A Task-Level Model of Distributed Computation for Sensory-Based Control of Complex Robot Systems [1]

Damian Lyons
Michael Arbib

COINS Technical report 85-30

October 18, 1985

Laboratory for Perceptual Robotics
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

A version of this paper was presented to the IFAC *Symposium on Robot Control*, 6-8th of November, 1985, Barcelona, Spain.

Abstract.

We describe the development of a task-level model of distributed computation specifically designed for complex robot systems. We start by describing two fundamental problems which characterize the complex robot domain, and then describe four examples by which our model can be judged. Emphasizing that the formal exploration of behavior is as important as the specification of behavior, we describe the syntax and semantics of our model. Noting that our emphasis in this paper is on representation, we detail the implementation of each of the four examples introduced earlier. We conclude by analysing our implementations and outlining future work on this model.

[1]Preparation of this paper was supported in part by grants ECS-8108818 and DMC-8511959 from NSF

1. Introduction

We are developing a computational model for *task-level, distributed* robot control. By task-level is meant that actions can be described by their effects on objects [9]. By distributed is meant that concurrent entities in the model cooperate and compete [12], generating a decentralized control architecture. Such a model would allow the specification of behavior for *complex* robot systems working in dynamic environments. We define a complex robot system as a robot with *many degrees of freedom* (possibly highly redundant) and *multiple sensory capabilities* [3]. Our interest in these robots stems from their potential for versatile behavior far outstripping current industrial robots.

Using existing methods for the specification of robot behavior, it is difficult to realize the full capabilities of complex robot systems. We wish to develop a model which not only allows the specification of complex behavior patterns for robots, but also allows the examination and exploration of this behavior formally.

In general, robot programming languages, both *robot-level* and *task-level* [9], are simply general purpose programming languages with special procedures for interfacing to the robot mechanism. Motor control and sensory input are essentially represented as output and input from peripheral devices. Some recent robot languages have followed a new approach; that motor control and sensory input are not just specific examples of the peripheral *write* and *read* operations, but highly important characteristics of the whole robot programming problem. An important example of this is Geschke's RSS [5]. In order to command the robot to do some action, the RSS programmer initiates a computing agent, called a *Servo Process*. Each such servo process is a combination of a sensory event and a motor action. SMS [17] defines action in terms of the cooperative activity of a *sense-process* and a *motor-process*, called a *function*. It differs from RSS in that it defines a *sensory-motor hierarchy* and associates a *judge process* with each function to determine if the function has achieved its objective.

Arbib [1] defines the *perceptual schema* as the unit of sensing and the *motor schema* as the unit of motor behavior. Motor and perceptual schemas are closly coupled in that a motor schema may be triggered by a perceptual schema and a motor schema may trigger task-specific perceptual schemas to analyse sensory information in a task context. In Overton's [13] tactile sensing robot system, a *schema* is an abstract type which monitors certain aspects of the current situation and becomes active when the situation matches the expected state. Both [1] and [13] represent a complex task or perceptual event as an *assemblage* of schemas.

2. Complex Robots

There are two fundamental difficulties in dealing with complex robot systems. The first is the *coherent control of many separate degrees of freedom*. Examples are the fingers of a dextrous hand or the legs of a multi-legged robot. The second is the *integration of sensory information into motor*

behavior. The only way to deal with a dynamic environment is to parameterize motor behavior with dynamic sensory information, sensory information which is on-line description of the surrounding environment.

A solution to the first problem would entail some versatile mechanism for linking degrees of freedom with each other. Since this problem will become more important as more and more complex systems are built, a solution must be general and not based on a particular robot mechanism. A solution to the second problem must address two areas: A robot system which lacks a focusing mechanism derived from task context is degraded by having to consider all sensory information, as is a robot system which receives appropriate information but in a non-task-oriented manner.

We use these two problems as the cornerstones of our model, and from them develop a number of particular examples. We shall judge our model by the facility with which it implements these examples, and hence, solves the fundamental problems.

To make our analysis more concrete we have chosen to work in the dextrous hand domain. Once we have suitably developed our model in simulation, we plan to implement it on a network of processors controlling a Salisbury hand [11]. Our previous work in this area [2], [7], [10] hasapproached the problems of grasping from the task-level, *not* from the mechanism level. Thus we have not constrained ourselves to a particular hand model as does much of the literature, and we attempt to integrate grasping and manipulation into an overall task-context. While such integration has been formulated in the context of the two-fingered gripper [8], both versatility and problems increase when a dextrous hand is employed.

We base our grasping on a general model of the human hand having a number of multi-jointed fingers based on a rigid palm, in turn mounted on a 6-degree of freedom (DOF) wrist. We define a *grasp* as characterizing a domain of interaction between the dextrous hand and objects in the world. Each grasp is defined as a triplet consisting of; a *preshape configuration*, some *acquisition information*, and some *manipulation* information. All grasp information is described in terms of logical units, *Virtual Fingers*, which are mapped to physical fingers on the basis of hand and object characteristics.

3. Test Examples

We outline here four test examples which embody much of the difficulties of representation in the complex robot domain. We shall judge our model on how well it implements specific instances of these examples in the hand domain.

Example I

The *guarded move*, a motor command with a sensory termination, has emerged as a powerful robot-level programming construct. Our model must be able to represent the classicial guarded

move for n sensors and m limbs.

Example II

The concept of the guarded move is as valid at the task-level as at the robot-level, but the classical form of the construct is robot-level. Object parameterization of motor commands is the task-level concept of which the sensory termination condition of the guarded move is the robot-level projection. Important aspects of this example are determining the nature of object representation and its relationship to sensor data.

