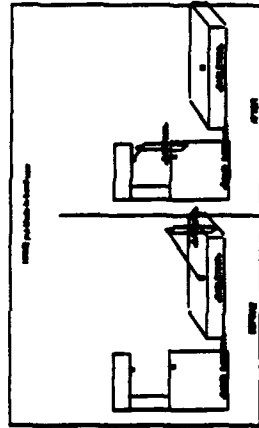
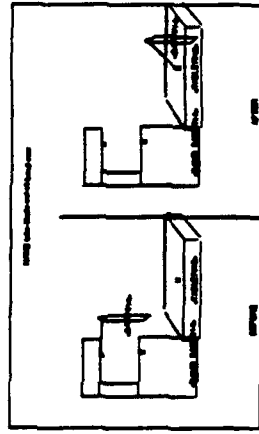
(operator plane-part
  (params
    ((<cut-size> part-size) (<planer14114> planer)
     (<vise14114> vise) (<obj> part-side)
     (<comb> part-surface-condition) (<dim> dimension-side)
     (<size> part-size) (<part14115> part)))
  (preconds
    (and (surface-condition <part14115> <obj> <comb>)
         (dimensions <part14115> <dim> <size>)
         (holding-vise-in-planer <vise14114> <planer14114>)
         (holding <part14115> <vise14114>)))
  (effects
    (del (surface-condition <part14115> <obj> <comb>))
    (del (dimensions <part14115> <dim> <size>))
    (add (surface-condition <part14115> <obj> rough))
    (add (dimensions <part14115> <dim> <cut-size>))))))

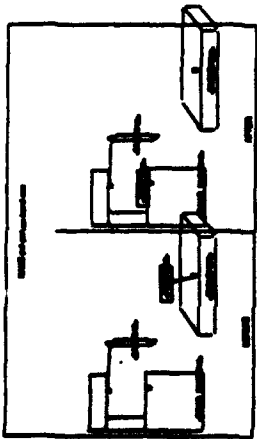
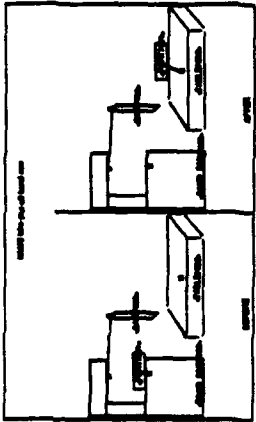
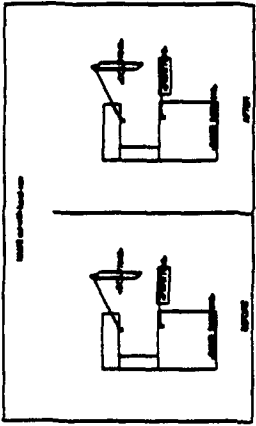
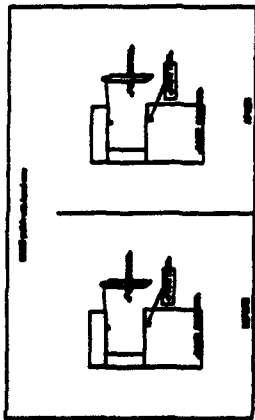
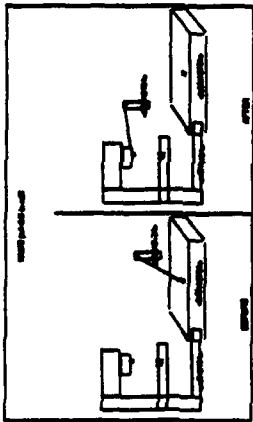
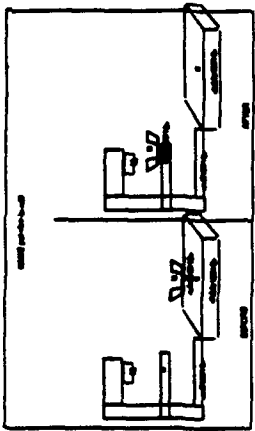
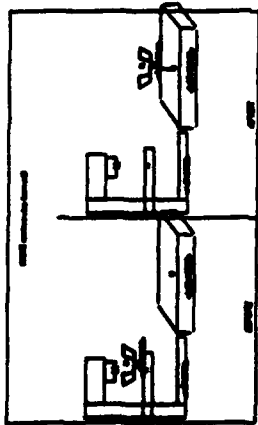
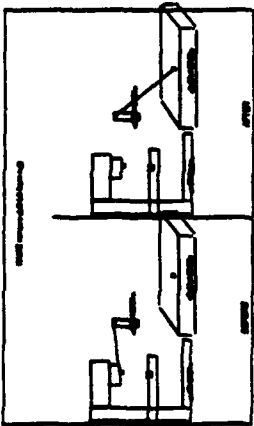
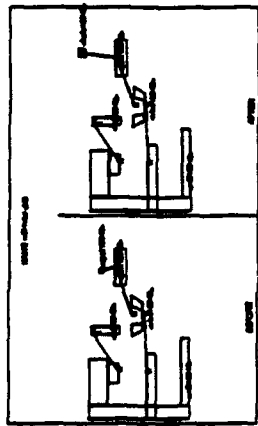
(operator set-up-mill-for-milling
  (params ((<obj> mill-drill) (<mill30262> milling_machine)))
  (preconds (set-up-for <mill30262> <obj>))
  (effects ((del (set-up-for <mill30262> <obj>))
            (add (set-up-for <mill30262> milling))))))

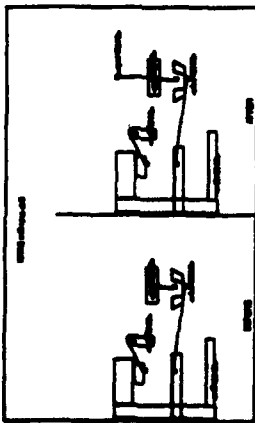
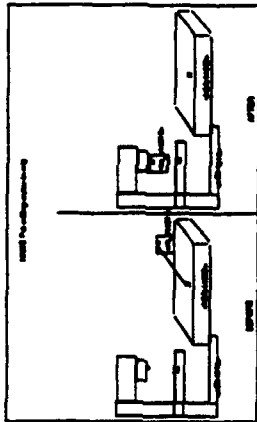
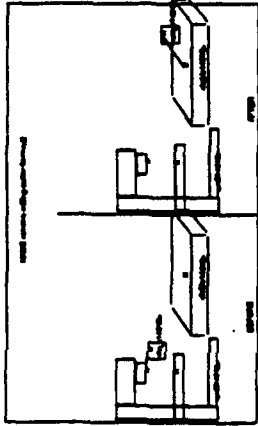
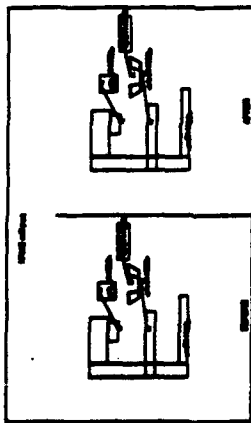
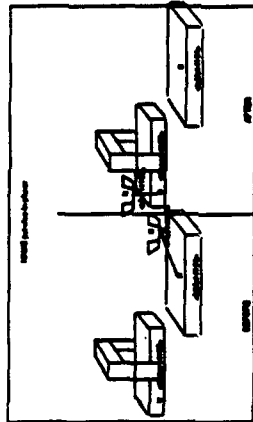
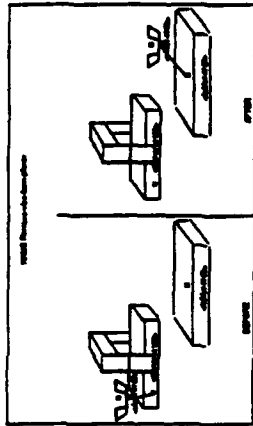
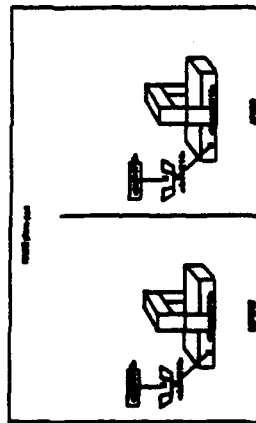
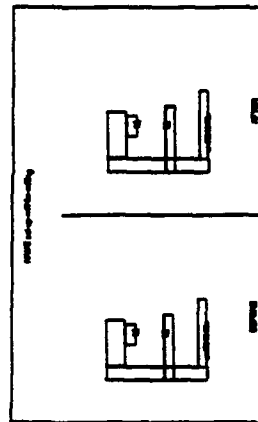
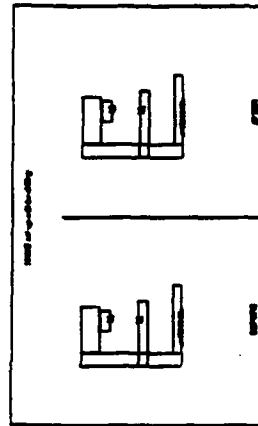
(operator set-up-mill-for-drilling
  (params ((<obj> mill-drill) (<mill30263> milling_machine)))
  (preconds (set-up-for <mill30263> <obj>))
  (effects ((del (set-up-for <mill30263> <obj>))
            (add (set-up-for <mill30263> drilling))))))

