AN OBJECT-ORIENTED PLANNING ENVIRONMENT

Technical Report CSE-91-10

Ivory J. Griskell

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

May 20, 1991

A Technical Report

on

AN OBJECT-ORIENTED PLANNING ENVIRONMENT

May 20, 1991

Ivory J. Griskell

Department of

Computer Science and Engineering

Auburn University

Auburn University, AL 36849

TABLE OF CONTENTS

I. INTRODUCTION

The problem/task to be addressed in this technical report is an object-oriented planning environment. In order to talk about what is meant by an object-oriented planning environment, planning should initially be discussed.

Planning is a process of formulating a sequence of actions needed to solve a specific problem. A planning environment is an environment in which specific plans can be generated, implemented, and/or reformulated.

There are four aspects to be considered when developing a planning environment. The first of these is the representation of the world (or environment). Proper consideration should be given to the configuration of the background, the objects that can be moved around or worked on along with their characteristics, as well as the agents that can perform the actions along with their capabilities.

Secondly, we must consider the representation of the actions that are to be performed by specific agents. The effects, either wanted or not wanted, the preconditions that must be satisfied before an action can be actually applied, and any other associated knowledge must be represented as actions in order to become a part of the planning environment.

The third aspect is the representation of goals. This representation is accomplished by creating a conjunction of subgoals.

Finally, a reasoning or search mechanism must be devised.

This involves approaches to search for the specified actions that are applicable in the world and to achieve the initial goals.

To date, at Auburn University, there has been little or no attempt to use an object-oriented Lisp programming approach to planning. As the evolution of Lisp continues, an attempt to explore object-oriented programming should be undertaken.

II. LITERATURE REVIEW

Before beginning to design a planning environment, one must have a basic knowledge of each of the elements and concepts needed in its creation. The basic concepts needed to design a planning environment are discussed. In subsequent sections, the elements and actual creation of a planning environment will be further discussed.

Robotic Planning

Research on robotic planning has led to many of our ideas about problem solving systems. In the typical formulation of a robot plan, we have a robot that has a repertoire of primitive actions that it can perform in its domain.

Robotic planning involves several intricate steps. Among these are function integration, including perception of the world around it, formulating plans of action and monitoring the execution of these plans. Thus, we are now confronted with the problem that arises when given some initial situation, how to synthesize a sequence of robot actions that will achieve some stated goal. [NIL80].

Problem-Solving

Problem-solving in AI currently reduces to states, or models, of the problem within the computer, operators to manipulate the states, and a control strategy that applies the operators to produce a problem solution. Two general methods used to represent problems are state-space representation and problem-reduction representation. State-space representation divides a problem into states, operators, and goals [STA87].

Problem-reduction representation reduces a problem into subproblems whose solutions solve the original problem.

Both state-space and problem-reduction networks are represented by graphs called search trees. State-space trees are basically OR trees that solve problems using forward reasoning, from the initial state to a goal state. Problem-reduction representation reduces to AND/OR trees that solve problems using backward reasoning, breaking the goal down to a set of primitive problems whose solutions solve the goal state [STE81].

There are two ways to define search trees: implicitly and explicitly. Search trees for which a diagram is used for definition purposes are called explicit. Explicit trees are easy for us to visualize, but difficult for a computer to work with. Even simple problems in AI have very large, almost infinite search trees. This causes a requirement of vast amounts of memory to store the tree. For this reason, most search trees used in AI programs are defined implicitly. An implicit definition for a

search tree is a rule or mathematical relationship that is used to generate the tree.

As the search for a solution proceeds through the tree and progresses, the search tree grows. This tree searching should attempt to accomplish three tasks:

1. Always find a solution to the problem if one exists.
2. Always find the best solution to the problem.
3. Always find the most efficient solution in terms of computer time and memory space.

[STA87]

Tree-searching techniques fit into two general categories: blind searches and heuristic searches. In a blind search, the expansion of the tree nodes is more or less arbitrary. Heuristic searches utilize additional knowledge about the properties of a problem to control the search process. Consequently, heuristic searching, sometimes called ordered searching, reduces and focuses the search. As a result, heuristic searches are much more efficient for solving complex problems than blind searches [STE81].

Searching techniques that fit into the two general search categories include breadth-first, depth-first, progressive-deepening, minimum-cost, hill-climbing, ordered depth-first, difference-reduction, and best-first searches [STE81].