Example III

Basic to a task-level formulation is a representation of an *object*; however it is really only relevant to the robot controller in that it *triggers* or *parameterizes* some task process. Our model must be able to represent two situations: A task process which when active, searches for some given object and is then parameterized by that object's properties; and a task process to be triggered by the recognition of some object and then grossly parameterized by that object's properties.

Example IV

A logical aggregation mechanism is a highly important tool in specifying the behavior of complex robots. Although the concept of the Virtual Finger (VF) [2] is hand specific, the notion of representing actions in terms of logical rather than physical units is equally valid in any robot domain, e.g. Virtual Legs [14]. We must be able to group an arbitrary number of fingers together to form a VF, and be able to issue common commands to all the members by issuing the command to the VF. We extend this by demanding that we be able to use the VF concept to specify *similar* but not identical conditions for each member of the VF. Consider closing all the fingers of a hand on an object. Depending on the local geometry some fingers may come to meet the object sooner that others. However, all the fingers are essentially carrying out the same command — moving until they grip the object surface.

4. Model

We use the Port Automata model of Steenstrup et al.[15] to provide the semantics of distributed computation. Essentially a *port automaton* (PA) is a formal machine equipped with a set of *ports* through which all communication with the environment takes place. Communication is achieved by connecting together ports on separate PA using a *port connection map*. Such a network may be considered as a single PA, its port set being composed of those ports which have not been connected by the port connection map. All communication is synchronous[16]; that is, a PA writing to a port will suspend execution (i.e. will not proceed to its next state) until some port connected to that port has been read. In similar fashion, a PA reading a port will suspend execution until some value is written to a port connected to that port.

We have constructed a programming environment based on our model on a VAX 11/780. The environment consists of a *compiler* which accepts programs in the notation introduced here, and produces code for a network of stack machines; an *emulator* for the network of stack machines; and a *robot simulator*[2] which accepts motor commands from the network, produces graphical output of the robot mechanism, and feeds back simulated sensory input to the network.

4.1 Schemas

Our syntactic unit of distributed computation is the *schema*. Semantically a *schema* is a *generic* port automaton description augmented with *instantiation* and *deinstantiation* operations. A *schema* description consists of: A list of input and output ports, which we associate with the ports of an equivalent PA; an internal variable list and a behavior section, which we associate with the *state transition* and *output* maps of the equivalent PA. The *instantiation operation* takes as input a *schema* and some *instantiation parameters*, and produces a computing agent, referred to as a *schema instantiation* or SI. The *instantiation parameters* consist of initial values for the internal variables, and a *connection map* specifying connections of the ports on this *schema* to ports on other SI. The behavior section is a program which cycles continuously until the SI is deinstantiated. These instructions can synchronously read from or write to the ports, access internal variables, instantiate other *schemas*, or deinstantiate other SIs. We use the following syntax to define a schema:

$$[\mathbf{N}\ (\mathbf{ip})\ (\mathbf{op})\ (\mathbf{v})\ (\mathbf{b})], \tag{1}$$

where,

- N is an identifying name for the schema
- ip, op are lists of input and output port names respectively
- v is a list of internal variable names
- b is a specification of behavior

All five components of the schema description must be present, however components other than the name may be empty, which we indicate by empty parenthesis (). We define the behavior section by the following syntax:

$$\begin{aligned}
&< \textit{behaviorsection} > ::= < \textit{Stat} >^{*} \\
&< \textit{Stat} > ::= < \textit{Assign} > | < \textit{If} > | < \textit{Instn} > | < \textit{Dinstn} > \\
&\qquad | < \textit{For} > | < \textit{Forall} > \\
&< \textit{Assign} > ::= < \textit{Var} > := < \textit{Expression} > | \\
&\qquad < \textit{OutputPort} > := < \textit{Expression} >
\end{aligned}$$

We embed reading and writing into the syntax of < *Assign* >; an input portname occuring in < *Expression* > is a read to that port, and an output portname on the left hand side of an < *Assign* > is a write to that port. Apart from this we assume standard syntax for an < *Expression* >.

$$< Instn > \quad ::= < Schemaname >_{Inst} \{V_0\}\{C_0\}$$
$$< Dinstn > \quad ::= \textbf{STOP}\{< Schemaname >_{Inst}\}$$

The V_0 parameter for the instantiation operation is a list of initial variable values of *Schemaname*; these are assigned *by position* to the internal variables of *Schemaname* as they are specified in its *v* list (the same syntax as is usual for procedure parameters). The C_0 parameter specifies connections for the ports of *schemaname* to ports on other SI, and is a list:

$$C_0 \quad ::= (< Couplets >^*)$$
$$< Couplets > \quad ::= Ipname \leftarrow SIPname \mid SIPname \leftarrow Opname$$
$$SIPname \quad ::= Schemaname_{Inst}(port)$$

where *Ipname* and *Opname* are names of input and output ports of *schemaname*, and *SIPname* identifies a port on an SI.

$$< If > ::= \textbf{IF} \ < condition > \ \textbf{THEN} \ < Stat >^* \ \textbf{ELSE} \ < Stat >^* \ \textbf{ENDIF}$$
$$< For > ::= \textbf{FOR} \ < index > = 1 \ldots n \ \textbf{DO} \ < Stat >^* \ \textbf{ENDFOR}$$

