INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 1336703

Hierarchical modelling of mobile, seeing robots

Luh, Cheng-Jye, M.S.

The University of Arizona, 1989

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

HIERARCHICAL MODELLING OF MOBILE, SEEING ROBOTS

by

Cheng-Jye Luh

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL ENGINEERING

In the Graduate College
The University of Arizona

1 9 8 9

STATEMENT OF AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Chengzhi Lu

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

B. P. Zeigler
Bernard P. Zeigler
Professor of Electrical
and Computer Engineering

May 22 / 89
Date

ACKNOWLEDGMENTS

I would like to express my gratitude to Dr. Bernard P. Zeigler, my advisor and chairman of the Committee, for his guidance and support throughout this work.

TABLE OF CONTENTS

	<u>Page</u>
LIST OF FIGURES	6
ABSTRACT	7
 CHAPTER 1 INTRODUCTION	 8
1.1 Objectives and Approaches	8
1.2 Overview of the Thesis	10
CHAPTER 2 DEVS-SCHEME SIMULATION ENVIRONMENT	12
2.1 Brief Overview of DEVS-Scheme	12
2.2 Implementation of Robot Models in DEVS-Scheme	14
2.2.1 Controlled-models	15
2.2.2 Rule Based Modelling: Forward-models	15
CHAPTER 3 SYSTEM ENTITY STRUCTURE KNOWLEDGE REPRESENTATION	 22
3.1 Inheritance	22
3.2 Specialization	29
3.3 Multiple Entities	30
CHAPTER 4 MULTILEVEL HIERARCHICAL ROBOT MODELS	34
4.1 Robot Models	34
4.1.1 SPACE-MANAGER	37
4.1.2 Robot's Brain	38
4.2 Robot-Managed Laboratory Model	40
CHAPTER 5 ROBOT VISION	43
5.1 CESM-like Expert System	45
5.2 Image-Attribute Generator	49

	5
5.3 Visual Data Acquisition & Identification Algorithm	52
CHAPTER 6 SIMULATION EXPERIMENTS	55
CHAPTER 7 CONCLUSIONS	62
APPENDIX 1 RULE BASE	64
APPENDIX 2 VISUAL MPU	67
REFERENCES	74

LIST OF FIGURES

Figure 1	DEVS-Scheme Class Specification Hierarchy	13
Figure 2	Structure of an Activity	17
Figure 3	Forward-model Specification of General MPU	24
Figure 4	Forward-model Specification of Generic TASK MPU	25
Figure 5	System Entity Structure for ROBOT-SYSTEM	35
Figure 6	Minimal Robot System Configuration	36
Figure 7	System Entity Structure for SPACE-STATION LAB	42
Figure 8	Basic Vision Functions	44
Figure 9	Class Definitions of Rules and Parameters	46
Figure 10	Inferencing Interface	47
Figure 11	Image Attributes Data Base	51
Figure 12	Image Evidence Generation Formula	51
Figure 13	Visual Data Acquisition and Identification Algorithm	54
Figure 14a	Laboratory Objects' Graphical Representations and Image Attributes	57
Figure 14b	Laboratory Objects' Image Generators	58
Figure 15	Spatial Relationships between the Seeing Robot and the Object	59
Figure 16	System Entity Structure for Expert-System	63

ABSTRACT

This thesis describes the implementation of a hierarchical robot simulation environment which supports the design of robots with vision and mobility. A seeing robot model applies a classification expert system for visual identification of laboratory objects. The visual data acquisition algorithm used by the robot vision system has been developed to exploit multiple viewing distances and perspectives. Several different simulations have been run testing the visual logic in a laboratory environment. Much work remains to integrate the vision system with the rest of the robot system.

CHAPTER 1

INTRODUCTION

Today's robots are applied in many different areas in the workplace. They are used to increase productivity, reduce costs, improve product quality, overcome skilled labor shortage and to free humans from boring, repetitive tasks or hazardous, unhealthy environments. However, such industrial robots are not intelligent. They carry out predefined routine procedures over and over again. All they "know" is how to accept external commands, to retrieve the sequence of movements corresponding to the external command and play it back at the right time. Although they work with exact precision, they are incapable of flexibly responding to an unknown or variable environment. To achieve greater autonomy, a robot must be capable of sensing its surroundings and possess enough "intelligence" to respond to a variable environment much the same as humans do. Much research is being done to develop a new generation of robots that will improve in all respects. Such intelligent robots will eventually incorporate elements of thinking, vision, sense, mobility and speech recognition[14], all of which require the direct application of artificial intelligence.

1.1 Objectives and Approaches

In this work, a combination of discrete event simulation and AI knowledge representation schemes, called knowledge-based simulation[7,9] has been employed to augment the intelligent robot models being developed in our research*[21] with vision and mobility. Such robots are intended to work as laboratory assistants in space-borne laboratory environments. In designing the robot models, we assume that necessary low level motion, manipulative and sensory capabilities exist so we

* Supported by NASA-Ames Co-operative Agreement No. NCC 2-525, "A Simulation Environment for Laboratory Management by Robot Organizations"

can focus on higher level task-related cognitive requirements.

The robot model and laboratory environment are being constructed on the basis of object-oriented and hierarchical models of robots and laboratory components at multiple levels of abstraction. The robot organization is first represented in the system entity structure knowledge representation[10,11,12,13]. After pruning, the structure is transformed into a hierarchical model containing controlled-models at two levels. Controlled-models is a class of DEVS-Scheme[16,17,22] which realizes a centralized control structure. A controlled-model contains an arbitrary number of components which communicate with each other and with the outside world via a controller[19]. At the top level, the ROBOT-SYSTEM is a controlled-model containing a SPACE-MANAGER as controller, and ROBOTS as components. The SPACE-MANAGER is a modelling device to conveniently represent knowledge of where robots are and whom they can communicate with. Each ROBOT contains a MOTION, a SENSORY, and a COGNITION subsystem. The COGNITION SYSTEM is itself a controlled-model containing a SELECTOR as controller, and MPUS (Model-Plan Units) as components. The MPUS are task specialists which can be designed by employing models of intended tasks and plans of actions based on such models. A mobile, seeing robot must at least contain VISUAL and NAVIG MPUS for vision and mobility, respectively.

To implement the robot's vision system, we have incorporated a classification system based on CESM (Classification Expert System Maker)[18,23]. The vision system can identify the laboratory objects on the basis of physical attributes (such as size, shape and transparency), functionality (its probable use as a container, transformer, etc.), and location (e.g., storage area or work area). Since what a robot sees will depend on it's viewing perspective and distance, a seeing robot will

move itself to try different perspectives and distances, until it can identify an object or decide to give up. Helped by its vision system, a robot model can interact with laboratory objects knowing where and what they are. It can travel to an object's site and perform tasks controlled by its other MPUs.

Throughout this work, the author has been extensively involved in the implementation of the hierarchical robot simulation environment, especially in the design and development of the specialized MPUs and the communication and coordination protocols among MPUs and robots. The most significant contribution of this work is in the design of the intelligent robot models with vision and mobility.

1.2 Overview of the Thesis

Chapter 2 and 3 provide basic and advanced concepts and techniques of DEVS-Scheme and the system entity structure knowledge representation scheme. We begin in Chapter 2 by providing a brief overview of DEVS-Scheme, then give more details on controlled-models and forward-models, which are the major modelling concepts and techniques employed in the implementation of robot models. Chapter 3 contains advanced concepts and techniques of the system entity structure knowledge representation. We first discuss the inheritance property in DEVS-Scheme, especially in the class of forward-models. We then present how specialization in the system entity structure is transformed into inheritance discussed above. Finally, we show how specialization of a multiple entity works to generate any number of desired alternatives. Examples taken from the implemented robot models show the real implementation in part.

Chapter 4 discusses the implementation of multilevel hierarchical robot models. We first implement a robot organization, using concepts and techniques represented

in Chapter 2 and 3. Then we extend the robot model into a robot-managed laboratory model in which robot visual logic is being tested. Both the robot model and laboratory model are constructed on the basis of object-oriented and hierarchical models of robot and laboratory components at multiple levels of abstraction.

Chapter 5 deals with the implementation of robot vision. A CESM-like expert system and an image-attribute generator have been developed for robot visual identification of laboratory objects. The visual data acquisition algorithm used by the robot vision system has been developed to exploit multiple viewing distances and perspectives.

Chapter 6 illustrates the simulation experiments we have done in a simple laboratory environment. Finally, Chapter 7 discusses the problems we have encountered, and some directions of our future work.

CHAPTER 2

DEVS-SCHEME SIMULATION ENVIRONMENT

DEVS-Scheme is a general purpose environment for constructing hierarchical discrete event models implemented in PC-Scheme which runs on DOS compatible microcomputers and under a Scheme interpreter for the Texas Instrument Explorer system. The environment realizes the DEVS formalism[3,15,16,20], a theoretically well-grounded means of expressing hierarchical, modular discrete event simulation models. Moreover, DEVS-Scheme is implemented as a shell that sits upon PC-Scheme in such a way that all the underlying Lisp-based and object-oriented programming language features are available to the users. The result is a powerful basis for combining AI and simulation techniques.

