1

CANADIAN THESES ON MICROFICHE

I.S.B.N.

THESES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us a poor photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de mauvaise qualité.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Division

Division des thèses canadiennes

Ottawa, Canada
K1A 0N4

59933

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moulées ou dactylographier

Full Name of Author — Nom complet de l'auteur

GILLES DUPONT

Date of Birth — Date de naissance

12 OCT 42

Country of Birth — Lieu de naissance

CANADA

Permanent Address — Résidence fixe

1213 BORDEAU GROVE
GLOUCESTER, ONT
K1C 2M6

Title of Thesis — Titre de la thèse

AN ACTOR SYSTEM FOR SUPPORTING A MILITARY APPLICATION

University — Université

CARLETON

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MENG

Year this degree conferred — Année d'obtention de ce grade

1982

Name of Supervisor — Nom du directeur de thèse

W.R. LALONDE

Permission is hereby granted to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈQUE NATIONALE DU CANADA de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans l'autorisation écrite de l'auteur.

Date

SEPT 23, 1982

Signature

[Signature]

**AN ACTOR SYSTEM
FOR SUPPORTING
A MILITARY APPLICATION**

by

© Gilles J.C. Dupont, P. Eng.

A thesis submitted to the
faculty of graduate studies and research
in partial fulfilment of the requirements for
the degree of
MASTER OF ENGINEERING

Carleton University

Ottawa Ontario

September 1982

The undersigned recommend to the
faculty of graduate studies and research
acceptance of the thesis

AN ACTOR SYSTEM
FOR SUPPORTING
A MILITARY APPLICATION

submitted by
GILLES J.C. DUPONT in partial fulfilment of
the requirements for the degree of
Master of Engineering

W. Laporte

Thesis Supervisor

B. Lagussek

Chairman

Department of Systems and
Computer Engineering

Carleton University
Department of Systems and Computer Engineering

22 September 1982

Acknowledgements

This work was made possible through the financial support of the Department of National Defence.

Much credit for this dissertation must go to my thesis supervisor professor Wilf LaLonde. His enthusiasm for the subject of artificial intelligence and patience at reviewing the several drafts enabled me to see this work through.

A special debt of gratitude is owed to my wife Lise who meticulously screened every draft of this document providing essential guidance on syntax and semantics.

Finally, a special thank you to professor John Pugh for his encouragement and interest in my research.

Abstract

When two modern armies meet on a battlefield, the opposing generals can no longer perch themselves on surrounding hills with a pair of binoculars to supervise the activities. Indeed, modern armies are highly mobile and operate over a very wide front. Generals are informed of the disposition of troops on the battlefield through the use of a map marking system whereby headquarters personnel place military symbols on a battle map.

Military map marking as well as all the other activities of an army headquarters has been the target of automation studies in the Canadian land force for some time. Automation of the map marking system is an excellent subject for an artificial intelligence technique called "actor" or "object-oriented" programming since both are highly hierarchical.

This thesis is concerned with the design and prototype implementation of an automated military map marking system using actor programming techniques. The design is divided into two components:

- 1) a universal actor system, and
- 2) a military map marking system.

Military map marking is used as a sample application of actor programming. The final system however turned out to be much more than just an automation of the manual map marking system used by the Canadian land force today.

Preface

The objective of my thesis was to find a way to automate the present map marking system of army headquarters using artificial intelligence techniques. Once I started to investigate the possibility of using actor programming, it became clear that the map marking system would have to be implemented in an actor environment, which was not available at Carleton University. The emphasis therefore shifted to designing a general actor system which could be used to program not only my particular application but anything that could be modeled using actors.

Once the proper environment had been created, programming of a model to draw military symbols on a map became so trivial that the design of the military system was extended to become a much more "intelligent" system where an actor representing a military unit could, not only show a symbol on a graphics terminal but answer queries about the unit it represents. This extension of the military system stopped where tactical information was required. This limitation was due partially to the non-availability of the information and also because the information could not have been used in a non-military publication.

Due to the nature of this document and because some explanation of army organizations was necessary to comprehend the application, I used well published world war II organizational concepts. Modern armies may or may not be organized the way this thesis depicts them. It is also possible that some military

Preface

operational concepts have been intentionally changed or falsified
in order not to reveal classified information.

TABLE OF CONTENTS

Page

1. INTRODUCTION

1.1	An Overview.....	11
1.2	Army Organization.....	12
1.3	Map Marking System.....	19
1.4	The Problem.....	24
1.5	The Chosen Solution.....	25

2. ACTOR SYSTEMS

2.1	A Sample Actor Program.....	28
2.2	Definition.....	31
2.3	Properties.....	34
2.4	Components.....	41
2.5	Manipulation.....	42
2.6	Actor Programming.....	44
2.7	Acting in Parallel.....	48
2.8	Dynamic Creation of Actors.....	48

3. OASYS

3.1	An Ordinary Actor System (OASYS).....	53
3.2	LISP Programming.....	62
3.3	Augmented LISP Support.....	65

TABLE OF CONTENTS

Page

3.4	Actor Representation.....	68
3.5	User Interface (Behavior Programming).....	72
3.6	How OASYS works.....	77
3.7	Human Engineering Aspects of OASYS.....	86

4. MMM

4.1	Conceptual View of MMM.....	92
4.2	Overall Structure of the MMM System.....	96
4.3	Basic Symbols.....	99
4.4	Size Symbols.....	104
4.5	Type Symbols.....	106
4.6	Overlay.....	108
4.7	Clock.....	109
4.8	A Session with MMM.....	111
4.9	MMM as a Decision Making Tool.....	116

TABLE OF CONTENTS

Page

5. OTHER APPLICATIONS AND RECOMMENDATIONS

5.1	General.....	120
5.2	Actor Systems in Distributed Environments.....	121
5.3	LISP as a Base for Actor Programming.....	123
5.4	Actor Systems Programming Style.....	125
5.5	Other Application Areas.....	126

APPENDICES:

A.	Package LISP Support.....	132
B.	Package Action.....	146
C.	Package Knowledge.....	150
D.	Package Actor.....	162
E.	Package Simulation.....	173
F.	Package MMM.....	180
G.	Actor SYSTEM of OASYS and the Patterns it Handles.....	190
H.	Sample Sessions with OASYS.....	197

CHAPTER 1

INTRODUCTION

1.1 AN OVERVIEW

This thesis is concerned with the design and prototype implementation of a military map marking system using actor programming techniques. A map marking system is a military information system that contains tactical and administrative data for military units. It is used to draw military symbols on a map overlay and to answer queries about the units they represent. Actor programming is a methodology where programs are controlled by the passing of messages between active-objects called actors. The design is divided into two components:

- 1) A general actor system called the Ordinary Actor System (OASYS), and
- 2) A Military Map Marking System (MMM) that uses OASYS.

OASYS is implemented in CP6 LISP, a subset of INTERLISP. It is an actor or object-oriented programming support system, complete with its own language. OASYS can be used for any type of application but it is mainly suited for artificial intelligence. Its only limitations are those of CP6 LISP on which it is based. It supports an actor system where all the actors reside in the same computer. The OASYS programming language is aimed at computer-naive people and its simplicity is due to a verbose syntax based upon pattern matching.

MMM is implemented in OASYS. It is designed to file and retrieve tactical military information and to draw military

symbols on a graphics terminal as an overlay to an existing geographic map. The system uses Canadian map marking symbols and can operate only in a Canadian land force environment. It is a real time system where symbols are moved in accordance with known information about the military units they represent. MMM can also be used as a decision making tool to show the consequences of certain movement of troops on the battlefield. The present version of MMM is demonstrated on ASCII terminals instead of the intended color graphics equipment, so it only prints alphanumeric information rather than the military symbol where a unit is located. Thus it should be clear that the emphasis of the thesis is on matters separate from the actual display and the corresponding protocols.

1.2 ARMY ORGANIZATION

MMM was designed using the organization of the land force of the Canadian Armed Forces as an application domain. An overall description of this organization is therefore justified.

The Canadian Armed Forces comprise three distinct forces:

- a naval force,
- a land force, and
- an air force.

The Canadian land force is a hierarchy of organizations where each level is subordinate to the level above it. The levels are

ARMY
CORPS
DIVISION
BRIGADE
BATTALION/REGIMENT
COMPANY/SQUADRON
PLATOON/TROOP

A brigade or higher organizations are referred to as formations, while a battalion/regiment is referred to as a unit and anything below it is called a sub-unit. Each unit is classified either as an arms unit or a service unit. The arms component is concerned with directly engaging the enemy while the service part of a formation supports the arms units with the necessary battle commodities and repair services. The arms units are further sub-divided into infantry, armoured, artillery, signals and engineers. The services are also sub-divided into the following: transport maintenance, supply, administration, postal, chaplain, and military police.

An {army}(*) is a self-contained fighting organization capable of carrying on a land battle, on any terrain, under any weather conditions for prolonged periods of time. The word self-contained here meaning that it owns the necessary support

(*) A word or expression is placed between squiggles
() at the place in the text where it is defined.

An army consists of an army headquarters, some highly specialized support organizations, called army troops, and two self-contained fighting formations called corps. It is commanded by a general. (what american movies call a four star general). The army does not depend on any other land force for support. It may, on occasion, require the support of tactical air forces or use naval transportation and long range fire power for amphibious operations.

A division is the last level of self-contained battle formations. It consists of a divisional headquarters, and the combat support and logistics service units necessary to support three fighting brigades. These three brigades can either be infantry brigades or armoured brigades; making the division an infantry division or an armoured division. A typical infantry division is shown in Figure 1.3. The division is subordinate to the corps and depends upon it for some specialized support.

In the Canadian land force, divisions do not exist in peacetime. Therefore, the infantry brigade group, used as a corps troop, is the last level of self-contained battle formations. Figure 1.4 presents a simplified view of the main Canadian land force battle formation. It consists of a headquarters and signals squadron, three infantry battalions, an armoured regiment, an artillery regiment, an engineers squadron,

[illegible]

A battalion or regiment is a highly mobile battle unit with very little administrative and logistics support. It normally comprises a headquarters, a few line companies (which are sub-units that carry out the unit's main task; e.g., an infantry battalion is made up of infantry companies), a signals troop that looks after its communications needs, and an administrative company. Figure 1.5 shows the structure of a battalion

organization.

1.3 MAP MARKING SYSTEM

When two modern armies meet on a battlefield, the opposing generals can no longer perch themselves on surrounding hills with a pair of binoculars to supervise the activities. Indeed, modern armies are highly mobile and operate over a very wide front. How is the modern general to keep track of the changing situation, and location of his troops? For years now, army headquarters personnel have been using a map marking system to keep their commanding general informed of the disposition of troops on the battlefield.

(Military map marking) is the set of activities necessary to properly display information about a quickly changing battlefield, in such a way that trained military personnel can easily assess the situation.

It is important to note that every formation or unit has a headquarters and that one of the tasks of the headquarters is to keep track of its sub-units, of other units and of other formations. To implement map marking, the headquarters personnel use military symbols. The symbol representing a particular unit or sub-unit is

made up of three distinctive parts:

- a) a base token,
- b) a size token, and
- c) a type token.

The list that follows is simply to illustrate the concept of military symbols, it is not intended to be an accurate all-inclusive list.

Base Tokens:

a unit's location

a unit's approximate location

a headquarters

Size Tokens:

an Army	XXXX
a Corps	XXX
a Division	XX
a Brigade	X
a Battalion/Regiment	
a Company/Squadron	
a Platoon/Troop	...

Type Tokens:

Infantry X

Armoured O

Mechanized Infantry X

Reconnaissance /

Artillery ●

Signals \

Engineers M

Service SVC

Transport ⚙

Supply ⊗

Maintenance H

The symbol representing a unit or a formation is a combination of one or several tokens. For example, to represent an infantry brigade, one draws a unit base token, then the brigade size token on top of it, and finally, the infantry type token inside the base token to yield:

A typical military symbols overlay is shown at Figure 1.6.

To adequately draw the appropriate symbol representing a unit, the "plotter" must know the relative position of the unit in the hierarchy of units known at the headquarters. It must know whether or not a unit is in movement, and if it is, what is its up-to-date location. The plotter must also know how much information is to be revealed about the unit; i.e., is it to be shown as being approximately at that position or are all the components of the unit to be shown? In other words, a map marking system must contain much more information than the military symbols listed above.

The present Canadian army map marking system consists of manually placing military symbols on a clear plastic overlay placed on top of an oversize map of the area for which the headquarters is responsible.

Some of the necessary information for plotting an up-to-date picture of the battlefield is obtained from daily update reports, and from infrequent reports which units send to the headquarters as situations occur on the battlefield. Most of the required information however resides in the marker's mind. From an automation point of view, map marking is only one of many applications using information received from subordinate units/sub-units or neighboring formations or superior formation and the expertise of headquarters personnel.

In an automated command and control system, the information handling and storage is done by computers. However, the high

resolution graphics terminal equipment required for showing the details of a map as well as the details of its overlay of military symbols is not yet cost effective. Nevertheless, the need exists for map marking application programs, the back-end of which can later be adapted to whatever graphics equipment is made available in the future.

1.4 THE PROBLEM

The problem addressed by this thesis is that of designing an application program suitable for the manipulation of data necessary for accurately plotting the changing battlefield, in a manner meaningful to military personnel. The software has to possess a great deal of knowledge about the units and other entities it has to represent to prevent the constant feeding of similar data. It must also be flexible enough to add and delete units without re-programming.

The application designed must be adaptable to whatever graphics equipment is used by field armies of the future when they automate the present map marking system.

The system must be simple in design and simple to use. Operators of this map marking system will be non-programmers, and therefore, the application program must have an interface that is natural and very close to the English language.

7.

1.5 THE CHOSEN SOLUTION

The selected approach is to use an actor-oriented programming methodology. The reasons for selecting the actor approach for the solution of the problem of marking military overlays in an automated army command and control system include:

- a) the simplicity and flexibility of actor programming;
- b) the ease of modeling army structures as actors; and
- c) the possibility of extending a military map marking system to replace several programs.

The traditional view of software systems is that they consist of two types of entities: data and procedures. Procedures act on data to produce the required results. One of the problems with traditional programming is that it treats data and procedures as if they were independent, when in fact, they are not, since all procedures make assumptions about the form of the data they manipulate [12]. Actor programming, on the other hand, consists of only one entity: the actor. An actor contains data and procedures, and is "put to work" by sending it a message. The message can be in plain English, instead of using the rigid parameter passing structure of procedure calls. From a programmer's point of view, this makes actor systems simpler than conventional software systems. In addition, in a highly volatile environment, actor systems are much more adaptable, since actors

can be added or deleted easily in an active program and actor's behaviors (the procedure part of an actor) can also be modified as the situation evolves.

As pointed out earlier, each level in an army is subordinate to and depends upon the level immediately above it. In an actor system, as will be explained in details in the next chapter, actors are linked together in a hierarchy where an actor depends upon one or many actors one level above it for its existence. It is, therefore, easy to draw a parallel between the army structure and a hierarchy of actors, and the modeling of one by the other is natural.

Symbol markings on maps is only one of several functions carried out at an army unit headquarters. With ordinary high level programming techniques, these different functions have to be automated in separate programs while with actor programming techniques, several functions can be programmed as one application.

In an actor system, the actor modeling an army unit is much more than just a symbol plotter. The primary function of the actor remains to show the headquarters personnel where the unit is on the map, but it is, in fact, an expert that knows everything about the unit it represents. It contains information about the mission of the unit, its actual strength, the name of its commanding officer and much more tactical and administrative information.

CHAPTER 2

CTOR SYSTEMS

2.1 A SAMPLE ACTOR PROGRAM

The sample actor program that follows should not be examined in detail at this stage. It will be referred to later when explaining the general concepts underlying actor programming. The sample is placed this early in the chapter to provide the reader with something concrete to latch onto when learning about the theoretical concepts. The particular syntax used for the sample program is not important and should not be regarded as the syntax used by all actor systems.

(* Sample Actor Program)

(* This program sets up the necessary actors to
handle bank accounts.)

(* It is designed to be used on personal computers.)

(ASK SYSTEM CREATE BANK_ACCOUNTS)

(ASK BANK_ACCOUNTS CREATE SAVINGS_ACCOUNT)

(ASK BANK_ACCOUNTS CREATE CHEQUING_ACCOUNT)

(ASK BANK_ACCOUNTS DO WHEN RECEIVING

(+ PRESENT BALANCE +)

((ASK !MYSELF RECALL YOUR BALANCE)))

```
(ASK BANK_ACCOUNTS DO WHEN RECEIVING
  (DEPOSIT ?AMOUNT)
  ((ASK !MYSELF SET YOUR BALANCE TO
    (ASK PLUS (ASK !MYSELF RECALL YOUR BALANCE)
      !AMOUNT))
    (ASK !MYSELF ADD (!AMOUNT 0 (ASK !MYSELF RECALL
      YOUR BALANCE))
      TO YOUR LIST OF TRANSACTIONS)
    (REPLY 'OK)))
```

```
(ASK BANK_ACCOUNTS DO WHEN RECEIVING
  (WITHDRAW ?AMOUNT)
  ((ASK !MYSELF SET YOUR BALANCE TO
    (ASK DIFFERENCE (ASK !MYSELF RECALL YOUR
      BALANCE)
      !AMOUNT))
    (ASK !MYSELF ADD (0 !AMOUNT (ASK !MYSELF RECALL
      YOUR BALANCE))
      TO YOUR LIST OF TRANSACTIONS)
    (REPLY 'OK)))
```

```
(ASK BANK_ACCOUNTS DO WHEN RECEIVING
  (TRANSFER ?AMOUNT FROM ?ACCOUNT_X TO ?ACCOUNT_Y)
  ((ASK !ACCOUNT_X WITHDRAW !AMOUNT)
    (ASK !ACCOUNT_Y DEPOSIT !AMOUNT)
    (REPLY 'DONE)))
```



```

(ASK SAVINGS_ACCOUNT DO WHEN RECEIVING
  (ADD INTEREST)
  ((ASK !MYSELF SET YOUR BALANCE TO
    (ASK PLUS (ASK !MYSELF RECALL YOUR BALANCE)
      (ASK TIMES (ASK !MYSELF RECALL YOUR
        BALANCE)
          (ASK QUOTIENT (ASK !MYSELF RECALL
            YOUR INTEREST)
              1200))))))
  (REPLY 'DONE)))
(ASK SAVINGS_ACCOUNT SET YOUR INTEREST TO 15.5)
(END)

```

Once the system is in place, a typical session could be:
 (user input is preceeded by the prompt "<->")

```

<->(ASK SAVINGS_ACCOUNT SET YOUR BALANCE TO 425.00)
425.00
<->(ASK CHEQUING_ACCOUNT SET YOUR BALANCE TO 200.25)
200.25
<->(ASK CHEQUING_ACCOUNT DEPOSIT 653.33)
OK
<->(ASK CHEQUING_ACCOUNT DEPOSIT 1234.15)
OK
<->(ASK CHEQUING_ACCOUNT ADD INTEREST)
(+++++ WARNING +++++ (ADD INTEREST) IS AN UNKNOWN
  MESSAGE PATTERN FOR ACTOR CHEQUING_ACCOUNT)
NIL

```

```
<->(ASK SAVINGS_ACCOUNT ADD INTEREST)
DONE
<->(ASK BANK_ACCOUNTS TRANSFER 632.00 FROM
<->CHEQUING_ACCOUNT TO SAVINGS_ACCOUNT)
DONE
<->(ASK SAVINGS_ACCOUNT WHAT IS MY PRESENT BALANCE)
1062.49
<->(ASK CHEQUING_ACCOUNT WHAT PRESENT BALANCE IS THERE)
1455.73
<->(ASK CHEQUING_ACCOUNT RECALL YOUR TRANSACTIONS)
((653.33 U 353.58)
(1234.15 D 2087.73)
(O 632.00 1455.73))
<->(END)
```

2.2 DEFINITION

The actor or object-oriented programming concept is rapidly gaining popularity among computer scientists, especially since the advent of SMALLTALK-80. The concept is not yet well defined and researchers are still trying to evaluate the full power of object-oriented techniques and to establish rules (and limitations to govern its use). A good definition of actor systems can nevertheless be found in Carl Hewitt's 1976 paper [9].

An {actor system} is a transparent medium for constructing models in which the control structure emerges as a pattern of passing messages among the objects being modeled.

In data/procedures-oriented software, control resides in procedures; the data is manipulated in accordance with the controlled sequence of steps that makes up the procedure. In an application program overall control is given to a super-procedure normally called the main program. In an object-oriented program, there is no overall control structure and the object which receives the message is the only control structure in existence. This implies that the main difference between conventional programming and object-oriented programming is that procedures return control to their caller, while actors do not even have to know who sent them the message.

The objects being modeled are called actors or sometimes active objects. In our example in section 2.1, SYSTEM, BANK_ACCOUNTS, SAVINGS_ACCOUNT, CHEQUING_ACCOUNT, PLUS, DIFFERENCE, TIMES and QUOTIENT are actors. An actor is the single type of entity handled in object-oriented software. Obviously, it must contain data and procedures if it is to do the same job as conventional data/procedures-oriented programs. An (actor) therefore is an active software object containing both knowledge and behaviors.

The knowledge of an actor is a collection of data structures required for manipulation of input data received through a message. In our example, the knowledge of actors SAVINGS_ACCOUNT and CHEQUING_ACCOUNT contains the current balance and a list of all transactions made. The behaviors of an actor are made up of a message pattern or selector, and a set of actions called the

script. The script is executed when the incoming message matches the pattern. What follows the word "RECEIVING" in "DO WHEN RECEIVING" messages of our sample program is a behavior: first the pattern, then the script.

Schematically behaviors look like this:

An important part of actor programming is the message passed to an actor. A message as it applies to actor systems is defined as follows by Adele Goldberg [18]:

A (message) is the specification of one of an actor's manipulations.

When an actor receives a message, it determines how to behave by matching the selector part of its behaviors with the message received.

At this stage, we can turn our attention to some important aspects

of actor systems such as:

- Properties,
- Components, and
- Manipulation.

2.3 PROPERTIES

Every researcher of object-oriented programming concepts seems

to agree that the main characteristics of actor systems are:

- 1) a hierarchical structure,
- 2) the inheritance of behaviors and attributes,
- 3) the sending and receiving of messages, and
- 4) intrinsic knowledge.

Actor systems, like army organizations, have a hierarchical structure where each level is subordinate to the level immediately above it. The relationship in the actor system is parental rather than the military type of subordination and since an actor can have several parents, the hierarchy should more correctly be termed a network. The hierarchical viewpoint, however, is more convenient to use than the network outlook.

In an actor system, the top level consists of generic actors that become increasingly specialized at lower levels. The bottom echelon is made up of very specific and highly individualized and specialized actors. The hierarchy created in section 2.1 contained generic actor `BANK_ACCOUNTS` and two subordinate actors `SAVINGS_ACCOUNT` and `CHEQUING_ACCOUNT`. The parents of actors in the hierarchy are also called superclasses or generic classes. Figure 2.1 illustrates the hierarchical structure of actor systems using a set of actors under the generic actor `ANIMAL` and another under the generic actor `SENSES`. There are two parallel

hierarchies except at the bottom level where actor FOX_EYES is a child of FOX and of EYES each belonging to a different hierarchy.

Figure 2.1 TYPICAL ACTOR NETWORK

A hierarchy induced by a parent relation on actors is called a class. All actors related to the same parent are members of the same class. Individual actors are subordinate to their parents in that they could not exist in the same form without the superclass (parent). In every actor system therefore, there must be an overall head of the structure or a super-superclass from which all other actors are descendants (actor SYSTEM in our

sample in section 2.1).

The most important characteristic of actor systems is that actors inherit behaviors and attributes from their parents. Some systems permit inheritance between all actors, but most allow it to happen only between actors of the same class. Actors are said to inherit the properties of their parents in the sense that, if a given attribute's value or a behavior is stored explicitly in a parent of the actor, but not in the actor itself, then that value or the script of the behavior is retrieved when the actor is requested to recall its value for the attribute or when the message it received matched the pattern of the behavior. When a matching pattern is found in a parent of the actor, the script is executed in the environment of the actor which received the message. Therefore, an actor uses only his own knowledge and inherited attributes to influence manipulations. For example, when actor CHEWING_ACCOUNT was asked to deposit \$653.33, it executed a procedure stored in actor BANK_ACCOUNTS to change its knowledge of "balance" and to add a new transaction to its knowledge of "transactions".

The inheritance property allows patterns to be more generic and therefore lets the same actions execute in answer to many different messages fitting the same pattern.

Once an actor has been created it may modify itself,

- by adding new attributes to its knowledge,
- by adding new behaviors (the corresponding messages

will not be understood by the parent), or

- by changing actions or behaviors of the parent.

The first two of the above modifications are self-explanatory, but the third may generate unwanted results. Should an off-spring or subclass be allowed to change the behavior of a superclass or parent thereby affecting all other subclasses ensued from the superclass? The obvious answer is no, although there might be cases when it is necessary to change the behavior of a superclass. In such a case, the message should be redirected to the superclass and the change implemented by it, if warranted. To change the actions or behaviors of the parent as performed by a subclass implies a two step operation:

- 1) obtain a copy of the behavior from the parent,
and
- 2) change the copy to suit the actor's need.

Some object-oriented systems allow actors to add attributes to a parent directly but I prefer the better-defined approach of sending a message to the superclass requesting it to add the new attribute to its knowledge. Attribute values, on the other hand, can only be changed by the owner of the attribute; e.g., actor `CHEQUING_ACCOUNT` could not change the value of attribute "balance" in actor `SAVINGS_ACCOUNT`.

To change the value of an attribute may require that some actions be taken by another actor or even by the same actor on some other data item. In order not to clutter scripts with

housekeeping chores, actor programming, and for that matter Artificial Intelligence programming in general, uses "demons".

(Demons) are procedures which are called automatically when specified changes are made to the knowledge or behavior of an actor.

There are two types of demons:

- 1) Private demons which are private to the actor which owns them, and
- 2) Non-private demons which are inherited by the children of an actor so that they are also called when changes are made to their knowledge or behaviors.

The kinds of demons normally used are termed IF-ADDED, IF-REMOVED, IF-CHANGED, and IF-NEEDED demons. For example, if a new attribute or attribute value is added to an actor's knowledge, the IF-ADDED demon is invoked; if an action is replaced by another in a behavior, the IF-CHANGED demon is called; if a value for an attribute is sought but not found, the IF-NEEDED demon is summoned if one exists for that attribute.