We assume standard syntax for < *Condition* >, and constrain < *Index* > to be an internal variable for simplicity. Apart from this **IF** and **FOR** are as one would expect. We extend the syntax to < *Instn* > to allow the **FORALL** statement.

$$< Forall > ::= \textbf{FORALL} \ Schemaname \ \textbf{DO} \ < Instn >^* \ \textbf{ENDFOR}$$

This statement uses a schema name as an *index* for definite iteration. The < *Instn* >* commands in the loop body will be executed *once for each instantiation of schemaname which currently exists.* Consider the following example of schema definition:

```
[TClosejoint (T_in P_in) (P_out)          ;;name and port definitions.
(Del)                                     ;;internal variable, step increment
(IF T_in=0        THEN P_out:=P_in+Del    ;;increment current position
```

ELSE P_{out}:=P_{in} ;;stay at this position

ENDIF)]

The internal variable *Del* must be initialized when *TClosejoint* is instantiated, and connections made to its ports. If T_{in} was a tactile input and P_{in} and P_{out} were actual and desired position respectively, then *TClosejoint* would continually increment the position of some joint until the tactile input was non-zero. Although we can specify robot programs at this level, we have not constructed anything in our model yet to facilitate their representation. It is for this purpose we present the following structure.

4.2 Assemblages

An assemblage SI is a computing agent in which the behavior is defined as the behavior of a number of other communicating SIs. The *port connection* automaton of Steenstrup et al. provides the semantics for our assemblage construct. This aggregate SI can be considered the instantiation of a single *schema*, an *assemblage schema*, which must contain information on how the individual SI are created and connected, and how the ports of the component SIs appear as the ports of the assemblage SI.

An *assemblage schema* description consists of an input and output port list, an *equivalence* list of assemblage port names with component SI port names, a list of component SI and an initialization behavior section to set up the SI network, and a *network map* detailing the connections of SI *within* the assemblage only. An assemblage SI terminates when all its component SIs terminate. We further strengthen modularity by defining instantiation numbering to be *local* to the assemblage in which the SI is a component. Also we now define the instantiation number to be an optional but unambiguous parameter to the instantiation operation; this paves the way for our definition of sensing in our model. We use the following syntax for assemblage definition:

$$[N\ (ip)\ (op)\ (s)\ (ib)\ (p)\ (n)], \tag{2}$$

where,

- N, ip, and op are the assemblage name and its input and output ports lists respectively,
- s and ib are a list of component schemas, and the commands to set up the network of component SIs, respectively,
- p defines the way in which the ports of the component SIs appear as the ports of the assemblage SI, a list of the form

  P ::= (< *Equivalence* >*)
  < *Equivalence* > ::= *assemblage port name* ≡ *component port name*

- n defines the port connection mappings between component SIs,

In an *implementation* of the assemblage construct we would expect that all the component SIs are close together in communication space; this is a process distribution criterion for a distributed computer system. We would also expect that an assemblage can be started and stopped as if it were a single SI and that all the component SIs of the assemblage are treated equally with respect to resource access.

Fan-in occurs if an input port on an SI is connected to more than one output port; fan-out occurs if an output port is connected to more than one input port. If a port a on A is connected to ports b on B and c on C, then any value written by A to its port a will be simultaneously passed to both b on B and c on C. After a write to a, SI A can only proceed if at least one SI reads a port connected to a. The semantics of fan-in is important to effectively implement *parallel searches*. If an SI A instantiates B and C to search some range in parallel, and connects its port a to result ports on B and C, then A would like any read on a to terminate if *either* of B or C produce a value first, since this makes most effective use of the parallelism. In general a read to some port a will terminate if *any* port connected to that port is written to; if more than one connected port is written to, then all written values will eventually be readable at a.

4.3 Sensing

We represent sensation in our model by predefining a list of *special schemas*. The *ith* instance of the *Jmotor* schema accepts input through its input port *desired* and has an implementation defined effect on the *ith* joint of the robot mechanism for some defined physical numbering system. For a stepper motor *Jmotor* might control the number of steps to take, for a DC-servo motor it might control torque. An important point to grasp is that to control the ith joint, make a local ith instance of the schema *Jmotor*. To disambiguate this situation we let the actual value of the controlled variable fed to the *ith* joint is the average of all $Jmotor_i$ values. *Jmotor* is sketched as:

$$[Jmotor(desired)(actual)\cdots < implementation\ dependent > \cdots] \quad (3)$$

In a similar fashion a schema *Jposition* is predefined, the *ith* instance of which reports on the position of the *ith* joint for some numbering of the physical mechanisms. We consider two other robot senses, a tactile sense and a visual sense. An SI $Tactile_i$ reports on the status of the *ith* tactile sensor, for some numbering of the tactile sensors.

$$[Tactile()(Contact)\cdots < implementation\ dependent > \cdots] \quad (4)$$

Vision is a more difficult sense to deal with in the fashion in which we have developed position and touch. The SEF schema is predefined (Separable Environmental Facet); an instance of the SEF schema is created automatically by the visual interface for each separable bunch of features detected. We demand that SEF_i always represent the same set of features; this we refer to as visual continuity. One way to implement such visual continuity is to define it in terms of continuity of the features measured by an SEF SI. SEF is defined to have no input ports, and one output

port per feature measured. It can then be described as an assemblage of *Feature* SIs, each of which measures one feature, and provides one port on the assemblage. Thus SEF has ports which can most extensibly be described as a *port array*; we denote this by subscripting the ports in the assemblage definition.

$$[SEF()(F_1 \ldots F_n) \cdots < \textit{implementation dependent} > \cdots] \tag{5}$$

Object continuity is not, of course, so simply related to feature continuity. However for the time being let us accept this approximation to proceed with model development. Note that SEF_i is essentially a *logical* sensor for a distinct physical set of features under some numbering system [6]. Also we assume SEF SIs local to all assemblages.