2.1 Brief Overview of DEVS-Scheme

DEVS-Scheme is principally coded in SCOOPS, the object-oriented superset of PC-Scheme. The class specification hierarchy is illustrated in figure 1.

All classes in DEVS-Scheme are subclasses of the universal class *entities* which provides tools for manipulating objects in these classes. The inheritance mechanism ensures that such general facilities need only be defined once and for all. *Models* and *processors*, the main subclasses of entities, provide the basic constructs needed for modelling and simulation. Models is further specified into the major classes *atomic-models* and *coupled-models*, which in turn are specialized into more specific classes. Class processors, on the other hand, has three specializations: *simulators*, *co-ordinators*, and *root-co-ordinators* which serve to handle all the simulation needs.

The class *atomic-models* realizes the atomic-level of the DEVS model formalism. It has variables corresponding to each of the parts of this formalism. For example, *atomic-models* has instance variables *int-transfn*, *ext-transfn*, *outputfn*,

Figure 1. DEVS-Scheme Class Specification Hierarchy.

and time-advancefn which specify a model's internal transition function, external transition function, output function and time-advance function, respectively. Forward-models, a subclass of atomic-models generates models which inherit all the instance variables and methods of atomic-models. However, forward-models is based on the forward-chaining paradigm for writing models in a rule-based manner. Greater details will be given later.

The class coupled-models is a major class which embodies the hierarchical model composition constructs of the DEVS formalism. In the DEVS formalism, a coupled-model is defined by specifying its components and the coupling relations which establish the desired communication links.

Digraph-models and kernel-models, the subclasses of coupled-models, enable specialization of coupled-models in a specific way. Digraph-models provide a means of specifying coupled-models which are composed of a finite set of explicitly given components with explicitly specified coupling. On the other hand, kernel-models provides a means of specifying coupled-models containing arbitrary numbers of components coupled in a uniform manner. Different specialized classes of kernel-models realize different internal and external coupling schemes. Four coupling schemes are supported: broadcast, hypercube, cellular, and centrally co-ordinated. Controlled-models, which realizes the centrally co-ordinated structure, is discussed in the next section. Greater detail on DEVS-Scheme is in [16,17,22].

2.2 Implementation of robot models in DEVS-Scheme

To demonstrate how the DEVS-Scheme environment supports the design of multilevel robot models, we discuss the class controlled-models as well as a new class, forward-models which are the major modelling concepts employed in the implementation of the robot models.

2.2.1 Controlled-models

Controlled-models is a subclass of kernel-models which embodies a centralized organization in a dynamic fashion. A controlled-model contains an arbitrary number of components which communicate with each other and with the outside world via a controller. The controller, an atomic-model, must have a state variable called *influencees* whose current value determines to which of the existing members its output will be sent. Thus, the internal coupling from the controller to members is variable. The internal coupling in the reverse direction is fixed; each member can send output only to the controller. The external coupling is also fixed; only the controller is coupled to the overall model.

The implemented robot organization is modeled in controlled-models at two levels. At the top level, motion and communication of robots is easily modeled in such a way. The controller is a SPACE-MANAGER which keeps track of locations and directions of robots as the robots move around. A message sent by a robot is routed by the SPACE-MANAGER to those robots that are located within range (determined by channel type) of the sender. The robot's cognition-system is also a controlled-model containing a SELECTOR as controller and MPUS as components.

The SELECTOR controls the number of sensory inputs the cognition-system must attend at one time depending on its state. In the open state, the SELECTOR broadcasts incoming sensory inputs to all MPUs. On the other hand, the SELECTOR passes on the incoming inputs only to the activated MPU in its closed state. The MPUS are developed as objects in the class forward-models of DEVS-Scheme, now to be discussed.

2.2.2 Rule Based Modelling: Forward-models

Forward-models, a specialized class of atomic-models, generates model objects

which inherit all the instance variables and methods of atomic-models. In addition, forward-models contains a forward-chaining inference engine that facilitates writing models in a rule based manner. As shown in Figure 2, a rule, called an activity, is defined by:

(define-structure activity condition action before-output after-output)

In addition to condition and action slots, such structure contains slots for specifying outputs to be produced before and/or after the action is performed. Rules for specifying internal and external transitions have the same format. In general, internal transition rule conditions test the phase and other state variables of the model while external transition rule conditions test more about the input content structure and elapsed time. Once its condition is true, the triggered rule becomes a candidate to execute its action and then to change the state of the model.

As an example, consider some of the rules for the VISUAL MPU first informally presented:

external activity:

- R1. if phase is wait-for-image
and receive x on port image
then record value of x as visual image
and hold in phase backward-chaining for 1 unit

internal activities:

- R2. if phase is backward-chaining
then record backward-chaining result
and hold in phase analyse-image for 1 unit
- R3. if phase is analyse-image
and the backward-chaining result is true
then send to port 'object-identify
and hold in phase send-analyse-result for 1 unit
and send to port 'finished
- R4. if phase is analyse-image
then hold in phase forward-chaining for 0 unit

Figure 2. Structure of an Activity.

R5. if phase is send-analyse-result
 then passivate

Rule R1 is an external activity which activates the model when an external event arrives on port 'image while the model is in phase 'wait-for-image. In DEVS-Scheme, this kind of external activity is provided to the model by use of the forward-models method *add-ext-activities*:

```
(send visual add-ext-activities
  (list
    ;;R1:
    (make-activity
      'condition '(and
        (equal? (content-port x) 'image)
        (equal? (state-phase s) 'wait-for-image)
      )
      'action '(begin
        (set! (state-image s) (content-value x))
        (hold-in 'backward-chaining 1)
      )
    )
  )
)
```

Note that the values specified in condition and action slots are quoted text fragments since they should only be evaluated under the appropriate circumstances: the condition slot when the condition is being tested and the action slot when the action is being executed. No output is specified in activity R1. If there were some, they are also quoted text fragments. Note that expression forms already known for writing atomic-models can be used.

Rules R2, R3, R4, and R5 are internal activities following on from rule R1. Rule R2 calls upon the backward-chaining process, and then places the model in phase 'analyse-image. Rules R3 and R4 provide alternative courses of action that follow depending on the backward-chaining result returned in R2. R3 claims that the VISUAL has confirmed the identification hypothesis. R4 bypasses such identi-

fication claim and immediately calls forward-chaining. R5 dictates what happens once the model finished its announcement. DEVS versions of these activities are sent to the model with the method *add-int-activities*:

```
(send visual add-int-activities
  (list
    ;;;R2:
      (make-activity
        'condition '(equal? (state-phase s) 'backward-chaining)
        'action '(begin
          (set! (state-result s)
            (bc-exp (state-image s) (state-object-data s)))
          (hold-in 'analyse-image 1)
        )
      )
    ;;;R3:
      (make-activity
        'condition '(and
          (equal? (state-phase s) 'analyse-image)
          (equal? (state-result s) #t)
        )
        'before-output '(make-content 'port 'object-identify)
        'action '(hold-in 'send-analyse-result 1)
        'after-output '(make-content 'port 'finished)
      )
    ;;;R4:
      (make-activity
        'condition '(equal? (state-phase s) 'analyse-image)
        'action '(hold-in 'forward-chaining 0)
      )
    ;;;R5:
      (make-activity
        'condition '(equal? (state-phase s) 'send-analyse-result)
        'action '(passivate)
      )
  )
)
```

Rule R3 provides an example where both before- and after- outputs are specified. The before-output is generated just before the action is evaluated while the after-output is generated at the end of the interval specified by the hold-in primitive. Graphically,


```

;;; otherwise call forward chaining
  (make-activity
    'condition '(equal? (state-phase s) 'analyse-image)
    'action '(hold-in 'forward-chaining 0)
  )
)

```

These rules come into play after backward chaining. Depending on the backward chaining result, VISUAL may recognize the object, try a different viewing distance or call forward chaining.

One advantage of employing rules is apparent in the above examples. Whether internal or external, activities that are closely associated can be placed contiguously. This avoids breaking sequences of external and internal transitions apart and thus aids model comprehension. A second benefit: since outputs may be specified within the rules, the output specification is not separated from the transition specification as in atomic-models. However, such benefits of rule-based model specification are achieved at some cost in execution time and necessary user care. Since the inference engine evaluates the rules in a head-to-tail order, the closer to the end of the external and/or internal activities list the fired rule is, the more execution time it takes. Moreover, any errors in the quoted text fragments specified in the slots of a rule can't be detected until the rule is evaluated. Therefore, much user care is needed for successful rule-based modelling and simulation.

CHAPTER 3

SYSTEM ENTITY STRUCTURE KNOWLEDGE REPRESENTATION

The system entity structure is a knowledge representation scheme for managing the family of hierarchical models. A system entity structure incorporates decomposition, taxonomy and coupling knowledge concerning a domain of real world[12,13]. A user prunes the system entity structure according to the objectives of the modelling study. This results in a so-called pruned entity structure. Upon invoking the transform procedure, the system searches the model base for components specified in the pruned entity structure and synthesizes the desired model by coupling them together in a hierarchical manner. The result is a simulation model expressed in DEVS-Scheme that is ready for simulation run.