Whatever method is used to program an application, there is a need to communicate with the "executor". In conventional programming a call to a procedure is the signal for that procedure to start operating. Sometimes, data must be passed to the procedure along with the call. The programmer of an object-oriented software sends a message to an actor rather than

call a procedure. A message consists of a string of English words or English like dialect describing what the programmer or another actor wants to happen, not how it should happen. The receiver of a message or one of its parents contains the actual description of the manipulation to be performed.

Procedures are called by name which specifies the exact sequence of events to happen. A message, however, may be interpreted in a different way by different actors. The message does not decide what will happen but the receiver of the message does [11]. When an actor receives a message, it tries to match it with a known pattern. If a match occurs, the associated manipulation takes place which may include the sending of messages to other actors starting a chain reaction which ends only when a reply is sent to the originator of the request. Such a chain reaction can be triggered by sending the message "TRANSFER ..." to an actor in our sample program in section 2.1.

The calling of procedures is rigid and sometimes difficult to comprehend. For instance, when a call to procedure SUBSTITUTE is made, say SUBSTITUTE (A B SET-A), does it mean

- substitute A for B in SET-A, or
- substitute A with B in SET-A?

With actor programming the message passed to actor SUBSTITUTE can be unambiguous. Furthermore, the same results can be achieved by passing different messages such as

- A for B in SET-A,
- Letter A for letter B in set SET-A,

- Character A for character B in list called SET-A.

The same message sent to the same actor using the same input data may yield different results the second time around than it did the first time because the actor may have changed its knowledge of certain facts between the first reception and the second. This phenomenon occurs because an actor contains its own operating environment which influences its behavior and that unlike procedures, it "remembers" from one activation to the next. The knowledge of an actor contains attributes as discussed earlier and it may also house a small relational database of facts which influences its behaviors also. The knowledge may include a set of rules for making decisions much like a set of rules used by humans to conduct deductive reasoning. As a matter of fact, an actor's knowledge may be made up of as many different kinds of data as needed. The items named here are merely examples, not a list of what must be in the knowledge of an actor. Therefore, a generalized definition of knowledge as applied to actor systems could be:

The (knowledge) of an actor is the set of data that composes its environment but, unlike the environment of conventional procedures, it is dynamic.

2.4 COMPONENTS

The only components of an actor system are the actors. By definition, actors are made up of two parts: knowledge and behaviors. Within the actor model of computation however, there is no way to decompose an actor into its components. An actor is defined by its behavior, not by its physical representation [9].

Actor programming is best analysed in comparison with conventional programming. To ease this process therefore, one has to look at an actor as being made up of knowledge (its data) and behaviors (its procedures) but must bear in mind that the distinction is purely academic.

The knowledge of an actor has been covered quite adequately in the previous section as a property of actors and will not be dwelled upon here. It suffices to remark that an actor could be made up of knowledge only. In that case, one compartment of the knowledge would comprise the behaviors of the actor or the procedures it executes when a message of the corresponding pattern is received.

The behaviors of an actor consist of two distinct components:

- a pattern, and
- a script.

Message patterns must be written using certain conventions. At the present time, there is no consensus on a convention and every system defines its own. For example, well known

conventions of written languages such as "...", which stands for a missing part of a sentence, and "etc" to denote an unfinished enumeration could

be incorporated to formulate some message patterns like:

- 1) (... he said ...)
- 2) (He recalled ...)
- 3) (... Paul, Ernie, Janice etc)

Messages that match these patterns could be:

- 1) (She said goodbye and he said hello.)
- 2) (He recalled his early years at the college.)
- 3) (I invited Paul, Ernie, Janice, Laura and Pat.)

The script of a behavior is made up of executable statements or a call to a procedure. Part of a script may be sending a message to another actor or manipulating the actor's knowledge or simply executing a computation.

2.5 ACTOR MANIPULATION

Actors manipulate themselves in accordance with the script associated with the pattern matching the message received. An actor system therefore must include facilities to support the manipulation of the different components of actors.

The facilities required can be compared with those provided by an operating system to high level programs. A Fortran programmer who issues the command "READ*, VARIABLE" does not have to worry about how a value is assigned to VARIABLE but he knows

that he can use this value later in the program. Similarly, a programmer who sends the message RECALL YOUR ATTRIBUTE to an actor should not have to worry about where and how to obtain the value but simply that the value will be made available.

The facilities provided by an actor system should include:

- manipulation of an actor as an entity,
- manipulation of an actor's knowledge, and
- manipulation of an actor's behaviors.

The very first thing that the user of an actor system wishes to be able to do is to determine whether an actor exists or not. If a needed actor is not present in the system, the programmer must then be able to either transfer the actor from its storage place into the system or define/create the desired actor. The saving of actors in their current state is a corollary to the transfer facility mentioned before and is a mandatory facility to be provided. Furthermore, every actor must be able to retrieve and restore its components as if they were physically separable; i.e., "get" and "put" facilities for the components of actors must exist.

An actor needs to be able to manipulate its knowledge in basically the same manner humans do. It has to be able to

1) learn,

2) forget,

- 3) change (its mind), and
- 4) recall.

The information about its parents and its children must be a special part of an actor's knowledge because one does not think of "learning" a parent or a child but of "adding" one to an existing list, and also because the parent attribute must be used in the search for other attributes and behaviors.

The very high degree of flexibility of an actor system is based on the fact that an actor can manipulate its behaviors. An actor system must provide facilities for handling behaviors and their components and the handling must be tailored to the nature of behaviors. For instance, one cannot add a pattern without the associated script or delete a script without also deleting the corresponding pattern.

The following table shows which facilities are provided for handling a behavior and its components.

FACILITY	BEHAVIOR	PATTERN	SCRIPT	ACTION
Add	Yes	No	No	Yes
Delete	Yes	No	No	Yes
Find	Yes	Yes	Yes	No

2.6 ACTOR PROGRAMMING

Programming in an actor environment consists primarily of defining the behaviors of each actor, each behavior being a response to specific messages. It may include operations that

change the actor's knowledge, its relationship to other actors, or its behaviors; and it may result in messages being sent to other actors. In some cases, a reply is required by the sender and in other cases the reply has to be sent to other actors, or sometimes no reply is necessary. When a reply is required, the specific script to be executed is usually a function of the contents of the reply. For example, if an actor sends an "IS IT COLD?" message, the sender will likely want to execute the `IT_IS_COLD` script if the reply is YES and the `IT_IS_NOT_COLD` script if the answer is NO. In conventional programming languages, we would expect the sender of the message to be suspended until the reply is received. At that point, the contents of the reply would be examined and used to decide which of the two options to pursue. We call this the synchronous approach. Although such an approach can be provided in an actor language, it is not typical because there is no concept of suspension in actor systems. Each actor is activated as soon as it gets a message which it processes to completion. It is active or inactive but never suspended. This leads naturally to a style of programming called the asynchronous approach. In this scenario, the actor would have completed his actions after sending the "IS IT COLD?" message and would have a specific script for a "YES IT IS COLD" message and another one for a "NO IT IS NOT COLD" message. The actor is either active or inactive depending on whether it has a message to process or not but never suspended, where suspended means being made inactive in the middle of processing a message.

The following shows how the part of an actor's behaviors dealing with the "IS IT COLD" message sequence could be programmed using the synchronous and the asynchronous styles:

Synchronous

Pattern: (DO SOMETHING)

Script: ((IF (EQ (ASK TEMPERATURE IS IT COLD?)

(YES IT IS COLD))

(THEN (TELL JOHN PUT ON YOUR JACKET)

(TELL MARY WEAR SLACKS))

(ELSE (TELL JOHN REMOVE YOUR JACKET)

(TELL MARY YOU MAY WEAR SHORTS)))

(REPLY "MARY AND JOHN ADVISED"))

Asynchronous

Pattern: (DO SOMETHING)

Script: ((ASK TEMPERATURE IS IT COLD?))

Pattern: (YES IT IS COLD)

Script: ((TELL JOHN PUT ON YOUR JACKET)

(TELL MARY WEAR SLACKS)

(REPLY "MARY AND JOHN ADVISED"))

Pattern: (NO IT IS NOT COLD)

Script: ((TELL JOHN REMOVE YOUR JACKET)

(TELL MARY YOU MAY WEAR SHORTS)

(REPLY "MARY AND JOHN ADVISED"))

Under the asynchronous concept, the control of the behavior of an actor depends only on the message that activated it and on the knowledge contained in the actor receiving the message. This facilitates the scheduling of messages but it makes an actor's behaviors harder to read and comprehend. This kind of approach may also cause more behaviors to be programmed, but may cut down on the number of messages passed. It may generate the creation of more actors which in a parallel environment would probably speed up execution since many actors can be activated at the same time.

An actor system that allows the synchronous approach for behavior programming must be able to support the concept by suspending and reactivating actors. An actor system that uses asynchronous programming, on the other hand, must also differentiate between messages which require a reply and others that do not. This fact leads to the concept of continuation. The system must know where to direct the reply to a message since the originator of the request may not necessarily be the recipient of the reply. For example, the "DO SOMETHING" message of the sample program above must tell our actor (call it myself) where to reply to. Similarly, the "IS IT COLD?" message sent to actor TEMPERATURE must specify that myself requires the reply. The continuation used by the system must "travel" with the messages since they are the only links between actors which in turn are the only entities of the system.

2.7 ACTING IN PARALLEL

In theory, all actors which have a message to act upon, could be executing at the same time. Of course, as in other programming languages there is sometimes a need to carry out certain steps in a specific sequence. An actor programming environment must recognize what can be done in parallel and what must be done sequentially.

An actor system activates as many actors as the supporting hardware can manage simultaneously. An actor can act upon only one message at a time necessitating careful ordering of messages. The order of execution should be that of arrival of messages but some systems provide a priority scheme allowing certain messages to be scheduled before others. Since the sending order is not guaranteed to be the receiving order, where a choice exists, an actor system must assign the script of messages that must be executed in series to the same computing facility.

Actor systems thrive on parallel processing by the very nature of their composition and the full power of actor programming will not be discovered until parallel processing is available on a larger scale.

2.8 DYNAMIC CREATION

One of the main advantages of an actor system over conventional programming is that actors can be created when required as a function of the current environmental conditions. This dynamic creation provides much programming flexibility.

Unlike in conventional programming, where the programmer has to foresee all the eventualities and solve them in procedures or subroutines which are called as required, an actor can be created differently every time the program is executed depending on the circumstances of its creation. In conventional programming, the creation of a subroutine or of a procedure is done in a fixed environment and depends only on the part of the calling environment which is passed through parameters. The creation of an actor on the other hand, can be tailored to the surroundings in which it is created. For example, in an air traffic control application the same airplane may visit an airfield every day, causing the creation of the same actor in the model of the airspace every time. That actor will behave differently from one day to the next because of different payload, different captain, different weather conditions, whether it is on schedule or not, and finally its behaviors would be adapted to the air traffic over the airfield for that particular day. It is worthy to note also that some of these factors do not have to be explicitly incorporated in the actor because of the inheritance property of actor systems.

Since the behavior of an actor can be changed while a program is executing, an actor can adapt a behavior to a changing environment. Assuming that the aircraft mentioned in the previous example comes into the airspace having lost half of the power of one engine, its approach and landing "behaviors" would certainly be different. Actor programming allows this situation

to be simulated without reprogramming any part of the model.

Dynamically changing the behavior of an actor does not necessarily change the behavior of other actors in the same class. Changing the behavior of one actor however, may affect the behavior of other actors. If this is the case, the behavior of the second actor is changed by the use of demons.

Dynamic creation of actors by actors is also a very important feature of an actor system. The full power of this particularity can be used to create required control structures. It is one step further than an actor asking another actor to do something, an actor creates another actor and tells it what to do when receiving certain messages. A good example of dynamic creation of an actor by another actor is the creation of a "control actor" to take over from the "parent" actor. If an actor needs a reply from another actor to continue executing a script, it creates an actor and tells it what to do when the reply comes. This frees the original actor to handle another request if needed, while its newly created helper takes care of the last message. This technique prevents "suspending" an actor while other actors it depends on are activated. To prevent the proliferation of control actors, it is normal to place a self-destruction instruction as the last action carried out by the dynamically created actor.

In theory, actors can be created or deleted at any level of an actor hierarchy. Of course, one must protect against users'

interference into the basic structure of a system which is also made up of actors. Removing an actor from a level other than the bottom level should not destroy the system; i.e., the system should be able to operate even after removing a part of a chain of actors. The system will remain alive as long that there is a path to the top level actor from any actor in the system in order to protect the inheritance property which is the corner stone of an actor system.

CHAPTER 3

O A S Y S

3.1 ORDINARY ACTOR SYSTEM (OASYS)

OASYS is an actor system implemented in CP6 LISP. It consists of one actor called SYSTEM and a supporting environment which is an extension to CP6 LISP.

OASYS can support all application programs which do not make use of floating point mathematics since CP6 LISP does not accommodate floating point numbers. It is an interactive system that provides facilities for manipulating actors. Sample sessions with OASYS are in appendix H. They show an actor program that sets up the actors and two interactive sessions with the newly created actor system. The operator interacts with the system by simply asking actors to do something. Requests to actors are classified in four general categories:

- Actor Creation and Deletion,
- Actor Knowledge Manipulation,
- Actor behaviors Manipulation, and
- Storage and Retrieval of Actors.

An actor may be asked to create another actor. The new actor, whose name is specified by its creator, is made a child of that actor unless the requesting message contained a list of parents for it. At the time of creation, an actor possesses no knowledge other than that of its name. Its behaviors component is void. It contains information about its parents but it has no children.

An actor may be asked to make an exact image of itself. The

duplicate actor bears another name and is not a child of its creator but a brother. At the time of creation of the clone, the system contains two actors distinct by their name but similar in all their components. The two actors however, are separate entities and may soon become different from one another.

There are times when an actor has run its course or served its purpose and must be purged from the hierarchy. Deleting an actor from the system can be done in two ways:

- 1) the actor can be removed leaving the rest of the structure intact or
- 2) the actor can be removed along with the entire subordinate branch of the hierarchy.

Removing the parent links of the actor (which must be concurrent to removing the actor's name from the list of children of its parents) simply disposes it from the hierarchy, it does not remove it from the system. Removing the parental links makes the actor stand by itself, outside the hierarchy, deprived of the essential inheritance property of actors, but it does not discard it completely from the system since it can still handle requests for which it contains the necessary script. To completely eliminate an actor, one must therefore destroy all of its components.

If it is necessary to remove an actor from the system without disturbing the structure, OASYS ensures that the behaviors of the actor leaving the system are passed to its children. Behaviors whose pattern already exists in a child however are not changed due to the departure of the parent. The system also ensures that the children of the departing actor become children of every one

of its parents. Figure 3.1 shows before and after pictures of the elimination process.

When a chain of actors must be destroyed, OASYS eliminates the actor specified in the message as the root of the hierarchy then it follows all of its children links to the bottom level destroying the actors which have only non-existent actors on their list of parents.

Assume the following network of actors:

and further assume that actor A receives the message: "Cease to exist and take your dependents with you". The system eliminates actor A then travels down AD to find that actor D's parents list consists only of actor A which no longer exists, so actor D is discarded and link DH travelled. Actor H is removed from the hierarchy and so is actor J. After travelling the JF link the system finds that actor F has actors J and E for parents and since actor E is still "alive", actor F is left in the system but actor J is removed from its list of parents. The only other actor to be "investigated" is actor E which is found to be still useful and left in.

The second category of requests to actors is handled by

OASYS's facilities for knowledge manipulation. An actor's knowledge comprises attributes. Attributes can be pictured as subjects under which information is stored or the title of a relation when the knowledge is viewed as a relational database. The variety of information stored has no limit: it can be a single value, the name of a procedure to be called, a pattern to be matched, a list of many other attributes, a list of actors and so on. There are also many ways to store the same information. For example, if the two facts "the sky is blue" and "the ocean is blue" have to be stored in an actor, it can be done

- by having a FACT attribute and placing the two values (SKY IS BLUE) and (OCEAN IS BLUE) in it or
- by having a BLUE attribute and storing the values (SKY) and (OCEAN) in it or
- by storing the value (BLUE) in both attributes SKY and OCEAN or
- by having two attributes (SKY IS BLUE) and (OCEAN IS BLUE) and storing the value (FACT) in both.

The manner in which attributes are utilized is left to the users of OASYS, the system simply provides for the manipulation of these attributes in the following ways:

- An actor may list all its attributes.
- An actor may set a particular attribute to a specific value as if it were an ordinary variable.
- An actor may recall an attribute or more

explicitely may recall the content of an attribute.

- An actor may show the operator the contents of an attribute.
- An actor may add a new value or remove an old one.
- An actor may carry out integer mathematics on numerically valued attributes.
- An actor may forget or remove a value from an attribute.
- An actor may forget an attribute.
- An actor may forget every attribute it has i.e., wipe out all of its knowledge but for its name.

Behavior manipulation is probably the most important part of any actor system. Above all, the user of OASYS must be able to tell the actor he created what to do when it receives certain messages. The messages specified need not be different from those of a parent but neither is it necessary to repeat a behavior if it is identical to one already described in a parent of the actor; i.e., since the inheritance property of actor systems is handled by OASYS, it is not any of the user's concern. A behavior consists of a message pattern and a script to be executed when the received message matches the corresponding pattern. A user cannot specify a pattern without its corresponding script and conversely, a script is meaningless without a message pattern. If an actor does not possess specific behaviors or if it receives a message for which it does not have a script, it relies on the inheritance property of actor systems

to reply to the message. In other words, the actor's behaviors are checked for a match with the received message and if one does not occur, its ancestors' behaviors are investigated. The first behavior encountered with a matching pattern sees its script executed in the context of the actor which received the message. If neither the actor nor its ancestors can respond to the message then a complaint is sent back to the originator stating that the request could not be handled.

OASYS caters to the manipulation of behaviors of actors by ensuring that every actor in the system can

- 1) store and remove a behavior
- 2) add and delete actions from a script, (the actor can only recognize the script to be modified by its associated message pattern), and
- 3) list its patterns or a script associated with a given pattern.

An actor may need to store behaviors of its own when it is created or later as the model develops to modify its participation. Similarly, an actor may need to remove some behavior either because it is no longer required or because its response to a message of that pattern must now conform to the script stored in one of its parent.

As the model handled by an application program develops, there might be a need to modify the script associated with a

message pattern in an actor. Every actor in OASYS can add or delete actions from a particular script. The system allows one or more actions to be added or deleted with a single command while it permits only one behavior to be handled per message sent. The distinction is made so that the user has good control over the process and also because when messages of that nature are sent from one actor to another, there is seldom a need for a large scale operation.

In the light of the above discussion, it is easier to understand why an actor can reply differently to the same message received several times: not only its knowledge may have changed as explained earlier but its behaviors could have changed too. It is also possible, of course, that a message that generated a system warning to the effect that it could not be handled by its receiver generates a reply when sent to the same actor a second time.

The last facility provided for behaviors manipulation is a "help" feature which lets every actor in OASYS assist a user by listing the message patterns peculiar to that actor. This assistance can be misleading if interpreted to mean that only messages of the listed patterns can be sent to the actor since other messages can be replied to by the actor if the pattern exists in one of its parents. OASYS lists only the patterns stored at the actor queried because a complete list of message patterns could be very long and therefore useless. The purpose of this feature is to assist a user who specifically wants to

verify that certain behaviors exist in an actor without being inconvenienced by the scripts.

For the user who wants to inspect the script of a behavior interactively, OASYS provides for every actor in the system to comply with a request to list the script for a specified message pattern.

Actors can be copied individually to a specified file or a complete application can be saved with one command.

It is sometimes desirable to save the present state of an actor as an individual rather than as part of a hierarchy. OASYS provides facilities to save and retrieve individual actors. More often however, a user will want to save all the actors participating in a model or application.

The saving and retrieval of actors, and of actor hierarchies, is normally done by actor SYSTEM but user-defined actors can do it. The saving of an application is most safely done by first creating an actor bearing the application's name at the head or root of the actors hierarchy for the application then asking either SYSTEM or the new head actor to save the application. If actor SYSTEM is asked to save an application which does not have a root actor of the same name, SYSTEM creates the root and assumes that its children belong to the application to be saved and places them on the list of children of the new root-actor before it saves the application.

Additional facilities of OASYS include tracing messages sent and received while a program is running and an actor definition facility to specify an actor interactively. Sending messages to actors when building up a model may be impractical due to the number of actors to be created. OASYS provides for interactive definition of actors where a user is prompted for the different components of the actor to be created until it is completely characterized.

Actor SYSTEM and a list of messages that it understands is described in more details in Appendix G. Since every actor in the system is related to actor SYSTEM, these messages are also understood by every actor in the system.

In order to grasp the details of OASYS, it is necessary to comprehend the LISP environment in which it operates. The next two sections explain very briefly this LISP environment.

3.2 LISP PROGRAMMING

LISP is a computer programming language specially designed for symbol manipulation. The main feature of LISP is that programs and data have the same syntactic form. It is the main computer language used for Artificial Intelligence programs.

Computers are well established as great manipulators of data which they represent as bits or strings of zeros and ones and the strength of computers in number crunching is legendary but people are just becoming aware of other capabilities of computers.

Once it is understood that bits can be grouped to represent characters, it is clear that characters can form words, words can be grouped into sentences, sentences into paragraphs, and so on. LISP uses a similar grouping notion, although the units have different names [1].

- The fundamental things formed from bits are word-like objects called ATOMS.
- Groups of atoms form LISTS. Lists themselves can be grouped together to form higher-level lists.
- Atoms and lists collectively are called symbolic expressions or S-EXPR.

A LISP interpreter classifies characters as either delimiters or non-delimiters. The delimiters play the role of reserved words of high level computer languages while the non-delimiters are grouped together to form atoms.

- (denotes the beginning of a list,
-) denotes the end of a list,
- [stands for one or more (,
-] stands for one or more),
- (blank character) separates atoms,
- % denotes that the delimiter that follows is to be interpreted as a non-delimiter,
- " brackets a string of characters, and
- . when preceded and followed by a blank, denotes the separation of a name from its value in a

DOTTED-PAIR.

A LISP program consists of a series of lists which are evaluated in sequence. The term evaluate rather than execute is used because LISP is made up of functions only, it does not have procedures or sub-programs like other high level languages. A function takes on or returns a value in accordance with a set of instructions applied to the parameters passed to it. In LISP the function to be evaluated is always placed at the head or first element of a list, followed by the parameters passed. For example, the list (PLUS 3 5) returns 8 as the value of the function PLUS.

An atom could be the name of a function or it could be the name of a variable, to which a value has been assigned. To generalize the notion of value and to give an atom many different values if needed, LISP uses the concept of properties. Each explicitly named property of an atom has a value associated with it and the atom keeps a list of all its defined properties in a property list. The value of a property can be any s-expr or a dotted pair. For example, the atom TEMPERATURE could be the name of a function which returns "hot", "warm" or "cold" depending on the outside temperature in degrees Celsius passed as a parameter. The same atom, used as a variable, could be assigned the value of the present outside temperature in degrees Fahrenheit. TEMPERATURE may also have properties FREEZING and BOILING in its property list where FREEZING is associated with the numerical atom 0 and BOILING with 100. Asking TEMPERATURE for its value

would yield the present outside temperature in degrees F while asking for its FREEZING property value returns 0. LISP provides functions to handle properties while the value of an atom is obtained by writing simply the name of the atom.

There are many variation of LISP, the three most important being:

- MACLISP
- INTERLISP
- LISP-MACHINE LISP

The differences between these dialects are not fundamental from a user's point of view since the syntax is basically the same for all. CP6 LISP is based on LISP F3 and is a subset of INTERLISP as documented in references [17] and [18].

3.3 AUGMENTED LISP SUPPORT

I have designed an extension to CP6 LISP to make the code in OASYS a little easier to read and to provide facilities which INTERLISP does not have but which are sometimes provided in other LISP interpreters.

The reasons for providing augmented LISP support to OASYS are:

- 1) to provide familiar control structures,
- 2) to provide additional pre-defined functions,
- 3) to rename some standard LISP functions, and
- 4) to provide for record-type handling.

LISP programming does not provide for the familiar control structures such as IF-THEN-ELSE, WHILE-LOOP, REPEAT-UNTIL, FOR-DO, etc. Although the control structures of LISP programming can become familiar to a LISP programmer, there is a need for OASYS to provide for the more familiar structures since it is aimed at computer-naive operators which may have to program behaviors of actors they create for their application. I have chosen to implement only a few structures, namely IF-THEN-ELSE, WHILE-LOOP and REPEAT-UNTIL because actor behavior programming mainly consists of sending messages to other actors and manipulating the knowledge of an actor which does not require many kinds of control structures. For readers interested in my code implementing OASYS, the added "familiar" control structures make reading easier but a familiarity with LISP is necessary to understand completely the design.

Some pre-defined functions were added to those already included in INTERLISP. For instance, I added a function MATCH (Pattern Data) that returns true or false depending on whether or not given data match a given pattern. The function is essential for deciding which script to evaluate when a message is received by an actor and for searching attributes or relations (in a relational database sense). I also added the function MISMATCH which generates an error message when an actor receives an unfamiliar message. OASYS contains some MACLISP functions which are not available in INTERLISP like FUNCALL, MAPCAN, NOT and REMAINDER. The interpretation of these functions can be found in

reference [13]. OASYS also includes the practical set functions UNION and INTERSECTION. Function REPLACE (Item1, By Item2 In Source) has been added as a complement to the standard LISP function SUBST. It differs from SUBST in three points:

- 1) it replaces only the first occurrence of an item in the source;
- 2) the search for that item is not made at the top level only, but at all levels in the source list; and
- 3) the passing of parameters has been made unambiguous by the addition of the keywords "By" and "In".

Some standard LISP functions were simply renamed to facilitate understanding of their meaning. Here is a list of these functions :

IN INTERLISP

CAR
CADR
CADDR
CDR
CDDR
CDDDR
CAAR
CDDAR
CONS

IN OASYS

FIRST
SECOND
THIRD
AFTER_FIRST
AFTER_SECOND
AFTER_THIRD
FIRST_OF_FIRST
AFTER_SECOND_OF_FIRST
CONSTRUCT_NEW_LIST

The value associated with an attribute may be a record with several fields. In order to alleviate the task of manipulating records and their associated fields, OASYS has the following functions:

- DEFINE_RECORD (Type_Name . Field_Names)
- MAKE_RECORD (Record_Name Type_Name)
- SET_RECORD (Record_Name . Field-Value_Pairs)
- RECORD_VALUE (Record_Name Field_Name)

the first two functions being declaration statements. DEFINE_RECORD establishes the fact that a record of type Type_Name has fields of certain names. MAKE_RECORD declares that Record_Name is a record of type Type_Name and makes it into a record with as many empty fields as required. The last two functions manipulate the values of the fields of record Record_Name.