4.4 Task-Unit

The assemblage construct is used to build the basic unit of task representation. In this we follow RSS and SMS in tightly coupling a sensory oriented process and a motor oriented process; where a *sensory* SI is primarily concerned with reading and processing sensor information and a *motor* SI is primarily concerned with motor control. We extend this structure by the addition of a third, linking, process between sensing and action. "Robotics is the *intelligent connection of perception to action*"[4], this third process provides the intelligent link. The special assemblage constructed in this way is referred to as a *task-unit* assemblage.

To specify a task-unit it is necessary to specify the sensory and motor components and the details of the linking process, which we will refer to as the *r*-schema. A task-unit has the same semantics as an assemblage, with the exception that the task-unit terminates if the *r*-schema terminates (this is necessary because of the way in which sensing and action were defined as special schemas).

$$\textbf{[N (ip) (op) (sl) (ml) (ib) (var)(b)]} \tag{6}$$

where,

- N, ip, and op are the name and port lists of the task-unit and *r*-schema
- sl and ml are the list of sensory components and motor components respectively, and ib is the instructions which initialize this network,
- var and b are the internal variables and behavior section of the *r*-schema.

It is easy to see how this can be rewritten as an assemblage definition (2). Although definition (6) characterizes a task-unit schema completely, it is advantageous sometimes to be able to abbreviate a task-unit, to leave implicit the linking process and detail only the sensory and motor SIs. For example we indicate a task unit assemblage $Joint_i$ consisting of $Jposition_i$ as a sensory SI and $Jmotor_i$ as a motor SI as:

$$Joint_i = [Jposition_i - Jmotor_i] \tag{7}$$

This task-unit specifies some form of position servo. When instantiated it position servos the *ith* physical joint for some numbering of the physical mechanism. The internals of $Joint_i$ are abbreviated to '—'. In a similar fashion we can define a force servo.

$$Jointf_i = [Jforce_i - Jmotor_i] \tag{8}$$

Depending which of (7) or (8) are instantiated per physical joint we can specify a particular hybred position/force controller. We can describe the guarded move which was the subject of our earlier schema example as:

$$Closejoint_i = [Tactile_i - Joint_i] \tag{9}$$

This 'shorthand' notation allows us to briefly explain our notation of a precondition. We can optionally associate a *precondition* with each task unit. In implementation terms a precondition is an SI which will test for some defined conditions before making an instance of the task unit. The precondition is not constrained to *terminate* when the assemblage has been created; it may terminate, or it may continue to be active, creating assemblages whenever it is satisfied. For example, if *Mug*? is an SI which recognises a mug from sensory input, then we denote the precondition relationship of *Mug*? to some task-unit for grasping the mug by:

$$Mug? : [Mug - Graspmug] \tag{10}$$

5. Implementation of examples

Example I

Consider the following much simplified problem: A dextrous hand has been placed in such a configuration around an object that evenly closing the joints on each finger will succeed in stably acquiring the object. Let us assume a numbering of the physical mechanism and sensors as in fig. 1. A task-unit schema can be described in terms of definition (6) as:

```
[ Closejoint (T_in P_in) (P_out)                    ;;name and port definitions
(Tactile) (Joint)                                   ;;sensory, motor components
( Tactile_4*i() (T_in ← Tactile(contact))           ;;one instance of Tactile
  Joint_i()(P_in ←Joint(actual), Joint(desired)← P_out) )   ;;and one of Joint
(i del)                                             ;;internal variables
( IF T_in = 0 THEN P_out :=P_in+del                 ;; no contact so move in
  ELSE P_out :=P_in                                 ;; otherwise stay here
 ENDIF) ]
```

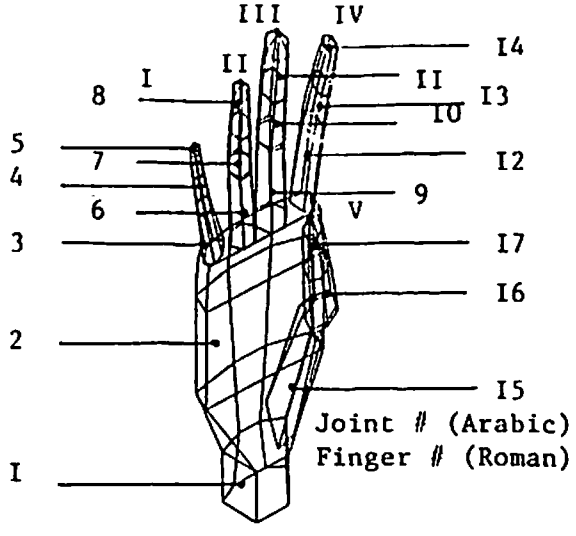
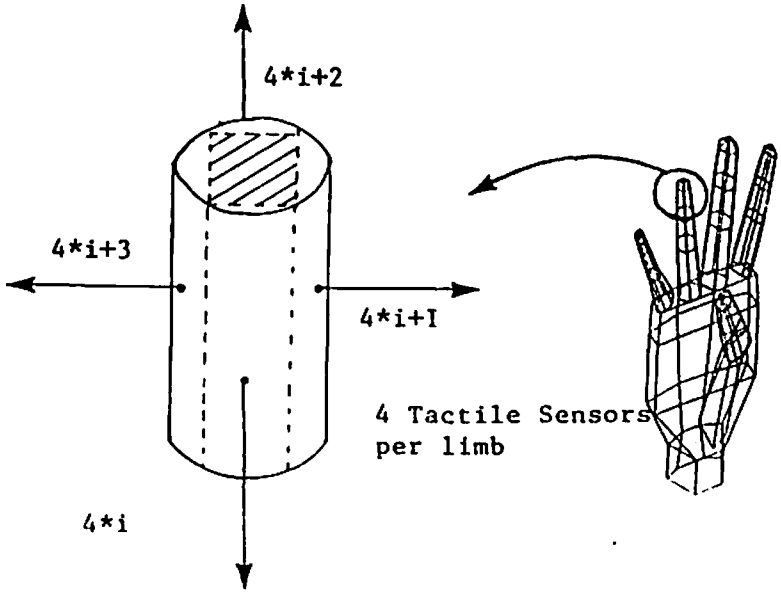