3.1 Inheritance

In the DEVS formalism, the smallest meaningful chunks of a model are the basic functions: internal transition, external transition and output functions. Atomic-models can share such functions but they can't share smaller parts of them. The ability to write models using rules provides an additional level of decomposition of model specification. Rules can be copied from one model and added to those of another. However, to define a meaningful model structure, rules inherited from a parent class must be combinable with more specialized rules provided by the child class.

In a system entity structure, a specialized forward-model inherits rules from a more general forward-model which are placed at the tails of external and internal activities lists, respectively. Since the inference engine underlying forward-models scans the external and internal activities list in head-to-tail order, placing inherited

rules at the tail of respective lists achieves the desired priority ordering.

The method *inherit-from* of forward-models carries out the juxtaposition of rules. Let *m* and *m'* both be forward-models. The message:

(send *m* inherit-from *m'*)

- extends the state-variables of *m* to include any new ones employed by *m'* (the method *def-state* will do this nicely)
- adds the internal-activities of *m'* to the tail of the internal-activities list of *m*
- adds the external-activities of *m'* to the tail of the external-activities list of *m*.

For example, in the robot's brain, the generic MPU (Figure 3) has a rule for self-identification and a default rule for normal continuation when no other rule applies for an external event.

The mpu which specializes in visual identification of objects has additional rules for the visual data acquisition and identification processes. To construct a vision system which also has the ability to identify itself, we use:

(send visual inherit-from mpu).

The two external activities of MPU will be appended at the end of the list of existing external activities. To make sure the identification protocol can work, a check for conflicts between its condition and those of the existing visual identification protocol should be done carefully. The default continuation rule always has its condition satisfied, however, being at the tail of the list, it can only fire when none of the other conditions holds. Thus, the inheritance mechanism ensures that all specialized MPUs, such as the Assistance-Requestor, Navigator, etc., also have the ability to identify themselves. In the same way, we have designed a generic TASK MPU (Figure 4) that has rules to request help, to call the Navigator, to apply

```

(make-pair forward-models 'mpu)

(send mpu def-state '(name))

;;;;;;;;identification protocol

(send mpu set-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'passive)
                        (equal? (content-port x) 'request-id))
      'before-output '(make-content 'port 'respond 'source
                                    (state-name s))
      'action '(passivate)
    )
  )
)

;;;;;;;;default continuation

(send mpu add-ext-activities
  (list
    (make-activity
      'condition '#t
      'action '(continue)
    )
  )
)

```

Figure 3. Forward-model Specification of General MPU.

```

;;; GENERIC TASK ;;;;;;

(make-pair forward-models 'task)

(send task def-state '(
    ;; call visual
    position
    direction
    object-class
    object-name
    object-position
    ;; call navig
    navig-destination
    echo-status
))

(send task set-s (make-state
    'interpreter-phase 'test-condition
    'ext-activities '()
    'int-activities '()
    'before-output-buffer '()
    'after-output-buffer '()
    'action-buffer '()
    'sigma 'inf
    'phase 'passive
    'name 'task
    )
)

;;;;;;;;identification protocol

(send task set-ext-activities
    (list
        (make-activity
            'condition '(equal? (content-port x) 'request-id)
            'before-output '(make-content 'port 'respond
                'source (state-name s)) ;;;
            'action '(continue)
        )
    )
)

;;;;;;;; request-help protocol ;;;;;;

;;;send request for help to assis

(send task add-int-activities
    (list
        (make-activity
            'condition '(equal? (state-phase s) 'send-need-help)
            'before-output '(make-content 'channel 'ready-help
                ;;;to simulate external input
                'port 'in)
            'action '(hold-in 'send-start-help 1)
        )
        (make-activity

```

```

'condition '(equal? (state-phase s) 'send-start-help)
'before-output '(make-content 'channel 'start-help
                             'port 'in)
'action '(passivate)
)
)
)

;;;;; help-offered & resume working, called by ASSIS or OFFER

(send task add-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'passive)
                       (equal? (content-port x) 'start-working))
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(hold-in 'resume-working 1)
    )
  )
)

;;;;;;; vision protocol ;;;;;;

;;; call visual

(send task add-int-activities
  (list
    (make-activity
      'condition '(equal? (state-phase s) 'send-start-visual)
      'before-output '(make-content 'port 'in
                                   'value (list (state-position s)
                                                (state-direction s))
                                   'channel 'start-visualization)
      'action '(passivate-in 'wait-for-object-data)
    )
  )
)

;;; object-identify result

(send task add-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'wait-for-object-data)
                       (equal? (content-port x) 'object-identify)
                       (equal? (state-object-class s)
                              (car (content-value x))))
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(begin
                (set! (state-object-position s)
                      (cadr (content-value x)))
                (set! (state-object-name s) (caddr (content-value x)))
                (hold-in 'object-expected 0)
              )
    )
  )
)

```

```

    )
    (make-activity
      'condition '(and (equal? (state-phase s) 'wait-for-object-data)
                       (equal? (content-port x) 'object-identify)
                       )
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(hold-in 'no-object-expected 1)
    )
  )
)

;;;;;;;; navigation protocol ;;;;;;;;;

;;;; call navig

(send task add-int-activities
  (list
    (make-activity
      'condition '(equal? (state-phase s) 'send-start-navig)
      'before-output '(make-content 'port 'in
                                   'value (state-navig-destination s)
                                   'echo-status (state-echo-status s)
                                   'channel 'start-navigation)
      'action '(passivate-in 'moving)
    )
  )
)

;;; arrived

(send task add-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'moving)
                       (equal? (content-port x) 'arrived)
                       )
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(hold-in 'arrived 0)
    )
  )
)

;;;;;;;; finished ;;;;

(send task add-int-activities
  (list
    (make-activity
      'condition '(equal? (state-phase s) 'finished)
      'action '(passivate)
    )
  )
)

```

```
;;;;; default ;;;;;;
(send task add-ext-activities
  (list
    (make-activity
      'condition '#t
      'action '(continue)
    )
  )
)
```

Figure 4. Forward-model Specification of Generic TASK.

vision, etc., under specific conditions. Thus, in addition to performing a particular task, each task specialist also has the ability to communicate and cooperate with other MPUs. Greater detail on forward-models is given in [22].

3.2 Specialization

In the system entity structure, the specialization relation provides a means of conveniently expressing alternatives for the model structure. Having pruned an entity structure and removed all specializations, the transform operator calls upon inheritance to synthesize models corresponding to the chosen specialized entities.

To show how specialization is transformed into inheritance, consider the following SES fragment:

In pruning, we select a specialized entity, S or T, to replace the general entity G. If S is chosen for example, the result will be the pruned entity structure:

S.G

i.e., a single specialized entity which may be called “S specialization of G”. The order, in which the specialization choice comes first, is natural.

The specialized entity S.G is transformed using the following rules:

- transform S into a model of the same name
- (rule 1) if both S and G are instances of forward-models (or its subclasses) then
 - transform G into a model of the same name

- the transform of S_G is the result of telling S to inherit-from G.
- (rule 2) otherwise, the transform of S_G is S.

The first rule expresses that a specialized entity S_G is synthesized through inheriting the structure of the general entity G to the specialized choice S if both S and G are forward-models. The second rule bypasses such inheritance and returns S as the result of transforming S_G.

For a concrete example, consider the SES for MPU shown as below:

Pruning might then result in the specialized entities, ASSIS_MPU (assistance requesting MPU), OFFER_MPU (assistance offering MPU), etc. The transform of ASSIS_MPU, for example, would load the files assis.m and mpu.m from the model base, setting up forward-models ASSIS and MPU in working memory. The forward-model ASSIS_MPU is then constructed and told to inherit from ASSIS and then MPU. The result is the juxtaposition of the protocol for assistance requesting with those for self-identification and normal continuation as discussed above.

3.3 Multiple Entities

In the robot models, the MPUS are defined as a multiple entity in the system entity structure. In DEVS-Scheme, specialization of such a multiple entity offers a powerful means to generate any number of desired alternatives. That is, pruning results not just in the inclusion of one, but of many, selected components in the

model. As an example, consider the entity structure for the robot's COGNITION-SYSTEM:

In general, this entity structure is transformed into a controlled-model or a digraph-model containing a kernel-model (other than controlled-model) corresponding to the multiple entity MPUS. If the user selects a specialized MPU in pruning, then components of the kernel-model are isomorphic copies of that MPU, the kernel, which would be generated under the user's control at simulation time (i.e., during the restart process).