Hierarchical Planning

The essence of hierarchical planning is the use of different levels of abstraction both in the planning process and in the description of the domain. "An abstraction level is distinguished by the granularity, or fineness of detail, of the discriminations it makes in the world" [STE81]. From a somewhat formal standpoint, a more abstract description (in whatever formalism is being used) will have a larger set of possible world-states that satisfy it. When less abstract descriptions are added, the size of this satisfying set diminishes as things in the world are discriminated against in increasingly finer detail. In complex worlds, abstract descriptions can often be idealizations. This means that a plan realizable at an abstract level may not be realizable in a finer grain. For example, one might ignore friction in an abstraction of the domain, but find that the effects of friction are included in the world description [SAC74].

To see how hierarchical planning can help to avoid the combinatorial explosion involved in reasoning about primitive actions, consider planning to build a house. At the highest abstraction level might be such steps as site preparation and foundation laying. The planner can plan sequences of these steps without considering the detailed actions of hammering a nail or opening a bag of cement. Each of these steps can be expanded into more detailed actions, finally getting down to the level of nail driving, but with the abstract plan eliminating all but a few of

the possible nail drivings at each point where the plan could drive a nail. Hierarchical abstraction levels provide the structure necessary for generating complex plans at the primitive level [STE81].

In order to design a planning environment from an object-oriented standpoint, hierarchical planning must be used. The ideas and basic concepts of hierarchical planning make programming from an object-oriented approach that much more practical. For this reason, hierarchical planning has been incorporated into the design and implementation of this planning environment.

Object-Oriented Programming

Object-oriented programming is a style of programming which makes it easy to create and manipulate objects [WIN89]. It is a natural evolution from earlier innovations to programming language design. It is more structured than previous attempts at structured programming [KEE89]. Also, object-oriented programming is more modular and abstract than previous attempts at data abstraction and detail hiding [OBJ89].

Three main properties characterize an object-oriented programming language. These are Encapsulation, Inheritance and Polymorphism. Encapsulation is achieved by combining a record with the procedures and functions that manipulate it to form a new data type; an object. Inheritance is defining an object and then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors' code and data. And Polymorphism is attained by giving an action one name that is shared up and down an object hierarchy, with each object in the hierarchy implementing the action in a way appropriate to itself [OBJ89].

An object-oriented approach to planning has several advantages. The first of which is modularization. This approach allows the designer to use modules to represent the different actions and agents of the environment. The second advantage in an object-oriented approach to planning is the concept of inheritance. As stated previously inheritance allows a good degree of sharing of

code to take place. The last advantage in an object-oriented approach to planning is the ease of modification. If the desire to add or remove modules arises, then this is very simple to achieve through this approach.

IBUKI Common Lisp

In the design of this object-oriented planning environment, IBUKI Common Lisp is used. It is a commercially supported, extended and enhanced version of KYOTO Common Lisp. It was originally developed by Dr. Taiichi Yuasa and Mr. Masami Hagiya under the direction of Professor Reiji Nakajima at the Research Institute for Mathematical Science (RIMS), Kyoto University and in cooperation with the Nippon Data General Corporation [IBU87].

CLOS

At the ACM Lisp and Functional Programming Conference in the summer of 1986, many Lisp users and implementors insisted that it was time to standardize. The sense of the community was that experimentation should continue. However, the people needed something practical that they could use until the experimenters came up with the ultimate object-oriented language, if such a thing could exist [KEE89].

There had been earlier calls for standardization, with little result. However, during 1986 a convergence of aesthetic, academic, and economic interests in favor of a standard was unstoppable. An ad hoc standardization group formed at the conference. Soon afterwards, the X3J13 committee for formal standardization of Common Lisp was formed and the group became a part of it [KEE89].

The initial idea was to adopt one of the existing dialects as the standard, but examination of the differences among dialects revealed some interesting facts. Many programmers were passionately committed to the particular dialect that they used and were unwilling to switch unless the standard offered comparable features. Although no existing dialect contained all the right features, several of the dialects had an underlying unity that once they saw past the superficial syntactic differences they were actually aiming towards the same idea from different directions [KEE89].

At this point, the group decided to develop a new dialect that

would combine the best features of the most popular existing dialects, while discarding features that were ill-defined or insufficiently useful. This dialect would be called CLOS, (the Common Lisp Object System). CLOS is a very integral part of Ibuki Common Lisp [KEE89].

III. PROBLEM STATEMENT

Planning, a process of formulating a sequence of actions needed to solve a specific problem, was discussed in the Introduction. Therefore, it will not be discussed again here. Instead, what is meant by a planning environment will be discussed.