Figure I: Numbering of Hand Limbs and Tactile Sensors.



When *Closejoint* is instantiated i and *del* must be appropriately parameterized. Notice that the r-schema degenerates to a simple IF statement; as long as $Closejoint_j$ exists this IF continually executes — the guarded move.

Having specified the required action for a single joint we now need to apply this to all the joints which need to be controlled. We could just go ahead and instantiate *Closejoint* for each necessary joint. However each $Closejoint_i$ is a locally tightly-coupled group of processes, left on their own, they will each proceed at varying paces depending on the physical mechanism and the processor scheduling algorithms. One explicit solution to this is to change our task-unit specifications so that each instance of *Closejoint* is synchronized with every other instance.

This provides complete synchronization at the expense of parallelism. If we weaken synchronization to say that all $Closejoint_j$ will be treated *equally* with respect to the resources they use, both computational and physical, then we can use our assemblage construct. We define all the instances of *Closejoint* to be components of some assemblage *Grip*:

```
[Grip ()()                  ;;no ports necessary
 (Closejoint)               ;;only one component schema

 (FOR i = 1...n DO          ;;for joints 1 to n
  Closejoint (i, 10)()      ;;set up local networks of SI
 ENDFOR)

 ()()]                      ;;no port maps
```

This assemblage instantiates n *Closejoint* SI in a tightly-coupled bundle. Although *Closejoint* SI do not communicate with each other, they are scheduled with equally fair resource allocations; an *implicit* synchronization. In addition effects due to setting up or stopping the network are eliminated, since by definition all components of an assemblage are treated as a single SI. If we use the concept of a *port array*, as used to specify the SEF (5), the task-unit generalizes easily to control any number of joints based on one sensor, or one joint on any number of sensors.

Example II

In the preshape phase of grasping, a dextrous hand must be configured to facilitate object acquisition and subsequent manipulation, and also conveyed to the object location. For reaching, an object model should provide a location and orientation for the object; however for preshaping, object shape and size are the important characteristics. The object model therefore, depends on the task to be accomplished, and consists of designated sensor readings or data computed from sensor readings. Let us assume the feature ports on an SEF, $F_1 \ldots F_4$, provide *position*, *orientation*, *shape*, and *size* respectively, and that we know SEF_k represents an object we wish to grasp. We write two task units *reach* and *preshape*:

$$Reach = [RObject - (MoveWrist, OrientWrist)] \tag{11}$$

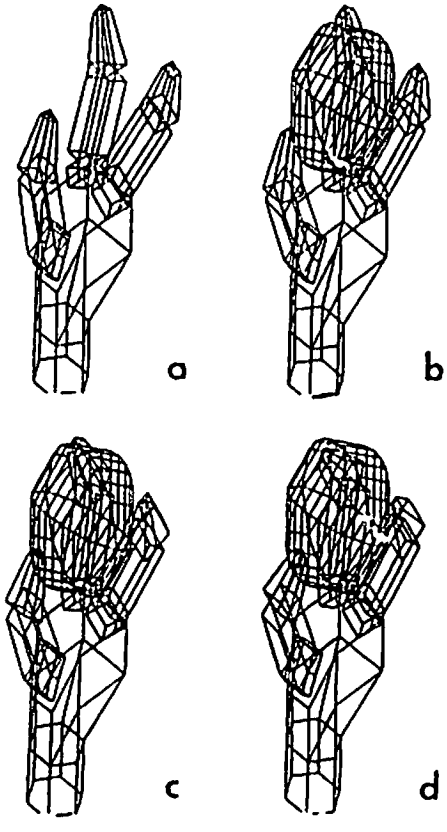


Figure 2: Graphic Output of Simulation of Closejoint$_i$ Task-Unit for Salisbury Hand Model.

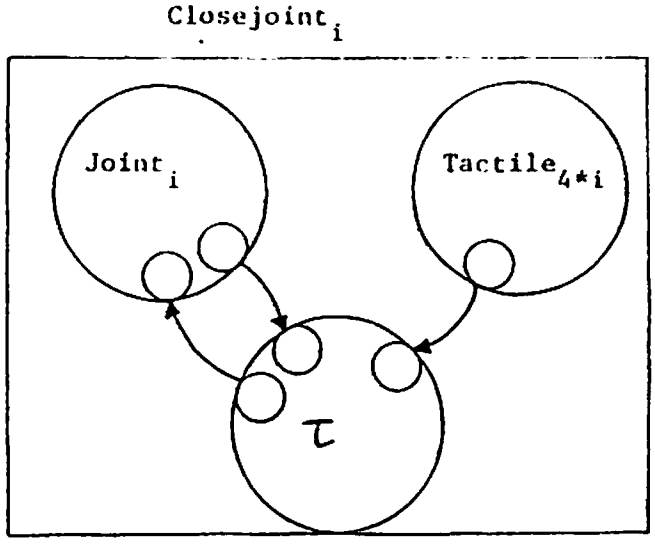


Figure 3: The Closejoint Task-Unit Schema.