Another possibility is now evident. When pruning a multiple entity such as MPUS, the subtree below the single entity, MPU, can be placed into the kernel-model and used, at simulation time, to generate any number of desired alternatives for the kernel. Consider for example, pruning e:cognition-system. During such pruning, we choose controlled-models as the desired subclass of kernel-models, as before. However, when pruning arrives at the single entity, MPU, we cut the entity structure at this point. This results in two actions:

- the pruned entity structure, p:cognition-system, will have the subtree below MPU removed:

- a new entity structure, e:mpu consisting of MPU and its substructure is created:

Now, when transforming p:cognition-system, the user can choose whether to have the entity structure e:mpu replace the model MPU as the originator in the pseudo class slot of the co-MPUS, the resulting kernel model. At simulation time, (during restart), the *init-new* method will check for the presence of an entity structure in the originator slot. If present, this entity structure, e:mpu, can be used to generate any number of any of the possible prunings of the structure. Thus, for e:mpu we can have any desired number of copies of ASSIS, OFFER, NAVIG, etc., as components for co-MPUS depending on the real application requirements. For example, a mobile, seeing robot can contain only one simple TASK, one VISUAL and one NAVIG MPU. In the same way, the SPACE-STATION LAB can

be decomposed into a SPACE-MANAGER and a multiple entity OBJECTS. Each OBJECT is specialized into ROBOT and EQUIP. Following the steps discussed above, we can generate any number of ROBOT and EQUIP in the laboratory.

CHAPTER 4

MULTILEVEL HIERARCHICAL ROBOT MODEL

In designing the robot models, we assume that necessary low level motion, manipulative and sensory capabilities exist so we can focus on higher level task-related cognitive requirements. The robot models and laboratory environment are being constructed on the basis of object-oriented and hierarchical models of robots and laboratory components at multiple levels of abstraction. In this chapter, we first focus on representing organizational aspects of the multi-robot system. This requires representing how robots can communicate and coordinate their activities. Then we construct a robot-managed laboratory model in which robots are intended to work as laboratory assistants.

4.1 Robot Models

A system entity structure for the ROBOT-SYSTEM is shown in Figure 5. After pruning, the pruned entity structure is transformed into a hierarchical model containing controlled-models at two levels (Figure 6). At the top level, the ROBOT-SYSTEM is a controlled-model containing a SPACE-MANAGER as controller, and ROBOTS as components. Each ROBOT is decomposed into MOTION-SYSTEM, SENSORY-SYSTEM, and COGNITION-SYSTEM. The MOTION-SYSTEM keeps track of ROBOT's motion information such as position, direction, speed, etc.. It accepts such motion commands as start & stop move, change-speed, etc. from the COGNITION-SYSTEM. When a robot changes its position, the MOTION-SYSTEM sends its new location and direction to the SPACE-MANAGER. A ROBOT receives and sends messages via its SENSORY-SYSTEM. Between the ROBOTS and the SPACE-MANAGER, messages are sent on channels. Among the ROBOT's subsystems, messages are sent on ports. The SENSORY-

Figure 5. System Entity Structure for ROBOT-SYSTEM.

Figure 6. Minimal Robot System Configuration.

SYSTEM must switch from channel to port and vice versa in receiving and sending messages, respectively. Thus, messages can be passed on to proper destination components. Messages on certain channels, such as touch, are reflected back to the SPACE-MANAGER by SENSORY-SYSTEMs upon receipt, as well as being transmitted to the COGNITION-SYSTEM. Such echo messages are used by the original sender to ascertain its spatial relationship to the receiving ROBOT.

The COGNITION-SYSTEM is itself a controlled-model consisting of a SELECTOR as controller, and MPUS as components. MPUs are task specialists which are designed by employing models of intended tasks and plans of action based on such models.

4.1.1 SPACE-MANAGER

The SPACE-MANAGER is the controller in the ROBOT-SYSTEM. It's a modelling device to conveniently represent knowledge of where robots are and whom they can communicate with. That is, motion and communication of ROBOTS are managed by the SPACE-MANAGER. When a ROBOT moves around, its MOTION-SYSTEM sends its new location and direction to the SPACE-MANAGER which keeps track of the ROBOTS' positions and directions. When a ROBOT wishes to communicate with other ROBOTS, it sends message via its SENSORY-SYSTEM to the SPACE-MANAGER which relays the message only to those ROBOTS within the range of the sender. The range is determined by the channel on which the message is sent. Thus different transmission media and sensory modalities could be modeled, such as light and vision, sound and hearing, pressure and touch, etc. To implement robot vision, the seeing ROBOT sends its "seeing" message via the LIGHT channel to the SPACE-MANAGER. Then the SPACE-MANAGER decides which OBJECTS (including other ROBOTS) are lo-

cated within the line of sight of the viewer. The OBJECT that is closest to the viewer is chosen as target. Since the SPACE-MANAGER has complete knowledge of locations and directions, it can easily compute the viewing distance and perspective, and attaches these two pieces of viewing data to the light message before routing it to the target. Upon receiving the light message, the target will reflect its image according to the viewing distance and perspective to the viewer. This implementation models human vision. What we see is our interpretation of light energy reflected from objects. More detail will be presented later.

Since the SPACE-MANAGER has complete knowledge of their locations, it can detect collisions between ROBOTS. Space is viewed as a kind of resource shared by its occupants so that collisions represent attempts to occupy the same space twice at the same time. The SPACE-MANAGER can report such an event but do nothing to prevent it. However, the SPACE-MANAGER may be given greater intelligence to co-ordinate the ROBOTS, for example to prevent collisions, and to do space resource management.

4.1.2 Robot's Brain

As indicated, each ROBOT's COGNITION-SYSTEM is also a controlled-model containing a SELECTOR as controller, and MPUS as components. The SELECTOR is essentially a bi-state device whose state is determined by the MPUs' responses. In the open state, all incoming sensory inputs are broadcast to MPUs. The first one to respond will become the activated MPU. Once a MPU activation has occurred, the SELECTOR is switched to the closed state in which it passes on the incoming sensory inputs to the activated MPU. The latter pays attention only to those inputs which matter to achieve its goals. Moreover, the "action-by-exception" control ensures that an MPU, once initiated, retains activation until

its plan is successfully executed, or until a significant discrepancy arises between the actual results of carrying out the plan and the results expected of its model. Thus, upon completion of the activation plans or receiving a discrepancy alert, the SELECTOR is switched back to the open state in which MPUs will compete for activation again. In such a way, we can minimize the sensory inputs the system must attend to at any one time.

The MPUs comprising the robot's brain are of two kinds: those specialized for carrying out specific tasks and those specialized for more general tasks involving communication, motion, vision, co-operation, etc. A prototype minimal configuration illustrated in Figure 6 contains two robots each with the following MPUs.

Task Specialist: MPU specialized for executing a particular task. When help or visual identification of object is needed in performing this task, it relinquishes control to the Assistance-Requestor or Visual, respectively.

Assistance-Requestor: MPU specialized for the task of requesting help from other robots. When it is activated, it initiates a protocol which tries to make contact with robots within its range and to engage one which can provide the needed assistance.

Assistance-Offeror: MPU specialized for the task of dealing with incoming requests for help emitted from Assistance-Requestors of other robots. When activated, it decides if help can be offered, and if so, engages in a dialogue with the Assistance-Requestor of the help-seeking robot and sets up a rendezvous. It relinquishes control to the Navigator to bring the robot to the requestor's work site.

Navigator: MPU specialized for directing the motion sub-system to bring the

robot to a given destination. It requests the current motion state from the motion component, and sends it new parameters (direction, speed, and time-step) for travelling to the vicinity of the destination. Once there, it directs the motion component in physically contacting the object or robot at the destination. The touch channel is used for judging when contact has been made.

Visual: MPU specialized for the task of visual identification of objects. It relinquishes control to Navigator to bring the robot to new locations when different viewing distances and perspectives are needed. Once there, it resumes control to accept a visual image and to identify objects by consulting its built-in classification system.

As shown in Figure 5, the MPUS are defined as a multiple entity in the system entity structure. As stated in Chapter 3, we can specialize such a multiple entity to generate any number of desired alternatives. Thus, we can have any desired copies of TASK, ASSIS, NAVIG, VISUAL, etc. as components for robot's brain depending on the real application requirements. For example, a mobile, seeing robot can contain only one simple TASK, one VISUAL, and one NAVIG MPU.

The MPUs are developed in the class of forward-models of DEVS-Scheme as discussed in Chapter 3.

4.2 Robot-Managed Laboratory Model

A system entity structure for the SPACE-STATION LAB is shown in Figure 7. After pruning, the pruned entity structure is transformed into a controlled-model containing a SPACE-MANAGER as controller, and OBJECTS as components. Each OBJECT is specialized into ROBOT and EQUIP. Each ROBOT is decomposed into MOTION-SYSTEM, SENSORY-SYSTEM, REFLECTOR-SYSTEM

and COGNITION-SYSTEM(BRAIN). The REFLECTOR-SYSTEM is employed to generate the OBJECT's own image upon receiving LIGHT signal. The EQUIP is a generic entity for laboratory equipment which is modeled much the same as the ROBOT. However, EQUIP has no BRAIN and its MOTION, SENSORY and REFLECTOR subsystems are always passive. Note that OBJECTS are also defined as a multiple entity, and with the pruning discussed earlier, we can have any desired number of ROBOT and EQUIP in the laboratory.

Figure 7. System Entity Structure for SPACE-STATION LAB.

