Philips TR-87-092. Accepted for Publication, IEEE J. Robotics & Automation

A Formal Model of Computation for Sensory-Based Robotics

D.M. Lyons† and M.A. Arbib‡

†*Dept. of Robotics & Flexible Automation*
Philips Laboratories
Briarcliff Manor, NY 10510

‡*Dept. of Computer Science*
University of Southern California
Los Angeles, CA 90089-0782

ABSTRACT

Almost all attempts to construct special purpose robot programming languages have proceeded by taking a computer programming language and adding some special primitives. We have taken the unique approach of trying to define computation at its most primitive level in terms of the characteristics of the robot domain. We construct a special model of computation, called $\mathcal{RS}$ (Robot Schemas) with properties designed to facilitate sensory-based robot programming. Our approach offers the potential to construct robot task representations which are easy to use, concise, and which execute in an efficient manner. We define the model formally using *port automata*. These definitions ensure consistency and well-definedness, but they also facilitate plan verification and automatic plan generation.

Preparation of this paper was supported in part by grant number DMC-8511959 from NSF (K.Ramamritham and M.A. Arbib, Co-Principal Investigators) to the Laboratory for Perceptual Robotics at the University of Massachusetts at Amherst.

1 Introduction

One common approach to robot programming is to use a general purpose programming language [12]. This has the advantages that such languages are well understood, they offer a large array of control and data structures, and they can be quite portable. Another, complementary approach is to look deeply at what is unique about robot programming, and try to develop a *model of computation* based solely on these characteristics. We adopt the latter approach because it offers the potential to construct task level specification mechanisms which are easy to use, concise, and which execute in an efficient manner.

We argue that this approach has never been attacked deeply enough; many so-called special purpose robot languages are conventional programming languages with some new data types and procedures (e.g. VAL [31] and Basic, AL [26] and Pascal). Other special purpose languages provide insight into the nature of task-level specification [18, 20] but don't address conciseness or efficiency of execution. Our goal in this paper is to formalize the key computational characteristics of robot programming into a single mathematical model, *not* to come up with a specific programming language. We offer a model of computation based completely on the characteristics of the robot programming domain. This work provides a 'deep structure' for subsequent program or plan representation languages.

The approach offers the potential of direct task-level specification of programs (easier for humans to deal with, and automatic planners to interface to), as well as efficient run-time behavior (because the model can be built with efficiency for specific primitives in mind). There is another advantage to this approach: An additional formal level at which robot behavior can be specified and analyzed is introduced. Since a robot system is real-time, and since 'intelligent' behavior is a major goal of robotics, it is essential to have a program/plan representation which is amenable to mathematical analysis.

We begin by presenting the key computational characteristics of the robot programming domain. Our goal is to construct a model of computation that explicitly captures these characteristics concisely and efficiently. The next section incrementally constructs the model, referring back to these characteristics. The remainder of the paper is then devoted to working out the formal definition of the model in detail.

2 Characteristics of the Robot Domain

Robot programming is a unique subclass of general purpose programming because the robot needs to interact directly with a noisy, dynamic and unpredictable environment [32]. Where general purpose programs accept input and produce output, the distinguishing feature of robot programs is that the input and output are directly linked to the perception of, and interaction with, the physical environment [3, 32]. This is our most basic characteristic:

Robot programs interact with the world. (C.1)

This has two immediate consequences. Firstly, a model of computation for robotics *must* contain some facility for a *plant/world model*! Secondly, we argue that because of (C.1):

The central paradigm in robot programming is that sensory input is linked with knowledge to produce appropriate action. (C.2)

A convenient term for this type of computation is *sensorimotor computation.* It is surprising to find that the guarded move is frequently the only structure which relates sensing and action in typical robot languages[16]. A notable exception is RSS[8], a language explicitly developed for sensory-based control. In RSS, to command the robot to do some action, the programmer initiates a computing agent called a Servo Process. Each such servo process is a *combination of a sensory query and a motor action* in an infinite loop. This is more expressive than a guarded move because it describes an actively maintained setpoint *and* a strategy for achieving that setpoint — the sensory query is not just a termination condition. RSS was designed to provide a versatile interface to the robot's control system, so its constructs are low-level. We argue that this type of connection of sensing events to motor actions is relevant *at all levels of abstraction in robot programming.*