```









Appendix G - Learning Study Domains

Hiking World Description

In this domain a person is hiking with a backpack from one campground to another campground. The objects in this domain are: backpack, person, and campgrounds. Relations between objects are as follows: 1) a person can hold a backpack; 2) a person can be at a campground; 3) a backpack can be at a campground; and 4) two campgrounds can be connected to each other. The actions that are performed in this domain are: 1) a person can pickup a backpack; 2) a person can put down a backpack; and 3) a person can move from one campground to another connected campground.

Task: *Define an initial state, goal state, and domain. Have the system find the plan needed to go from initial to goal state of the following problems.*

Problem 1: For the initial state create two campgrounds (camp1, camp2), a person, and a backpack. Camp1 and camp2 are connected together. Have the person and backpack start at camp1. The goal is to have the backpack sitting at camp2.

Problem 2: Use the work from the previous problem. For the initial state create three campgrounds (camp1, camp2, camp3), a person, and a backpack. Camp1 is connected to camp2 and camp2 is connected to camp3. Have the person start at camp1 and the backpack at camp2. This time the goal state is to have the backpack sitting at camp3.

Loading Truck World Description

In this domain a person will be loading a truck with packages from a warehouse. The objects in this domain are: truck, warehouse, person and packages. The relations that exist between the objects are: 1) a package can be on the truck; 2) a package can be in the warehouse; 3) a person can be on the truck; and 4) a person can be in the warehouse. The actions that can be performed in this domain are: 1) pick a package up at the warehouse; 2) Take a package from the warehouse to the truck; 3) put a package in the truck; and 4) Go from the truck back to the warehouse.