CHAPTER 5

ROBOT VISION

Vision involves three basic functions: image transformation, image analysis, and image understanding[14] as shown in Figure 8. In a machine vision system, the light images sensed by the video camera are digitized into an array of levels. Each picture element, called pixel, represents the light intensity of the original scene at the corresponding region of the image. Image analysis involves various numerical operations. Noise reduction, edge detection, line finding, region splitting, etc. are applied to extract such image information as object edges, regions, boundaries, color and texture. For example, the edges of an object define its shape providing basic information needed to identify the object.

To implement robot vision, we assume the robot contains image processing mechanisms to perform the image transformation and analysis tasks, so we can focus on the image understanding task. In that stage, visual data has been interpreted into physical attributes, such as size and shape of the object to be identified.

As a basic MPU for robot visual identification of objects, VISUAL applies a CESM-like approach[23] to identify laboratory objects on the basis of physical attributes (such as size, shape, and transparency), functionality (such as container, transformer, etc.) and location (e.g., storage-area or work-area). Since the classification process is invoked whenever visual data is being recorded by the robot's eye (say a video camera), this is an on-line, non-interactive process. Thus rule-based inferencing procedures have been built to call backward chaining and/or forward chaining on the given visual data.

In general, what the vision system senses will depend on the orientation of its eye and the distance between it and the object to be identified. So the visual data

Figure 8. Basic Vision Functions.

acquisition algorithm used by the VISUAL will try possible viewing distances and perspectives to gather feasible evidence.

Meanwhile, the visual data come in as a list of image attributes. We need an image-attribute model to generate an initial image and to enlarge the image when the viewer comes closer.

5.1 CESM-like Expert System

CESM(Classification Expert System Maker) is an expert system shell which is designed for developing classification systems[23]. Like DEVS-Scheme, CESM is implemented in PC-Scheme with object-oriented features. All the rules and parameters in the knowledge base are instances of the classes rules and parameters, respectively. The class definitions of rules and parameters are illustrated in Figure 9. In addition to the necessary condition, action and certainty-factor instance variables, rules are interlinked to speed up the inference process. Two inference strategies are supported: backward chaining and forward chaining. The uncertainty management mechanism employs a modified version of Dempster-Shafer operators to provide well-defined evidence accumulation properties[18].

An inferencing interface (Figure 10) is designed to call backward chaining and/or forward chaining on a predefined rule base of laboratory objects. For example, a rule base for identification of bottles, faucets and funnels is listed in Appendix 1. The image attributes and objects are treated as parameters in this case. The inferencing interface also works as knowledge acquisition interface to pass incoming image attributes to the system's working memory. The procedure ASSIGN-CF performs this function. In addition, the procedure CLEAR-CF will clear the parameters' certainty factors for the next inference process.

The inferencing interface consists of two inferencing procedures: BC-EXP and

```

(define-class rules
  (instvars condition action text temp1 temp2
    influencee-classes influencer-classes triggerable-rules
    certainty-factor)
  (options inittable-variables settable-variables gettable-variables)
  (mixins entities)
)

(compile-class rules)
(mk-ent rules goal)

(define-class parameters
  (classvars
    (ef-ea-level '(.7 .3))
    (grain 100)
    (threshold .2))
  (instvars
    (demf demmod)
    (triple '(0 0 1))
    (picture '())
    (expression '()) ;;boolean expression using other parameters
    (function '())   ;;expression referring to defined variables, return boolean
    (need-ask #f)    ;;whether query this name
    (ask-user-first #f) ;; evaluate expression first before asking for value
    (updated-by '())
  )
  (options settable-variables gettable-variables inittable-variables)
  (mixins entities)
)

(compile-class parameters)

```

Figure 9. Class Definitions of Rules and Parameters.

```

(define (assign-cf attribute-list)
  (let (
    (param-list (getcv parameters lst))
  )
    (for-each (lambda(pair)
      (let (
        (param (car pair))
        (c-f (cadr pair))
      )
        (when (member (eval param) param-list)
          (if (>= c-f 0)
            (send (eval param) input-cef c-f)
            (send (eval param) input-cea (abs c-f))))
        ))
      attribute-list)
  )
)

(define (clear-cf)
  (let (
    (param-list (getcv parameters lst))
  )
    (for-each (lambda(param)
      (send (eval param) clear-triple))
      param-list)
  )
)

(define (backward-chain object)
  (let (
    (goal (symbol-append 'is- object))
  )
    (evaluate (eval goal))
    (list object (send (eval goal) get-ef?))
  )
)

(define (bc-exp attribute-list object)
  (clear-cf) ;; clear triple for next inference
  (assign-cf attribute-list)
  (backward-chain object)
)

(define (fc-exp attribute-list)
  (clear-cf)
  (assign-cf attribute-list)
  (forward-chain)
)

```

```

(define (forward-chain)
  (define (conflict-resolve lst)(car lst))
  (if (send goal test-condition) (send goal execute-action)
      (let* (
        (rules (send (%sc-class-env rules) get-lst))
        (conflict-set (clean (map (lambda(r)(send r
          test-condition)) rules)))
        )
        (for-each(lambda(rl)
          (writeln "rule: " (send rl get-name) "----fired----")
          (newline)
          (send rl execute-action)
          )
          conflict-set)
          (forward-chain)
          )))
  )

```

Figure 10. Inferencing Interface.

FC-EXP for backward chaining and forward chaining, respectively. Each procedure contains three steps: CLEAR-CF, ASSIGN-CF, and BACKWARD-CHAIN or FORWARD-CHAIN, except that BC-EXP requires one more argument, an object that is used as a goal in backward chaining.

Backward chaining tries to confirm a given hypothesis that the unknown object is in a particular class. Forward chaining tries to assign the given object to one of the leaf classes. Since the robot's next action depends on the inferencing result, we categorize the evidential status values into TRUE(confirm), UNKNOWN, and FALSE(disconfirm). The UNKNOWN value will drive the seeing robot to try different viewing distances or perspectives.

5.2 Image-attribute Generator

In our robot models, the vision system receives images by scanning the scene with a beam of light. The image comes from the object that reflects the light. Thus we have incorporated a REFLECTOR component in each laboratory EQUIP object to reflect light and to generate image attributes with certainty factors.

In general, what the vision system senses will depend on its location and viewing perspective. For example, when the robot is facing the object, or is close to it, it can receive clearer pictures. In terms of image attributes, they contain more positive evidence or less negative evidence. Moreover, at some distance, the robot can get a maximum evidence for one or more image attributes. When approaching the object further, the evidence will stay at maximum until the robot is very close to the object, in which case the evidence will decrease. Let's call the distance from the maximum evidence location to object "saturation distance" and the distance from the location where the evidence starts to decrease "blind distance". Graphically,

In some critical situation, the saturation distance and the blind distance may be the same. In simulation runs, the blind distance is set to 20 since we have set 10 as TOUCH range. The seeing robot is prohibited to move across this distance.

To represent this situation, we have developed a relational data base (defined by module RELATION) as shown in Figure 11 which stores an object's image features. Orientation and image-attributes are treated as data base attributes. The data associated with each image-attribute is a pair consisting of minimum and maximum evidence values. The minimum value E_{min} is the evidence the viewer can get when it is farthest away from the object in that orientation. The image data reflected by the REFLECTOR contains attribute, evidence pairs, e.g. ((orientation right) (transparent 0.8) (container 0.9)). Assuming the evidence increases linearly as the viewing distance decreases, we compute the image attribute's evidence by the formula listed in Figure 12. This formula is implemented in each OBJECT's REFLECTOR subsystem.

```

(define image-attr (relation))
(set-attributes image-attr
  '(orientation transparent conical-shaped container))
(assert-rel iamge-attr '(right (0.5 1) (-0.4 -1) (0.6 1)))
(assert-rel image-attr '(face (0.5 1) (-0.5 -1) (0.8 1)))
(assert-rel image-attr '(left (0.5 1) (-0.4 -1) (0.6 1)))

```

Figure 11. Image Attributes Data Base.

$$\begin{aligned}
 E_{cur} = E_{min} + (E_{max} - E_{min}) * \frac{D_{max} - D_{cur}}{D_{max} - D_{sat}} & \quad \text{if } E_{cur} < E_{max} \\
 \text{or } E_{max} & \quad \text{if } E_{cur} \geq E_{max}
 \end{aligned}$$

where E_{cur} = current evidence
 E_{max} = absolute maximum evidence
 E_{min} = absolute minimum evidence
 D_{max} = maximum viewing distance
 D_{cur} = current viewing distance
 D_{sat} = saturation distance

Figure 12. Image Evidence Generation Formula.

5.3 Visual Data Acquisition & Identification Algorithm

The vision system maintains a relational data base which is a world map of objects at their locations. When the vision system is invoked to start the visualization, it checks the world map first to see whether there are objects located within the line of sight of the seeing robot. If so, the object that is closest to the seeing robot is chosen as expected target, otherwise the vision system will inform the calling MPU that there is no object there.

The visual data identification process is based on the assumption that the world map is basically correct. Therefore, most of the time, all it needs is to confirm the object as expected. Thus, backward chaining is called first, and only if no conclusion can be reached or the expectation is disconfirmed, then forward chaining is invoked.