In order to generate a plan of actions, the initial state, the goal state, and the world must be given. This information is user-defined and must be supplied in order to generate a plan. The proposed environment provides a mechanism which allows a user to input this information. In order to increase the user interface efficiency, certain predefined objects and actions are available for the user's adoption. Once all pertinent information is supplied to the planning environment, a suitable plan of actions is generated. The problem domain of the planning environment is a blocks world.

Object Representation

Objects model the characteristics and behavior of the elements of the planning environment. The objects presented in this environment include tables, bricks, balls, wedges, and rooms. These objects are represented in the source code for the environment in the form of classes. All the objects except for rooms fall neatly under the class of basic blocks. This class is divided into two other categories: load-bearing blocks and movable blocks.

Load-bearing blocks and movable blocks have one property which distinguishes them from one another. A load-bearing block is capable of supporting another object, and a movable block can be moved from one place to another.

The wedge and ball are considered movable blocks, whereas the table and the brick are considered to be both load-bearing and movable.

Rooms do not fit neatly under these two classes. Therefore, it is necessary to classify them separately.

It is quite apparent that by having used an object-oriented approach to the development of the planning environment that the definition of the initial objects in the environment has been very simple.

Agent Representation

Agents, with respect to a planning environment, are responsible for performing the necessary actions on the objects represented in the environment. There is only one agent type or class presented in this environment. It is the robot. Through the aid of the "defclass" function, the robot is given four properties: name, inroom, nextto, and hand-state. The name property is used to associate a label (or name) with the robot. The inroom property is used to keep track of which room the robot is presently occupying in the environment. The nextto property is used for keeping track of what is presently next to the robot. And finally, hand-state is used to keep track of what the robot is holding.

The environment is designed to accommodate at most three of these robot agents. This limitation was made to keep the capacity of the analyzer relatively reasonable.

Basic Actions

There are several basic actions which are pre-defined for use in the planning environment. They are:

1. PICK-UP -- Robot picks up the designated object
2. PUT-DOWN -- Robot puts down the object it is holding
3. GO-TO -- Robot moves to designated door
4. GO-NEXT-TO -- Robot moves next to designated object
5. GO-THROUGH -- Robot moves through designated door
6. GO-INTO -- Robot moves into specified room
7. STACK -- Robot stacks one object on another
8. UNSTACK -- Robot removes one object from the top of another

This list of actions can be extended by the user if he/she sees a need to add additional actions. For each action, the list of preconditions, deletions, additions, and the arguments must be defined. Preconditions are those conditions of the environment that must be satisfied before an action can be applied. The list of additions are conditions that will become true once that action has been applied. For example, once the robot has travelled from one room to another, the new location of the robot must be reflected in the current state. This predicate would then be added to the current state. Similarly, once the robot has moved from one room to the next, the predicate stating that it was in the old room

must be deleted/changed since the robot is no longer there. Therefore, each action has not only a list of additions, but also a list of deletions that must be applied to the current state.

The action PICK-UP will now be described to illustrate these concepts. Assume robot1 is to pick up box1.

PRECONDITIONS:

-- robot1's hand must be empty

-- robot1 must be next to box1

-- robot1 must be in same room as box1

-- nothing must be stacked on box1

DELETIONS:

-- robot1's hand must be empty

-- robot1 must be next to box1

ADDITIONS:

-- robot1 is holding box1

In order to perform the actions in the environment, several small utility functions are needed. Functions such as WHICH-ROOM and NEXTTO-WHAT, which are used to determine in what room an object or robot is in, and what other object is an object currently next to, are necessary for some of the basic actions to be carried out.

Plan Generation Mechanism

Finding a planning scheme that will always produce a workable plan is not trivial. The idea is to generate a plan which is correct for all possible cases. The only way to accomplish such a tremendous task is to set restrictions on how certain factors are defined.

In the design of this planning environment, there have been some restrictions set for the user. The first restriction is that the user can only choose a maximum of three agents (or robots). The reason for this was discussed previously. The second and most important restriction is that the user must create his/her new classes as descendants of the base classes. Without some foundation/base class to descend back to, the creation of generic actions would not have been possible.

Through the aid of the above restrictions and some hierarchical planning, the following planning strategy has been developed.

-- Move all objects from their initial room state to their goal room state, if they aren't already there

-- Move all objects next to whatever object they are supposed to be next to, according to the goal state

-- Place all objects which should be placed on top of another in their proper locations

User Interface

In the design of any programming (or problem solving) tool, one of the important parts is the user interface. The better the design of the user interface, the easier the tool is to use. A lot of effort and time have gone into the development of the user interface for this planning environment. The user interface contains numerous examples to help the user setup his/her environment.