Task: *Define an initial state, goal state, and domain. Have the system find the plan needed to go from initial to goal state of the following problems.*

Problem 1: For the initial state create a truck, a package, a warehouse, and a person. The person is at the warehouse along with the package. The goal is to have the package loaded onto the truck.

Problem 2: Use the work from the previous problem. For the initial state this time have the person start out at the truck and the package at the warehouse. The goal is again to have the package loaded onto the truck.

Robot Picking Tulip Description

In this domain a robot is going around to different locations and picking up tulips. The objects in this domain are: robot with a basket, tulips, and locations. Relations between objects are as follows: 1) a robot with a basket can hold a tulip in the basket; 2) a robot with a basket can be at a location; 3) a tulip can be at a location; and 4) two locations can be connected to each other. The actions that are performed in this domain are: 1) a robot with a basket can pickup a tulip and put it in a basket; and 2) a robot can move from one location to another connected location.

Task: *Define an initial state, goal state, and domain. Have the system find the plan needed to go from initial to goal state of the following problems.*

Problem 1: For the initial state create two location (Loc1, Loc2), a robot, and a tulip. Loc1 and Loc2 are connected together. Have the robot start at Loc1 and the tulip start at Loc2. The goal is to have the robot hold the tulip in its basket.

Problem 2: Use the work from the previous problem. For the initial state create three locations (Loc1, Loc2, Loc3), a robot, and two tulips. Loc1 is connected to Loc2 and Loc2 is connected to Loc3. Have the robot at Loc1, a tulip at Loc2, and a tulip at Loc3. This time the goal state is to have the robot holding two tulips in its basket.