The detail of the above functions can be found in Appendix A.

3.4 ACTOR REPRESENTATION

There are several ways of physically representing an actor. I have chosen to maintain each actor on the "ACTOR" property of the atom bearing the actor's name. The value of the property, as shown in Figure 3.2, is a list of four components:

- the actor's knowledge,
- the actor's behaviors,
- the actor's parents, and

- the actor's children.

Figure 3.2 ACTOR REPRESENTATION

I have chosen to separate parents and children from the actor's knowledge because, although they are attributes of an actor, they have to be handled differently since they show the actor's place in the system. Special attention must be paid to the parents of an actor since the parents list cannot be empty, an actor must have at least one parent: actor SYSTEM. The children list is not as critical but it deserves special status because it is a safety feature or back pointer and because it is used to load actor hierarchies from their storage files.

Each component of an actor is also a list. The knowledge of an actor is a list which has a frame structure.

A {frame} is a data structure for representing a stereotyped situation. [1] One can think of a frame as a network of nodes and relations or a tree structure where the first level is the frame.

identifier, the second is a slot name, the third is a facet, and the fourth is a value. Each path from the root to a leaf can be summarized as:

```
(Frame (Slot (Facet (Value))))
```

Applied as the knowledge of an actor, frame is the actor's name, slot is an attribute's name and facet must be one of VALUE, IF-ADDED, IF-REMOVED, IF-CHANGED, IF-NEEDED, PRIVATE-IF-ADDED, PRIVATE-IF-REMOVED, PRIVATE-IF-NEEDED or PRIVATE-IF-CHANGED. When facet is VALUE, value is interpreted as the attribute's value in the usual sense; otherwise, value is a demon's name.

This representation was chosen because it is a natural extension of a list and it is more flexible than the alternate dotted pair approach since the facet part of a slot can categorize value as the value of the attribute or a demon to be executed when appropriate changes are made to the attribute. Figure 3.3 shows a typical knowledge frame for actor ADAM.

```
*****  
*  
*          (ADAM  
*            (AGE  
*              (VALUE (35)))  
*            (SEX  
*              (VALUE (MALE)))  
*            (STATUS  
*              (VALUE (MARRIED)))  
*            (SPOUSE_NAME  
*              (VALUE (EVE))  
*              (IF-CHANGED  
*                (AMEND_SPOUSE_ACTOR))))  
*  
*  
*  
*  
*  
  
Figure 3.3 TYPICAL KNOWLEDGE OF AN ACTOR
```

Actor ADAM has knowledge of four attributes: age, sex, status and spouse name. The first three only have a value associated with them while the last one also has an IF-CHANGED demon to be executed when the spouse name is changed.

The behaviors of an actor is a list of its behaviors where a behavior is a list containing a message pattern and its associated script. Figure 3.4. shows the structure of the


```

*****
*
*      ?      Matches any S-Expression
*
*
*      +      Matches zero or more S-Expressions
*
*
*  ?VARIABLE  Assigns the S-Expression matching at that
*              position to VARIABLE
*
*
*  +VARIABLE  Assigns the list of S-Expressions matching
*              at that position to VARIABLE
*
*
*
*  Anything else matches exactly
*
*
*
*
*      Figure 3.5  MATCHING SYMBOLS CONVENTION
*
*****

```

When specifying a message pattern, care must be taken not to be so general as to cause confusion and misinterpretation. For example, message pattern (+ ?VARIABLE +) is so general that it is in fact worthless. Message patterns must be sparkled with "keywords" to ensure that variable assignments are made properly.

On the other hand, message patterns should be as general as possible to allow a user to formulate meaningful messages in his own words.

Script specifications are written in OASYS. The OASYS programming language uses the synchronous approach to actor behavior programming. The LISP-like syntax it uses is extremely simple.

(ASK/TELL actor_name message_text)

OASYS treats pre-defined INTERLISP functions as actors and a programmer can refer to them through functions ASK and TELL or in the normal INTERLISP fashion.

ASK and TELL are OASYS-defined LISP functions that first evaluate the parameters that need evaluation, and then activate the designated actor returning the value of the last evaluated S-Expr called the reply. Every parameter or parts of parameter that needs to be evaluated is preceded by the evaluation symbol, "!". The function SCRIPT may be used to identify the script of a behavior and to ensure that it is not evaluated as part of the present ASK environment but stored intact with its message pattern.

Every OASYS evaluable statement has the same form. Computer-naive people can therefore learn to use it with minimal training. Figure 3.6 shows a behavior for actor FACTORIAL.

```

*****
*
*
*
*      L(+ ?Number)
*
*      (IF (ASK LESSP !Number 2)
*
*          (THEN (REPLY 1))
*
*          (ELSE (REPLY (ASK *
*
*                      !(ASK FACTORIAL
*
*                        !(ASK - !Number 1))
*
*                          !Number))))])
*
*
*
*
*
*
*      Figure 3.6 THE FACTORIAL EXAMPLE
*
*****

```

Actor FACTORIAL responds to only one message pattern in addition to those in actor SYSTEM. The pattern is very general since it simply calls for the assignment of its last atom to the variable named Number.

For example, messages

- Compute Factorial of 5
- Compute 5
- What is the factorial of 5
- 5

sent to actor FACTORIAL would all yield the same result.

The synchronous approach to behavior programming used in OASYS allows an actor to "ask" another actor for a value it needs. In general, however, the reply does not need to return to the "asker", it can be passed on to an explicit receiver (sometimes called the continuation).

In the case of actor FACTORIAL, the script at Figure 3.6 shows a choice of two replies based on the value returned to FACTORIAL by actor LESSP. LESSP is also the name of a LISP function of two parameters, therefore the script could have been written IF (LESSP Number 2). Note that function ASK requires the identification of parameters to be evaluated while function LESSP does not. Once actor LESSP has replied, actor FACTORIAL replies either 1 or whatever reply it receives from actor *. The message sent to actor * (which plays the role of the INTERLISP function TIMES) consists of two numerical values, the first being the reply to a request to actor FACTORIAL and the second the value of Number. The nested request to actor FACTORIAL in turn comprises one numerical value supplied by actor - which is an actor equivalent to the LISP function DIFFERENCE. It is interesting to note that the request (ASK - !Number 1) could have been written in many other ways:

- (ASK DIFFERENCE !Number 1),
- (DIFFERENCE Number 1), or
- (SUB1 Number).

OASYS allows the non-programmer to use the simple actor system syntax for every action while a more experienced LISP programmer could simplify a script by using pre-defined LISP functions underlying OASYS. For instance, one could rewrite the script at Figure 3.6 as:

```
(IF (LESSP Number 2)
    (THEN 1)
    (ELSE (TIMES
           (ASK FACTORIAL COMPUTE FACTORIAL FOR
                ((SUB1 Number))
                Number)))
```

When an actor asks another actor to do something, the originator of the request normally depends on the answer to continue the execution of its script. When the operator asks an actor to do something he normally needs to see the reply. There are times however, when a user may not require a reply or when an actor does not depend on the answer to continue. In those cases, the TELL function is used. TELL returns an empty reply as its value.

3.6 HOW OASYS WORKS

By now, you may or may not have constructed a conceptual model of OASYS. Figure 3.7 presents the components of OASYS as a series of "black boxes" through which requests and replies are processed. Studying this model will provide the answer to the question: How does OASYS work?

Figure 3.7. ACTIVITY FLOW DIAGRAM FOR OASYS

When a request comes into OASYS, it is received by the request/reply processor which separates it into its components: the sender, the message and the receiver. The message is given to the message processor while the name of the receiver and that of the sender are passed to the actor activator. The same process takes place for requests that originate inside OASYS.

When a reply is fed into OASYS, it is handled in the same manner as a request except that the name of the sender is not passed to the actor activator.

When a message has been processed completely, it is passed to the actor executor and a signal is sent to the actor activator so that the name of the appropriate receiver is transmitted at the same time. The message processor may have to generate a request in the process of constructing a message. If this happens, the new request is formulated and sent through the system via the request/reply processor.

In addition to receiving names of actors for processing, the actor activator reacts to requests from other parts of OASYS. When the message processor has to suspend processing a message, the actor activator is requested to send the appropriate names of sender and receiver to it. Similarly, when advised by the actor executor that a script has to be suspended, the actor activator sends the name of the sender to the executor for storage. When requested by the reply sender, the actor activator sends the name of the appropriate sender to the reply sender for inclusion into the reply. Whenever the message processor sends a processed message to the actor executor, the actor activator is informed and sends the name of the receiver to the actor executor.

When a request or a reply and the name of its receiver or addressee are received at the actor executor, the required script is executed. When executing a script, the actor executor

interacts with the actor creator and the actor manipulator in a two-way exchange of information. If the script can be processed to the end a reply message is sent to the reply sender. If an error condition is encountered by the actor executor, the error generator is asked to produce the necessary error message to the user and the executor continues. The actor executor may have to generate a new request as part of a script. When this happens, the executor generates the request and feeds it back into the system via the request/reply processor.

Now that we have an idea of how OASYS works, let's open the "black boxes" and see what composes them. Figure 3.8 shows the sub-components of the request/reply processor, the front-end component of OASYS.

The REQUEST/REPLY PROCESSOR receives a request or a reply and breaks it into the receiver, the message and the sender. The processor has two sub-components:

- 1) a request solicitor which obtains exterior requests and
- 2) a request/reply handler which does the actual processing of both type of requests and of the replies.

When the request/reply processor passes the components of a request or a reply to the message to the message processor and the actor activator, it also tells them whether they are receiving a request or a reply.

The MESSAGE PROCESSOR is responsible for constructing meaningful messages. It also has the task of tracing messages going through the system when asked by the user. In the process of assembling a message, the message processor may generate a new request. When this happens, the partly processed message is placed in a list of partially processed messages pending the reply to this new request. The components of the message

processor are shown in Figure 3.9.

The message processor handles requests and replies quite differently. A request message is queued into the incoming messages queue while a reply message causes a search of the interrupted messages list to determine whether the reply is needed for processing a previously suspended message. If the reply is required by the message processor the partially processed message is removed from the list and placed at the tail of the incoming messages queue and the name of the actor to which the reply is to be sent is passed to the actor activator. If the reply is not required by the processor it is immediately sent to the actor executor along with some sign which identifies it as a reply message and the actor activator is informed of the action taken.

The ACTOR ACTIVATOR is that part of OASYS which keeps track of the active actors. An active actor is one which is the

addressee or receiver of a message in the system or one that is expected to receive a reply. The actor activator must therefore keep two queues: one for each class of activated actors. The originator of an "ask" is deemed to be the target of the reply whereas no reply is expected of a "tell" message. The reply target is deemed to be "nobody". The queue of actors awaiting reply shown in Figure 3.10 therefore contains a mix of "nobodies" and actor names. When actor1 sends a message to actor2, the actor activator enqueues actor1 on the queue of actors awaiting reply and actor2 on the queue of actors awaiting execution. Whenever the message processor sends a completed message to the actor executor, the actor activator removes the name of the actor at the head of the queue of actors awaiting execution and sends it to the actor executor. This strategy assumes that messages are processed in the order in which they arrive in the system. This orderly process can be interrupted as pointed out earlier when the message processor generates a new request in order to replace part of a message by the reply to this new request. In this case, processing of the message is suspended or interrupted until the reply is received. The actor activator must be advised of this delay in processing so that the right message is matched with the right receiver. When told of a suspension by the message processor, the actor activator simply removes the head of the awaiting execution queue and the head of the awaiting reply queue and sends these two names to the message processor for identification of the partially processed message. Similarly, a

delay can occur while the actor executor is executing a script. In this case, the actor activator removes the head of the awaiting reply queue and sends the name to the actor executor. Conversely, when told that processing is to resume at either the message processor or the actor executor the names are re-queued.

The actor activator must be told whether it is receiving parts of a request requiring a reply, a request not requiring a reply, or a reply. Although the actor activator always queues the name of the receiver of a message, it handles the name of the sender differently depending on this additional piece of information. A request requiring a reply sees the name of the sender placed at the tail of the queue of actors awaiting reply; a request not requiring a reply causes the name "nobody" to be placed at the tail of the awaiting reply queue; and finally, when handling a reply the actor activator ignores the name of the sender.

The responsibility of the ACTDR EXECUTOR is to find and execute the script associated with the message passed to an

actor. While executing a script, the actor executor may generate a new request. If this request requires a reply is required the actor is suspended while the new request is processed, otherwise the actor executor continues after sending the request to the request/reply processor.

If the message received is a reply necessary for continuing execution of a suspended actor, the actor executor retrieves the suspended environment from its list of suspended processes, tells the actor activator to re-queue the sender and places the partially executed script at the end of a queue after fitting in the reply just received.

The actor executor interfaces with two OASYS components that are not in the chain of "boxes" which process request and replies: the actor creator and the actor manipulator.

The ACTOR CREATOR component of OASYS creates actors and answers queries about the existence of actors.

The ACTOR MANIPULATOR can handle actors as a whole and any of the four components of an actor as if it were a separate entity. Note that actor manipulation also includes the destruction of actors no longer required and the saving and loading of actors to and from exterior storage.

The next component in the chain that processes requests and replies is the REPLY SENDER which ensures that whomever placed a request receives the reply. To properly function, it must cooperate with the actor activator. properly function. The reply sender determines where to send the reply by asking the actor activator for the name of the head of the awaiting reply queue. If the queue is empty, the reply is sent to the user, otherwise it is sent to the actor through the request/reply processor.

Finally, the ERROR GENERATOR is responsible for issuing error or warning messages to the user. There is no such thing as a fatal error in OASYS. The system always continues executing after issuing the error message.

3.7 HUMAN ENGINEERING ASPECTS OF OASYS

The primary data types in LISP are "atom", "list", "integer", "real", "character", "string" and "array". Our version of INTERLISP provides no real or array types and as in other LISP provides no means for users to define other data types. Furthermore, LISP does not formally support user defined data

types as in PASCAL or ADA. This, of course, could create problems for long programs that might become so entangled that there is no practical and easy way of making changes without affecting, unknowingly, other parts of the program. INTERLISP is also hard to read because it does not use standard control structures like other high level programming languages. Several approaches have been suggested to alleviate the problem caused by this non-definition of data types. One of the simpler suggestions comes from Charniak, Riesbeck and McDermott [3]: They suggested naming functions to reflect the data type of their parameters. For instance, a function that handles actors could have a name like GET_ACTION:ACTOR, or something that handles the behaviors of actors, a name like FIND_CODE_FOR:BEHAVIORS. This identifies the function with the data type it handles, and if a change is made to the way of representing actors or behaviors, then only the functions that have the appropriate data type notation in their name would have to be looked at for possible changes. Every other function in that program should work after the change. This scheme adds to the clarity of the code but it lengthens the function names and since another good practice is to make the names of functions as descriptive as possible to eliminate documentation, this additional descriptor might mean that most functions in a LISP program would have an excessively long name. This may lead to typing errors and omissions defeating the original purpose of adding clarity.

The organization of code in a structured manner is mainly

for human interface. In other words, to write code which is readable and understandable by people but that is not necessarily easier for the LISP interpreter to process. For that reason, the organization of code a la ADA which is an artificial way of organizing LISP code might be better.

Regardless of the programming language used there is a need to split a complex program into smaller manageable units which can be individually designed and checked [4]. An Ada program is made up of a number of logical pieces each of which covering a particular aspect of the problem to be solved. One form of such logical pieces is called a package. It defines a set of facilities which the rest of the program may use. A package consists of a package specification and may also include a package body. The package specification may contain a public part and a private part. The public part is the main portion of a specification since it provides users of the package with the necessary information about the entities it contains. The package body is required to house the body of sub-programs described in the package specification.

In OASYS the supporting elements have been grouped into five packages:

- 1) LISP SUPPORT
- 2) ACTION
- 3) KNOWLEDGE
- 4) ACTOR
- 5) SIMULATION

These packages are shown in Appendices A to E. Actor SYSTEM has also been "packaged" and is shown in Appendix G.

The package bodies which are written in LISP are not available to the behavior programmer but the package specifications which are looked at as comments by the interpreter give a good description of the facilities to the user. When packages are loaded the interpreter ignores the specifications and sets up a work space for the loaded program as if only the function definitions existed.

Packaging a la Ada allows identification of data types with their manipulations without the use of long descriptive and "auto-typing" function names. If a change has to be made to a data type, for instance, if one decides to change the way that the knowledge of an actor is to be stored, then the knowledge package only has to be reviewed; every other packages in the program should still work. Sometimes, a data type is used by a package other than the one where it is defined. The ADA structure informs the programmer about which packages contain data types not defined in the present package by using of the keyword "with" at the beginning of the specification. In our previous example, packages that state "with knowledge" at the beginning of their specification would also be reviewed since they contain facilities that manipulate one or more data types defined in package knowledge. The grouping of code in packages a la ADA also provides for a very easy to understand program documentation, and allows the hiding of the LISP code in the

Chapter 3 OASYS

90

package body.

CHAPTER 4

THE
MMM
SYSTEM

4.1 CONCEPTUAL VIEW OF THE MILITARY MAP MARKING (MMM) SYSTEM

The military map marking (MMM) system is an actor system designed to draw military map symbols on computer graphics equipment. The system is implemented in CASYS and allows computer-naïve personnel to manipulate actors representing army units.

MMM consists of seven components:

- 1) a TACTICAL INFORMATION DATABASE,
- 2) a SYMBOLS DATABASE,
- 3) a UNITS DATABASE,
- 4) a SYMBOLS DRAFTER,
- 5) a SYMBOLS MOVER,
- 6) a PROJECTION CONTROLLER, and
- 7) a real time CLOCK.

Figure 4.1 shows the relationship between all the components of MMM.

The tactical information database is a gathering of classified military information on the tactical behaviors of army units. The database is organized as a set of rules to be followed in certain situations. This part of MMM is not implemented in the current version of the system for reasons mentioned in the preface.

The symbols database contains information on military map marking symbols. The data is broken down into

- tokens representing a unit,
- tokens representing the size of a unit, and
- tokens representing the type of unit.

Theoretically, once the symbols database is set up, its mechanism handles almost only queries.

The units database used by MMM is very dynamic and consists

of data about known units. Every piece of information known about a unit is stored under a wide variety of relations and whenever requests are made through OASYS, updates to this database may occur.

The symbols drafter is the component of MMM responsible for showing the military map marking symbols at the appropriate location on the map overlay. It interfaces with the symbols database to obtain the composite symbol to be drawn and with the units database to get its actual location.

The symbols mover relieves the user from having to update the location of a moving units. It obtains information about the unit to be moved from the units database and the required timing information from a real time clock.

The component of MMM which extends its capabilities beyond drafting symbols on a map is called the projection controller. This "black box" allows the user to place the system in the projection mode in order to analyse the possible consequences of certain movement of troops. As shown in Figure 4.2, the components of the projection controller are:

- the GLOBAL STATE SAVER/RE-LOADER,
- the TACTICAL INFORMATION ANALYSER, and
- a QUERY PROCESSOR.

2

4.2 OVERALL STRUCTURE OF MMM

A military map can be viewed as three superimposed layers of information:

- 1) the contour lines and the features of the region represented by the map,
- 2) military grid lines which divide the map into several smaller areas easily referenced, and
- 3) an overlay containing military symbols marking the position of army units.

The MMM system addresses only the last component^o of a military map: the overlay of military symbols. The other two are assumed to be managed by other systems, possibly other actor systems.

MMM consists of a hierarchy of actors as shown in Figure 4.3.

Actor `MMM`, as head of the structure, is a child of actor `SYSTEM` of `OASYS`. Its children are generic actors which represent the three components of a military symbol (see section 1.3 of chapter 1), the overlays on which the symbols are drawn and a real time clock used to move symbols as required.

Actor `BASIC_SYMBOLS` is a generic actor and overall parent of all user-created actors representing units or formations known at the headquarters. Each unit drawn on an overlay uses one of the base tokens listed in section 1.3:

- 1) (a unit's location),
- 2) (a unit's approximate location), or
- 3) (a unit's headquarters location).

and since the instructions for drawing these symbols are known only to actor `BASIC_SYMBOLS`, every unit must be a child of `BASIC_SYMBOLS` to inherit these attributes.

In the military map marking system, every actor created by an operator has to specify a list of parents. One of these parents has to be related to actor `BASIC_SYMBOLS`. Trying to create an actor without specifying a list of parents produces a warning that the actor was created but that it is to operate outside the `MMM` system. If none of the enumerated parents are related to

BASIC_SYMBOLS, the actor is not created and an error message is returned.

Actors SIZE_SYMBOLS and TYPE_SYMBOLS are generic actors grouping under their class actors which contain the size tokens and type tokens explained in section 1.3. As explained in that section, a military map marking symbol is a combination of one or several tokens. An actor representing a unit of a certain type and of a certain size is therefore a child of the appropriate type actor and of the appropriate size actor.

Actor OVERLAY is the generic actor head of a class of actors representing overlays. An overlay is a screenful of information. Every overlay has a specific name or number which binds it to a specific map and, to a specific grid system. An instance of OVERLAY knows which units are part of it and where they are to be located on the screen.

Actor CLOCK is a central control and storage facility for all movements of symbols. It does not plan nor execute the moves but it contains a calendar of future moves to be made by units.

An actor representing a moving unit registers the next time it is due to move with actor CLOCK and relies on it for timely execution of the movement. Actor CLOCK stores the time of move and the name of the associated actor. It can only "remind" actors to move, it cannot specify how far or where the movement is to take place.

In addition to the generic actors, MMM contains a navigation package which provides facilities for placing a unit's symbol on the screen and for moving unit's symbols in accordance with their recorded velocity. The package includes:

- Functions to convert miles per hour and kilometers per hour into movement units for the I/O device it operates on.
- Functions to convert a unit's location into X and Y coordinates on the terminal.

The details of package navigation can be found in Appendix F.

The present prototype of MMM uses standard alpha-numeric terminals rather than graphics terminals due to lack of access to graphics equipment on CP6. For that reason it simply prints the name of a unit at the place on the overlay where it should draw the military symbol. This thesis, therefore, does not place the emphasis on the outputs of the chosen application but depicts MMM as a convertible and extendable system. Convertible in that the drawing instructions contained in the actors are easily changed to adapt to the terminal equipment used. Extendable in that it can be much more than just a military map marking system.

4.3 BASIC SYMBOLS

Actor BASIC_SYMBOLS and all of its children are the core of the MMM system. The generic actor is a child of actor MMM and contains attributes and behaviors that characterize all of its

siblings. BASIC_SYMBOLS contains attributes Base_Symbol, Locating_Symbol and Headquarters_Symbol. The value of these attributes is a set of instructions for drawing the appropriate symbol. Children of BASIC_SYMBOLS, which represent military units, can contain many attributes. The choice of attribute names is left to the user. Since the primary function of MMM is to show the position of military units on an overlay, each descendant of BASIC_SYMBOLS must contain attribute Location. They may also comprise attributes Velocity and Last_Update (this will be explained later).

The value of attribute Location must be a valid military grid reference since it is the way that military personnel refer to points on a map. A grid reference is a six-digit number referring to the grid line system of a map. The first three numbers indicate the eastings (it is a reference to the vertical lines numbered from left to right or west to east) while the last three indicate the northing of the position (it is a reference to the horizontal lines numbered from bottom to top or south to north). The three digits 205, for instance, would indicate a point half way between lines no 20 and 21, i.e. point 20.5. For example 355425 is the grid reference of the point situated at the center of the grid square depicted below:

45	----- ----- -----		
		(355425)	
42	----- ----- -----		
35			36

The position of a unit is the location of its headquarters. It is not necessarily the center of mass of its deployment.

Velocity is a necessary attribute for a unit that is in movement. It is used to automatically move a unit in accordance with its speed and compass direction. The value of attribute Velocity therefore, is a record of two fields:

- 1) a compass direction of the form SW or N for example, and
- 2) the number of spaces the symbol must be moved in one time unit. (the time unit chosen for the experimental model was one minute)

Last_Update is the attribute that contains the last date-time-group (DTG) at which the location of the unit was

changed or added. The DTG is a military compaction of information like only the military can inflict upon itself. It is basically a string of numbers and characters indicating the time and the date. For example, 2:10 pm on April 25th 1985 in England would be expressed as 251410ZAPR85. The first two numbers are the date, the next four are the time of day, the letter that follows indicates the time zone and finally, the first three letters of the month and the last two digits of the year close the string. The value of attribute Last_Update is kept current via demons which means that the user need not worry about it.

Actor BASIC_SYMBOLS contains behaviors that allow its children:

- to register their location with a particular overlay,
- to move from one location to another, and
- to draw themselves on an output device.

A unit may be drawn on an overlay using one of the three base tokens as pointed out earlier:

The symbol used depends on the amount of details that can be shown on the map to which the overlay belongs. For instance, a unit may be represented by its locating symbol on a very large scale map while its base symbol may be used to mark its position on a smaller scale map. On a map that allows showing the components of the unit, the headquarters symbol may be drawn at the location of the unit.

To register its presence on an overlay, a unit must specify:

- the type of basic symbol to be used on that overlay, and
- the location where it is to be placed.

The choice of basic symbol to use is made by the operator. If the message asking a unit to register its presence on a map does not contain any specific basic symbol, MMM defaults to the base symbol. If the unit actor does not specify any overlays in its message to generic actor OVERLAY, the actor is registered with each overlay that contains the grid reference of the unit.

Every actor that represents a moving unit may receive from actor CLOCK a message ordering it to move. When such a message is received the actor updates its location in accordance with the time lapsed since the latest update and its current Velocity value. The new location is then re-registered with all the overlays where the unit is represented. In addition to moving messages, a moving unit may receive messages from the user amending its Velocity attribute. These messages conform with

general UASYS message patterns updating attribute values explained in part 3.

When an actor is asked to draw itself on an output device, it must be told also which basic symbol and which X-Y coordinate pairs to use. This behavior of BASIC_SYMBOLS is device dependent and will not be detailed here. Suffice it to know that it contains the necessary instructions to output the required information on the device.