Whenever the image data come in, the vision system applies a predefined classification expert system for visual identification of laboratory objects. In the identification process, the vision system will call backward chaining first to verify whether the image attributes match those of the expected object. Then there are two cases in which the vision system will call forward chaining. First, if the hypothesis is disconfirmed, the vision system will call forward chaining to identify the object as different than expected; the subsequent inference processes are all forward chaining no matter what the robot's viewing distance and perspective is. Second, if the robot has tried to increase its certainty, and still can't confirm the identification hypothesis, it will give up backward chaining and then use forward chaining to identify the object. Either in backward chaining or in forward chaining, while the UNKNOWN result is reported, the robot will come closer or turn to face the object if such a movement is possible, otherwise it gives up the inference process

and calls the diagnostics. In general, the viewer can receive a more accurate image as it moves closer to the object. However, the seeing robot is prohibited to move across the “blind distance”.

Figure 13 shows the identification control strategy we have implemented in the VISUAL MPU forward-model. Note that a robot can attempt to increase its certainty by either moving closer or turning to face the object. If several such attempts fail to help, the identification process is abandoned.

Details of the design are included in Appendix 2.

Figure 13. Visual Data Acquisition and Identification Algorithm.

CHAPTER 6

SIMULATION EXPERIMENTS

To test the visual logic, we have run several simulation experiments of a small laboratory scenario. The simulation experiments concern three types of objects: bottle, funnel and faucet. Their graphical representations and image attributes are illustrated in Figure 14a. The objects' image generators are initially set up as shown in Figure 14b. Where the data listed under each image attribute is a pair containing a minimum and a maximum value. The positive values represent evidence for while negative values stand for evidence against that attribute. The minimum value is the evidence that a seeing robot can get when it is farthest away from the object in that orientation. The maximum viewing distance is set to 100 units in our simulation experiments. Note that the "transparent" attribute is common to bottle and funnel. Thus backward chaining on the bottle's image attributes will get the UNKNOWN result for the funnel identification hypothesis, and vice versa. In addition, positive "transparent" evidence will disconfirm the "faucet" identification hypothesis, since objects of class faucet are not commonly transparent. The simulation experiments are listed as follows:

- a) The vision system confirms the identification hypothesis in its first backward chaining attempt. In this case, we have located one bottle at position (0, 0), direction (0, 1) (It's an unit vector), and one seeing robot at position (0, 40), direction (0, -1) as shown in Figure 15a so that the seeing robot has faced the object and it is close enough to the latter to gather strong positive evidence for the "bottle" identification hypothesis.
- b) The vision system confirms the identification hypothesis in backward chaining after the seeing robot has moved closer to the object twice. Case b) is similar

to case a), except that the seeing robot is initially located at position (0, 80). According to our image-attributes generation and evidence accumulation approaches, the vision system will get the UNKNOWN result in backward chaining at that point. Thus the seeing robot will move closer to the object. Since we assume each close move distance is 20 units, the vision system can confirm the identification hypothesis after two close movements. This is just the same situation as that in case a).

- c) The vision system won't confirm the identification hypothesis in backward chaining until the seeing robot has turned to face the object. Assuming the bottle's direction is changed to (1, 0) in case b), so that the seeing robot is in the left orientation of the bottle (Figure 15c). Since the seeing robot is prohibited to move across the blind distance defined in section 5.2, three close movements are possible in this case. However, such close movements don't help to confirm the identification hypothesis. Therefore the seeing robot will turn to face the object. This results in two actions: 1) The seeing robot moves to the position that is 20 units distant from the object at the object's direction. In the case, it is position (20, 0). 2) The seeing robot rotates itself to face the object, that is, to change its direction to the opposite direction of the object, e.g. (-1, 0) in this case.
- d) The vision system gets the UNKNOWN results in possible backward chaining attempts, and then identifies the object in forward chaining. Given the "funnel" identification hypothesis in case c), even though the seeing robot has turned to face to object, the vision system still gets the UNKNOWN result in backward chaining, then it calls forward chaining to identify the object as a bottle.

OBJECT	GRAPHICAL REPRESENTATION	IMAGE ATTRIBUTES
BOTTLE		transparent cup-shaped container
FUNNEL		transparent conical-shaped
FAUCET		has-knob not transparent

Figure 14a. Laboratory Objects' Graphical Representations and Image Attributes.

object	orientation	transparent	cup-shaped	container	conical-shaped	has-knob
BOTTLE	face	(0.8 1)	(0.8 1)	(0.7 1)	N/A	N/A
	right	(0.8 1)	(0.7 1)	(0.6 1)	N/A	N/A
	left	(0.8 1)	(0.7 1)	(0.6 1)	N/A	N/A
FUNNEL	face	(0.8 1)	N/A	N/A	(0.8 1)	N/A
	right	(0.8 1)	N/A	N/A	(0.7 1)	N/A
	left	(0.8 1)	N/A	N/A	(0.7 1)	N/A
FAUCET	face	(-0.8 -1)	N/A	N/A	N/A	(0.8 1)
	right	(-0.8 -1)	N/A	N/A	N/A	(0.5 1)
	left	(-0.8 -1)	N/A	N/A	N/A	(0.5 1)

Figure 14b. Laboratory Object's Image Generators.

(a)

(b)

(c)

(d)

Figure 15. Spatial Relationships between the Seeing Robot and the Object

- e) The vision system disconfirms the identification hypothesis in backward chaining, and then identifies the object in forward chaining. Assuming that the spatial relations between the object and the seeing robot is the same as that in case c) and the object is replaced by a funnel. The vision system will disconfirm the given "faucet" identification hypothesis in its first backward chaining attempt. Then the vision system calls forward chaining instead, and the subsequent inference processes are all forward chaining no matter what the robot's viewing distance or perspective is. In this case, We have made a small change in the image attributes data base, so that the vision system identifies the object as a funnel in forward chaining after the seeing robot has turned to face the object.
- f) Possible evidence-gathering and inferencing attempts fail to help, the vision system gives up the identification process and calls diagnostics. This case is similar to case e). Let the seeing robot be initially located at position (0, 100) as shown in Figure 15f. When the UNKNOWN result is reported in backward chaining, the seeing robot will try to increase evidence by moving closer to and then turning to face the object. Note that after three closing movements the seeing robot will turn to face the object, since such movements don't increase the evidence much in that orientation. Let all possible backward chaining processes get UNKNOWN results, the last inference process will be forward chaining in the face orientation. If the result is still UNKNOWN, the vision system gives up the identification process and calls diagnostics.

CHAPTER 7

CONCLUSIONS

In several simulation experiments we have verified the visual logic. However, much work remains to integrate the VISUAL MPU into the ROBOT brain. For example, we must consider the case where the ROBOT must search for the OBJECT it expected to be at a certain location but wasn't. Scenarios in which a complete sequence of operations must be performed (and if a mishap occurs, recovered from) will be developed.

There is a problem with the consultation of CESM-like expert systems. Since the image attributes and objects are treated as parameters in our approach, when more than one robot consults the same classification system simultaneously, the evidences will interfere with each other. One of many alternatives is to generate a temporary copy of the classification system for each robot. Thus, many seeing robots can work properly at the same time. Following the same idea, we will try to develop a more generic inferencing interface that can call upon expert systems with different inference engines and knowledge bases combinations[2] as shown in Figure 16.

Figure 16. System Entity Structure for Expert-System.

APPENDIX 1. RULE BASE

```

(mk-ent parameters is-bottle)
(mk-ent parameters is-faucet)
(mk-ent parameters is-funnel)
(mk-ent parameters transparent)
(mk-ent parameters cup-shaped)
(mk-ent parameters conical-shaped)
(mk-ent parameters container)
(mk-ent parameters has-knob)

(send goal set-condition
  '(begin (ueval (or is-bottle is-faucet is-funnel))
    (def-var-val 'bottle (send is-bottle get-ef?))
    (def-var-val 'faucet (send is-faucet get-ef?))
    (def-var-val 'funnel (send is-funnel get-ef?))
  )
)

(send goal set-action
  '(cond
    ( (equal? bottle #t) (list 'bottle))
    ( (equal? faucet #t) (list 'faucet))
    ( (equal? funnel #t) (list 'funnel))
  )
)

(send goal set-influencer-classes
  (list is-bottle is-faucet is-funnel))

(send goal set-influencee-classes ())

(mk-ent rules r0)

(send r0 set-condition '(ueval transparent))

(send r0 set-action
  '(begin (send is-bottle input-ef certainty-factor)
    (send is-funnel input-ef certainty-factor)))

(send r0 set-influencer-classes (list transparent))

(send r0 set-influencee-classes (list is-bottle is-funnel))

(mk-ent rules r1)

(send r1 set-condition '(ueval cup-shaped))

(send r1 set-action
  '(send is-bottle input-ef certainty-factor))

(send r1 set-influencer-classes (list cup-shaped))

(send r1 set-influencee-classes (list is-bottle))