Albus [1] has suggested a hierarchical model with this sensorimotor structure. However, for different robot tasks, or in response to a changing environment, it is convenient to be able to dynamically reconfigure the way sensors and effectors are linked, resulting in a more concise and efficient robot program. Our second characteristic is:

Robot programs exhibit a flexible, hierarchical, sensorimotor structure (C.3)

Albus' system, with its separate sensing, modeling and control hierarchies and fixed set of levels, is too rigid. But recursively defining an *RSS-like* network of sensing and motor actions generates a much more versatile tool for building high-level robot programs. At any level of abstraction, the program is described as a *network* consisting of a sensory model for the task and the primitive motor actions of the task. In turn, these elements[1] themselves can be decomposed into sensory and motor networks.

If (C.2) and (C.3) are combined, then higher-level robot programs have the interesting property that object models (the sensory sub-network) only explicitly contain those aspects of the environment that are directly relevant to the task(s) at hand. We call this an *action-oriented* view of sensing. Conversely, the sensory network can be be though of as an embedded 'logical sensor' for the program. This has some similarity to the concept of *logical sensors* [13], but with one difference: the 'sensor' is integrated directly with the set of actions which use it. This approach to sensorimotor computation provides the basis for a general and efficient way to interact robustly with an unstructured environment.

The schema [3, 22] or skeletal procedure mechanism reoccurs often in robot programming. Taylor [30] discusses an approach to robot programming with sensors based on *procedure skeletons*, which represent prototypical motion strategies. Parameterizing a skeleton produces a specific instance of that motion strategy. Recently this approach has been extended by Lozano-Perez and Brooks [20] with the notion of geometric constraint propagation, to a general system for task-level planning. Objects are well represented by a prototype or template mechanism. This is particularly appropriate in an action-oriented view of sensing; an object can be represented as an instance of a 'template' whose parameters are simply that object-model data, or sensory input about the object, which is relevant for the task in progress. We list this as our third characteristic:

Robot programs are defined recursively using a schema or class structure (C.4)

It is widely agreed that multiprocessing can help in making robot programs more efficient; the real issue is how to make best use of a multiprocessor. We argue that a robot programming model should have structures which explicitly represent the *inherent* parallelism of the domain, perhaps in addition to, but distinct from, the accidental parallelism resulting from the way any specific program just happens to be written. Robot hardware is implicitly distributed — a set of actuators, each controlling a degree of freedom (DOF) on the robot, and a set of sensors constantly sampling the environment. We want to be able to represent the ways in which these can be collected into groups with common control needs[2] and controlled in parallel. In addition, it is important to be

[1]The sensory model can, therefore, contain nested motor commands, e.g., to control the panning or zooming of a camera.

[2]E.g., the grouping of the wrist vs. the arm actuators on a robot in which the position and orientation DOFs can be separated kinematically, or the fingers of a dextrous hand, etc.

able to represent *logical groupings* that can be controlled in parallel, e.g., *Virtual Fingers* [4] and *Oppositions* [15]. That is, specific actuators on the robot are identified, which can be grouped together for the purpose of simplifying the description of the ongoing task. Such a logical grouping may change from task to task.

Robot programs are inherently distributed (C.5)

Of course, the nested network task representation can also express another major source of inherent parallelism — the parallelism which exists between the gathering of the sensations necessary to construct a task model of the object, and the actions to be carried out on the object, once they are parameterized by the task model of the object. This allows us to represent the 'overlap' between sensing and action. For example, in a program which directs the robot to reach to an object, the action can be started once a coarse estimate of the object position has been computed, and the destination refined as more sensory information becomes available [3].