4.4 SIZE SYMBOLS

Actor SIZE_SYMBOLS is a generic actor child of actor MMM, which simply provides a convenient way of grouping all the different size symbol actors together. It does not contain any attributes or behaviors. Its only role is that of superclass of the hierarchy shown at Figure 4.4.

The children of actor SIZE_SYMBOLS contain an attribute named Size_Symbol. The value of that attribute is the set of instructions necessary to draw the particular size symbol. Every child is different in that it contains a different symbol. Each child of SIZE_SYMBOLS also include a Strength attribute which houses the number of personnel normally manning a unit of that size.

Organizing SIZE_SYMBOLS in such an actor hierarchy may seem wasteful at first but since every actor representing a unit of a certain size is created a child of the size actor, the particular Size_Symbol attribute is available to all unit actors of that size through the inheritance property. This implementation provides for the value of a Size_Symbol attribute to be stored only once and yet made available to the many units that need it.

4.5 TYPE SYMBOLS

Figure 4.5 TYPE SYMBOLS

Actors ARMS and SERVICES, in addition to being superclasses of the hierarchies shown at Figures 4.4 and 4.5, contain attributes Type_Symbol and Role. The value of Type_Symbol is the set of instructions used to draw the particular type symbol while

attribute Role has for value the text string describing the general role of all arms or service units. This role attribute is used mainly as a default value when units are not given any particular role or mission of their own.

Figure 4.6 ARMS HIERARCHY

Figure 4.7 SERVICES HIERARCHY

Each instance or child of ARMS and of SERVICES has its own

Type_Symbol attribute. They also contain the Role attribute of units of their type.

None of the actors in the TYPE_SYMBOLS hierarchy contain specific behaviors. Of course for different attributes, the hierarchy serves the same purpose than the SIZE_SYMBOLS organization.

4.6 OVERLAY

Keeping track of military operations involves keeping several maps up-to-date. Military maps are marked using military symbols drawn on overlays. Actor OVERLAY is the generic actor which characterizes the behavior of all the overlays which are stored in a particular MMM workspace.

When an overlay is asked to place a unit at a certain location, it first converts the grid reference into an X-Y coordinate pair representing that point on the overlay. It then stores the information as a record of three fields:

- 1) X-Y coordinate.
- 2) Name of Unit (or actor representing it).
- 3) Kind of Basic Symbol to be Used.

If two or more units fall into the same X-Y coordinate location, the conflict is not resolved at this point but is dealt with at the time of printing the overlay.

When asked to show itself, an overlay resolves conflicts if

any, by moving some units to neighboring points until no more than one symbol remains at any occupied position on the screen. It then asks each actor representing the units to be shown to draw themselves at the designated X-Y point on the screen using the specified basic symbol. At the bottom of the overlay an identification is printed showing the date and time of the printing. The identification is of the form:

Overlay No XXX 251410ZAPR85

An overlay may be asked to remove a certain unit from a particular location. The given name-location pair specified in the message must match some information stored at the overlay. Otherwise the removal does not take place and an error message is issued.

There is sometimes a need in the military to make composite maps from existing maps; i.e., to take several maps and to merge them into one. In that case, the composite overlay is made from several actors but it does not become a new actor itself. If there is a need to "create" a new actor which is a composite overlay, a "create" message must be used.

4.7 CLOCK

When a symbol is placed on an overlay, it represents current information but if the unit is moving, the information quickly becomes out of date. In order not to burden the operator with frequent updates of positions, the MM system changes the

location of moving symbols automatically. This feature is implemented by activating actor CLOCK which in turn sends a message to all actors which have to change their position at the present time.

Actor CLOCK is activated whenever an overlay is asked to show itself on the screen. When CLOCK receives a message, it sends a message to all actors requiring an update of location, not just those of the overlay that activated it.

The chain reaction generated by the activation of actor CLOCK is as follows:

- 1) A message is sent to all units which registered a move for the present time or for an earlier time. Actor CLOCK sends the current real time as part of its message so that the receivers can determine how far to move. The move is deleted from the calendar of future events after the message is sent to the unit.
- 2) An actor receiving a "move" message determines where to move to in accordance with its last time of move and its current speed. It sends a message to all the overlays where it is registered to change its location and finally sends a message to CLOCK to register its next move.
- 3) When actor CLOCK receives a registration message, it simply places the information in its

chronologically sorted calendar.

Update messages sent to overlays and to CLOCK are sent by a demon which is activated when the value of the Location attribute of a unit is changed or added. Whenever an actor changes, it "de-registers" whatever planned move it had with CLOCK and registers a new one.

The real time at which a symbol has to be moved, not only depends on the speed of the unit, but on the scale of the map. The same actor therefore, shown on different maps, may move at different times on the different overlays.

4.8 A SESSION WITH MMM

The following is an actor program written in OASYS that sets up actors which will serve to illustrate the way MMM can be used. The program is divided into two parts:

- 1) the creation of the necessary actors, and
- 2) the setting of certain attributes for these actors.

```
(ASK MMM CREATE 10CIBG CHILD OF
```

```
  (BASIC_SYMBOLS BRIGADE INFANTRY))
```

```
(ASK MMM CREATE 1RML CHILD OF (10CIBG BATTALION))
```

```
(ASK MMM CREATE 2RML CHILD OF (10CIBG BATTALION))
```

```
(ASK MMM CREATE 3RML CHILD OF (10CIBG BATTALION))
```

```
(ASK MMM CREATE 12RCB CHILD OF
```

(10CIBG BATTALION ARMOURED))
(ASK MMM CREATE 105RR CHILD OF
(10CIBG BATTALION ARTILLERY))
(ASK MMM CREATE 10SVC CHILD OF
(10CIBG BATTALION SERVICE))
(ASK MMM CREATE 27MAINTCOY CHILD OF
(10SVC COMPANY MAINTENANCE))
(ASK MMM CREATE 35SUPCOY CHILD OF
(10SVC COMPANY SUPPLY))
(ASK MMM CREATE 21PTCOY CHILD OF
(10SVC COMPANY TRANSPORT))
(ASK BASIC_SYMBOLS CREATE ENEMY?)
(ASK OVERLAY CREATE OVER001)

(ASK 10CIBG SET YOUR MISSION TO
"Protect the right flank of First
Canadian Division along the Wrangley road")
(TELL OVER001 SET YOUR CORNERS TO
(901010 960010 901200 960200))
(ASK 10CIBG SET YOUR LOCATION TO 932101)
(ASK 1RML SET YOUR MISSION TO
"Attack and destroy enemy at 946184")
(TELL 1RML SET YOUR LOCATION TO 946110)
(TELL 2RML SET YOUR LOCATION TO 910073)
(ASK 3RML SET YOUR LOCATION TO 944134)
(ASK 12RCB SET YOUR LOCATION TO 927127)
(ASK 10SVC SET YOUR LOCATION TO 912005)

(ASK ENEMY? SET YOUR LOCATION TO 946184)

(ASK 1RML SET YOUR VELOCITY TO

(N !(MPH_TO_SPEED 20)))

With the stage set by the above program, a typical session with MMM could be as follows (user input is preceeded by "<->") (added comments are preceeded by ";"):

<->(ASK OVERDU1 SHOW YOURSELF)

Overlay No: 001 141632ZAPR82

<->(ASK 1RML RECALL YOUR MISSION)

Attack and destroy enemy at 946184

<->(ASK 2RML RECALL YOUR MISSION)

Protect the right flank of First Canadian Division along
the Wrangley road. ;2RML was not given a specific mission,
therefore the mission of its parent
10C1BG becomes his.

<->(ASK 10SVC RECALL YOUR ROLE)

Support 10C1BG with combat supplies and services. ;Actor
10SVC was not specifically given a role in
the set-up, it inherited the role attribute
of actor SERVICES, one of its parents.

<->(ASK 105RR RECALL YOUR LOCATION)

932101 ;Actor 105RR is not assigned a specific location,
so it "assumes" that it is co-located with
10C1BG which is its only parent with a location
attribute value.

<->(ASK MM TRACE MESSAGES)

(MM REPLIES: (TRACE ON))

(TRACE ON)

<->(ASK OVER001 SHOW YOURSELF AT 141640ZAPR82)

(OVER001 receives: (SHOW YOURSELF AT 141640ZAPR82))

(CLOCK RECEIVES: (UPDATE AT 141640ZAPR82))

(1RML RECEIVES: (MOVE AT 141640ZAPR82))

(OVER001 receives: (REGISTER 1RML AT LOCATION 946158))

(OVER001 REPLIES: (DONE))

(1RML REPLIES: (DONE))

(CLOCK REPLIES: (UPDATED))

(OVER001 RECEIVES: (RECALL YOUR UNITS))

(OVER001 REPLIES: (ENEMY? 10CIBG 1RML 2RML 3RML 12RCB 1CSVC))
(OVER001 RECEIVES: (ADD (ENEMY? (46 18)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((ENEMY? (46 18))))
(OVER001 RECEIVES: (ADD (10CIBG (32 10)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((10CIBG (32 10))))
(OVER001 RECEIVES: (ADD (1RML (46 15)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: (((1RML (46 15))))
(OVER001 RECEIVES: (ADD (2RML (10 7)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((2RML (10 7))))
(OVER001 RECEIVES: (ADD (3RML (44 13)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((3RML (44 13))))
(OVER001 RECEIVES: (ADD (12RCB (27 12)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((12RCB (27 12))))
(OVER001 RECEIVES: (ADD (1CSVC (12 1)) TO YOUR LIST OF MAP))
(OVER001 REPLIES: ((1CSVC (12 1))))

(OVER001 REPLIES: (" Overlay No: 001
 141640ZAPR82"))
 Overlay No: 001 141640ZAPR82

<->(END)

(OASYS TERMINATED - - YOU ARE NOW IN CP6 LISP - - THANK YOU)

4.9 MMM AS A DECISION MAKING TOOL

MMM is designed to depict tactical situations by drawing military map marking symbols on overlays. The actors used however, can contain enough information to become decision making tools.

For instance, if generic size and type actors contained an attribute whose value is a set of tactical rules applicable to units of their size or type, tactical decisions could be made by a unit based on these rules. Assuming that an actor stores

rules. Assuming that an actor stores information about

- the enemy units facing it,
- the friendly forces along side of it,
- the surrounding terrain,
- the present weather conditions,
- and other relevant attributes about itself and about its enemy;

it could be asked to assess the chances of success of an attack on the enemy at the present time. This new role for actors can be achieved simply by adding information or new attributes to the actors in MMM and ensuring that generic actors contain the necessary behaviors to answer the new type of queries.

To use MMM as a decision making tool, the user would have to place the system in a different mode, say a "look_ahead" mode. This could be done by storing the present state of every actor in the system at the change of mode and restoring MMM when returning to "normal" operation.

Using MMM in the look_ahead or projection mode provides for an opportunity to assess the possible results of certain movements of troops without issuing the implementation orders. MMM could be very useful to Commanding Officers in helping them to assess the current situation and the possible outcomes of certain battlefield moves but the system will never be able to make the necessary decisions for them.

MMM has not been developed into an expert system which could

be used as a decision making tool in the present prototype. It is clear however, that it could be done as an extension to the existing MMM system.

CHAPTER 5

OTHER APPLICATIONS
AND
RECOMMENDATIONS,

5.1 GENERAL

This section gathers subjects that are intrinsically unrelated. Their only common element is that they are related to actor systems.

At one point in the research, I implemented a simulation of MMM in a distributed environment. This venture proved to be very expensive and did not unveil any unexpected behaviors. Section 5.2 [Actor System in Distributed Environment] discusses some of the general concepts derived from the attempt at distributing MMM.

Actor systems can be implemented in any high level computer language. Some languages, however, are better suited than others for this very particular application. In section 5.3 [LISP as a Base for Actor Programming], I specify why LISP seems to be the best language.

Section 5.4 [Actor Systems Programming Style] could have been fitted into chapter 2 but I kept it separate because my recommendations pertain to the programming of applications using actors and not to programming the actor system itself.

As I developed MMM, I kept resisting the temptation to change the thrust of my research to some other area where I could visualize an actor program solution. Some of the areas where I thought actor programming could be applied are listed and briefly discussed in section 5.5 [Other Application Areas].

5.2 ACTOR SYSTEMS IN DISTRIBUTED ENVIRONMENTS

A distributed computing environment is said to exist when two or more computers are interconnected by a physical medium [19]. The medium could be anything from a co-axial cable to a satellite link. The ultimate aim of distributed systems is to make the network of computers transparent to users of any particular computer in the system. In effect, this implies that a user anywhere in the system is virtually linked through his terminal or interaction device to all the computers of the network.

An actor system operating in a distributed environment must provide a link, virtual or physical, between actors related to one another but not physically co-located. This link can be achieved through an interface with the supporting network protocol. The actor system interacts with the distributed system protocol in the same way any other application programs do. The details of such an interface are beyond the scope of this thesis and will not be covered here.

The present discussion of actor systems in distributed environments therefore limits itself to examining the principles to be applied when determining how to distribute actors' hierarchies in a computer network. The prominent factors affecting the distribution of actors are:

- 1) communication delays, and
 - 2) actors' relationship.
-

In an environment where communication delays are significantly long, an actor system can be implemented by duplicating the generic actors in every computer where their children are present. This presents a new problem however: How to maintain consistency when changes are made to actors that are supposed to remain duplicates of each other.

The alternate solution is to implement the inheritance property of actors by duplication: i.e., a child physically contains all the behaviors and attributes of its parents. In most systems, it is easier to keep several identical copies of the same actor; than to keep several behaviors and attributes identical in several actors not necessarily located in the same computer.

In an environment where communication delays are insignificant, the structure of an actor system can be broken anywhere. Messages may be sent to actors as if they were all implemented in the same computer. This kind of environment also provides for a choice in the implementation:

- 1) once the script of a behavior has been found, it can be sent to the actor who needs it and executed in its environment, or
- 2) the knowledge of the actor which received the request can be passed in the operating system's search request and, wherever the script is found, execution can take place in the proper,

environment.

For the army command and control system which is to be implemented with very long communication delays, the MMM generic actors structure should be implemented in every computer of the network. There is no problem with such an implementation as long as the knowledge and the behavior of a duplicated actor does not change. When a change to the knowledge or to the behavior of the actor becomes necessary however, a mechanism has to be devised to ensure that the copies are identical at all times. This problem has been studied by researchers in distributed database management systems and protocols for the synchronization of update transactions in distributed database systems have been designed which could be used for a distributed actor system [8].

Each computer of a network where an actor system is to be distributed has to contain the necessary support environment. In other words, to support a distributed MMM system, every computer where a part of MMM is implemented must be equipped with OASYS.

5.3 LISP AS A BASE FOR ACTOR PROGRAMMING

In the field of artificial intelligence, LISP is by far the most popular language of implementation. Actor programming as a technique used in artificial intelligence, is normally based on LISP. The advantages of LISP are:

- 1) the provision of interactive development,

- 2) the provision of interactive modifications,
- 3) the powerful handling of symbols,
- 4) the provision of automatic garbage collection for the data structures,
- 5) the provision of dynamic scoping, and
- 6) the easy implementation of natural language interface.

Languages such as LISP allow the development of programs to be done interactively using an interpreter. Once a program is working satisfactorily, a compiled version of it may be used to speed up future execution.

Interactive development can be followed by interactive modification of an actor program. Using LISP as a base for actor programming allows changes to be made to the behaviors or attributes of actors without stopping the system as it would in an environment that requires compilation.

Handling of symbols is the most important feature of an actor system. The fact that data and program are structured in exactly the same way in LISP provides for very powerful symbol manipulation. In addition, LISP as a base language provides the data structuring and handling of garbage collection which frees the actor system to handle the passing of messages.

Since LISP uses dynamic scoping, the value of a free-variable used by a function is determined by the so-called activation environment [1]. This feature makes implementation of the

inheritance property of actor systems a natural. Setting of global variables such as "myself" further allows scripts to be executed in the right environment no matter where they are physically stored. One last advantage that comes to mind is that programming in LISP, or in a dialect of LISP, allows user interaction to be made to look like english sentences placed in parentheses to denote their context.

The main disadvantage of using LISP for large programs of any kind, whether it is an actor system or not, is the fact that data type declarations cannot be made explicitly as part of the system. As I explained earlier, some artificial means have to be used to modularize data type declarations, and their supporting and handling functions. Although actor system programming is by nature very modular, the underlying LISP programming can be a muddle. For that reason, most people developing actor systems or working in any other artificial intelligence fields, design their own dialect which inevitably has a strong LISP flavor. In general, LISP programs are hidden from unfamiliar users so that the intricacies of the base does not deter from the simplicity of the top level.

5.4 ACTOR SYSTEMS PROGRAMMING STYLE

Actor programming allows a very flexible approach to getting things done and can lead to the elimination of much documentation of programs if messages sent have English sentences structures. This point was illustrated earlier using the example of actor

SUBSTITUTE VS function SUBSTITUTE. Actor programming based on LISP can be almost as readable as natural language programming without using a natural language parser.

Programming is kept simple because it uses only one syntactical structure: a list comprising an actor command, the name of an actor and message to be passed to that actor. The user can also help readability by applying the following rules to actor behavior programming:

- Use longer but more descriptive names to avoid confusion.
- Use message patterns that are complete sentences rather than the telegram type.
- Use object oriented structures as much as possible, even if it means programming an extra behavior, or even creating an extra actor.
- Use asynchronous style only when absolutely necessary.

5.5 OTHER APPLICATION AREAS

Actor systems can be designed for several areas. The MMM system described in this thesis has been designed for one particular application: marking military map overlays in an army command and control system. Other applications could be in the area of:

- Automated office,

- Management information systems, or
- Training simulators.

An automated office environment is made up of several sub-systems each capable of operating on its own, but linked together by a communication media. Three such activities where an actor system could be implemented are:

- 1) word processing,
- 2) database query, and
- 3) communications.

Word processors are becoming more "intelligent" with every new generation released. Words can be manipulated in the text editor sense and corrected if misspelled. A word processor could be implemented using an actor program where each word is an actor. The actor could contain a dictionary definition of itself, some synonyms, syntactical and semantic information among other attributes. The system could provide the operator with an online help service, a misspelling correction facility and a warning system for syntactical mistakes. All of these facilities could be provided by actors activated by the operator when he needs help. The same actors can be activated also by the system when running a verification of text. For instance, when the "misspell" actor receives a message from the operator, it checks the last word typed in and corrects it if only one possibilities exists but prints a list of possible words if it cannot make the correction. When actor "misspell" receives a message from the

operating system as part of a text verification scheme, one or several words are passed in the message. The script executed in this case could call for a correction or for sending a list of possible words to actor "syntax" so that the latter can try to correct the misspelled words.

In general, a database query is answered by checking every tuple or record stored against a given pattern. If the database is an actor and every relations or tables of information that compose it are attributes, database queries in an automated office environment could be done using several small databases organized in a hierarchical manner. The added flexibility of such an implementation can be imagined easily.

An automated office relies heavily on its communication system. Centralized internal switchboards have been commercially available for some time and are getting better with every new product developed. If one were to attempt to implement an office communication system using actors, each telephone line connected to the switchboard could be made an actor that knows everything about that line and whatever is available about its habituated user.

Management information systems are decision making tools that package information in a very specific way. The scope of a management information system is normally very narrow. Implementing such a system using actor techniques could see each report generator implemented as an actor which is an expert in a

very specific aspect of the system. For instance, an actor could be a financial report maker. It could contain the necessary knowledge not only to draw up the report but also to recommend certain actions or to warn of projected overspending or underspending.

There is a continuous requirement for an army operating in peacetime to use simulators to train troops for wartime action. It is evident that many activities of an army have to be analysed without going through the real thing but relying on a faithful simulation that reveals behavioral properties without incurring the loss of human lives.

In most dynamic systems to be simulated, one can identify several component objects that have distinct properties and behave in predictable ways to given inputs. Actor programming techniques therefore seem to be a logical choice for implementing simulation. Indeed MMM simulates the movement of troops by moving their symbols on the overlays.

The RUSS language, developed at the Rand Corporation in California is the leader of simulation systems using object-oriented or actor concepts.

The application areas discussed above are by no means an exhaustive list of possible applications of actor systems. The advent of powerful micro-computers and the proliferation of personal computers make the use of actor programming techniques very attractive since actors are small and very flexible but each

actor can be made into an easily available expert in a very narrow field of operation.

5.6 CONCLUSION

This thesis is concerned with the design and implementation of a general actor based system called OASYS. The system supports multiple inheritance with demons. It was designed to support military applications.

In order to test OASYS, a prototype military map marking system (MMM) was also implemented. MMM is a decision making tool for army headquarters personnel.

It is my belief that actor based systems, because of their added flexibility over conventional programming, will enjoy a rise in popularity as increasingly complex applications are tackled. Hopefully, we have made a small contribution towards this evolutionary pattern.

A P P E N D I X A

P A C K A G E

L I S P S U P P O R T

```
*****
*
*                               PACKAGE LISP SUPPORT
*
*****
```

Type THEN is

Record

KEYWORD: Constant := THEN

EXECUTABLE_PART: Array [*] of S-Expr

End Record;

Type ELSE is

Record

KEYWORD: Constant := ELSE

EXECUTABLE_PART: Array [*] of S-Expr

End Record;

Type DO_STATEMENT is

Record

KEYWORD: Constant := DO

EXECUTABLE_PART: Array [*] of S-Expr

End Record;

Type UNTIL is

Record

KEYWORD: Constant := UNTIL

EXECUTABLE_PART: Array [*] of S-Expr

End Record;

Type PV_PAIR is

Record

FIELD_NAME: Atom

VALUE: S-Expr

End Record;

Procedure IF (CONDITION: S-Expr; THEN_PART: Then;
ELSE_PART: Else);

Procedure WHILE (CONDITION: S-Expr; DO_PART: Do_Statement);

Procedure REPEAT (DO_PART: Do_Statement; UNTIL_PART: Until);

Procedure DEFINE_ACTOR (ACTOR_NAME: Atom);

Procedure DO_NOTHING;

Procedure FUNCALL (FUNCTION: Atom; ARGUMENTS: Atom ...);

Procedure MISMATCH (MSG: S-Expr);

-- Sends an error message when an actor receives
a message of an unknown pattern to him --

Procedure SET_RECORD (RECORD_NAME: Atom;
FIELD-VALUE_PAIRS: PV_Pair);

Procedure MAKE_RECORD (RECORD_NAME, TYPE_NAME: Atom);

Function AFTER_FIRST (LIST: List) Return List;

Function AFTER_FOURTH (LIST: List) Return List;

Function AFTER_SECOND (LIST: List) Return List;

Function AFTER_SECOND_OF_FIRST (LIST: List) Return List;

Function AFTER_THIRD (LIST: List) Return List;

Function CONSTRUCT_NEW_LIST (A: S-Expr; B: List) Return List;

Function FIRST (LIST: List) Return S-Expr;

Function FIRST_OF_FIRST (LIST: List) Return S-Expr;

Function FOURTH (LIST: List) Return S-Expr;

Function INTERSECTION (SETA, SETB: List) Return List;

Function MAPCAN (ARGUMENTS: List;
FIRST_FUNCTION, AFTER_FIRST_FUNCTION:
S-Expr) Return List;

Function MATCH (PATTERN, DATA: List) Return Boolean;

Function NOT (X: S-Expr) Return Boolean;

Function RANDOM_NUMBER_1 () Return Digits Range 0..9;

Function RANDOM_NUMBER_2 () Return Digits Range 0..9;

Function REMAINDER (DIVIDEND, DIVISOR: Integer)
Return Integer;

Function SECOND (ARG: List) Return S-Expr;

Function THIRD (ARG: List) Return S-Expr;

Function UNION (SETA, SETB: List) Return List;

Function RECORD_VALUE (RECORD_NAME, FIELD_NAME:
Atom) Return S-Expr;

Function ALL_AFTER (KEY: S-Expr; SOURCE: List) Return List;

Function REPLACE (?ITEM1 BY ?ITEM2 IN ?SOURCE) Return List;

```

*****
*
*          PACKAGE BODY LISP SUPPORT
*
*****

```

```

(PRIND 'PACKAGE_BODY_LISP_SUPPORT 0 T)
(PRINT '(VERSION 31))
[DEFINEQ
[IF
(NLAMBDA (CONDITION THEN_PART ELSE_PART &LOCAL RETURN_VALUE)
  [COND [(EVAL CONDITION)
    (MAPC (AFTER_FIRST THEN_PART)
      '(LAMBDA (PART)
        (SETQ RETURN_VALUE
          (EVAL PART))
        (T (MAPC (AFTER_FIRST ELSE_PART)
          '(LAMBDA (PART)
            (SETQ RETURN_VALUE
              (EVAL PART))
            RETURN_VALUE))]]
[WHILE
(NLAMBDA (CONDITION DO_PART)
  (PROG NIL
    LOOP (COND ((EVAL CONDITION)
      (MAPC (AFTER_FIRST DO_PART)
        'EVAL)
      (GO LOOP))
    (T (RETURN))])
[REPEAT
(NLAMBDA (DO_PART UNTIL_PART)
  (PROG NIL
    LOOP (MAPC (AFTER_FIRST DO_PART)
      'EVAL)
    (COND ((EVAL (SECOND UNTIL_PART))
      (RETURN))
    (T (GO LOOP))])
[DEFINE_ACTOR
(NLAMBDA (ACTOR_NAME &LOCAL PARENTS CHILDREN PATTERN ACTIONS)
  [IF (SETQ PARENTS
    (PROMPTREAD "PARENTS: "))
    (THEN [IF (ATOM PARENTS)
      (THEN (SETQ PARENTS
        (LIST PARENTS)
        (CREATE_ACTOR ACTOR_NAME PARENTS'))