```

```

(mk-ent rules r2)

(send r2 set-condition '(ueval (not cup-shaped)))

(send r2 set-action
  '(send is-bottle input-cea certainty-factor))

(send r2 set-influencer-classes (list cup-shaped))

(send r2 set-influencee-classes (list is-bottle))

(mk-ent rules r3)

(send r3 set-condition '(ueval container))

(send r3 set-action
  '(send is-bottle input-ef certainty-factor))

(send r3 set-influencer-classes (list container))

(send r3 set-influencee-classes (list is-bottle))

(mk-ent rules r4)

(send r4 set-condition '(ueval (not container)))

(send r4 set-action
  '(send is-bottle input-cea certainty-factor))

(send r4 set-influencer-classes (list container))

(send r4 set-influencee-classes (list is-bottle))

(mk-ent rules r5)

(send r5 set-condition '(ueval conical-shaped))

(send r5 set-action
  '(send is-funnel input-ef certainty-factor))

(send r5 set-influencer-classes (list conical-shaped))

(send r5 set-influencee-classes (list is-funnel))

(mk-ent rules r6)

(send r6 set-condition '(ueval (not conical-shaped)))

(send r6 set-action
  '(send is-funnel input-cea certainty-factor))

(send r6 set-influencer-classes (list conical-shaped))

```

```

(send r6 set-influencee-classes (list is-funnel))

(mk-ent rules r7)

(send r7 set-condition '(ueval (not transparent)))
(send r7 set-action
  '(begin
    (send is-bottle input-cea certainty-factor)
    (send is-funnel input-cea certainty-factor)
    (send is-faucet input-ef certainty-factor)))

(send r7 set-influencer-classes (list transparent))

(send r7 set-influencee-classes
  (list is-bottle is-funnel is-faucet))

(mk-ent rules r8)

(send r8 set-condition '(ueval has-knob))

(send r8 set-action
  '(send is-faucet input-ef certainty-factor))

(send r8 set-influencer-classes (list has-knob))

(send r8 set-influencee-classes (list is-faucet))

(mk-ent rules r9)

(send r9 set-condition '(ueval (not has-knob)))

(send r9 set-action
  '(send is-faucet input-cea certainty-factor))

(send r9 set-influencer-classes (list has-knob))

(send r9 set-influencee-classes (list is-faucet))

```

APPENDIX 2. VISUAL MPU

```

(make-pair forward-models ' visual)

(send visual def-state '(name position direction map object-data image
                        orient image-source view-cnt inf-status temp))

(send visual set-display)

(send visual set-s (make-state      'interpreter-phase 'test-condition
                                'ext-activities '()
                                'int-activities '()
                                'before-output-buffer '()
                                'after-output-buffer '()
                                'action-buffer '()
                                'sigma 'inf
                                'phase 'passive
                                'name ' visual
                                'map (relation) ;;object classes, locs, orient
                                'view-cnt 0
                                )
)

;(set-attributes (state-map (send visual get-s)) '(class location orientation))
;(assert-rel (state-map (send visual get-s)) '(bottle #(0 0) #(0 1)))
;(assert-rel (state-map (send visual get-s)) '(funnel #(70 70) #(1 0)))
;(assert-rel (state-map (send visual get-s)) '(faucet #(40 50 ) #(0 1)))

;;;;;;;;; respond to command to start  visualization

(send visual add-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'passive)
                      (equal? (content-port x) 'start-visualization)
                      (vector? (car (content-value x)))
                      (vector? (cadr (content-value x))))
      )
    'before-output  '(make-content 'port 'respond 'source
                                   (state-name s))
    'action          '(begin
                      (set! (state-position s)
                            (car (content-value x)))
                      (set! (state-direction s)
                            (cadr (content-value x)))
                      (hold-in ' visual-start 1)
                      )
    )
  )
)

(send visual add-ext-activities
  (list
    (make-activity
      'condition '(and
                  (equal? (content-port x) 'image)
                  (equal? (state-phase s) 'wait-for-image)
                  )
    )
  )
)

```

```

      'action '(begin
        (set! (state-orient s) (cadr (content-value x)))
        (set! (state-image s) (cdr (content-value x)))
        (set! (state-image-source s) (content-source x))
        (hold-in 'inferencing 1)
      )
    )
    ;;; example: content-value x =
    ;;; ((orientation right) (container 0.5) (cup-shaped 0.8))
  )
)

(define (get-nearest-tuple relation direction position)
  (define (colinear direction vector)
    (< (geometric-distance (vector-normalize vector) direction) .1)
  )
  (let* (
    (locations (get-range-rel relation 'location))
    (loc-in-direction (clean (map (lambda(loc)
      (when (colinear direction
        (vector-sub loc position))
        loc)) locations)))
    (nearest-loc (closest-fn position loc-in-direction)) ;; for now
  )
  (car (select-rel relation (list '- nearest-loc '-)))
))

;;; return example : (box #(0 10) #(1 0))

(send visual set-int-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'visual-start)
        (eval '(get-nearest-tuple ,(state-map s)
          ,(state-direction s),(state-position s)))
      )
      'before-output '(make-content 'port 'light 'echo-status #!true
        'source (state-name s)
        'value (clip
          (geometric-distance
            (state-position s)
            (cadr
              (eval '(get-nearest-tuple
                ,(state-map s)
                ,(state-direction s)
                ,(state-position s))))))
          0))
      'action '(begin
        (set! (state-object-data s)
          (eval '(get-nearest-tuple ,(state-map s)
            ,(state-direction s),(state-position s))))
        (set! (state-view-cnt s) (1+ (state-view-cnt s))));;;
        (passivate-in 'wait-for-image)
      )
    )
    (make-activity
      'condition '(equal? (state-phase s) 'visual-start)
      'before-output '(make-content 'port 'in
        'source (state-name s)

```



```

'channel 'object-identify
'value 'no-object-there)
'action '(passivate-in 'visual-start)
)
)
)

;;;;; start inference process ;;;;;;
;;;;; default : backward chaining ;;;

(send visual add-int-activities
(list
  (make-activity
    'condition '(and (equal? (state-phase s) 'inferencing)
                     (equal? (state-inf-status s) 'forward-chain)
                    )
    'action '(hold-in 'forward-chaining 1)
  )
  (make-activity
    'condition '(equal? (state-phase s) 'inferencing)
    'action '(hold-in 'backward-chaining 1)
  )
)

;; backward-chaining ;;

(make-activity
  'condition '(equal? (state-phase s) 'backward-chaining)
  'action '(begin
    (set! (state-temp s)
      (bc-exp (state-image s) (car (state-object-data s))))
    (set! (state-inf-status s) 'backward-chain)
    (hold-in 'analyse-image 1))
  )
  ;; bc-exp returns (class truth-value)
)
(make-activity
  'condition '(and
    (equal? (state-phase s) 'analyse-image)
    (equal? #t (cadr (state-temp s)))
  )
)

;;; confirm object class
'action '(hold-in 'send-analyse-bc-result 1)
'after-output '(make-content 'port 'finished
  'source (state-name s))
)
;; let selector open

(make-activity
  'condition '(equal? (state-phase s) 'send-analyse-bc-result)
  'before-output '(make-content 'port 'in
    'source (state-name s)
    'channel 'object-identify
    'value (list
      (car (state-object-data s))
      (cadr (state-object-data s))
      (state-image-source s))
    )
    ;; now it's model name
    ;; used by space manager
  )
  'action '(passivate)
)