The best way to describe the details of the user interface is to give the following brief example from the actual code which I have written.

> The most important object to be defined in this environment is ROBOT-ROOM. ROBOT-ROOM defines the structure of the blocks world. Please use a variation of the following example to initialize ROBOT-ROOM.
>
> Example:
> Always remember to have a sketch of how you expect the setup to be configured; see below.

```
-------------------------------------------------
|              |                    |                |
|  room 1   door 1    room 2    door 2    room 3 |
|              |                    |                |
|--- door 3 --------- door 4 ---------------------
|              |                    |
| room 4    door 5    room 5       |
|              |                    |
-----------------------------------
```

> In order to represent the above setup, ROBOT-ROOM would be entered as a list as follows:
> ((room1 door1) (room1 door3) (door1 room2) (door3 room4)
> (room2 door2) (room2 door4) (door2 room3) (door4 room5)
> (room4 door5) (door5 room5))
> The number of doors in a room will represent .5 times the number of pairs needed to represent that room. For example, room 1 has 2 doors, so (room1 door1) and (room1

door3) are needed to show what doors are present in room 1. (door1 room2) and (door3 room4) are needed to show what rooms are adjacent/connected to room 1.

Please enter your variation now:

IV. CLOS OVERVIEW

Brief Overview

The Common Lisp Object System (CLOS) supports a style of programming called object-oriented programming, which makes it easy to create and manipulate objects. CLOS encourages the software developer to create a working model that describes the various classes of objects in terms of their structure and behavior. Often, the working model includes classes that are related to one another. They are similar but not identical. For example, window systems usually need to support different kinds of windows for different purposes. One kind of window might have a border; another might have a label; another might have both a border and a label. The design of a window system would likely include several classes of windows.

CLOS makes it easy to represent relationships among classes, and it supports a flexible means of inheriting (sharing) structure and behavior. Inheritance allows the design and implementation of an application program to be highly modular. It also alleviates the need for maintaining several bodies of nearly identical code.

Any CLOS program can be written using the traditional style of Lisp programming. An important advantage of using CLOS lies in the automatic control of the interaction among the objects.

Example Sessions

The best way to understand any development tool is to have an example session of that tool available. A couple of example sessions using the planning environment design discussed in this report are shown in Figures 1 to 14.

There are several special features or flaws which are common to this version of CLOS. The first is the supported editor. Vi is the editor supported by this version of CLOS, via the "ed" command. The second feature is that this version of CLOS is only supported fully on WILLOW (the CSE Sun server).

```
willow{griskell}2: clos

IBUKI Common Lisp  Version 2 release 01.021.

This software is provided by IBUKI pursuant to a written license
agreement and may be used, copied, transmitted and stored only in
accordance with the terms of such license.

;; Copyright (c) 1990-86 IBUKI  All rights reserved.
;; Copyright (c) 1986-84 T. Yuasa and M. Hagiya All rights reserved

For more information: (describe 'copyright) or (describe
'acknowledgements)

>(load "planner.lsp")
Loading planner.lsp
Finished loading planner.lsp
T

>(planner)
```

Let's Create a Planning Environment

The most important object to be defined in this environment is ROBOT-ROOM. ROBOT-ROOM defines the structure of the blocks world. Please use a variation of the following example to initialize ROBOT-ROOM.

Example:
Always remember to have a sketch of how you expect the setup to be configured, such as:

```
-------------------------------------------------
|               |                  |              |
|  room 1    door 1   room 2   door 2    room 3  |
|               |                  |              |
|--- door 3 --------- door 4 --------------------
|               |                  |
|   room 4   door 5    room 5      |
|               |                  |
-----------------------------------
```

In order to represent the above setup, ROBOT-ROOM would be entered as a list as follows:

```
((room1 door1)  (room1 door3)  (door1 room2)  (door3 room4)
 (room2 door2)  (room2 door4)  (door2 room3)  (door4 room5)
 (room4 door5)  (door5 room5))
```

The number of doors in a room represent .5 times the number of pairs needed to represent that room. For example, room 1 has 2 doors, so (room1 door1) and (room1 door3) are needed to show

Figure 1.

what doors are present in room 1. (door1 room2) and (door3 room4) are needed to show what rooms are adjacent/connected to room 1.