```

```
(ELSE (PRINT "An actor must have at least
              one parent.")
      (TERPRI)
      (MAPC (LIST "Therefore, " ACTOR_NAME
                  " has been made a child of
                    Actor SYSTEM.")
            'PRINQ)
      (TERPRI)
      (CREATE_ACTOR ACTOR_NAME
                    '(SYSTEM]
    LIF (SETQ CHILDREN
          (PROMPTREAD "CHILDREN: "))
        (THEN [IF (ATOM CHILDREN)
                (THEN (SETQ CHILDREN
                          (LIST CHILDREN)
                            (ADD_CHILDREN ACTOR_NAME CHILDREN)
                            (MAPC CHILDREN
                                '(LAMBDA (CHILD)
                                      (ADD_PARENT CHILD
                                                (LIST ACTOR_NAME])
                                  )
                        )
          ]
        WHILE (SETQ PATTERN
                    (PROMPTREAD "PATTERN: "))
              (DO (SETQ ACTIONS
                      (PROMPTREAD "ACTIONS: "))
                  [IF (ATOM (FIRST ACTIONS))
                      (THEN (SETQ ACTIONS
                              (LIST ACTIONS)
                                (SETQ MYSELF ACTOR_NAME)
                                (ADD_BEHAVIOR
                                  (APPEND (LIST PATTERN)
                                           (LIST ACTIONS)
                                         )
                                )
                                ACTOR_NAME))]
    [LOAD_MODULES
    (LAMBDA (LIST_OF_MODULES)
      (MAPC LIST_OF_MODULES
            '(LAMBDA (PACKAGE)
                  (OPEN PACKAGE 'I PACKAGE)
                  (LOAD PACKAGE)
                  (CLOSE PACKAGE)
                  (MAPC (EVAL (MKATOM
                               (CONCAT PACKAGE 'VARS)))
                       'EVAL)))
            'DONE)]
[DO_NOTHING
(NLAMBDA NIL NIL)]
[FUNCALL
(LAMBDA (FUNCTION . ARGUMENTS)
  (APPLY FUNCTION ARGUMENTS))]
[MISMATCH
(LAMBDA (MSG)
```

(* ERROR MESSAGE GIVEN WHEN A MESSAGE OF WRONG
FORMAT IS SENT TO AN ACTOR)

```

      (PRINT (LIST '+++++ 'WARNING '+++++ MSG 'IS 'AN
                  'UNKNOWN 'MESSAGE 'PATTERN 'FOR 'ACTOR
                  MYSELF)))]

[AFTER_FIRST
 (LAMBDA (LIST)
   ((SUBR . CDR)
    LIST))]

[AFTER_SECOND
 (LAMBDA (LIST)
   ((SUBR . CDDR)
    LIST))]

[AFTER_SECOND_OF_FIRST
 (LAMBDA (LIST)
   ((SUBR . CDDAR)
    LIST))]

[AFTER_THIRD
 (LAMBDA (LIST)
   ((SUBR . CDDDR)
    LIST))]

[AFTER_FOURTH
 (LAMBDA (LIST)
   (AFTER_THIRD
    (AFTER_FIRST LIST)))]

[CONSTRUCT_NEW_LIST
 (LAMBDA (A B)
   (CONS A B))]

[FIRST
 (LAMBDA (ARG)
   ((SUBR . CAR)
    ARG))]

[FIRST_OF_FIRST
 (LAMBDA (LIST)
   ((SUBR . CAAR)
    LIST))]

[FOURTH
 (LAMBDA (ARG)
   (CADDR (CDR ARG)))]

[INTERSECTION
 (LAMBDA (SET_A SET_B)
   (COND ((NULL SET_A)
          NIL)
         ((MEMBER (FIRST SET_A)
                    SET_B)
          (CONSTRUCT_NEW_LIST
           (FIRST SET_A)
           (INTERSECTION
            (AFTER_FIRST SET_A)
            SET_B)))))

```

```

      (T (INTERSECTION
          (AFTER_FIRST SET_A)
          SET_B))
[MAPCAR
 (LAMBDA (ARGUMENTS FIRST_FUNCTION AFTER_FIRST_FUNCTION)
  (APPLY 'NCONC
    (MAPCAR ARGUMENTS FIRST_FUNCTION
      AFTER_FIRST_FUNCTION))))]
[MATCH
 (LAMBDA (PATTERN DATA)
  (* ----- PATTERN MATCHER -----)

  (* "?" MATCHES ANY ONE ATOM)

  (* "+" MATCHES ZERO OR MORE ATOMS)

  (* "?VARIABLE" IS ASSIGN THE VALUE AT THAT POSITION
    IN DATA IF A MATCH OCCURS)

  (* "+VARIABLE" IS ASSIGN A VALUE WHICH IS A LIST OF
    ATOMS THAT OCCUPY THAT POSITION IN MATCHING DATA)

  (COND (OR (AND (NULL PATTERN)
    (NULL DATA)))
    T)
  (LOR (NULL PATTERN)
    (AND (NULL DATA)
      (NOT (EQUAL PATTERN '(+ ]
        NIL)
      (LAND (NOT (ATOM (FIRST PATTERN)))
        (EQ (FIRST_OF_FIRST PATTERN)
          'RESTRICT)
        (EQ (FIRST (AFTER_FIRST
          (FIRST PATTERN)))
            '?))
      (APPLY 'AND
        (MAPCAR
          (AFTER_SECOND_OF_FIRST PATTERN)
          '(LAMBDA (PRED)
            (FUNCALL PRED
              (FIRST DATA)
              (MATCH (AFTER_FIRST PATTERN)
                (AFTER_FIRST DATA))))
        ((AND (NOT (ATOM (FIRST PATTERN)))
          (EQ (FIRST_OF_FIRST PATTERN)
            'RESTRICT)
          (EQ (FIRST (UNPACK (FIRST
            (AFTER_FIRST
              (FIRST PATTERN)
              '?))

```

```

      APPLY 'AND
      (MAPCAR
        (AFTER_SECOND_OF_FIRST PATTERN)
        '(LAMBDA (PRED)
          (FUNCALL PRED
            (FIRST DATA)
            (MATCH (AFTER_FIRST PATTERN)
              (AFTER_FIRST DATA))))
        (SET [PACK (AFTER_FIRST
          (UNPACK (FIRST (AFTER_FIRST
            (FIRST PATTERN]
            (FIRST DATA)))
          T)
      ((OR (EQ (FIRST PATTERN)
        '?')
        (EQ (FIRST PATTERN)
          (FIRST DATA)))
        (MATCH (AFTER_FIRST PATTERN)
          (AFTER_FIRST DATA)))
      ((AND (ATOM (FIRST PATTERN))
        (EQ (FIRST (UNPACK (FIRST PATTERN)))
          '?')
        (MATCH (AFTER_FIRST PATTERN)
          (AFTER_FIRST DATA)))
      (SET [PACK (AFTER_FIRST
        (UNPACK (FIRST PATTERN]
        (FIRST DATA)))
      T)
      ((EQ (FIRST PATTERN)
        '+')
      (COND ((MATCH (AFTER_FIRST PATTERN)
        DATA))
        ((MATCH PATTERN
          (AFTER_FIRST DATA]
      ((AND (ATOM (FIRST PATTERN))
        (EQ (FIRST (UNPACK (FIRST PATTERN)))
          '+'))
      (COND ((MATCH (AFTER_FIRST PATTERN)
        (AFTER_FIRST DATA))
        (SET [PACK (AFTER_FIRST
          (UNPACK (FIRST PATTERN]
          (LIST (FIRST DATA)))
        T)
      ((MATCH PATTERN
        (AFTER_FIRST DATA))
      [SET [PACK (AFTER_FIRST
        (UNPACK (FIRST PATTERN]
        (CONS (FIRST DATA)
          (EVAL
            (PACK
              (AFTER_FIRST
                (UNPACK
                  (FIRST PATTERN]
          T)

```

```

((MATCH (AFTER_FIRST PATTERN)
  DATA)
 (SET [PACK (AFTER_FIRST
  (UNPACK (FIRST PATTERN)
    NIL)
  T]))

[NOT
 (LAMBDA (X)
  ((SUBR . NULL)
   X))]

[SECOND
 (LAMBDA (ARG)
  ((SUBR . CADDR)
   ARG))]

[RANDOM_NUMBER_1
 [NLAMBDA NIL
  (SECOND (REVERSE (UNPACK (ABS (CLOCK))

[THIRD
 (LAMBDA (ARG)
  ((SUBR . CADDR)
   ARG))]

[RANDOM_NUMBER_2
 [NLAMBDA NIL
  (THIRD (UNPACK (ABS (CLOCK))

[REMAINDER*
 (LAMBDA (DIVIDEND DIVISOR)
  (IF (EQUAL DIVISOR 0)
    (THEN 0)
    (ELSE (DIFFERENCE DIVIDEND
      (TIMES DIVISOR
        (QUOTIENT DIVIDEND
          DIVISOR))

[CALL_AFTER
 (LAMBDA (KEY SOURCE)
  (AFTER_FIRST
   (MEMBER KEY SOURCE))))]

[REPLACE
 [NLAMBDA (ITEM1 BY ITEM2 IN SOURCE)
  [IF (NEW (FIRST ITEM1)
    'NOBIND)
    (THEN (SETQ ITEM1
      (EVAL ITEM1))
  [IF (NEW (FIRST ITEM2)
    'NOBIND)
    (THEN (SETQ ITEM2
      (EVAL ITEM2))
  [IF (NEW (FIRST SOURCE)
    'NOBIND)
    (THEN (SETQ SOURCE
      (EVAL SOURCE))

```



```

(PROG (OUTCOME)
  [WHILE (NOT ((NULL SOURCE)
    (RETURN OUTCOME)))
    (DO [COND
      ((AND
        (ATOM (FIRST SOURCE))
        (EQ (FIRST SOURCE) ITEM1)))
        [SETQ
          OUTCOME
          (APPEND
            OUTCOME
            (COND
              ((ATOM ITEM2)
               (CONSTRUCT_NEW_LIST ITEM2
                (AFTER_FIRST SOURCE)))
              (T (APPEND ITEM2
                (AFTER_FIRST SOURCE)]
            (SETQ SOURCE NIL))
          [(LISTP (FIRST SOURCE))
            (SETQ
              OUTCOME
              (APPEND
                OUTCOME
                (LIST
                  (REPLACE ITEM1 ITEM2
                    (FIRST SOURCE))
                (T (SETQ
                  OUTCOME
                  (APPEND
                    OUTCOME
                    (LIST (FIRST SOURCE))
                  (SETQ SOURCE
                    (AFTER_FIRST SOURCE))
                (RETURN OUTCOME)))]
[UNION
[LAMBDA (SET_A SET_B)
  (IF (NULL SET_A)
    (THEN SET_B)
    (ELSE (IF (MEMBER (FIRST SET_A) SET_B)
      (THEN (UNION (AFTER_FIRST SET_A)
        SET_B))
      (ELSE (CONSTRUCT_NEW_LIST
        (FIRST SET_A)
        (UNION (AFTER_FIRST SET_A)
          SET_B)]
[OASYS
(NLAMBDA (INPUT_FILE &LOCAL NAME)
  (PRIN1 " Welcome to OASYS")
  (TERPRI)
  (TERPRI)
  (TERPRI)
  (IF (EQ (FIRST SIMULATIONFNS) 'NOBIND)
    (THEN (LOAD_MODULES
      '(SIMULATION ACTOR ACTION KNOWLEDGE))

```

```

        (TERPRI)
        (TERPRI)
        (LOAD_ACTOR 'SYSTEM)
        (TERPRI)
        (TERPRI)))
(IF (NULL INPUT_FILE)
  (THEN
    (PRIN1 "Any applications to be loaded?")
    (TERPRI)
    LIF (EQ 'YES
      (PROMPTREAD " "))
      (THEN (TERPRI)
        (TERPRI)
        (WHILE (SETQ
          NAME
          (PROMPTREAD
            "Application Name: "))
          (DO (LOAD_APPLICATION NAME)
            (TERPRI)
            (TERPRI))
          (TERPRI)
          (TERPRI))
        (PRIN1 (EVAL (PROMPTREAD '<->)))
        (TERPRI)
        (GO LOOP)))
    (PRIN1 "Do you wish to save some
      applications?")
    (TERPRI)
    LIF (EQ
      'YES
      (PROMPTREAD
        " "))
      (THEN (TERPRI)
        (TERPRI)
        (PROG (APPLICATION)
          (WHILE (SETQ
            APPLICATION
            (PROMPTREAD
              "Application Name: "))
            (DO (SAVE_APPLICATION
              APPLICATION)
              (TERPRI)
              (TERPRI))
            (PRIN1 (OASYS_TERMINATED - - YOU ARE NOW IN CP6 LISP
              - - THANK YOU))
            (ELSE (INFILE INPUT_FILE)
              [RECORD_VALUE.
              [LAMBDA (RECORD_NAME FIELD_NAME &LOCAL_FIELDS RECORD)
              [IF (NULL RECORD)
              (THEN (SETQ RECORD
                (EVAL RECORD_NAME)

```

```

[IF (NULL FIELDS)
  (THEN (SETQ FIELDS
              (GETP (GETP RECORD_NAME 'TYPE)
                    'FIELDS))
  (IF (EQUAL (FIRST FIELDS)
              FIELD_NAME)
    (THEN (FIRST RECORD))
    (ELSE (IF (NOT (NULL (AFTER_FIRST FIELDS)))
              (THEN (RECORD_VALUE
                    RECORD_NAME FIELD_NAME &LOCAL
                    (AFTER_FIRST FIELDS)
                    (AFTER_FIRST RECORD)))
              (ELSE (MAPC (LIST RECORD_NAME
                                " DOES NOT HAVE A
                                FIELD NAMED "
                                FIELD_NAME)
                          'PRINT)
                (TERPRI))
    (SET_RECORD
     (LAMBDA (RECORD_NAME . FIELD-VALUE_PAIRS)
       (MAPC FIELD-VALUE_PAIRS
             '(LAMBDA (PAIR)
               (IF (MEMBER
                   (FIRST PAIR)
                   (GETP
                    (GETP RECORD_NAME 'TYPE)
                    'FIELDS))
                 (THEN
                  (PROG
                   ((FIELD_NAMES
                    (GETP
                     (GETP RECORD_NAME 'TYPE)
                     'FIELDS))
                   (RECORD (EVAL RECORD_NAME)))
                   LOOP (IF
                       (NULL FIELD_NAMES)
                       (THEN (RETURN))
                       (ELSE
                        (IF
                         (EQ
                          (FIRST PAIR)
                          (FIRST
                           FIELD_NAMES))
                         (THEN (RFLACA
                              RECORD
                              (SECOND
                               PAIR)
                              (SETQ
                               FIELD_NAMES
                               (AFTER_FIRST
                                FIELD_NAMES))
                              (SETQ
                               RECORD
                               (AFTER_FIRST

```

```

                                RECORD))
                                (GO LOOP]

                                (ELSE
                                  (MAPC
                                    (LIST
                                      (FIRST PAIR)
                                      " IS NOT A VALID FIELD
                                      FOR RECORDS OF TYPE "
                                      (GETP RECORD_NAME 'TYPE))
                                      'PRIN0)
                                    (TERPRI)])

[MAKE_RECORD
[LAMBDA (RECORD_NAME TYPE_NAME)
  (PUT RECORD_NAME 'TYPE TYPE_NAME)
  (IF (NOT (GETP TYPE_NAME 'FIELDS))
    (THEN
      (LIST
        'TYPE TYPE_NAME " HAS NOT BEEN DEFINED"))
    (ELSE (SET RECORD_NAME NIL)
      (MAPC (GETP TYPE_NAME 'FIELDS)
        '(LAMBDA (FIELD_NAME)
          (SET
            RECORD_NAME
            (APPEND (EVAL RECORD_NAME)
              (LIST NIL)))

[DEFINE_RECORD
[INLAMBDA (TYPE_NAME . FIELD_NAMES)
  (IF (GETP TYPE_NAME 'FIELDS)
    (THEN
      (LIST
        "TYPE " TYPE_NAME
        " HAS ALREADY BEEN DEFINED."))
    (ELSE (PUT TYPE_NAME 'FIELDS FIELD_NAMES)))
]
(SETQ LISP_SUPPORTFNS
  (IF WHILE REPEAT DEFINE_ACTOR LOAD_MODULES DO NOTHING
    FUNCALL MISMATCH AFTER_FIRST AFTER_SECOND
    AFTER_SECOND_OF_FIRST AFTER_THIRD AFTER_FOURTH
    CONSTRUCT_NEW_LIST FIRST FIRST_OF_FIRST FOURTH
    FIRST_OF_FIRST FOURTH INTERSECTION MAPCAN MATCH NOT
    SECOND RANDOM_NUMBER_1 THIRD RANDOM_NUMBER_2
    REMAINDER ALL_AFTER REPLACE UNION OASYS RECORD_VALUE
    SET_RECORD MAKE_RECORD DEFINE_RECORD))
(SETQ LISP_SUPPORTCOMS PACKAGE_BODY_LISP_SUPPORT)
(SETQ LISP_SUPPORTGENNR 31)
STOP

```

A P P E N D I X B

P A C K A G E A C T I O N

```
*****
*                                     *
*                                PACKAGE ACTION                                *
*                                     *
*****
```

Type PATTERN is List;

Type CODE is List of S-Expr;

Type BEHAVIOR is

 Record

 SELECTOR: Pattern

 ACTIONS: List of Code

 End Record;

Procedure ADD_BEHAVIOR (BEHAVIOR: Behavior);

Procedure FIND_BEHAVIOR (SELECTOR: Pattern);

Procedure ADD_CODE (BEHAVIOR: Behavior);

Procedure DELETE_BEHAVIOR (SELECTOR: Pattern);

Procedure DELETE_CODE (BEHAVIOR: Behavior);

Function FIND_CODE_FOR (SELECTOR: Pattern)
 Return List of Code;

```

*****
*
*               PACKAGE BODY ACTION
*
*****

```

```
With SIMULATION, LISP_SUPPORT;
```

```
Use ACTOR, LISP_SUPPORT;
```

```
(PRIN1 'PACKAGE_BODY_ACTION 0 T)
```

```
(PRINT '(VERSION 21))
```

```
[DEFINEQ
```

```
[ADD_BEHAVIOR
```

```
[LAMBDA (BEHAVIOR)
```

```
  (PUT_BEHAVIORS MYSELF
```

```
    (APPEND (GET_BEHAVIORS MYSELF)
```

```
      (LIST BEHAVIOR))
```

```
[FIND_BEHAVIOR
```

```
[LAMBDA (SELECTOR)
```

```
  (PROG ((ACTIONS (GET_BEHAVIORS MYSELF)))
```

```
    (MAPC (AFTER_FIRST ACTIONS)
```

```
      '(LAMBDA (BEHAVIOR)
```

```
        (IF (MATCH (FIRST BEHAVIOR)
```

```
            SELECTOR)
```

```
            (THEN (RETURN BEHAVIOR))
```

```
[ADD_CODE
```

```
[LAMBDA (BEHAVIOR)
```

```
  (MAPC (SECOND BEHAVIOR)
```

```
    '(LAMBDA (CODE)
```

```
      (RPLACD (LAST (FIND_BEHAVIOR
```

```
        (FIRST BEHAVIOR)))
```

```
      (LIST CODE))
```

```
[DELETE_BEHAVIOR
```

```
[LAMBDA (SELECTOR)
```

```
  (PUT_BEHAVIORS MYSELF
```

```
    (REMOVE (FIND_BEHAVIOR SELECTOR)
```

```
      (GET_BEHAVIORS MYSELF))
```

```
[DELETE_CODE
```

```
[LAMBDA (BEHAVIOR)
```

```
  (PUT_BEHAVIORS MYSELF
```

```
    (SUBST (PROG ((NEW_ACTION
```

```
      (FIND_BEHAVIOR
```

```
        (FIRST BEHAVIOR)))
```

```
      (CODE (SECOND BEHAVIOR)))
```

```
    [WHILE (NOT (NULL CODE))
```

```
      (DO (SETQ NEW_ACTION
```

```
          (REMOVE (FIRST CODE)
```

```
            NEW_ACTION))
```

```
        (SETQ CODE
```

```
          (AFTER_FIRST CODE)
```

```
        (RETURN NEW_ACTION))
```

```

(FIND_BEHAVIOR
 (FIRST_BEHAVIOR))
(GET_BEHAVIORS MYSELF]]

[FIND_CODE_FOR
[LAMBDA (SELECTOR)
  (COND (SECOND
    (FIND_BEHAVIOR SELECTOR)))
  (T (PROG ((PARENT (GET_PARENT MYSELF)))
    LOOP (IF (NULL PARENT)
      [THEN (IF (ACTOR? MYSELF)
        (THEN
          (MISMATCH_SELECTOR)))
        (RETURN '((DO_NOTHING))
        (ELSE [MAPC
          (AFTER_FIRST
            (GET_BEHAVIORS
              (FIRST_PARENT)))
            '(LAMBDA (BEHAVIOR)
              (IF (MATCH
                FIRST_BEHAVIOR)
                SELECTOR)
              (THEN
                (RETURN
                  (SECOND
                    BEHAVIOR))
                [SETQ PARENT
                  (UNION
                    (AFTER_FIRST_PARENT)
                    (GET_PARENT
                      (FIRST_PARENT)
                    (GO_LOOP]]]]

(SETQ ACTIONFNS
  (ADD_BEHAVIOR FIND_BEHAVIOR ADD_CODE DELETE_BEHAVIOR
    DELETE_CODE FIND_CODE_FOR))
(SETQ ACTIONVARS
  ((DO_NOTHING)))
(SETQ ACTIONCOMS PACKAGE_BODY_ACTION)
(SETQ ACTIONGENNK 21)
STOP

```


A P P E N D I X C

P A C K A G E K N O W L E D G E

```

*****
*                                     *
*                                PACKAGE KNOWLEDGE                                *
*                                     *
*****

```

```

Private Type FACET_NAME is (VALUE, IF-ADDED, IF-REMOVED,
                           IF-CHANGED, IF-NEEDED,
                           PRIVATE-IF-ADDED,
                           PRIVATE-IF-REMOVED,
                           PRIVATE-IF-NEEDED,
                           PRIVATE-IF-CHANGED);

```

```

Private Type VALUE is List of S-Expr;

```

```

Private Type FACET is

```

```

    Record

```

```

        FACET_NAME: Facet_Name

```

```

        VALUES: Array [*] of Value

```

```

    End Record;

```

```

Type SLOT is

```

```

    record

```

```

        SLOT_NAME: Atom

```

```

        FACETS: List of Facet

```

```

    End Record;

```

```

Procedure LEARN (NEW: Array [1..3] of Atom);

```

```

Procedure FORGET (OLD: Array [1..3] of Atom);

```

```

Procedure CHANGE (BAD: Array [1..4] of Atom);

```

```

Function RECALL (FACT: Array [1..2] of Atom) Return List;

```

```

-- The second argument, when used, denotes a demon or
-- a particular field of a record to be retrieved. --
-- Note that RECALL is local to an actor. --

```

```

Function IS_THERE (QUERY: Array [1..3] of Atom) Return List;

```

-- All the above procedures and functions can be called with one two or three arguments. --

-- For example:

```
LEARN (SLOT_NAME)
LEARN (SLOT_NAME VALUE)
LEARN (SLOT_NAME FACET_NAME VALUE)
```

The procedure itself turns theses parameters into a list of parameters. --

Function REMEMBER (ATTRIBUTE: Atom) Return List;

Procedure DEFINE_DEMON (DEMON_NAME: Atom; DEFINITION:
List of S-Expr);

Function LIST_ATTRIBUTES (ACTOR_NAME: Atom) Return List;

```
(ELSE NIL)]
```

```

[PUT
(LAMBDA (SLOT FACET VALUE)
  (IF (MEMBER VALUE
        (FGET MYSELF SLOT FACET))
      (THEN NIL)
      (ELSE [FASSOC VALUE
                    (FASSOC FACET
                     (FASSOC SLOT
                      (GET_KNOWLEDGE
                       MYSELF)
                     VALUE))]))]

[REMOVE
(LAMBDA (SLOT FACET VALUE)
  [IF (NULL VALUE)
    (THEN (SETQ VALUE (FGET MYSELF SLOT FACET)))
    (ELSE (SETQ VALUE (LIST VALUE)
  (PROG ((SLOTS (GET_KNOWLEDGE MYSELF))
        FACETS VALUES TARGET)
    (SETQ FACETS
      (SASSOC SLOT
       (AFTER_FIRST SLOTS)))

    (SETQ VALUES
      (SASSOC FACET
       (AFTER_FIRST FACETS)))

    (RETURN
      (MAPCAR
        VALUE
        (LAMBDA (E)
          (SETQ TARGET
            (SASSOC E
              (AFTER_FIRST VALUES)))

          LPUT_KNOWLEDGE
            MYSELF
            (IF (AFTER_FIRST
              (REMOVE TARGET VALUES))
              (THEN (SUBST
                (REMOVE TARGET VALUES)
                VALUES SLOTS))
              (ELSE (IF (AFTER_FIRST
                (REMOVE
                  VALUES FACETS))
                (THEN (SUBST
                  (REMOVE
                    VALUES FACETS)
                    FACETS SLOTS))
                (ELSE (SUBST
                  (REMOVE
                    FACETS SLOTS)
                    SLOTS SLOTS)

          E))

  (REMOVE+
(LAMBDA (SLOT FACET VALUE &LOCAL SAVED_VALUE)
  (IF (EQ FACET 'VALUE)
    (THEN

```

```

(IF (SETQ SAVED_VALUE
  (REMOVE SLOT FACET VALUE))
  (THEN
    [IF
      (FCHECK SLOT 'PRIVATE-IF-REMOVED)
      (THEN
        (MAPC
          (FGET MYSELF SLOT
            'PRIVATE-IF-REMOVED)
          'FUNCALL))
      (ELSE
        (IF
          (FCHECK SLOT 'IF-REMOVED)
          (THEN
            (MAPC
              (FGET MYSELF SLOT
                'IF-REMOVED)
              'FUNCALL))
          (ELSE
            (PROG ((PARENT
              (GET_PARENT MYSELF))
              (SAVEDME MYSELF))
              [WHILE
                (NOT (NULL PARENT))
                (DO (SETQ
                  MYSELF
                    (FIRST PARENT))
                  (IF (FCHECK SLOT
                    'IF-REMOVED)
                    (THEN
                      (MAPC
                        (FGET
                          MYSELF
                          SLOT
                            'IF-REMOVED)
                        ((LAMBDA
                          (DEMON)
                          (SETQ
                            MYSELF
                              SAVEDME)
                          (FUNCALL
                            DEMON))))
                    (RETURN)))
                  (ELSE
                    (SETQ
                      PARENT
                        (UNION
                          (AFTER_FIRST
                            PARENT)
                          (GET_PARENT
                            MYSELF))))
                    (SETQ
                      PARENT
                        (REMOVE