Where parentheses are used to group motor SIs together. *MoveWrist* is an assemblage of *Joint* SIs which control the base position of the wrist, and in similar fashion *OrientWrist* is an assemblage of *Joint* SI which control the wrist orientation. The exact nature of these two assemblages will depend on the nature of the physical mechanism. The object model *RObject* filters general position and orientation to provide task oriented information:

```
[RObject (Pn On)-(Pt Ot)        ;; Name and Ports
 (handlength grotate)           ;; internal variables, set up on instantiation
 (Pt:=Pn-handlength             ;; Decrement target location by hand size
  Ot:=grotate·On)]              ;; Rotate object coordinate frame to suit task
```

This allows for some static hand offset *handlength*, and rotates the object coordinate system (grasp dependent) by *grotate*. Initialization of *Reach* must of course set these filtering variables up. We can define *Reach* more fully as:

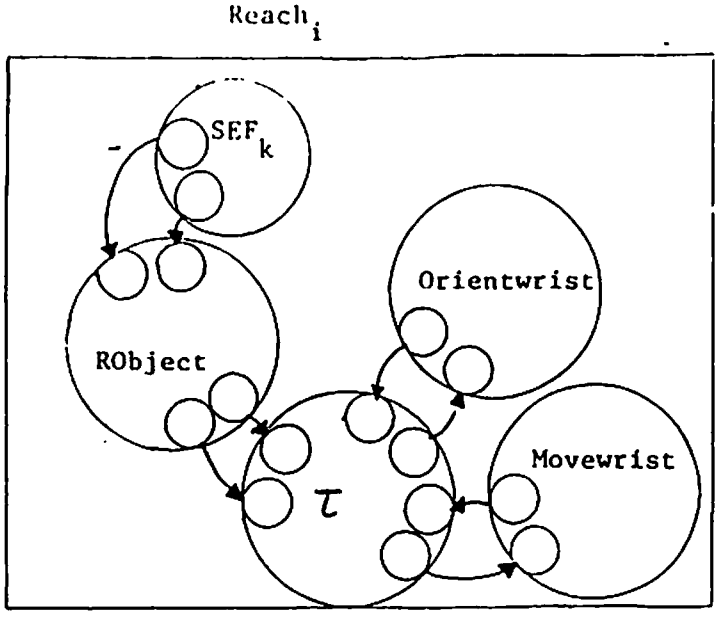


Figure 4: The Reach Task-Unit Schema.

```
[Reach (M_P M_O O_P O_O) (Pos Ornt)                    ;;Ports
 ( RObject SEF)                                        ;;Sensory Components
 ( MoveWrist OrientWrist)                              ;;Motor Components
 ( RObject(hlength,GRot)                               ;;setup object model
   (O_P ←RObject(Pt), O_O ←RObject(Ot),
    RObject(Pn)←SEF_k(F_1),
    RObject(On)←SEF_k(F_2))                            ;;connect to SEF_k
  MoveWrist ()                                         ;;Set up Movewrist
   (M_P ←MoveWrist(actual), MoveWrist(desired)←Pos)    ;;connect motor o/p
  OrientWrist ()                                       ;;Set up Orientwrist
   (M_O ←OrientWrist(actual), OrientWrist(desired)←Ornt) ;;connect motor o/p
 (hlength GRot Ptemp Otemp del)                        ;;local var of reach
 (ptemp:= M_P                                          ;;read current pos
  Otemp:= M_O                                          ;;and orientation
  IF Ptemp≠O_P THEN Pos:=Ptemp+Del ENDIF               ;;move to object
  IF Otemp≠O_O THEN Ornt:=Otemp+Del ENDIF              ;;and orient
  IF (Ptemp=O_P)AND(Otemp=O_O) THEN STOP ENDIF)]       ;;terminate reach
```

Example III

Now we consider the problem of triggering some task-unit T_j when a certain object, or object configuration, is perceived. In our example we shall just use visual sensing, and indicate how this can be extended easily for any sensory description. The object is characterized as a set of visual features; values of the feature ports of an SEF (5). We define a precondition *Object?* whose function is to search in parallel all SEF instances for one whose feature port values correspond to values of internal variables of *Object?*. *Object?* is structured to search all SEF instances in parallel using the **FORALL** statement. *Object?* creates one instance of another SI *Objtest* for each instance of SEF, the internal variables of *Objtest* are initialized to describe the desired object, and the ports of *Objtest* are connected to ports on the corresponding SEF instance. Each $Objtest_i$ also has a result output port connected to the input port *result* on *Objects?* (a fan-in situation). Each $Objtest_i$ simply tests the values it receives on its input ports against the values stored in its internal variables; if they match, a one is written to the result port, otherwise a zero is written to the result port. In either case $Objtest_i$ terminates after testing SEF_i. Having started all *Objtest* instance, all *Object?* need do is to wait for a one to be written to its input port *result*, and instantiate T_j when that happens. We can describe this as:

```
[Objtest (F1 ... F4) (judgement)                ;;
(DF1 ... DF4)                                   ;;desired feature vector of object
(IF (F1 =DF1) AND ... (F4 =DF4)                 ;;Object Test
  THEN judgement:=1                             ;;Object matches description
  ELSE judgement:=0                             ;;No match with Object
ENDIF                                           ;;
STOP)]                                          ;;terminate SI

[Object? (result) ()                            ;;
(T j                                            ;;task unit specification
  DF1 ... DF4)                                  ;;feature vector of desired object
(FORALL SEFi DO                                 ;;start all Objtest SI
   Objtesti (DF1 ... DF4)
   (Objtest(F1 ... F5) ← SEF1(F1 ... F4),   Object(result) ← Objtesti(judgement))
 ENDFOR
 IF result=1 THEN                               ;;wait for a match
  Tj(...)(...)                                  ;;if matched, setup and
  STOP                                          ;;start the task-unit
 ENDIF)]                                        ;;
```