Please enter your variation now:
((room1 door1) (door1 room2))

The maximum number of allowable agents (robots) is 3 (default is 1).
How many agents will there be present in this environment? 1

There are four classes of objects available for use in the design of your planning environment: brick, wedge, ball, table. An option open to you as a user of this program is to define additional classes if you choose to. There is only one stipulation which applies if this option is chosen. The stipulation is that these additional classes must fit under the base classes of movable-block and/or load-bearing-block.

Here is an example to follow if you wish to add a class(es):

```
(defclass chair (movable-block load-bearing-block) ())
```

A new class named chair has been created with both movable- and load-bearing-block characteristics. It could have easily been created with only one of the base classes as its predecessor.

Again, creating new classes is optional. Use of the existing classes is fine.

Enter your new classes now. Type (define-objects) after your last new class.

""

>(define-objects)

Now let's define the initial and final configuration of our environment using our classes.

However, before we can do that we must first create the objects to be used in the environment.

How many objects will there be in your environment? 2

Object #1
    Class name: table
    Object name: table

Object #2
    Class name: brick

Figure 2.

```
     Object name:   box

==================================================================

         Type in the Initial State of the Domain.

***  NOTE:  All input is assumed to be correct and in        ***
***                        correct form.                       ***

TABLE is in room1
BOX is in room1
ROBOT1 is in room2

TABLE is next to nil
BOX is next to nil
ROBOT1 is next to nil

What is stacked on TABLE?  box
What is stacked on BOX?  nil

What is ROBOT1 holding?  nil

==================================================================

                 Now Type in the Goal State of the Domain.

TABLE is in room2
BOX is in room2
ROBOT1 is in room2

TABLE is next to nil
BOX is next to nil
ROBOT1 is next to nil

What is stacked on TABLE?  box
What is stacked on BOX?  nil

What is ROBOT1 holding?  nil

Now type (new-actions?)

""

>(new-actions?)
```

Finally, all the necessary things have been done to initialize the planner. There is however one last thing you the user may optionally choose to do. This is to create additional actions for the agent to perform. At this point, there are 8 actions the agent is able to perform: stack, unstack, pick-up, put-down, go-through,

Figure 3.

go-into, go-next-to, and go-to.

An example of how PUT-DOWN was created follows:

```
; PUT-DOWN a box BOX and change HAND-STATE property and
; INROOM property of BOX to reflect this action.

(defmethod put-down ((box basic-block) (robt robot))
   (cond ((and (eq (which-room robt) (which-room box))
               (eq (state-of-hand robt) box))
          (setf (robot-hand-state robt) 'hand-empty)
          (setf (block-nextto box) (block-nextto robt))
          (write-line (write-to-string (list 'put-down (block-name
           box)))))))
```

The condition statements contain the necessary preconditions which must be satisfied before PUT-DOWN will execute. They are:

1. Both box and robt must be in the same room
2. Robt must be holding box

The two lines which follow the if statement are the additions. They are:

1. Robt hand will be empty
2. Box will be next to robt

Again, creating new actions is optional. Use of the existing actions is fine.

Enter your new actions now. Type (generate-plan) after your last new action.

""

```
>(generate-plan)

==================================================================

The plan is as follows:

** First move all objects to their prospective goal rooms **
(NOTE: Agents work in parallel as much as possible)

ROBOT1 PLAN

(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM1)
(UNSTACK BOX TABLE)
(PUT-DOWN BOX)
(PICK-UP TABLE)
(GO-TO DOOR1)
```

Figure 4.

```
(GO-THROUGH DOOR1)
(GO-INTO ROOM2)
(PUT-DOWN TABLE)
(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM1)
(PICK-UP BOX)
(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM2)
(PUT-DOWN BOX)

** Move all objects next to whatever they should be next to **


** Stack all necessary objects **

ROBOT1 PLAN

(PICK-UP BOX)
(GO-NEXT-TO TABLE)
(STACK BOX TABLE)

** Move agents to their prospective goal rooms **


** Move agents next to whatever they should be next to **


** Finally, if necessary, agent(s) should pick up any objects
it/they need to **

NIL

>Bye.
Bye.
```

Figure 5.

```
willow{griskell}3: clos

IBUKI Common Lisp  Version 2 release 01.021.

This software is provided by IBUKI pursuant to a written license
agreement and may be used, copied, transmitted and stored only in
accordance with the terms of such license.

;; Copyright (c) 1990-86 IBUKI  All rights reserved.
;; Copyright (c) 1986-84 T. Yuasa and M. Hagiya All rights reserved

For more information: (describe 'copyright) or (describe
'acknowledgements)

>(load "planner.lsp")
Loading planner.lsp
Finished loading planner.lsp
T

>(planner)

                    Let's Create a Planning Environment!!!!!
```

The most important object to be defined in this environment is ROBOT-ROOM. ROBOT-ROOM defines the structure of the blocks world. Please use a variation of the following example to initialize ROBOT-ROOM.