```

```

'NO_PARENT
PARENT]
      (SETQ MYSELF SAVEDME)
      (RETURN)
    SAVED_VALUE)))
  (ELSE (REMOVE SLOT FACET VALUE))
[FPUT+
[LAMBDA (SLOT FACET VALUE)
  (IF (EQ FACET 'VALUE)
    (THEN
      (IF (FPUT SLOT FACET VALUE)
        (THEN
          LIF (FCHECK SLOT 'PRIVATE-IF-ADDED)
            (THEN (MAPC (FGET MYSELF
                          SLOT
                          'PRIVATE-IF-ADDED)
                        'FUNCALL))
          (ELSE (IF (FCHECK SLOT 'IF-ADDED)
                    (THEN (MAPC (FGET MYSELF
                                      SLOT
                                      'IF-ADDED)
                                'FUNCALL))
                    (ELSE (PROG
                          ((PARENT
                           (GET_PARENT MYSELF))
                           (SAVEDME MYSELF))
                           WHILE
                           (NOT (NULL PARENT))
                           (DO (SETQ
                                MYSELF
                                (FIRST PARENT))
                                (IF
                                 (FCHECK
                                  SLOT
                                  'IF-ADDED)
                                 (THEN
                                  (MAPC
                                   (FGET
                                    MYSELF
                                    SLOT
                                    'IF-ADDED)
                                   (LAMBDA
                                    (DEMON)
                                    (SETQ
                                     MYSELF
                                     SAVEDME)
                                     (FUNCALL
                                      DEMON))))
                                (RETURN)))

```

```

(ELSE
  (SETQ
    PARENT
    (UNION
      (AFTER_FIRST
        PARENT)
      (GET_PARENT
        MYSELF)))
  (SETQ
    PARENT
    (REMOVE
      'NO_PARENT
      PARENT])
  (SETQ MYSELF SAVEDME)
  VALUE)))
(ELSE (PUT SLOT FACET VALUE))
[FCHECK+
[LAMBDA (SLOT FACET OLD_VALUE NEW_VALUE)
  (IF (EQ FACET 'VALUE)
    (THEN
      (REMOVE SLOT FACET OLD_VALUE)
      [IF (FCHECK SLOT 'PRIVATE-IF-CHANGED),
        (THEN
          (MAPC (FGET MYSELF SLOT 'PRIVATE-IF-CHANGED)
            'FUNCALL))
        (ELSE
          (IF (FCHECK SLOT 'IF-CHANGED)
            (THEN
              (MAPC (FGET MYSELF SLOT 'IF-CHANGED)
                'FUNCALL))
            (ELSE
              (PROG ((PARENT
                (GET_PARENT MYSELF))
                (SAVEDME MYSELF))
                [WHILE
                  (NOT (NULL PARENT))
                  (DO (SETQ MYSELF
                    (FIRST PARENT))
                    (IF (FCHECK SLOT 'IF-CHANGED)
                      (THEN
                        (MAPC
                          (FGET MYSELF
                            SLOT
                            'IF-CHANGED)
                          '(LAMBDA
                            (DEMON)
                            (SETQ MYSELF
                              SAVEDME)
                              (FUNCALL DEMON)))
                        (RETURN))
                    (ELSE
                      (SETQ
                        PARENT
                        (UNION

```



```

(AFTER_FIRST
 PARENT)
(GET_PARENT
 MYSELF)))
(SETQ
 PARENT
 (REMOVE
 'NO_PARENT
 PARENT])
(SETQ MYSELF SAVEDME]
(FPUT SLOT FACET NEW_VALUE))
(ELSE (FREMOVE SLOT FACET OLD_VALUE)
(FPUT SLOT FACET NEW_VALUE)])
[LEARN
[LAMBDA NEW
  (SELECTQ (LENGTH NEW)
    (1 (SETQ NEW
        (LIST (FIRST NEW)
              'VALUE NIL)))
    (2 (SETQ NEW
        (LIST (FIRST NEW)
              'VALUE
              (SECOND NEW))
        (3 (DO_NOthing))
        (SETQ NEW NIL))
    (IF (NOT (NULL NEW))
      (THEN
        (IF (OR (NOT (GETP (FIRST NEW)
                          'TYPE))
                (AND (GETP (FIRST NEW)
                          'TYPE)
                     (EQUAL (LENGTH (THIRD NEW))
                           (LENGTH
                             (GETP
                              (GETP (FIRST NEW)
                                    'TYPE)
                                    'FIELDS))
                           (THEN
                            (FPUT+ (FIRST NEW)
                                   (SECOND NEW)
                                   (THIRD NEW)))
                           (ELSE
                            (PRINT
                              (LIST "THE VALUE OF ATTRIBUTE "
                                    (FIRST NEW)
                                    " MUST BE A LIST OF "
                                    (LENGTH
                                      (GETP

```

```

                                (GETP (FIRST NEW)
                                  'TYPE)
                                'FIELDS))
                                " ELEMENTS."]]

[FORGET
[LAMBDA OLD
  (SELECTQ (LENGTH OLD)
    (1 (SETQ OLD
        (LIST (FIRST OLD)
              'VALUE NIL)))
    [2 (SETQ OLD
        (LIST (FIRST OLD)
              'VALUE
              (SECOND OLD)
              (3 (DO_NOTHING))
              (SETQ OLD NIL))
        (IF (NOT (NULL OLD))
            (THEN (FREMOVE+ (FIRST OLD)
                            (SECOND OLD)
                            (THIRD OLD))

[CHANGE
[LAMBDA BAD
  (SELECTQ (LENGTH BAD)
    [2 (SETQ BAD
        (LIST (FIRST BAD)
              'VALUE NIL
              (SECOND BAD)
              [3 (SELECTQ (SECOND BAD)
                          [(IF-ADDED IF-REMOVED IF-NEEDED
                            IF-CHANGED PRIVATE-IF-ADDED
                            PRIVATE-IF-REMOVED
                            PRIVATE-IF-NEEDED
                            PRIVATE-IF-CHANGED)
                          (SETQ BAD
                            (LIST (FIRST BAD)
                                  (SECOND BAD)
                                  NIL
                                  (THIRD BAD)
                                  (SETQ BAD
                                    (LIST (FIRST BAD)
                                          'VALUE
                                          (SECOND BAD)
                                          (THIRD BAD)
                                          (4 (DO_NOTHING))
                                          (SETQ BAD NIL))
                          (IF (NOT (NULL BAD))
                              (THEN (FCHANGE+ (FIRST BAD)
                                                  (SECOND BAD)
                                                  (THIRD BAD)
                                                  (FOURTH BAD))

[RECALL
[LAMBDA FACT
  (SELECTQ (LENGTH FACT)
    (1 (SETQ FACT

```

```

      (LIST MYSELF
        (FIRST FACT)
        'VALUE)))
    (2 (SETQ FACT
      (CONSTRUCT_NEW_LIST MYSELF FACT)))
    (SETQ FACT NIL))
  (IF (NOT (NULL FACT))
    (THEN (IF (OR (NOT (GETP (SECOND FACT)
      'TYPE))
      (INTERSECTION
        (LIST (THIRD FACT))
        '(VALUE IF-NEEDED IF-ADDED
          IF-REMOVED IF-CHANGED
          PRIVATE-IF-NEEDED
          PRIVATE-IF-ADDED
          PRIVATE-IF-REMOVED
          PRIVATE-IF-CHANGED))))
      (THEN (FGET (FIRST FACT)
        (SECOND FACT)
        (THIRD FACT)))
      (ELSE (SET (SECOND FACT)
        (FGET (FIRST FACT)
          (SECOND FACT)
          'VALUE))
        (RECORD_VALUE
          (SECOND FACT)
          (THIRD FACT)))
    ]IS_THERE
  (LAMBDA QUERY
    (FCHECK (FIRST QUERY)
      (SECOND QUERY)
      (THIRD QUERY))))]
[DEFINE_DEMON
  (LAMBDA (DEMON_NAME DEFINITION)
    (CURFILE USER)
    (PUTD DEMON_NAME
      (CONS 'NLAMBDA
        (CONS NIL DEFINITION))))
    [IF (EQ [FIRST (EVAL '(MKATOM (CONCAT MYSELF 'FNS)
      'NOBIND)
      (THEN (SET (MKATOM (CONCAT MYSELF 'FNS)
        USERFNS))
      (ELSE (SET (MKATOM (CONCAT MYSELF 'FNS)
        (APPEND (EVAL
          (MKATOM
            (CONCAT MYSELF 'FNS)))
            USERFNS]
      (SETQ USERFNS NIL)
      (CURFILE))]]

```

```

[REMEMBER
[LAMBDA (ATTRIBUTE)
  (COND ((RECALL ATTRIBUTE))
        ((MAPCAR (RECALL ATTRIBUTE 'PRIVATE-IF-NEEDED)
                  'FUNCALL))
        ((MAPCAR (RECALL ATTRIBUTE 'IF-NEEDED)
                  'FUNCALL))
        (T (PROG (RESULT)
                  (PARENT (GET_PARENT MYSELF))
                  (SAVEDME MYSELF))
           [WHILE
            (NOT (NULL PARENT))
            (DO (SETQ MYSELF
                     (FIRST PARENT))
                (IF [OR (SETQ RESULT
                           (RECALL ATTRIBUTE))
                    (SETQ
                     RESULT
                     (MAPCAR
                      (RECALL ATTRIBUTE
                       'IF-NEEDED)
                      '(LAMBDA
                       (DEMON)
                       (SETQ MYSELF
                            SAVEDME)
                       (FUNCALL DEMON]
                     (THEN (RETURN))
                     (ELSE (SETQ
                              PARENT
                              (UNION
                               (AFTER_FIRST
                                PARENT)
                               (GET_PARENT
                                MYSELF)))
                            (SETQ PARENT
                                (REMOVE
                                 'NO_PARENT
                                 PARENT]
                     (SETQ MYSELF SAVEDME)
                     (RETURN RESULT]))

[LIST_ATTRIBUTES
(LAMBDA (ACTOR_NAME)
  (MAPCAR (AFTER_FIRST
           (GET_KNOWLEDGE ACTOR_NAME))
          'FIRST)))]
(SETQ KNOWLEDGEFNS
  (FASSOC FGET FCHECK FPUT FREMOVE FREMOVE+ FPUT+
    FCHANGE+ LEARN FORGET CHANGE RECALL IS_THERE
    DEFINE_DEMON REMEMBER LIST_ATTRIBUTES))
(SETQ KNOWLEDGEVARS
  ((DO_NOHING)))
(SETQ KNOWLEDGECOMS PACKAGE_BODY_KNOWLEDGE)
(SETQ KNOWLEDGEGENNR 26)
STOP

```

A P P E N D I X D

P A C K A G E A C T O R

```

*****
*
*                               PACKAGE ACTOR
*
*****

With KNOWLEDGE, ACTION

Type DO is List of Behavior;

Type KNOW is
    Record
        ACTOR_NAME: Atom
        ASPECT: List of Slot
    End Record;

Function ACTOR? (ACTOR_NAME: Atom) Return Boolean;

Procedure LOAD_ACTOR (ACTOR_NAME: Atom);

Procedure SAVE_ACTOR (ACTOR_NAME: Atom);

Function GET_KNOWLEDGE (ACTOR_NAME: Atom) Return Know;

Function GET_BEHAVIORS (ACTOR_NAME: Atom) Return DO;

Function GET_PARENT (ACTOR_NAME: Atom) Return List;

Function GET_CHILDREN (ACTOR_NAME: Atom) Return List;

Procedure PUT_KNOWLEDGE (ACTOR_NAME: Atom;
                        NEW: List of Slot);

Procedure PUT_BEHAVIORS (ACTOR_NAME: Atom; UNDO: DO);

Procedure PUT_CHILDREN (ACTOR_NAME: Atom; NEW_LIST: List);

Procedure ADD_PARENT (ACTOR_NAME, NEW_PARENT: List);

Procedure ADD_CHILDREN (ACTOR_NAME: Atom; LUCKY_CHILD: List);

Procedure PUT_PARENT (ACTOR_NAME: Atom; NEW_LIST: List);

Procedure DELETE_PARENT (ACTOR_NAME, OLD_PARENT: ATOM);

Procedure DELETE_CHILDREN (ACTOR_NAME: Atom;
                          POOR_CHILD: List);

Procedure CREATE_ACTOR (ACTOR_NAME: Atom; FATHER: List);

```

Procedure KILL_ACTOR (ACTOR_NAME: Atom);

Procedure DESTROY_ALL (ACTOR_NAME: Atom);

Procedure DESTROY_AND_PRESERVE (ACTOR_NAME: Atom);

Procedure LOAD_APPLICATION (NAME: Atom);

Procedure SAVE_APPLICATION (NAME: Atom);

```

*****
*
*          PACKAGE BODY ACTOR
*
*****

```

With KNOWLEDGE, ACTION, SIMULATION;

Use LISP_SUPPORT;

Type ACTOR is

Record

KNOWLEDGE: Know

BEHAVIORS: Do

PARENTS: List

CHILDREN: List

End Record;

(PRIN0 'PACKAGE_BODY_ACTOR 0 T)

(PRINT '(VERSION 32))

[DEFINEQ

[ACTOR_ERROR

(LAMBDA (ACTOR_NAME)

(LIST '***** 'ERROR '***** ACTOR_NAME 'DOES 'NOT 'EXIST)))]

[ACTOR?

(LAMBDA (ACTOR_NAME)

(* ANSWERS THE QUESTION: IS ACTOR_NAME AN ACTOR?))

(IF (GETP ACTOR_NAME 'ACTOR)

(THEN T)

(ELSE NIL))))]

[LOAD_ACTOR

(LAMBDA (ACTOR_NAME)

(* THIS PROCEDURE LOADS AN ACTOR FROM ITS STORAGE
FILE ONTO THE PROPERTY LIST "ACTOR" OF
ATOM "ACTOR_NAME")

(IF (OPEN ACTOR_NAME 'I ACTOR_NAME)

(THEN (LOAD ACTOR_NAME)

(CLOSE ACTOR_NAME)


```

[PUT ACTOR_NAME 'ACTOR
  (EVAL (MKATOM
    (CONCAT ACTOR_NAME 'VARS)
    ACTOR_NAME)))]

```

```

[LOAD_TREE
  [LAMBDA (ROOT)

```

* THIS PROCEDURE LOADS AN ACTOR TREE STARTING AT THE ORIGINAL VALUE OF ROOT)

```

  (IF (NOT (ACTOR? ROOT))
    (THEN (LOAD_ACTOR ROOT)
      (MAPC (GET_CHILDREN ROOT)
        '(LAMBDA (CHILD)
          (IF (NEQ CHILD 'NO_CHILDREN)
            (THEN (LOAD_TREE CHILD))

```

```

[SAVE_ACTOR
  [LAMBDA (ACTOR_NAME)
    (IF (OPEN ACTOR_NAME 'O ACTOR_NAME)
      (THEN (SET (MKATOM (CONCAT ACTOR_NAME 'VARS))
        (GETP ACTOR_NAME 'ACTOR))
      (MAKEFILE ACTOR_NAME 2 100)
      (CLOSE ACTOR_NAME))]

```

```

[SAVE_TREE
  [LAMBDA (ROOT)
    (SAVE_ACTOR ROOT)
    (MAPC (GET_CHILDREN ROOT)
      '(LAMBDA (CHILD)
        (IF (NEQ CHILD 'NO_CHILDREN)
          (THEN (SAVE_TREE CHILD))

```

```

[GET_KNOWLEDGE
  [LAMBDA (ACTOR_NAME)
    (IF (ACTOR? ACTOR_NAME)
      (THEN (FIRST (GETP ACTOR_NAME 'ACTOR))

```

```

[GET_BEHAVIORS
  [LAMBDA (ACTOR_NAME)
    (IF (ACTOR? ACTOR_NAME)
      (THEN (SECOND (GETP ACTOR_NAME 'ACTOR))

```

```

[GET_PARENT
  [LAMBDA (ACTOR_NAME)
    (IF (ACTOR? ACTOR_NAME)
      (THEN (THIRD (GETP ACTOR_NAME 'ACTOR))

```

```

[GET_CHILDREN
  [LAMBDA (ACTOR_NAME)
    (IF (ACTOR? ACTOR_NAME)
      (THEN (FOURTH (GETP ACTOR_NAME 'ACTOR))

```

```

[PUT_KNOWLEDGE
  [LAMBDA (ACTOR_NAME NEW)
    (IF (ACTOR? ACTOR_NAME)
      (THEN (PUT ACTOR_NAME 'ACTOR
        (SUBST NEW
          (GET_KNOWLEDGE ACTOR_NAME)
          (GETP ACTOR_NAME 'ACTOR))

```

```

        (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME)))
[PUT_BEHAVIORS
[LAMBDA (ACTOR_NAME UNDO)
  [IF, (NULL UNDO)
    (THEN (SETQ UNDO
              '(NO_ACTION)
            (IF (ACTOR? ACTOR_NAME)
              [THEN (PUT ACTOR_NAME 'ACTOR
                        (SUBST UNDO
                          (GET_BEHAVIORS ACTOR_NAME)
                          (GETP ACTOR_NAME 'ACTOR)
                        (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME)))
[PUT_CHILDREN
[LAMBDA (ACTOR_NAME NEW_LIST)
  [IF (NULL NEW_LIST)
    (THEN (SETQ NEW_LIST
              '(NO_CHILDREN)
            (IF (ACTOR? ACTOR_NAME)
              [THEN (PUT ACTOR_NAME 'ACTOR
                        (SUBST NEW_LIST
                          (GET_CHILDREN ACTOR_NAME)
                          (GETP ACTOR_NAME 'ACTOR)
                        (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME)))
[PUT_PARENT
[LAMBDA (ACTOR_NAME NEW_LIST)
  (IF
    (EQ ACTOR_NAME 'SYSTEM)
    (THEN
      (PRINT "You are not allowed to give a parent to
            actor SYSTEM")
      (TERPRI))
    (ELSE [IF (NULL NEW_LIST)
      (THEN (SETQ NEW_LIST
                '(SYSTEM))
            (ADD_CHILDREN 'SYSTEM
                          (LIST ACTOR_NAME)
            (IF (ACTOR? ACTOR_NAME)
              [THEN (PUT ACTOR_NAME 'ACTOR
                        (SUBST NEW_LIST
                          (GET_PARENT ACTOR_NAME)
                          (GETP ACTOR_NAME 'ACTOR)
                        (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME)))
[ADD_CHILDREN
[LAMBDA (ACTOR_NAME LUCKY_CHILD)
  (IF (ACTOR? ACTOR_NAME)
    [THEN (IF (EQUAL (GET_CHILDREN ACTOR_NAME)
                    '(NO_CHILDREN))
      (THEN
        (PUT_CHILDREN ACTOR_NAME LUCKY_CHILD))
      (ELSE (MAPC LUCKY_CHILD
                  (LAMBDA (CHILD)
                    (IF
                      [AND
                        (NEQ CHILD 'NO_CHILDREN)

```

```

        (NOT
          (MEMBER
            CHILD
            (GET_CHILDREN
              ACTOR_NAME])
          (THEN (RPLACD
            (LAST
              (GET_CHILDREN
                ACTOR_NAME))
            (LIST CHILD]
            (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME]))
[ADD_PARENT
[LAMBDA (ACTOR_NAME PARENT)
  (IF (EQ ACTOR_NAME 'SYSTEM)
    (THEN
      (PRINT "You are not allowed to add parents
        to actor SYSTEM")
      (TERPRI))
    (ELSE (IF (ACTOR? ACTOR_NAME)
      (THEN (IF (EQUAL (GET_PARENT ACTOR_NAME)
        ' (SYSTEM))
        (THEN
          (PUT_PARENT ACTOR_NAME PARENT))
        (ELSE
          (MAPC
            PARENT
            '(LAMBDA (PARENT)
              (IF
                (NOT
                  (MEMBER
                    PARENT
                    (GET_PARENT
                      ACTOR_NAME)))
                (THEN (RPLACD
                  (LAST
                    (GET_PARENT
                      ACTOR_NAME))
                  (LIST PARENT])
                (ELSE (PRINT (ACTOR_ERROR ACTOR_NAME]))
[DELETE_CHILDREN
[LAMBDA (ACTOR_NAME POOR_CHILD)
  (* WHEN CALLED WITH ONLY ONE PARAMETER THIS
    THIS PROCEDURE DELETES ALL CHILDREN OF THE ACTOR)

  (IF (ACTOR? ACTOR_NAME)
    (THEN (IF (NULL POOR_CHILD)
      (THEN (PUT_CHILDREN ACTOR_NAME))

```

```

(ELSE (MAPC POOR_CHILD
          '(LAMBDA (CHILD)
                (PUT_CHILDREN
                 ACTOR_NAME
                 (REMOVE
                  CHILD
                  (GET_CHILDREN
                   ACTOR_NAME)]))

[DELETE_PARENT
 (LAMBDA (ACTOR_NAME OLD_PARENT)

  (* WHEN CALLED WITH ONLY ONE PARAMETER THIS
     PROCEDURE DELETES ALL PARENTS OF THE ACTOR)

  (IF (ACTOR? ACTOR_NAME)
      (THEN (IF (NULL OLD_PARENT)
                (THEN (PUT_PARENT ACTOR_NAME))
                (ELSE (MAPC OLD_PARENT
                          '(LAMBDA (PARENT)
                                (PUT_PARENT
                                 ACTOR_NAME
                                 (REMOVE
                                  PARENT
                                  (GET_PARENT
                                   ACTOR_NAME)]))

[CREATE_ACTOR
 (LAMBDA (ACTOR_NAME FATHER)
  (IF (NULL FATHER)
      (THEN (SETQ FATHER
                  '(SYSTEM)
              (MAPC FATHER
                    '(LAMBDA (PAPA)
                          (ADD_CHILDREN PAPA
                           (LIST ACTOR_NAME)
                           (IF (ACTOR? ACTOR_NAME)
                               (THEN (MAPC FATHER
                                             '(LAMBDA (PAPA)
                                                  (ADD_PARENT ACTOR_NAME
                                                              (LIST PAPA)
                                                              (ELSE (SET (MKATOM (CONCAT ACTOR_NAME 'COMS))
                                                                (MKATOM (CONCAT 'ACTOR ACTOR_NAME)))
                                                                (PUT ACTOR_NAME 'ACTOR
                                                                    (LIST (LIST ACTOR_NAME)
                                                                    '(NO_ACTION)
                                                                    FATHER
                                                                    '(NO_CHILDREN)]))

[KILL_ACTOR
 (LAMBDA (ACTOR_NAME)
  (PROG1 (GET_CHILDREN ACTOR_NAME)
        (MAPC (GET_PARENT ACTOR_NAME)
              '(LAMBDA (FATHER)
                    (DELETE_CHILDREN FATHER
                     (LIST ACTOR_NAME)

```

```
[MAPC (GET_CHILDREN ACTOR_NAME)
      '(LAMBDA (CHILD)
        (DELETE_PARENT CHILD
          (LIST_ACTOR_NAME]
          (HEMPROP ACTOR_NAME 'ACTOR)))]

[DESTROY_ALL
(LAMBDA (ACTOR_NAME &LOCAL CHILDREN)
  (IF (NOT (EQUAL (SETQ CHILDREN
                    (KILL_ACTOR ACTOR_NAME))
                  '{NO_CHILDREN}))
    (THEN (MAPC CHILDREN
                 '(LAMBDA (CHILD)
                   (IF (EQUAL (GET_PARENT CHILD)
                             '(SYSTEM))
                     (THEN (DESTROY_ALL CHILD]))

[DESTROY_AND_PRESERVE
(LAMBDA (ACTOR_NAME &LOCAL PARENTS BEHAVIORS)
  (SETQ PARENTS
        (GET_PARENT ACTOR_NAME))
  (SETQ BEHAVIORS
        (GET_BEHAVIORS ACTOR_NAME))
  (MAPC (KILL_ACTOR ACTOR_NAME)
        '(LAMBDA (CHILD)
          (IF (NEQ CHILD 'NO_CHILDREN)
              (THEN (PUT_BEHAVIORS
                      CHILD
                      (UNION
                        (GET_BEHAVIORS
                          CHILD)
                        BEHAVIORS))
                    (ADD_PARENT CHILD PARENTS]))

[LOAD_APPLICATION
(LAMBDA (NAME)
  (ADD_CHILDREN 'SYSTEM
    (LIST NAME))
  (OPEN NAME 'I NAME)
  (LOAD NAME)
  (CLOSE NAME)
  (MAPC (EVAL (MKATOM (CONCAT NAME 'VARS)))
        '(LAMBDA (ACTOR)
          (PUT (FIRST_OF_FIRST ACTOR)
               'ACTOR ACTOR)))
  (MAPC (LIST "Application " NAME " loaded.")
        'PKIND)))

[SAVE_APPLICATION
(LAMBDA (NAME &LOCAL CHILDREN)
  [IF (NOT (ACTOR? NAME))
    (THEN (CREATE_ACTOR NAME)
          (PUT_CHILDREN NAME
            (GET_CHILDREN 'SYSTEM))
          (DELETE_CHILDREN NAME (LIST NAME))
          (MAPC (GET_CHILDREN NAME)
                '(LAMBDA (ACTOR)
                  (DELETE_PARENT ACTOR '(SYSTEM))
```

```

        (ADD_PARENT ACTOR (LIST NAME))))
      (SET (MKATOM (CONCAT NAME 'COMS))
        (MKATOM (CONCAT 'APPLICATION_ NAME)))
      (SET (MKATOM (CONCAT NAME 'FNS)) NIL]
    (OPEN NAME 'O NAME)
    (SET (MKATOM (CONCAT NAME 'VARS))
      (LIST (GETP NAME 'ACTOR)))
    (SETQ CHILDREN
      (GET_CHILDREN NAME))
    [WHILE CHILDREN
      (DO [SET (MKATOM (CONCAT NAME 'VARS))
        (UNION (EVAL
          (MKATOM (CONCAT NAME 'VARS))
          (LIST (GETP (FIRST CHILDREN)
            'ACTOR)
            (SET (MKATOM (CONCAT NAME 'FNS))
              (APPEND (EVAL (MKATOM
                (CONCAT NAME 'FNS)))
                (IF (NEQ
                  (FIRST
                    (MKATOM
                      (CONCAT
                        (FIRST CHILDREN)
                        'FNS)))
                    'NOBIND)
                  (THEN
                    (EVAL (MKATOM
                      (CONCAT
                        (FIRST
                          CHILDREN)
                          'FNS))))
                    (ELSE NIL]
          [SETQ CHILDREN
            (UNION CHILDREN
              (GET_CHILDREN
                (FIRST CHILDREN)
              (SETQ CHILDREN
                (REMOVE 'NO_CHILDREN
                  (AFTER_FIRST CHILDREN]
            (MAKEFILL NAME 2 100)
            (CLOSE NAME))]
    ]
    (SETQ ACTOREFNS
      (ACTOR_ERROR ACTOR? LOAD_ACTOR LOAD_TREE SAVE_ACTOR
        SAVE_TREE GET_KNOWLEDGE GET_BEHAVIORS GET_PARENT
        GET_CHILDREN PUT_KNOWLEDGE PUT_BEHAVIORS PUT_CHILDREN
        PUT_PARENT ADD_CHILDREN ADD_PARENT DELETE_CHILDREN
        DELETE_PARENT CREATE_ACTOR KILL_ACTOR DESTROY_ALL
        DESTROY_AND_PRESERVE LOAD_APPLICATION
        SAVE_APPLICATION))