Essentially this is an example of the parallel search problem we discussed earlier on. Note that *Object?* will cycle through its behavior section continually creating *Objtest* processes. The

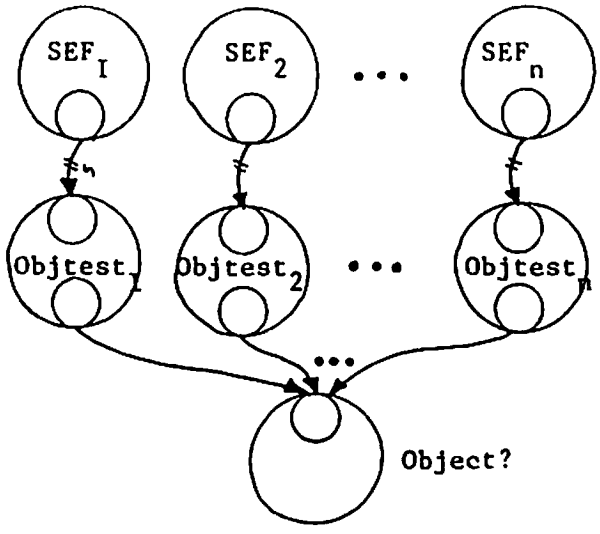


Figure 5: The parallel search initiated by Object?

unnecessary proliferation of *Objtest* processes is stopped since *Object?* specifies the instantiation number of *Objtest*. In this way if *Object?* tries to create more than one SI to examine some SEF_i, we can recognise the case.

To connect the instanced task-unit to the triggering SEF requires a different precondition: we can do with by a minor modification to the previous example. *Object?* creates one instance of *Objtest* for each instance of SEF and then dies. Each *Objtest* tests its SEF: If it fails the test the *Objtest* SI dies; otherwise the *Objtest* SI creates the task unit. Since every particular instance of *Objtest* is connected to only one SEF, the task-unit can be connected to the triggering SEF. We can combine both of these preconditions and have a precondition which waits around for some object to exist and when it does, creates an instance of some task unit T connected to the SEF describing the object. It is easy to generalize the nature of *Objtest* so that it tests not only some SEF, but also a combination of tactile or position data etc.

Example IV

Up to now we have been describing networks in terms of motor actions such as the control of a particular joint or finger. However logical grouping is also useful for motor specifications. For example, the virtual finger mechanism allows us to specify motor actions in terms of some *logical* unit, the VF, which will later be mapped on to a real mechanism. Assume we have some task unit N,

$$N = [T_i - F_i] \tag{12}$$

written in general terms of some finger control SI F_i, where i indicates which physical finger (according to some numbering of the robot mechanism) is being controlled, and is represented by some internal variable in N. The allotment of physical fingers to virtual fingers is a function of object and hand characteristics. The task to be accomplished is described in terms of VFs, which are mapped to appropriate physical fingers when the task is executed for a particular object(s). So if N describes a task to be done, then the function of the virtual finger mechanism is to instantiate N for each finger F_j which is included in some designated virtual finger VF_k.

Let us define a schema VFk, where k indicates which virtual finger VFk will be used to represent. Let VFk have no input or output ports, no internal variables, and have the behavior of a null process.

$$[VFk()()()()] \tag{13}$$

We indicate that some (not necessarily consecutive) set of fingers are to be included in VF_k:

$$f_k = \{i \mid F_i \textit{ is considered to be in } VF_k \textit{ for this task}\} \tag{14}$$

by making a corresponding instance of VFk for each $i \in f_k$. The virtual finger to physical finger mapping is accomplished by making appropriate *local* instantiations of VFk. For example if $VF2_1$, $VF2_2$ and $VF2_4$ exist in some local context that indicates that for this context $f_2 = \{1, 2, 4\}$. The **FORALL** statement can be used to make one instance of N for each instance of VFk:

$$(FORALL\ VFk_i\ DO\ N(i)()\ ENDFOR) \tag{15}$$

If for example $k = 2$ and $f_2 = \{1, 2, 4\}$, then (15) will result in three instantiations of N with i equal to 1,2 and 4 respectively. Since i determines which finger N controls, (15) has the required virtual to physical mapping behavior.

6. Conclusion

In analysing our solution of the test examples our criteria are how well the model brings out the parallelism inherent in the task and in the robot mechanism, and how well the task is represented. For the moment we are not interested in the brevity of our programs; if we are convinced that our model captures the appropriate semantics for the robot domain we can introduce appropriate syntax later.

In the first example our model representation correctly breaks down the problem into its most concurrent form: a process to evaluate the sensory termination condition continuously and one to persue the motor action continuously, linked by a monitoring process. The grouping constructs we build into the model facilitate extension; with the use of a port array to abbreviate syntax we can extend our first example to deal with any number of sensory process, and any number of motor processes.