Example:

Always remember to have a sketch of how you expect the setup to be configured, such as:

```
-----------------------------------------------
|               |                 |             |
|  room 1    door 1   room 2   door 2    room 3 |
|               |                 |             |
|--- door 3 --------- door 4 -------------------
|               |                 |
|   room 4   door 5    room 5     |
|               |                 |
-----------------------------------
```

In order to represent the above setup, ROBOT-ROOM would be entered as a list as follows:

```
((room1 door1)  (room1 door3)  (door1 room2)  (door3 room4)
 (room2 door2)  (room2 door4)  (door2 room3)  (door4 room5)
 (room4 door5)  (door5 room5))
```

The number of doors in a room represent 2 times the number of pairs needed to represent that room. For example, room 1 has 2 doors, so (room1 door1) and (room1 door3) are needed to show

Figure 6.

what doors are present in room 1. (door1 room2) and (door3 room4) are needed to show what rooms are adjacent/connected to room 1.

Please enter your variation now:
((room1 door1) (room1 door3) (door1 room2) (door3 room4)
(room2 door2) (room2 door4) (door2 room3) (door4 room5)
(room4 door5) (door5 room5))

The maximum number of allowable agents (robots) is 3 (default is 1).
How many agents will there be present in this environment? 3

There are four classes of objects available for use in the design of your planning environment: brick, wedge, ball, table. An option open to you as a user of this program is to define additional classes if you choose to. There is only one stipulation which applies if this option is chosen. The stipulation is that these additional classes must fit under the base classes of movable-block and/or load-bearing-block.

Here is an example to follow if you wish to add a class(es):

```
(defclass chair (movable-block load-bearing-block) ())
```

A new class named chair has been created with both movable- and load-bearing-block characteristics. It could have easily been created with only one of the base classes as its predecessor.

Again, creating new classes is optional. Use of the existing classes is fine.

Enter your new classes now. Type (define-objects) after your last new class.

""

```
>(defclass chair (movable-block load-bearing-block) ())
#<instance 00630e40>

>(define-objects)
```

Now let's define the initial and final configuration of our environment using our classes.

However, before we can do that we must first create the objects to be used in the environment.

How many objects will there be in your environment? 7

Object #1

Figure 7.

```
     Class name:   table
     Object name:   table

Object #2
     Class name:   chair
     Object name:   chair

Object #3
     Class name:   brick
     Object name:   box1

Object #4
     Class name:   brick
     Object name:   box2

Object #5
     Class name:   brick
     Object name:   box3

Object #6
     Class name:   brick
     Object name:   box4

Object #7
     Class name:   brick
     Object name:   box5

==================================================================

         Type in the Initial State of the Domain.

***  NOTE:  All input is assumed to be correct and in        ***
***                        correct form.                      ***

TABLE is in room1
CHAIR is in room1
BOX1 is in room1
BOX2 is in room2
BOX3 is in room5
BOX4 is in room2
BOX5 is in room5
ROBOT1 is in room3
ROBOT2 is in room3
ROBOT3 is in room3

TABLE is next to chair
BOX1 is next to nil
BOX2 is next to nil
BOX3 is next to nil
BOX4 is next to nil
BOX5 is next to nil
```

Figure 8.

```
ROBOT1 is next to robot2
ROBOT3 is next to robot2

What is stacked on TABLE?  box1
What is stacked on CHAIR?  nil
What is stacked on BOX1?  nil
What is stacked on BOX2?  nil
What is stacked on BOX3?  box5
What is stacked on BOX4?  box2
What is stacked on BOX5?  nil

What is ROBOT1 holding?  nil
What is ROBOT2 holding?  nil
What is ROBOT3 holding?  nil

==================================================================

                  Now Type in the Goal State of the Domain.

TABLE is in room4
CHAIR is in room4
BOX1 is in room4
BOX2 is in room4
BOX3 is in room4
BOX4 is in room4
BOX5 is in room4
ROBOT1 is in room4
ROBOT2 is in room4
ROBOT3 is in room4

TABLE is next to box4
CHAIR is next to nil
BOX1 is next to nil
BOX2 is next to nil
BOX3 is next to nil
BOX4 is next to nil
BOX5 is next to nil
ROBOT1 is next to nil
ROBOT2 is next to nil
ROBOT3 is next to nil

What is stacked on TABLE?  box5
What is stacked on CHAIR?  box1
What is stacked on BOX1?  nil
What is stacked on BOX2?  box3
What is stacked on BOX3?  nil
What is stacked on BOX4?  nil
What is stacked on BOX5?  box2

What is ROBOT1 holding?  nil
What is ROBOT2 holding?  nil
```

Figure 9.