```

```
(SETQ ACTORVARS  
      (DO_NOTHING))  
(SETQ ACTORCOMS PACKAGE_BODY_ACTOR)  
(SETQ ACTORGENNR 32)  
(SETQ DO_NOTHING NIL)  
STOP
```

A P P E N D I X E

P A C K A G E S I M U L A T I O N


```
*****
*
*                               PACKAGE SIMULATION
*
*****
```

Private Type MAIL is

Record

MYSELF: Atom

MESSAGE: List

CONTINUATION: List

End Record;

MAILBOX: List of Mail := NIL;

MESSAGE_TRACE: Boolean := NIL;

Procedure REPLY (MSG: List);

Procedure SCRIPT (SOME_ACTIONS: S-Expr);

Procedure EVALD (MSG_LIST: List);

Function ASK (SOMEBODY: Atom; SOMETHING: Text) Return S-Expr;

Procedure TELL (SOMEBODY: Atom; SOMETHING: Text);

Procedure I (TEXT: Text);

```

*****
*
*                               PACKAGE BODY SIMULATION
*
*****

```

With ACTION;

Use ACTION, ACTOR, LISP_SUPPORT;

SYSTEM-CLOCK: Integer;

CURRENT_CYCLE: List of Mail;

```

(PRINT 'PACKAGE_BODY_SIMULATION C T)
(PRINT '(VERSION 37))
[DEFINEQ
[SEND
[LAMBDA (MAIL &LOCAL DESTINATION MESSAGE)

```

(* THIS PROCEDURE PLACES A PIECE OF MAIL IN
THE MAILBOX)

(* MAIL CONSISTS OF THREE PARTS:)

(* THE ADDRESSEE)

(* THE MESSAGE)

(* THE CONTINUATION)

(SETQ DESTINATION
(FIRST MAIL))

(SETQ MESSAGE
(SECOND MAIL))

(SETQ CONTINUATION
(THIRD MAIL))

(SETQ MAILBOX
(NCONC MAILBOX
(LIST MAIL)))

(IF MESSAGE_TRACE
(THEN (PRINT (LIST MYSELF

'SENDS 'TO DESTINATION '
MESSAGE 'WITH 'CONTINUATION:
CONTINUATION))

[ACTIVATE

(LAMBDA (MAIL &LOCAL MYSELF MESSAGE)

(SETQ MYSELF
(FIRST MAIL))

(SETQ MESSAGE

```

      (SECOND MAIL))
    [IF MESSAGE_TRACE
      (THEN (PRINT (LIST MYSELF
        'RECEIVES 'MESSAGE:
        MESSAGE 'WITH 'CONTINUATION:
        CONTINUATION])
      (MAPC (FIND_CODE_FOR MESSAGE)
        'EVAL)))]
[SIMULATE
[LAMBDA (&LOCAL CURRENT_CYCLE)

  (* THIS PROCEDURE DRIVES THE SIMULATION FOR ONE
  CLICK DIVIDING IT INTO SEVERAL CYCLES)

  (* ONE CLICK STARTS WITH CALLING "SIMULATE" AND
  ENDS WHEN "MAILBOX" CONTAINS A MESSAGE
  ADDRESSED TO AUTHOR OR WHEN THE MAILBOX IS EMPTY)

  (PROG (RMAIL)
    LPROG ((SYSTEM-CLOCK 1))
    AGAIN
      (SETQ CURRENT_CYCLE MAILBOX)
      (SETQ MAILBOX NIL)
      (IF MESSAGE_TRACE
        (THEN (TERPRI)
          (PRINT (LIST '***** 'CYCLE
            'NUMBER
            SYSTEM-CLOCK
            '*****))
          (TERPRI)))
      (MAPC CURRENT_CYCLE 'ACTIVATE)
      (SETQ SYSTEM-CLOCK
        (ADD1 SYSTEM-CLOCK))
      (IF (OR (ASSOC 'AUTHOR MAILBOX)
        (NULL MAILBOX))
        (THEN (RETURN "FINI ONE CLICK"))
        (ELSE (GO AGAIN)
      (MAPC MAILBOX
        '(LAMBDA (MAIL)
          (MATCH '(AUTHOR ?MESSAGE ?)
            MAIL)))
      (SETQ RMAIL MESSAGE)
      (IF RMAIL
        (THEN (RETURN (IF (LESSP (LENGTH RMAIL)
          2)
          (THEN (FIRST RMAIL))
          (ELSE RMAIL))
        (ELSE (RETURN " ")))

[REPLY
[LAMBDA MSG
  [IF MESSAGE_TRACE
    (THEN (PRINT (LIST MYSELF 'REPLIES: MSG)
    (IF (EQUAL (LENGTH MSG)
      1)

```

```
[THEN (IF (ATOM (FIRST MSG))
            (THEN (FIRST MSG))
            (ELSE (IF (EQUAL (LENGTH (FIRST MSG))
                              1)
                      (THEN (FIRST_OF_FIRST MSG))
                      (ELSE (FIRST MSG))))
      (ELSE MSG)))]

[SCRIPT
(NLAMBDA SOME_ACTIONS SOME_ACTIONS)]
[EVALD
[LAMBDA (MSG_LIST)
  (IF (EQ (FIRST MSG_LIST)
          '!')
      (THEN (SETQ MSG_LIST
                  (AFTER_FIRST MSG_LIST))
             [RPLACA MSG_LIST
                (EVAL (EVALD (FIRST MSG_LIST)
                             MSG_LIST))]
           (ELSE (IF (EQ (FIRST (UNPACK (FIRST MSG_LIST)))
                       '!')
                     [THEN (RPLACA MSG_LIST
                                   (EVAL
                                    (PACK
                                     (AFTER_FIRST
                                      (UNPACK
                                       (FIRST MSG_LIST)
                                         )
                                     )
                                   )
                               (ELSE (IF MSG_LIST
                                           (THEN (EVALD
                                                  (AFTER_FIRST
                                                   MSG_LIST))
                                               (REMOVE '!' MSG_LIST))])
                   ]
              ]
    ]
[ASK
[NLAMBDA (SOMEBODY . SOMETHING)
  (PUT 'MYSELF 'MYSELF_STACK
       (CONS (FIRST 'MYSELF)
              (GETP 'MYSELF 'MYSELF_STACK)))
  (IF (ATOM SOMEBODY)
      (THEN (IF (EQ SOMEBODY '!')
                 (THEN (SETQ SOMEBODY
                             (EVAL (FIRST SOMETHING)))
                        (SETQ SOMETHING
                             (AFTER_FIRST SOMETHING)))
               (ELSE (IF (EQ (FIRST
                              (UNPACK SOMEBODY))
                          '!')
                         (THEN
                          (SETQ SOMEBODY
                                (EVAL
                                 (PACK
                                  (AFTER_FIRST
                                   (UNPACK
                                    SOMEBODY)
                                      )
                                  )
                                )
                            (SETQ SOMETHING
                                (PROG (MESSAGE)
```

```

LWHILE (NOT (NULL SOMETHING))
(DO [SETQ
      MESSAGE
      (APPEND
        MESSAGE
        (COND ((AND
                  (ATOM
                    (FIRST
                      SOMETHING)))
                 (NEQ (FIRST SOMETHING)
                       '!))
                (NEQ (FIRST
                      (UNPACK
                        (FIRST
                          SOMETHING)))
                      '!)))
          (LIST (FIRST SOMETHING)))
      [(EQ (FIRST SOMETHING) '!)]
      (SETQ SOMETHING
              (AFTER_FIRST
                SOMETHING))
      (LIST (EVAL
              (EVALD
                (FIRST SOMETHING]
            [(EQ (FIRST
                  (UNPACK
                    (FIRST SOMETHING)))
                  '!)]
            (LIST (EVAL
                   (PACK
                     (AFTER_FIRST
                       (UNPACK
                         (FIRST
                           SOMETHING]
                     ((LISTP (FIRST SOMETHING))
                      (LIST (EVALD
                            (FIRST SOMETHING))))
                    (T (LIST (FIRST SOMETHING]
              (SETQ SOMETHING
                      (AFTER_FIRST SOMETHING]
      (RETURN MESSAGE)))
[IF MESSAGE_TRACE
  (THEN (PRINT (LIST SOMEBODY 'RECEIVES
                        'MESSAGE: SOMETHING]
(PROG1 (IF (ACTOR? SOMEBODY)
           [THEN (SETQ MYSELF SOMEBODY)
                 (PROG ((CODE (FIND_CODE_FOR SOMETHING)))
                       [WHILE (GREATERP
                               (LENGTH CODE)
                               1)
                             (DO (EVAL (FIRST CODE))
                                   (SETQ CODE
                                           (AFTER_FIRST CODE])
               (RETURN (EVAL (FIRST CODE])

```

```

        (ELSE (IF (GETD SOMEBODY)
                  (THEN (APPLY SOMEBODY SOMETHING))
                  (ELSE (ACTOR_ERROR SCMEBCDY]
        (SETQ MYSELF (FIRST (GETP 'MYSELF 'MYSELF_STACK)))
        (PUT 'MYSELF 'MYSELF_STACK
        (AFTER_FIRST (GETP 'MYSELF 'MYSELF_STACK]

[TELL
(NLAMBDA SOMETHING
  (EVAL (CONSTRUCT_NEW_LIST 'ASK SOMETHING))
  " ")
]
[END
• (NLAMBDA NIL
  (MAPC (OPENP NIL 'OUTPUT)
        '(LAMBDA (FILE)
          (OUTPUT FILE)
          'STOP
          (OUTPUT T)))
  (CLOSEALL 'SAVE)
  (RETURN)))
]
[
[NLAMBDA TEXT
  (EVAL (CONSTRUCT_NEW_LIST 'ASK
    (CONSTRUCT_NEW_LIST 'MYSELF
    (SUBST 'YOUR 'MY TEXT]))
]
]
(SETQ
  SIMULATIONFNS
  (SEND ACTIVATE SIMULATE REPLY SCRIPT EVALD ASK TELL AND I))
(SETQ SIMULATIONVARS
  ((SETQ MAILBOX NIL)
   (SETQ MESSAGE_TRACE NIL)))
(SETQ SIMULATIONCOMS PACKAGE_BODY_SIMULATION)
(SETQ SIMULATIONGENR 37)
STOP

```

A P P E N D I X F

P A C K A G E M M M

```

*****
*                                     *
*                                     *
*                                     *
*                                     *
*                                     *
*****

```

PACKAGE MMM

Type COMPASS is (SW, SE, S, NW, NE, N, E, W);

Private Type COORDINATE is

Record

X-POSITION: Integer Range 1..60

Y-POSITION: Integer Range 1..20

End Record;

Type VELOCITY is

Record

DIRECTION: Compass_Direction

SPEED: Integer

End Record;

Type LOCATION is Integer Range 0..999999;

Type UPDATE is String;

Type DATE is Integer Range 010100..311299;

Type TIME is Integer Range 0..2400;

Procedure CALCULATE_INCREMENTS;

Function MPH_TO_SPEED (MPH: Integer) Return Integer;

Function KPH_TO_SPEED (KPH: Integer) Return Integer;

Function LOCATION_TO_COORDINATE (GRID_REFERENCE: Location)
Return Coordinate;

Function DATE (DTG: Update) Return Date;

Function TIME (DTG: Update) Return Time;


```

      (THEN (LIST (DIFFERENCE
                  (QUOTIENT GRID_REFERENCE 1000)
                  (QUOTIENT
                   (FIRST (FIRST (RECALL 'CORNERS)))
                   1000))
                (QUOTIENT [DIFFERENCE GRID_REFERENCE
                           (FIRST (FIRST (RECALL 'CORNERS)
                                         10)))
                (ELSE (ASK OVERLAY REDIRECT !SOMETHING]))
[DATE
[LAMBDA (DTG)
  (MKATOM (CONCAT (SUBSTRING DTG 1 2)
              (SELECTQ (MKATOM (SUBSTRING DTG 8 10))
                       (JAN "01")
                       (FEB "02")
                       (MAR "03")
                       (APR "04")
                       (MAY "05")
                       (JUN "06")
                       (JUL "07")
                       (AUG "08")
                       (SEP "09")
                       (OCT "10")
                       (NOV "11")
                       "12"))
          (SUBSTRING DTG 11 12]))
[TIME
[LAMBDA (DTG)
  (MKATOM (SUBSTRING DTG 3 6)))]
[SETUP_MMM
[NLAMBDA NIL
  (DEFINE_RECORD VELOCITY DIRECTION SPEED)
  (DEFINE_RECORD COORDINATE X-POSITION Y-POSITION)
  (MAKE_RECORD 'VELOCITY 'VELOCITY)
  (MAKE_RECORD 'SCREEN_POSITION 'COORDINATE))]
]
[SETQ MMHENS
  (CALCULATE_INCREMENTS MPH_TO_SPEED KPH_TO_SPEED
    LOCATION_TO_COORDINATE DATE TIME SETUP_MMM))
[SETQ MMMVARS
  (((MMM)
    (NO_ACTION)
    (SYSTEM)
    (BASIC_SYMBOLS SIZE_SYMBOLS TYPE_SYMBOLS
      OVERLAY CLOCK))
    (BASIC_SYMBOLS
      (BASE_SYMBOL
        (VALUE
          ("Instructions for drawing base symbol"))))
      (LOCATING_SYMBOL
        (VALUE
          ("Instructions for drawing
            locating symbol"))))
      (HEADQUARTERS_SYMBOL

```

```

(VALUE
  ("Instructions for drawing headquarters
   symbol")
LNO_ACTION ((MOVE AT ?DTG)
            ((ASK OVERLAY DE-REGISTER !MYSELF AT
              !(I RECALL MY LOCATION))
             (UPDATE_LOCATION
              (I RECALL MY VELOCITY)
              (I RECALL MY LAST_UPDATE)
              DTG)
             (ASK OVERLAY REGISTER !MYSELF AT
              LOCATION
              !(I RECALL MY LOCATION])

(MMM)
(NO_CHILDREN))

((SIZE_SYMBOLS)
 (NO_ACTION)
 (MMM)
 (ARMY CORPS DIVISION BRIGADE BATTALION COMPANY
  PLATOON))

((TYPE_SYMBOLS)
 (NO_ACTION)
 (MMM)
 (ARMS SERVICES))

(OVERLAY
 (CORNERS (IF-ADDED
            (CALCULATE_INCREMENTS]
LNO_ACTION [(SHOW YOURSELF AT ?DTG)
            ((ASK CLOCK UPDATE AT !DTG)
             (MAPC (I RECALL MY UNITS)
                   '(LAMBDA (UNIT)
                     (I ADD
                      !(LIST UNIT
                       (LOCATION_TO_
                        COORDINATE
                       (ASK !UNIT
                        RECALL
                        YOUR
                        LOCATION
                       )))
                     TO MY LIST OF MAP)))
            (SHOW_MAP)
            (FORGET MAP)
            (REPLY
             (CONCAT
              "
             (CONCAT
              "Overlay No: "
             (CONCAT

```



```

                                ))
                                100Q))
                                (THEN
                                  (ASK
                                    !CHILD
                                    ADD !UNIT
                                    TO YOUR
                                    LIST OF
                                    UNITS])

                                (REPLY 'DONE)))
                                ((DE-REGISTER ?UNIT AT ?GRID_REFERENCE)
                                  ((MAPC (GET_CHILDREN MYSELF)
                                    '(LAMBDA (CHILD)
                                      (ASK !CHILD REMOVE !UNIT
                                        FROM YOUR LIST OF
                                        UNITS)))

                                (REPLY 'OK])

                                (MMM)
                                (NO_CHILDREN))

                                ((CLOCK)
                                  LNO_ACTION ((UPDATE AT ?DTG)
                                    ((MAPC (I RECALL MY MOVING_UNITS)
                                      '(LAMBDA (UNIT)
                                        (ASK !UNIT MOVE AT !DTG)))

                                  (REPLY 'UPDATED])

                                (MMM)
                                (NO_CHILDREN))

                                (EARMY
                                  (SIZE_SYMBOL
                                    (VALUE ("Instructions for drawing XXXX
                                      over base symbol")))
                                  (STRENGTH (VALUE (100000))
                                  (NO_ACTION)
                                  (SIZE_SYMBOLS)
                                  (NO_CHILDREN))

                                (LCORPS
                                  (SIZE_SYMBOL
                                    (VALUE ("Instructions for drawing XXX
                                      over base symbol")))
                                  (STRENGTH (VALUE (30000))
                                  (NO_ACTION)
                                  (SIZE_SYMBOLS)
                                  (NO_CHILDREN))

                                (DIVISION
                                  (SIZE_SYMBOL
                                    (VALUE ("Instructions for drawing XX
                                      over base symbol")))
                                  (STRENGTH (VALUE (15000))
                                  (NO_ACTION)
                                  (SIZE_SYMBOLS)

```

```
(NO_CHILDREN))

(LBRIGADE
  (SIZE_SYMBOL
    (VALUE ("Instructions for drawing X
            over base symbol"))))
  (STRENGTH (VALUE (5000))
  (NO_ACTION)
  (SIZE_SYMBOLS)
  (NO_CHILDREN))

(LBATTALION
  (SIZE_SYMBOL
    (VALUE ("Instructions for drawing II
            over base symbol"))))
  (STRENGTH (VALUE (400))
  (NO_ACTION)
  (SIZE_SYMBOLS)
  (NO_CHILDREN))

(LCOMPANY
  (SIZE_SYMBOL
    (VALUE ("Instructions for drawing I
            over base symbol"))))
  (STRENGTH (VALUE (110))
  (NO_ACTION)
  (SIZE_SYMBOLS)
  (NO_CHILDREN))

(LPLATOON
  (SIZE_SYMBOL
    (VALUE ("Instructions for drawing ...
            over base symbol"))))
  (STRENGTH (VALUE (35))
  (NO_ACTION)
  (SIZE_SYMBOLS)
  (NO_CHILDREN))

((ARMS)
  (NO_ACTION)
  (TYPE_SYMBOLS)
  (INFANTRY ARMoured ARTILLERY ENGINEERS SIGNALS))

(LSERVICES
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing SVC
            inside base symbol"))
  (NO_ACTION)
  (TYPE_SYMBOLS)
  (TRANSPORT SUPPLY MAINTENANCE PAY POSTAL
  CHAPLAIN MILITARY_POLICE))

(LINFANTRY
  (TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing infantry  
symbol inside base symbol")
```

```
(NO_ACTION)
```

```
(ARMS)
```

```
(NO_CHILDREN))
```

```
([ARMoured
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing armoured  
symbol inside base symbol")
```

```
(NO_ACTION)
```

```
(ARMS)
```

```
(NO_CHILDREN))
```

```
([ARTILLERY
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing artillery  
symbol inside base symbol")
```

```
(NO_ACTION)
```

```
(ARMS)
```

```
(NO_CHILDREN))
```

```
([ENGINEERS
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing engineers  
symbol inside base symbol")
```

```
(NO_ACTION)
```

```
(ARMS)
```

```
(NO_CHILDREN))
```

```
([SIGNALS
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing  
communications symbol inside  
inside base symbol")
```

```
(NO_ACTION)
```

```
(ARMS)
```

```
(NO_CHILDREN))
```

```
([TRANSPORT
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing  
transportation symbol inside  
base symbol")
```

```
(NO_ACTION)
```

```
(SERVICES)
```

```
(NO_CHILDREN))
```

```
([SUPPLY
```

```
(TYPE_SYMBOL
```

```
(VALUE ("Instructions for drawing supply  
symbol inside base symbol")
```

```
(NO_ACTION)
```

```
(SERVICES)
```

```
(NO_CHILDREN))

(LMAINTENANCE
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing
            maintenance symbol inside
            base symbol"])
  (NO_ACTION)
  (SERVICES)
  (NO_CHILDREN))

(LPAY
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing pay
            services symbol inside base symbol"])
  (NO_ACTION)
  (SERVICES)
  (NO_CHILDREN))

(LPOSTAL
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing postal
            symbol inside base symbol"])
  (NO_ACTION)
  (SERVICES)
  (NO_CHILDREN))

(LCHAPLAIN
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing chaplain
            symbol inside base symbol"])
  (NO_ACTION)
  (SERVICES)
  (NO_CHILDREN))

(LMILITARY_POLICE
  (TYPE_SYMBOL
    (VALUE ("Instructions for drawing MP symbol
            inside base symbol"])
  (NO_ACTION)
  (SERVICES)
  (NO_CHILDREN))

(SETQQ MMMCOMS APPLICATION_MMM)
(SETQ MMMGENNR 1)
STOP
```


APPENDIX G
PACKAGE
ACTOR SYSTEM

```

*****
*
*                               MESSAGE PATTERNS
*                               HANDLED BY ACTOR SYSTEM
*
*****

```

NOTE: [] denote words or group of words that are optional.
i.e. they may be replaced by any other words, group
of words or may be left out entirely.

(CREATE ?ACTOR_NAME)

(CREATE ?ACTOR_NAME [CHILD OF] ?LIST_OF_PARENTS)

(DUPLICATE AND NAME IT ?ACTOR_NAME)

(CLONE AND NAME IT ?ACTOR_NAME)

(CEASE TO EXIST)

(CEASE TO EXIST AND BRING YOUR DEPENDENTS)

(LIST YOUR ATTRIBUTES)

(SET YOUR ?ATTRIBUTE TO ?NEW_VALUE)

(CHANGE YOUR ?ATTRIBUTE TO ?NEW_VALUE)

(PRINT YOUR ?ATTRIBUTE)

(RECALL YOUR ?ATTRIBUTE)

(SET ?DEMON_NAME AS A ?DEMON_TYPE DEMON FOR ?ATTRIBUTE)

(DEFINE [A] DEMON ?DEMON_NAME WITH [THE FOLLOWING LIST
OF ACTIONS:]

?FUNCTION_DEFINITION)

(RECALL YOUR ?DEMON FOR ?ATTRIBUTE)

(ADD ?NEW_VALUE [TO YOUR LIST OF VALUES FOR] ?ATTRIBUTE)

(REMOVE ?OLD_VALUE [FROM YOUR LIST OF VALUES FOR] ?ATTRIBUTE)

(REMOVE ALL ?ATTRIBUTE)

(INCREMENT YOUR ?NUMERICAL_ATTRIBUTE BY ?AMOUNT)

(MULTIPLY YOUR ?NUMERICAL_ATTRIBUTE BY ?FACTOR)
(FORGET THAT ?ATTRIBUTE HAD ?VALUE [AS ONE OF ITS VALUES])
(FORGET YOUR ?ATTRIBUTE)
(FORGET EVERYTHING)
(DO WHEN RECEIVING ?MESSAGE_PATTERN ?LIST_OF_ACTIONS)
(LIST YOUR PATTERNS)
(LIST YOUR SCRIPT FOR ?MESSAGE_PATTERN)
(ADD_ACTION ?SOME_CODE TO ?MESSAGE_PATTERN)
(DELETE ?SOME_CODE FROM SCRIPT FOR ?MESSAGE_PATTERN)
(DELETE ALL YOUR BEHAVIORS)
(REMOVE THE BEHAVIOR [FOR] ?MESSAGE_PATTERN)
(REMOVE THE BEHAVIOR [WHOSE PATTERN IS] ?PATTERN_NAME)
(SAVE [YOUR PRESENT/CURRENT] STATE)
(LOAD ACTOR ?ACTOR_NAME)
(TRACE MESSAGES)
(UNTRACE MESSAGES)

```

*****
*
*               ACTOR SYSTEM
*
*****

```

```

(PRIND 'ACTOR_SYSTEM 0 T)
(PRINT '(VERSION 27))
(SETQ SYSTEMVARS
  ((SYSTEM)

```

```

[NO_ACTION ((CREATE ?ACTOR_NAME)
  ((CREATE_ACTOR ACTOR_NAME
    (LIST MYSELF))
    (REPLY ACTOR_NAME)))

```

```

((CREATE ?ACTOR_NAME + ?LIST_OF_PARENTS)
  ((CREATE_ACTOR ACTOR_NAME
    LIST_OF_PARENTS)
    (REPLY ACTOR_NAME)))

```

```

((CEASE TO EXIST AND BRING
  YOUR DEPENDENTS)
  ((DESTROY_ALL MYSELF)
    (REPLY 'ALL 'GONE)))

```

```

((? AND NAME IT ?ACTOR_NAME)
  ((PUT ACTOR_NAME 'ACTOR
    (GETP MYSELF 'ACTOR))
    (REPLY 'DONE)))

```

```

((DO WHEN RECEIVING
  ?MESSAGE_PATTERN ?LIST_OF_ACTIONS)
  ((ADD_BEHAVIOR
    (LIST MESSAGE_PATTERN ACTIONS))
    (REPLY 'OK)))

```

```

((PRINT YOUR ?ATTRIBUTE)
  ((PRINT (RECALL ATTRIBUTE))
    (REPLY " ")))

```

```

((SAVE + STATE)
  ((SAVE_ACTOR MYSELF)
    (REPLY MYSELF 'SAVED))

```

```

((LOAD ACTOR ?ACTOR_NAME)
  ((LOAD_ACTOR ACTOR_NAME)
    (MAFC (GET_PARENT ACTOR_NAME)
      '(LAMBDA (FATHER)
        (ADD_CHILDREN
          FATHER
          (LIST ACTOR_NAME))))))