```

```
;; unknown result; move closer
```

```
(make-activity
  'condition '(and
    (equal? (state-phase s) 'analyse-image)
    (equal? (cadr (state-temp s)) 'U) ;; EFX or EFA
    (<= (state-view-cnt s) 2)
    (> (geometric-distance (state-position s)
      (cadr (state-object-data s))) 20)
  )
  'action '(hold-in 'move-closer 1)
  'after-output '(make-content 'port 'finished
    'source (state-name s))
)
```

```
;; unknown result; turn to face the object
```

```
(make-activity
  'condition '(and
    (equal? (state-phase s) 'analyse-image)
    (equal? (cadr (state-temp s)) 'U)
    (or (> (state-view-cnt s) 2)
      (<= (geometric-distance (state-position s)
        (cadr (state-object-data s))) 20))
    (not (equal? (state-orient s) 'face))
  )
  'action '(begin
    (set! (state-view-cnt s) 0)
    (hold-in 'move-to-face 1)
  )
  'after-output '(make-content 'port 'finished
    'source (state-name s))
)
```

```
(make-activity
  'condition '(equal? (state-phase s) 'analyse-image)
  'action '(hold-in 'forward-chaining 0)
)
```

```
;; forward-chaining
```

```
(make-activity
  'condition '(equal? (state-phase s) 'forward-chaining)
  'action '(begin
    (set! (state-temp s) (fc-exp (state-image s)))
    (set! (state-inf-status s) 'forward-chain)
    (hold-in 'identify-object 1)
  )
  )
  ;;; fc-exp returns (class)
  (make-activity
    'condition '(and (equal? (state-phase s) 'identify-object)
      (state-temp s)
    )
  )
)
```

```
;; identify object as different than expected
```

```
  'action '(hold-in 'send-analyse-fc-result 1)
  'after-output '(make-content 'port 'finished
    'source (state-name s))
  )
  ;; let selector open
  (make-activity
```

```

'condition '(equal? (state-phase s) 'send-analyse-fc-result)
'before-output '(make-content 'port 'in
                  'source (state-name s)
                  'channel 'object-identify
                  'value (state-temp s)
                  )
'action '(eval '(begin
                (retract ,(state-map s) ',(state-object-data s))
                (assert-rel ,(state-map s)
                  ',(append (state-temp s)
                    (cdr (state-object-data s))))
                (set! (state-inf-status s) '())
                (passivate)
                ))
)

;; unknown result; move closer

(make-activity
  'condition '(and
    (equal? (state-phase s) 'identify-object)
    (<= (state-view-cnt s) 2)
    (> (geometric-distance (state-position s)
      (cadr (state-object-data s))) 20)
  )
  'action '(hold-in 'move-closer 1)
  'after-output '(make-content 'port 'finished
    'source (state-name s))
)

;; unknown result; turn to face the object

(make-activity
  'condition '(and
    (equal? (state-phase s) 'identify-object)
    (or (> (state-view-cnt s) 2)
      (<= (geometric-distance (state-position s)
        (cadr (state-object-data s))) 20))
    (not (equal? (state-orient s) 'face))
  )
  'action '(begin
    (set! (state-view-cnt s) 0)
    (hold-in 'move-to-face 1)
  )
  'after-output '(make-content 'port 'finished
    'source (state-name s))
)

;; can't identify object

(make-activity
  'condition '(equal? (state-phase s) 'identify-object)
  'action '(hold-in 'send-analyse-result 1)
  'after-output '(make-content 'port 'finished
    'source (state-name s))
)
;; let selector open
(make-activity
  'condition '(equal? (state-phase s) 'send-analyse-result)
  'before-output '(make-content 'port 'in
    'source (state-name s)
    'channel 'object-identify

```

```

                                'value 'cant-identify-object
                                )
                                'action '(begin
                                    (set! (state-inf-status s) '())
                                    (passivate))
                                )
                                )
                                )
(send visual add-int-activities
 (list
  (make-activity
   'condition '(equal? (state-phase s) 'move-to-face)
   'before-output '(make-content 'port 'in
                        'value
                        (vector-add
                         (cadr (state-object-data s))
                         (scalar-mult 20 (caddr (state-object-data s)))
                        )
                        'echo-status #t
                        'channel 'start-navigation)
   'action '(begin
              (set! (state-position s)      ;; update position
                    (vector-add              ;; update direction ??
                     (cadr (state-object-data s))
                     (scalar-mult 20 (caddr (state-object-data s)))
                    )
              )
              (passivate-in 'moving-to-face))
            )
  (make-activity
   'condition '(equal? (state-phase s) 'move-closer)
   'before-output '(make-content 'port 'in
                        'value (vector-add (state-position s)
                                             (scalar-mult 20 (state-direction s)))
                        'echo-status #t
                        'channel 'start-navigation)
   'action '(begin
              (set! (state-position s)
                    (vector-add (state-position s)
                                (scalar-mult 20 (state-direction s))))
              (passivate-in 'moving)
            )
  (make-activity
   'condition '(equal? (state-phase s) 'view-again)
   'before-output '(make-content 'port 'light
                        'echo-status #t
                        'source (state-name s)
                        'value (clip
                                (geometric-distance
                                 (state-position s)
                                 (cadr (state-object-data s))) 0))
   'action '(begin
              (set! (state-view-cnt s) (1+ (state-view-cnt s)))
              (passivate-in 'wait-for-image)
            )
  )
 )
 )
 )
 )

```

```

(send visual add-ext-activities
  (list
    (make-activity
      'condition '(and (equal? (state-phase s) 'moving)
                        (equal? (content-port x) 'arrived))
      )
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(hold-in 'view-again 0)
    )
    (make-activity
      'condition '(and (equal? (state-phase s) 'moving-to-face)
                        (equal? (content-port x) 'arrived))
      )
      'before-output '(make-content 'port 'respond
                                   'source (state-name s))
      'action '(hold-in 'rotate 1)
    )
  )
)

;;; rotate visual to face on object

(send visual add-int-activities
  (list
    (make-activity
      'condition '(equal? (state-phase s) 'rotate)
      'before-output '(make-content 'port 'change-direction
                                   'source (state-name s)
                                   'value (scalar-mult -1
                                                       (caddr (state-object-data s))))
      'action '(begin
                (set! (state-direction s)
                      (scalar-mult -1 (caddr (state-object-data s))))
                (hold-in 'view-again 0))
            )
    )
  )
)

```

REFERENCES

- [1] Arkin, R.C., "Path Planning of a Vision-Based Autonomous Robot", *Proc. of The SPIE Int'l Conf. on Mobile Robots*, vol. 727, pp. 240-250, Oct. 1986.
- [2] Cho, Taeho, "Representation of Rule-Based Systems", ECE579 Term Project, University of Arizona, December, 1988.
- [3] Concepcion, A.I. and B.P. Zeigler, "DEVS Formalism: A Framework for Hierarchical Model Development", *IEEE Trans. on Software Engineering*, Vol. 14, No. 2, Feb. 1988, pp. 228-241.
- [4] Firschein, Oscar et al., *Artificial Intelligence for Space Station Automation*, Noyes Publications, Park Ridge, New Jersey, 1986.
- [5] Garzia R.F., M.R. Garzia, and B.P. Zeigler, "Discrete Event Simulation", *IEEE Spectrum*, December 1986, pp. 32-36.
- [6] Hunt, V. Daniel, *Smart Robots*, Chapman and Hall, New York, 1985.
- [7] Kim, Tag Gon, "A Knowledge-Based Environment for Hierarchical Modelling and Simulation", Doctoral Dissertation, University of Arizona, Tucson, AZ 85721, 1988.
- [8] Rao, A. Ravishankar and Ramesh Jain, "Knowledge Representation and Control in Computer Vision System", *IEEE Expert*, Vol. 3, No. 1, Spring 1988, pp. 64-79.
- [9] Reddy, Y.V., M.S. Fox, N. Husain, and M. McRoberts, "The Knowledge-Based Simulation System", *IEEE Software*, March 1986, pp. 26-37.
- [10] Rozenblit, J.W., "A Conceptual Basis for Model-Based System Design", Doctoral Dissertation, Dept. of Computer Science, Wayne State University, Detroit, MI, 1985.
- [11] Rozenblit, J.W. and B.P. Zeigler, "Concepts for Knowledge-Based System Design Environments", *Proc. Winter Simulation Conference*, San Francisco, CA, 1985, pp. 223-231.
- [12] Rozenblit, J.W., S. Sevinc, and B.P. Zeigler, "Knowledge-based Design of LANs Using System Entity Structure Concepts", *Proc. Winter Simulation Conf.*, Washington, D.C., 1986, pp. 885-865.
- [13] Sevinc, S. and B.P. Zeigler, "Entity Structure Based Design Methodology: A LAN Protocol Example", *IEEE Trans. on Software Engineering*, Vol. 14, No. 3, March 1988, pp. 375-383.
- [14] Staugaard, Andrew C., *ROBOTICS and AI*, Prentice-Hall, Englewood Cliffs, N.J., 1987.

- [15] Zeigler, B.P., *Multifaceted Modelling and Discrete Event Simulation*, Academic Press, London and Orlando, FL, 1984.
- [16] Zeigler, B.P., "DEVS-Scheme: A Lisp-Based Environment for Hierarchical Modular Discrete Events Models", Tech. Rep. AIS-2, CERL Lab., Dept. of ECE, Univ. of Arizona, Tucson, AZ 85721, 1986.
- [17] Zeigler, B.P., "Hierarchical, Modular Discrete Event Modelling in an Object Oriented Environment", *Simulation J.*, Vol. 49:5, 1987, pp. 219-230.
- [18] Zeigler, B.P., "Some Properties of Modified Dempster-Shafer Operators in Rule Based Inference System", *Int. J. General Systems*, Vol. 14, 1988, pp. 345-356.
- [19] Zeigler, B.P., "Implementation of Methodology Based Tool in the DEVS-Scheme Environment", In: *Modelling and Simulation Methodology: Knowledge System Paradigms*, M.S. Elzas, T.I. Ören, B.P. Zeigler (Eds.), North-Holland, Amsterdam, 1988.
- [20] Zeigler, B.P., "DEVS Representation of Dynamic Systems: Event-Based Intelligent Control", *Proceedings of the IEEE*, Vol. 77, No. 1, Jan. 1989, pp. 72-80.
- [21] Zeigler, B.P., F.E. Cellier, and J.W. Rozenblit, "Design of a Simulation Environment for Laboratory Management by Robot Organizations", *J. Intelligent and Robotic Systems*, Vol. 1, 1988, pp. 299-309.
- [22] Zeigler, B.P., *Knowledge Representation in Simulation Environments*, (in preparation)
- [23] Zhu, S., "CESM Documentation", Dept. of ECE, University of Arizona, Tucson, AZ 85721, 1988.