What is ROBOT3 holding? nil

Now type (new-actions?)

""

>(new-actions?)

Finally, all the necessary things have been done to initialize the planner. There is however one last thing you the user may optionally choose to do. This is to create additional actions for the agent to perform. At this point, there are 8 actions the agent is able to perform: stack, unstack, pick-up, put-down, go-through, go-into, go-next-to, and go-to.

An example of how PUT-DOWN was created follows:

```
; PUT-DOWN a box BOX and change HAND-STATE property and
; INROOM property of BOX to reflect this action.

(defmethod put-down ((box basic-block) (robt robot))
   (cond ((and (eq (which-room robt) (which-room box))
               (eq (state-of-hand robt) box))
          (setf (robot-hand-state robt) 'hand-empty)
          (setf (block-nextto box) (block-nextto robt))
          (write-line (write-to-string (list 'put-down (block-name
           box)))))))
```

The condition statements contain the necessary preconditions which must be satisfied before PUT-DOWN will execute. They are:

1. Both box and robt must be in the same room
2. Robt must be holding box

The two lines which follow the if statement are the additions. They are:

1. Robt hand will be empty
2. Box will be next to robt

Again, creating new actions is optional. Use of the existing actions is fine.

Enter your new action now. Type (generate-plan) after your last new action.

""

```
>(defmethod get-on-chair ((ch1 basic-block) (robt robot))
   (cond ((eq (which-room robt) (which-room ch1))
          (go-next-to robt ch1)
          (setf (block-nextto-to ch1) nil)
          (setf (robot-nextto-to robt) nil)
```

Figure 10.

```
            (setf (block-clear ch1) robt)
            (write-line (write-to-string (list 'get-on-chair
                  (block-name robt)))))))
#<instance 006278ac>

>(defmethod get-off-chair ((ch1 basic-block) (robt robot))
    (cond ((eq (is-clear ch1) robt)
             (cond ((not (eq (is-clear robt) nil))
                      (put-down (is-clear robt) robt)))
             (setf (block-nextto ch1) robt)
             (setf (robot-nextto robt) ch1)
             (setf (block-clear ch1) nil)
             (write-line (write-to-string (list 'get-off-chair
                (robot-name robt)))))))
#<instance 00622b70>

>(generate-plan)

==================================================================

The plan is as follows:

** First move all objects to their prospective goal rooms **
(NOTE: Agents work in parallel as much as possible)

ROBOT1 PLAN

(GO-TO DOOR2)
(GO-THROUGH DOOR2)
(GO-INTO ROOM2)
(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM1)
(GO-NEXT-TO TABLE)
(UNSTACK BOX1 TABLE)
(PUT-DOWN BOX1)
(PICK-UP TABLE)
(GO-TO DOOR3)
(GO-THROUGH DOOR3)
(GO-INTO ROOM4)
(PUT-DOWN TABLE)

ROBOT2 PLAN

(GO-TO DOOR2)
(GO-THROUGH DOOR2)
(GO-INTO ROOM2)
(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM1)
(GO-NEXT-TO CHAIR)
```

Figure 11.

```
(PICK-UP CHAIR)
(GO-TO DOOR3)
(GO-THROUGH DOOR3)
(GO-INTO ROOM4)
(PUT-DOWN CHAIR)

ROBOT3 PLAN

(GO-TO DOOR2)
(GO-THROUGH DOOR2)
(GO-INTO ROOM2)
(GO-TO DOOR1)
(GO-THROUGH DOOR1)
(GO-INTO ROOM1)
(GO-NEXT-TO BOX1)
(PICK-UP BOX1)
(GO-TO DOOR3)
(GO-THROUGH DOOR3)
(GO-INTO ROOM4)
(PUT-DOWN BOX1)

ROBOT1 PLAN

(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM5)
(GO-TO DOOR4)
(GO-THROUGH DOOR4)
(GO-INTO ROOM2)
(UNSTACK BOX2 BOX3)
(GO-TO DOOR4)
(GO-THROUGH DOOR4)
(GO-INTO ROOM5)
(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM4)
(PUT-DOWN BOX2)

ROBOT2 PLAN

(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM5)
(UNSTACK BOX5 BOX3)
(PUT-DOWN BOX5)
(PICK-UP BOX3)
(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM4)
(PUT-DOWN BOX3)
```