```

3 3

OF / DE

```
(REPLY ACTOR_NAME 'INSERTED']  
  
((REMOVE THE BEHAVIOR + ?MESSAGE_PATTERN)  
((DELETE_BEHAVIOR MESSAGE_PATTERN)  
(REPLY 'DONE)))  
  
((SET YOUR ?ATTRIBUTE TO ?NEW_VALUE)  
((FORGET ATTRIBUTE)  
(REPLY (LEARN ATTRIBUTE ?NEW_VALUE))))  
  
[(INCREMENT YOUR ?NUMERICAL_ATTRIBUTE  
BY ?AMOUNT)  
((REPLY  
  (CHANGE  
    NUMERICAL_ATTRIBUTE  
    (PLUS  
      (FIRST  
        (RECALL  
          NUMERICAL_ATTRIBUTE))  
      AMOUNT])  
  .[(MULTIPLY YOUR ?NUMERICAL_ATTRIBUTE  
  BY ?FACTOR)  
  ((REPLY  
    (CHANGE  
      NUMERICAL_ATTRIBUTE  
      (TIMES  
        (FIRST  
          (RECALL  
            NUMERICAL_ATTRIBUTE))  
        FACTOR])  
  ((ADD ?NEW_VALUE ?ATTRIBUTE)  
  ((REPLY (LEARN ATTRIBUTE ?NEW_VALUE)))  
  
  ((REMOVE ?OLD_VALUE + ?ATTRIBUTE)  
  ((IF (EQ OLD_VALUE 'ALL)  
    (THEN (SETQ OLD_VALUE NIL)))  
  (REPLY (FORGET ATTRIBUTE OLD_VALUE)))  
  
  ((FORGET YOUR ?ATTRIBUTE)  
  ((FORGET ATTRIBUTE)  
  (REPLY 'OK)))  
  
  ((CEASE TO EXIST)  
  ((DESTROY_AND_PRESERVE MYSELF)  
  (REPLY 'GOODBYE)))  
  
  ((FORGET EVERYTHING)  
  ((PUT_KNOWLEDGE MYSELF  
    (LIST MYSELF))  
  (REPLY 'EUH!)))  
  
  ((RECALL YOUR ?DEMON FOR ?ATTRIBUTE)
```

```

((RECALL ATTRIBUTE DEMON)))

((RECALL YOUR ?ATTRIBUTE)
 ((REPLY (REMEMBER ATTRIBUTE))))

((FORGET THAT ?ATTRIBUTE HAD ?VALUE +)
 ((REPLY (FORGET ATTRIBUTE VALUE))))

((LIST YOUR PATTERNS)
 ((MAPC (LIST
        "The following is a list of
        patterns understood by"
        " actor "
        MYSELF)
        'PRINT)
 (TERPRI)
 (MAPC (MAPCAR (AFTER FIRST
                (GET_BEHAVIORS MYSELF))
                '(LAMBDA (BEHAVIOR)
                    (FIRST BEHAVIOR)))
        'PRINT)
 (TERPRI)))

((LIST YOUR SCRIPT FOR ?MESSAGE_PATTERN)
 ((FIND_CODE_FOR MESSAGE_PATTERN)))

((TRACE MESSAGES)
 ((SETQ MESSAGE_TRACE T)
 (REPLY 'TRACE 'ON)))

((UNTRACE MESSAGES)
 ((SETQ MESSAGE_TRACE NIL)
 (REPLY 'TRACE 'OFF)))

((ADD_ACTION ?SOME_CODE TO
 ?MESSAGE_PATTERN)
 ([IF (ATOM (FIRST SOME_CODE))
      (THEN (SETQ SOME_CODE
                (LIST SOME_CODE)
                (ADD_CODE (APPEND (LIST MESSAGE_PATTERN)
                                (LIST SOME_CODE)))
                (REPLY 'DONE)))

      (DELETE ?SOME_CODE FROM SCRIPT FOR
 ?MESSAGE_PATTERN)
      ([IF (ATOM (FIRST SOME_CODE))
           (THEN (SETQ SOME_CODE
                     (LIST SOME_CODE)
                     (DELETE_CODE
                      (APPEND (LIST MESSAGE_PATTERN)
                              (LIST SOME_CODE)))
                      (REPLY 'DONE)))

           (LIST YOUR ATTRIBUTES)

```

```
((REPLY (LIST_ATTRIBUTES MYSELF))))
```

```
((DELETE ALL YOUR BEHAVIORS)
((PUT_BEHAVIORS MYSELF)
(REPLY "I CANNOT DO A THING
ANYMORE"))))
```

```
((DEFINE + DEMON ?DEMON_NAME WITH +
?FUNCTION_DEFINITION)
((DEFINE_DEMON DEMON_NAME
FUNCTION_DEFINITION)
(REPLY 'DONE)))
```

```
((SET ?DEMON_NAME AS A ?DEMON_TYPE DEMON
FOR ?ATTRIBUTE)
((LEARN ATTRIBUTE DEMON_TYPE DEMON_NAME)
(REPLY 'OK)))
```

```
((CHANGE YOUR ?ATTRIBUTE TO ?NEW_VALUE)
((REPLY (CHANGE ATTRIBUTE NEW_VALUE)
```

```
(NO_PARENT)
```

```
(NO_CHILDREN)))
```

```
(SETQ SYSTEMCOMS ACTOR_SYSTEM)
(SETQ SYSTEMGENK 27)
STOP
```


APPENDIX H

SAMPLE SESSIONS

WITH OASYS

```

*****
*
*                      SETUP PROGRAM
*                      FOR
*                      APPLICATION COMPUTER_SCIENCE
*
*****

```

```

(* This program sets-up and saves
   application COMPUTER_SCIENCE.)
(* The application contains generic actors for course
   scheduling, instructor and students allocation to
   courses in a computer science department)
(ASK SYSTEM CREATE PROFESSORS)
(ASK PROFESSORS DO WHEN RECEIVING
 (WHAT COURSES ARE YOU TEACHING?)
 ((PRIN "I teach ")
  (PROG ((COURSE_LIST (I RECALL MY COURSES)))
   (IF (NULL COURSE_LIST)
    (THEN (PRIN "nothing."))
    (ELSE (IF (EQUAL (LENGTH COURSE_LIST) 1)
     (THEN (PRIN
      (CONCAT
       (FIRST COURSE_LIST)
       ".)))
     (ELSE (WHILE
      (NOT
       (EQUAL (LENGTH COURSE_LIST)
        1))
      (DO (PRIN
       (CONCAT
        (FIRST COURSE_LIST)
        ", "))
       (SETQ COURSE_LIST
        (AFTER_FIRST
         COURSE_LIST))))
      (PRIN (CONCAT
       "and "
       (CONCAT
        (FIRST COURSE_LIST)
        ".)))))))
    (TERPRI)
    (REPLY "% )))
(ASK PROFESSORS DO WHEN RECEIVING
 (HOW MANY STUDENTS DO YOU TEACH?)
 ((PROG ((COUNT 0))
  (MAPC (I RECALL MY COURSES)
   (LAMBDA (COURSE)
    (SETQ COUNT
     (PLUS COUNT
      (ASK !COURSE
       HOW MANY STUDENTS?))))))
  (REPLY COUNT)))

```

```

(ASK PROFESSORS DEFINE A DEMON CHANGE_PROF WITH ACTIONS:
  ((ASK !COURSE_NUMBER SET YOUR PROFESSOR TO !MYSELF)))
(ASK PROFESSORS SET CHANGE_PROF AS A IF-ADDED DEMON
  FOR COURSES)
(ASK PROFESSORS DEFINE A DEMON NO_PROF WITH
  ((ASK !COURSE_NUMBER SET YOUR PROFESSOR TO
    Not_allocated)))
(ASK PROFESSORS SET NO_PROF AS A IF-REMOVED DEMON
  FOR COURSES)
(ASK PROFESSORS DO WHEN RECEIVING
  (+ TEACH ?COURSE_NUMBER)
  ((IF (LESSP (LENGTH (I RECALL MY COURSES))
    3)
    (THEN (I ADD !COURSE_NUMBER TO MY LIST
      OF COURSES)
      (REPLY 'OK))
    (ELSE (PRIN0 "Sorry, I already have three")
      (PRIN0 " courses to teach this term.")
      (TERPRI)
      (ASK !MYSELF WHAT COURSES ARE YOU
        TEACHING?))))))
(ASK PROFESSORS DO WHEN RECEIVING
  (YOU ARE NO LONGER ASSIGNED TO ?COURSE_NUMBER)
  ((I REMOVE !COURSE_NUMBER FROM MY LIST OF COURSES)
  (REPLY "Thank you.")))
(ASK SYSTEM CREATE STUDENTS)
(ASK STUDENTS DEFINE A DEMON REMOVE_STUDENT WITH ACTIONS:
  ((ASK !COURSE_NUMBER REMOVE !MYSELF FROM YOUR LIST
    OF STUDENTS)))
(ASK STUDENTS SET REMOVE_STUDENT AS A IF-REMOVED DEMON
  FOR COURSES)
(ASK STUDENTS DO WHEN RECEIVING
  (REGISTER ?STUDENT_NUMBER)
  ((I CREATE !STUDENT_NUMBER)
  (ASK !STUDENT_NUMBER SET YOUR NAME TO
    ! (PROMPTREAD "Name? "))
  (PROG (COURSE_NUMBER)
    LOOP (SETQ COURSE_NUMBER
      (PROMPTREAD
        "Course Number? "))
    (IF COURSE_NUMBER
      (THEN (IF (EQ (ASK !COURSE_NUMBER
        ANY ROOM?)
        'YES)
        (THEN (ASK !COURSE_NUMBER ADD
          !STUDENT_NUMBER
          TO YOUR
          LIST OF STUDENTS)
          (ASK !STUDENT_NUMBER ADD
            !COURSE_NUMBER
            TO YOUR
            LIST OF COURSES)
          (PRINT 'OK))
        (ELSE (PRIN0 "Sorry, course ")

```

```

                                (PRIN0 COURSE_NUMBER)
                                (PRIN0 " is full.")
                                (TERPRI)))
                                (GO LOOP)))
                                (ELSE (RETURN)))
    (IF (ASK !STUDENT_NUMBER RECALL YOUR COURSES)
        (THEN (REPLY STUDENT_NUMBER
                    (ASK !STUDENT_NUMBER RECALL YOUR NAME)
                    'REGISTERED))))))
    (ASK STUDENTS DO WHEN RECEIVING
      (ADD ?COURSE_NUMBER)
      ((IF (EQ (ASK !COURSE_NUMBER ANY ROOM?)
                'YES)
          (THEN (ASK !COURSE_NUMBER ADD !MYSELF TO
                    YOUR LIST OF STUDENTS)
                (I ADD !COURSE_NUMBER TO MY LIST OF
                  COURSES)
                (REPLY 'DONE))
          (ELSE (REPLY 'course COURSE_NUMBER 'full))))))
    (ASK STUDENTS DO WHEN RECEIVING
      (DROP ?COURSE_NUMBER)
      ((I REMOVE !COURSE_NUMBER FROM MY LIST OF COURSES)))
    (ASK SYSTEM CREATE COURSES)
    (ASK COURSES DO WHEN RECEIVING
      (HOW MANY STUDENTS?)
      ((REPLY (LENGTH (I RECALL MY STUDENTS)))))
    (ASK COURSES DO WHEN RECEIVING
      (+ A LIST OF STUDENTS)
      ((IF (LESSP (LENGTH (I RECALL MY STUDENTS))
                  1)
          (THEN (REPLY 'None))
          (ELSE (MAPC (I RECALL MY STUDENTS)
                      '(LAMBDA (STUDENT)
                        (PRIN0 STUDENT)
                        (PRIN0 " ")
                        (PRIN0 (ASK !STUDENT
                                RECALL YOUR NAME))
                        (TERPRI)))
                      (REPLY "End of Student List")))))
    (ASK COURSES DO WHEN RECEIVING
      (ANY ROOM?)
      ((IF (LESSP (LENGTH (I RECALL MY STUDENTS))
                  (I RECALL MY QUOTA))
          (THEN (REPLY 'YES))
          (ELSE (REPLY 'NO))))))
    (SAVE_APPLICATION 'COMPUTER_SCIENCE)
    (CLOSEF)

```

*
* SESSION NO 1 *
*

DRIBBLE ON W. 09:01 06/01/82

T

-
(* This session uses SAMPLE_SETUP as the input file to OASYS)
This

-
(* The file contains the program mentioned above)
The

-
(OASYS SAMPLE_SETUP)

welcome to OASYS

PACKAGE_BODY_SIMULATION(VERSION 37)
SIMULATION - DP#ACAD03/SIMULATION.PA088542 LOADED
PACKAGE_BODY_ACTOR(VERSION 32)
ACTOR - DP#ACADU3/ACTOR.PA088542 LOADED
PACKAGE_BODY_ACTION(VERSION 21)
ACTION - DP#ACADU3/ACTION.PA088542 LOADED
PACKAGE_BODY_KNOWLEDGE(VERSION 26)
KNOWLEDGE - DP#ACAD03/KNOWLEDGE.PA088542 LOADED

ACTOR_SYSTEM(VERSION 27)
SYSTEM - DP#ACADU3/SYSTEM.PA088542 LOADED

T
This
The
PROFESSORS
OK
OK
DONE
OK
DONE
OK
OK
OK
STUDENTS
DONE
OK
OK
OK
OK
COURSES

OK

OK

OK

COMPUTER_SCIENCE - DP#ACAD03/COMPUTER_SCIENCE.PA088542

COMPUTER_SCIENCE

DP#ACAD03/SAMPLE_SETUP.PA088542

(IBEX "DONT DRIBBLE")

DRIBBLE OFF @ 09:06 06/01/82

```
*****
*
*                               SESSION NO 2
*
*****
```

DRIBBLE ON @ 09:16 06/01/82

T

(* In this session, the section head decides what courses will
be taught during the term and assigns an instructor to each)
In

(OASYS)

welcome to OASYS

PACKAGE_BODY_SIMULATION(VERSION 37)
SIMULATION - DP#ACAD03/SIMULATION.PA088542 LOADED
PACKAGE_BODY_ACTOR(VERSION 32)
ACTOR - DP#ACAD03/ACTOR.PA088542 LOADED
PACKAGE_BODY_ACTION(VERSION 21)
ACTION - DP#ACAD03/ACTION.PA088542 LOADED
PACKAGE_BODY_KNOWLEDGE(VERSION 26)
KNOWLEDGE - DP#ACAD03/KNOWLEDGE.PA088542 LOADED

ACTOR_SYSTEM(VERSION 27)
SYSTEM - DP#ACAD03/SYSTEM.PA088542 LOADED

Any applications to be loaded?
YES

Application Name:
COMPUTER_SCIENCE

APPLICATION_COMPUTER_SCIENCE(VERSION 0)
COMPUTER_SCIENCE - DP#ACAD03/COMPUTER_SCIENCE.PA088542 LOADED
Application COMPUTER_SCIENCE loaded.

Application Name:
NIL

<->
(ASK COURSES CREATE 95.101)
95.101
<->
(ASK 95.101 SET YOUR QUOTA TO 100)
100
<->
(ASK COURSES CREATE 95.102)
95.102

<->
(ASK 95.102 SET YOUR QUOTA TO 30)
30
<->
(ASK COURSES CREATE 95.103)
95.103
<->
(ASK 95.103 SET YOUR QUOTA TO 110)
110
<->
(ASK COURSES CREATE 95.104)
95.104
<->
(ASK 95.104 SET YOUR QUOTA TO 10)
10
<->
(ASK COURSES CREATE 95.105)
95.105
<->
(ASK 95.105 SET YOUR QUOTA TO 50)
50
<->
(ASK COURSES CREATE 95.106)
95.106
<->
(ASK 95.106 SET YOUR QUOTA TO 120)
120
<->
(ASK PROFESSORS CREATE TEACHNONE)
TEACHNONE
<->
(ASK PROFESSORS CREATE TEACHALL)
TEACHALL
*->
(ASK PROFESSORS CREATE TEACHSOME)
TEACHSOME
<->
(ASK TEACHALL PLEASE TEACH 95.101)
OK
<->
(ASK TEACHALL YOU ARE TO TEACH 95.102)
OK
<->
(ASK TEACHALL TEACH 95.103)
OK
<->
(ASK TEACHALL CAN YOU TEACH 95.104)
Sorry, I already have three courses to teach this term.
I teach 95.101, 95.102, and 95.103.

<->
(ASK TEACHSOME CAN YOU TEACH 95.104)
OK

<->
(ASK TEACHSOME TEACH 95.105)
OK
<->
(ASK TEACHSOME TEACH 95.106)
OK
<->
(ASK TEACHNONE WHAT COURSES ARE YOU TEACHING?)
I teach nothing.
<->
(ASK TEACHSOME YOU ARE NO LONGER ASSIGNED TO 95.104)
Thank you.
<->
(ASK TEACHNONE TEACH 95.104)
OK
<->
(ASK TEACHSOME WHAT COURSES ARE YOU TEACHING?)
I teach 95.105, and 95.106.
<->
(END)
Do you wish to save some applications?
YES

Application Name:
COMPUTER_SCIENCE

COMPUTER_SCIENCE - DPHACAD03/COMPUTER_SCIENCE.PA088542
Application Name:
NIL

(OASYS TERMINATED - - YOU ARE NOW IN CP6 LISP - - THANK YOU)

(IBEX "DONT DRIBBLE")
DRIBBLE OFF @ 09:45 06/01/82

*
* SESSION NO 3 *
*

DRIBBLE ON 10:06 06/03/82

-
(* In this session, students are registered in courses)
In-
(OASYS)

welcome to OASYS

PACKAGE_BODY_SIMULATION(VERSION 37)
SIMULATION - DP#ACAD03/SIMULATION.PA088542 LOADED
PACKAGE_BODY_ACTOR(VERSION 32)
ACTOR - DP#ACADU3/ACTOR.PA088542 LOADED
PACKAGE_BODY_ACTION(VERSION 21)
ACTION - DP#ALADU3/ACTION.PA088542 LOADED
PACKAGE_BODY_KNOWLEDGE(VERSION 26)
KNOWLEDGE - DP#ACAD03/KNOWLEDGE.PA088542 LOADEDACTOR_SYSTEM(VERSION 27)
SYSTEM - DP#ACADU3/SYSTEM.PA088542 LOADEDAny applications to be loaded?
-YESApplication Name:
COMPUTER_SCIENCEAPPLICATION_COMPUTER_SCIENCE(VERSION 2)
COMPUTER_SCIENCE - DP#ACAD03/COMPUTER_SCIENCE.PA088542 LOADED
Application COMPUTER_SCIENCE loaded.Application Name:
NIL

<->

(ASK STUDENTS REGISTER S123521)

Name?

SMITH

Course Number?

95.101

OK

Course Number?

95.104

OK

Course Number?

95.106

OK

Course Number?

NIL

(S123521 SMITH REGISTERED)

<->

(ASK STUDENTS TRACE MESSAGES)

(STUDENTS REPLIES: (TRACE ON))

(TRACE ON)

<->

(ASK STUDENTS REGISTER S132561)

(STUDENTS RECEIVES MESSAGE: (REGISTER S132561))

(STUDENTS REPLIES: (S132561))

Name?

SMITH

(S132561 RECEIVES MESSAGE: (SET YOUR NAME TO SMITH))

(S132561 REPLIES: (SMITH))

Course Number?

95.104

(95.104 RECEIVES MESSAGE: (ANY ROOM?))

(95.104 RECEIVES MESSAGE: (RECALL YOUR STUDENTS))

(95.104 REPLIES: ((S123521)))

(95.104 RECEIVES MESSAGE: (RECALL YOUR QUOTA))

(95.104 REPLIES: ((10)))

(95.104 REPLIES: (YES))

(95.104 RECEIVES MESSAGE:
(ADD S132561 TO YOUR LIST OF STUDENTS))
(95.104 REPLIES: (S132561))
(S132561 RECEIVES MESSAGE:
(ADD 95.104 TO YOUR LIST OF COURSES))
(S132561 REPLIES: (95.104))
OK
Course Number?
95.102

(95.102 RECEIVES MESSAGE: (ANY ROOM?))
(95.102 RECEIVES MESSAGE: (RECALL YOUR STUDENTS))
(95.102 REPLIES: (NIL))
(95.102 RECEIVES MESSAGE: (RECALL YOUR QUOTA))
(95.102 REPLIES: ((30)))
(95.102 REPLIES: (YES))
(95.102 RECEIVES MESSAGE:
(ADD S132561 TO YOUR LIST OF STUDENTS))
(95.102 REPLIES: (S132561))
(S132561 RECEIVES MESSAGE:
(ADD 95.102 TO YOUR LIST OF COURSES))
(S132561 REPLIES: (95.102))
OK
Course Number?
NIL

(S132561 RECEIVES MESSAGE: (RECALL YOUR NAME))
(S132561 REPLIES: ((SMITH)))
(STUDENTS REPLIES: (S132561 SMITH REGISTERED))
(S132561 SMITH REGISTERED)
<->
(ASK STUDENTS UNTRACE MESSAGES)

(STUDENTS RECEIVES MESSAGE: (UNTRACE MESSAGES))
(TRACE OFF)
<->
(ASK STUDENTS REGISTER S130665)

Name?
ZIMMER

Course Number?
95.101

OK
Course Number?
95.104

OK
Course Number?
NIL

(S130665 ZIMMER REGISTERED)

<->
(ASK STUDENTS REGISTER S32C125)

Name?
ZAP

Course Number?
95.104

OK
Course Number?
NIL

(S320125 ZAP REGISTERED)

<->
(ASK STUDENTS REGISTER S300001)

Name?
ALLEY

Course Number?
95.104

OK
Course Number?
NIL

(S300001 ALLEY REGISTERED)

<->
(ASK STUDENTS REGISTER S566437)

Name?
BRAND

Course Number?
95.104

OK
Course Number?
NIL

(S566437 BRAND REGISTERED)

<->
(ASK STUDENTS REGISTER S321882)

Name?
DOPE

Course Number?
95.104

OK

Course Number?
NIL

(S321882 DOPE REGISTERED)
<->
(ASK STUDENTS REGISTER S992118)

Name?
SCOTCH

Course Number?
95.104

OK
Course Number?
NIL

(S992118 SCOTCH REGISTERED)
<->
(ASK STUDENTS REGISTER S00291)

Name?
SMITH

Course Number?
95.104

OK
Course Number?
NIL

(S00291 SMITH REGISTERED)
<->
(ASK STUDENTS REGISTER S126652)

Name?
COLBE

Course Number?
95.104

OK
Course Number?
NIL

(S126652 COLBE REGISTERED)
<->
(ASK STUDENTS REGISTER S199002)

Name?
TOOLATE

Course Number?
95.104

Sorry, course 95.104 is full.
Course Number?
NIL

(NO REGISTRATION)
<-->
(ASK 95.104 HOW MANY STUDENTS?)

10
<-->
(ASK STUDENTS REGISTER S100999)

Name?
VERYLATE

Course Number?
95.104

Sorry, course 95.104 is full.
Course Number?
95.102

OK
Course Number?
NIL

(S100999 VERYLATE REGISTERED)
<-->
(ASK 95.104 GIVE ME A LIST OF STUDENTS)

S123521	SMITH
S132561	SMITH
S130665	ZIMMER
S320125	ZAP
S300001	ALLEY
S566437	BRAND
S321882	DUPE
S992118	SCUTCH
S00291	SMITH
S126652	CULBE

End of Student List

<-->
(END)

Do you wish to save some applications?

YES

Application Name:
COMPUTER_SCIENCE

COMPUTER_SCIENCE - DP#ACAD03/COMPUTER_SCIENCE.PA088542
Application Name:
NIL

(OASYS TERMINATED - - YOU ARE NOW IN CP6 LISP - - THANK YOU)

(IBEX "DONT DRIBBLE")
DRIBBLE OFF a 10:44 06/03/82

BIBLIOGRAPHY

- [1] Patrick Henry Winston, and Berthold Klaus Paul Horn; LISP; Addison-Wesley Publishing Company, Inc. 1981.
- [2] Patrick Henry Winston; ARTIFICIAL INTELLIGENCE; Addison-Wesley Publishing Company Inc. 1979.
- [3] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermot; ARTIFICIAL INTELLIGENCE PROGRAMMING; Lawrence Erlbaum Associates Inc. 1980.
- [4] J.C. Pyle; THE ADA PROGRAMMING LANGUAGE; Prentice-Hall International Inc. 1981.
- [5] Kenneth M. Kahn; DIRECTOR GUIDE; Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1979.
- [6] Dave McArthur; THE ROSS LANGUAGE USER'S MANUAL; Rand Corporation Santa Monica, 1981.
- [7] TACCDAS - TACTICAL ARMY COMMAND AND CONTROL DATA SYSTEM; National Defence Headquarters, 1978.
- [8] Philip A. Bernstein, James B. Rothnie, and David W. Shipman; DISTRIBUTED DATABASE MANAGEMENT; IEEE Catalog No EHO 141-2.
- [9] Carl Hewitt; VIEWING CONTROL STRUCTURES AS PATTERNS OF PASSING MESSAGES; Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1976.

BIBLIOGRAPHY

- [10] Carl Hewitt, and Henry Baker; ACTORS AND CONTINUOUS FUNCTIONAL
Massachusetts Institute of Technology, Artificial Intelligence
Laboratory, 1977.
- [11] The Xerox Learning Research Group; THE SMALLTALK-80 SYSTEM;
BYTE Publications Inc, August 1981.
- [12] Adele Goldberg; INTRODUCING THE SMALLTALK-80 SYSTEM; BYTE
Publications Inc, August 1981.
- [13] Richard O. Duda; KNOWLEDGE-BASED EXPERT SYSTEMS COME OF AGE;
BYTE Publications Inc, Sept 1981.
- [14] Perry W. Inorndyke, Dave McArthur, and Stephanie Cammarata;
AUTOPILOT - A DISTRIBUTED PLANNER FOR AIR FLEET CONTROL; The
Proceedings of the Seventh International Joint Conference
of Artificial Intelligence (IJCAI), August 1981.
- [15] David McArthur, and Henry Sowizral; AN OBJECT-ORIENTED LANGUAGE
FOR CONSTRUCTING SIMULATIONS; The Proceedings of the Seventh
IJCAI, August 1981.
- [16] Luc Steels; REASONING MODELED AS A SOCIETY OF COMMUNICATING
EXPERTS; Massachusetts Institute of Technology, Artificial
Intelligence Laboratory, June 1979.
- [17] Mats Nordstrom; CP-6 LISP USERS GUIDE; Uppsala Sweden, June
1978. (Revised May 1980 by Nick Briggs and Andrew Gullen;
Academic Support Group, Carleton University).

BIBLIOGRAPHY

- [18] Anders Haraldsson; LISP-DETAILS INTERLISP/360-370; Uppsala University, 1975, Second Edition 1978.
- [19] International Organization for Standardization; OPEN SYSTEMS INTERCONNECTION; August 1979.

END

05109184

FIN