One advantage we see with our model is the way in which the task-level concept of object naturally occurs. An object is defined only within a task context, and acts as a 'filter' on sensory data; conversly each task has an implicit 'expected' object description. The concurrency inherent in this is a simple extension of the previous example; the object process is continually active gathering and filtering the most up to date sensory information, the motor process is then fed data in the most appropriate form. We can, as for the previous example, extend the task-unit to any number of objects and motor processes. However in addition we can suitably nest object representations; the resultant object model (in the AI sense) is an active structure of linked SIs spanning a set of task-units. Since the object model is incrementally constructed in this fashion we have no difficulty in deciding what to put in an object representation. What a human may regard as the same object however, may generate many different object models, depending on how it is to be used in a task. This structure also simplifies somewhat the represention of multi-sensory object models; our example deals with reaching, at the gripping stage however a task object model would include tactile expectations as well.

Our third example starts to deal in areas discussed more usually with respect to AI systems; searching for objects or types of objects. We feel this is the area where our model can best make contact with AI models of behavior. Our search technique brings out all the inherent parallelism of the problem. We can extend the object search to include any number of different (possibly mult-modal) sensor readings, each of which we can check in parallel.

Our final example implements a crucial concept in our dextrous hand work, the virtual finger. In it we use an *active set* notation to represent the members of a virtual finger; in this fashion we create the process network for each finger in the VF in parallel. For our purposes here we define a virtual finger to have the same degrees of freedom and structure as a physical finger, this simplifies exposition. It is easy to see that we could make the logical unit as abstract as we wish.

Having satisfied ourself that our semantics is appropriate to represent the complex robot domain our next step is the formal exploration of behavior. We were careful to construct our model with formal semantics corresponding to the port automata model. For purposes of verification we shall utilize this basis to represent programs in our model and associate axioms with each of the special schema to reason about the effects of programs.

Our model represents the four test examples quite well. Currently the syntax of the model is very verbose; this was intended, and provides the platform from which we will explore behavior formally. However for specification of behavior, a less expansive notation would be favored. In essence we feel our model captures the appropriate semantics for the robot domain.

REFERENCES

[1] Arbib, M.A., "Perceptual Structures and Distributed Motor Control," in V.B. Brooks (Ed.), *Handbook of Physiology — The Nervous System, II. Motor Control*, Bethesda, MD: American

Physiological Society, 1981, pp. 1449–1480.

[2] Arbib, M.A., Iberall, T., and Lyons, D., "Coordinated Control Programs for Movements of the Hand," *Exp. Brain Res. Supplements* 10, 1985, pp. 111–129.

[3] Arbib, M.A., Overton, K.J., and Lawton, D.T. "Perceptual Systems for Robots". *Interdisciplinary Science Reviews*, 9, pp 31–46.

[4] Brady, M., "Artificial Intelligence and Robotics" *MIT A.I. Lab Memo* 756, February 1984.

[5] Geschke, C., "A System for Programming and Controlling Sensor-Based Robot Manipulators," *IEEE Trans. on PAMI*, Vol. PAMI-5, No. 1, Jan. 1983, pp. 1–7.

[6] Henderson, T.C., Wu, S.F., Hansen, C., "MKS: A Multisensor Kernel System" *IEEE Trans. SMC* Vol. SMC-14 No.5, 1984, pp. 784–791.

[7] Iberall, T, and Lyons, D.M., "Towards Perceptual Robotics" *1984 IEEE Conference on Systems, Man and Cybernetics* Nova Scotia, October 9-12, 1984.

[8] Lozano-Perez, T., "Automatic Planning of Manipulator Transfer Movements" *IEEE Transactions on SMC.* Vol.SMC-11, No.10, October 1981, pp 681 – 698.

[9] Lozano-Perez, T., "Task Planning," in: Robot Motion Planning and Control, *MIT Press* Cambridge Ma, and London England, 1983.

[10] Lyons, D.M., "A Simple Set of Grasps for a Dextrous Hand," *Proceedings of the 1985 International Conference on Robotics and Automation*, Mar. 25–28, 1985, St. Louis.

[11] Lyons, D.M., and Pocock, G. "A Distributed Network for Control of the Salisbury Hand" *LPR Internal Memo* 8. 1984.

[12] MacQueen, D.B., "Models for Distributed Computing," *Rapport de Recherche*, 351 RIA Laboria, Rocquencourt, France, April 1979.

[13] Overton, K.J., "The Aquisition, Processing, and Use of Tactile Sensor Data in Robot Control," Ph.D. Thesis, Dept. of Computer and Information Science, University of Massachusetts, 1984.

[14] Raibert, M., "Legged Locomotion – The Robotics of Running" *SIAM Conf. on Geom. Modelling and Rob.* July 15-19th, Albany NY., 1985.

[15] Steenstrup, M., Arbib, M.A., and Manes, E.G., "Port Automata and the Algebra of Concurrent Processes," *Journal of Computer and System Sciences*, Vol. 27, No. 1, Aug. 1983, pp. 29–50.

[16] Shatz, S.M., "Communication Mechanisms for Programming Distributed Systems" *IEEE Computer* Vol. 3 No.4 June 1984, pp 21 –27.

[17] Tsukune, H., "A Formalization of the Cooperative Activity of the Sensor Motor System" *Bul. Electrotech. Lab.* Vol.43 Nos. 7&8, 1979, pp 34-53, 1979.