Figure 12.

```
ROBOT3 PLAN

(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM5)
(GO-TO DOOR4)
(GO-THROUGH DOOR4)
(GO-INTO ROOM2)
(PICK-UP BOX4)
(GO-TO DOOR4)
(GO-THROUGH DOOR4)
(GO-INTO ROOM5)
(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM4)
(PUT-DOWN BOX4)

ROBOT1 PLAN

(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM5)
(PICK-UP BOX5)
(GO-TO DOOR5)
(GO-THROUGH DOOR5)
(GO-INTO ROOM4)
(PUT-DOWN BOX5)

** Move all objects next to whatever they should be next to **

ROBOT1 PLAN

(PICK-UP TABLE)
(GO-NEXT-TO BOX4)
(PUT-DOWN TABLE)

** Stack all necessary objects **

ROBOT1 PLAN

(GO-NEXT-TO BOX5)
(PICK-UP BOX5)
(GO-NEXT-TO TABLE)
(STACK BOX5 TABLE)

ROBOT2 PLAN

(PICK-UP BOX1)
(GO-NEXT-TO CHAIR)
```

Figure 13.

```
(STACK BOX1 CHAIR)
```

```
ROBOT3 PLAN

(PICK-UP BOX3)
(GO-NEXT-TO BOX2)
(STACK BOX3 BOX2)

ROBOT1 PLAN

(GO-NEXT-TO BOX2)
(UNSTACK BOX3 BOX2)
(PUT-DOWN BOX3)
(PICK-UP BOX2)
(GO-NEXT-TO BOX5)
(STACK BOX2 BOX5)

ROBOT1 PLAN

(GO-NEXT-TO BOX3)
(PICK-UP BOX3)
(GO-NEXT-TO BOX2)
(STACK BOX3 BOX2)

** Move agents to their prospective goal rooms **


** Move agents next to whatever they should be next to **


** Finally, if necessary, agent(s) should pick up any objects
it/they need to **

NIL

>(bye)
Bye.
```

Figure 14.

CONCLUSION

Planning is a process of formulating a sequence of actions needed to solve a specific problem. The problem addressed in this report was the design of an object-oriented planning environment. An object-oriented planning environment is an environment in which specific plans can be generated using an object-oriented approach.

In order to generate a plan of actions, the initial state, the goal state, and the world must be defined. Through the aid of CLOS functions such as "defclass" and "make-instance", the definition of these entities is very straightforward. CLOS makes it easy to represent relationships among classes, and supports a flexible means of inheriting (sharing) structure and behavior.

In order to increase the user interface efficiency, certain predefined objects and actions are made available for the user's adoption. The objects presented in this environment include tables, bricks, balls, wedges, and rooms. The basic actions available include PICK-UP, PUT-DOWN, GO-TO, GO-NEXT-TO, GO-THROUGH, GO-INTO, STACK, and UNSTACK.

Once all pertinent information is supplied to the planning environment, a suitable plan of actions is generated.

After designing this environment using IBUKI's CLOS, it has become very clear that object-oriented Lisp programming approaches to planning have been overlooked far too long. The programmer benefits from a modular implementation. CLOS enables the programmer to define an organization of classes that models the

relationships among the various kinds of objects. And finally, the program more closely resembles the world it is modeling. An object-oriented program is designed at a higher level of abstraction than a traditional program.

REFERENCES

[IBU87] IBUKI Common Lisp User's Guide, IBUKI, Mountain View, California, 1987.

[KEE89] Sonja E. Keene, Object-Oriented Programming in Common Lisp, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[MAK90] I. M. Makarov, Engineering Artificial Intelligence, Hemisphere Publishing Company, New York, 1990.

[NIL80] Nils J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, California, 1980.

[OBJ89] Object-Oriented Programming Guide, Borland International, Incorporated, Scotts Valley, California, 1989.

[SAC74] E. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, v 5: 115-135, 1974.

[STA87] A. Stangaard, "An Introduction to Applied Machine Intelligence," Robotics and AI, Prentice Hall, Englewood Cliff, New Jersey, 1987.

[STE81] M. Stefik, "Planning and Metaplanning," Readings in Artificial Intelligence, Nilsson and Webber, eds., Tioga Publishing Company, Palo Alto, California, 1981.

[WIN89] Patrick H. Winston and Berthold K. Horn, Lisp, 3rd ed., Addison-Wesley Publishing Company, Massachusetts, 1989.