In order to emphasize the continuity of this work with the style of computation described informally in [3, 27] we call our model *Robot Schemas* (or $\mathcal{RS}$).

3 The $\mathcal{RS}$ Model

$\mathcal{RS}$ is a model of distributed computation embodying our nested network approach to robot programming. This section introduces the $\mathcal{RS}$ model and shows how it can be used to represent robot programs in an efficient and concise manner.

3.1 Overview

Computation is performed in a distributed model by the *interaction* of a number of *concurrent* computing agents. We will use this concurrency to bring out what we have called inherent parallism (characteristic (C.5)). A *schema* is a generic specification of a computing agent in $\mathcal{RS}$ (characteristic (C.4)). We shall always write a schema name in `teletype` font. A computing agent is created from a schema by the instantiation operation, and the term *schema instance*, or SI, denotes a computing agent. Instantiation creates an SI, sets some initial parameter values, and connects the SI to other SIs. Using the schema `Joint` as an example, by $\texttt{Joint}_{i=a}$ we denote an SI in which some variable i has been initialized to value a. When an SI has only one parameter which can be initialized, and the meaning is clear, we shall abbreviate this to $\texttt{Joint}_a$.

In order to built the sensorimotor structure demanded by (C.2), we need to have SIs communicate with each other. Each SI has a set of communication objects called *ports*. At instantiation, connections can be specified between these ports and ports on other SIs. We write an SI of the schema `T` as $\mathtt{T}_V(i_1, i_2, \ldots)(o_1, o_2, \ldots)$, where V describes how the variables of the SI have been initialized, the i's are input port names, and the o's are output port names. A data type is associated with each port. Only ports of the same type can be connected together. Communication occurs when an SI writes a value to one of its output ports, which has been connected at instantiation time to an input port on some SI, and the value is subsequently read by that SI from its input port.

The practical issues of communication in a robot workcell has received attention recently [10, 28]. Since our emphasis is on the construction of a mathematical model, we choose a communication approach which can be easily represented formally, but which yields all the kinds of communication necessary. This communication mechanism is a form of *synchronous message passing* [2]. The combination of synchronous communication and instantiation allows us to support synchronous with timeout, and asynchronous, communication (we show how later in the text).

Associated with each schema is a *behavioral description* that defines how an instance of that schema will behave in response to communication. In many cases this description will actually be a network of other SIs. To ground this recursion, we provide a simple procedural behavior specification language in Appendix A. Schemas defined using this syntax are called a *basic schemas.*

We use the notation $(\mathtt{A}_i, \ldots)^C$ to indicate that the SIs $\mathtt{A}_i, \ldots$ are connected together as described by the port to port connection map C. With (C.2) as our motivation, we define an $\mathcal{RS}$ program to be a network of SIs, some of which collect and process sensory data, some of which control robot motion. Such a network may *grow* and *shrink* as computation proceeds (a *dynamic* network).

<u>Example: Position Control Network</u>

Consider a simple, robot-level example: servoing a robot joint to a particular position. Let us assume that this can be characterized by the equation $u_i = PC(x_i - \bar{x}_i)$ where u_i is the motor control signal to joint i, x_i the current position of joint i, $\bar{x}_i$ the desired position, and PC some position control procedure. In $\mathcal{RS}$, this task is represented by a network of three SIs: one for each of the components in (C.2). Let `Jpos` be a schema for reading joint positions, which has an internal variable *joint* to determine which joint it inspects. $\mathtt{Jpos}_{joint=i}$, or for simplicity, $\mathtt{Jpos}_i$, is an instance of `Jpos` parameterized to read the *ith* joint. Let `Jpos` have a single (output) port x in which it (continually) writes the current position; we denote this $\mathtt{Jpos}_i()(x)$.

Let there also be a schema `Jmot`, which has a single internal variable *joint*, and which (continually) accepts through its only (input) port u a motor signal for joint number i and sends the signal to the motor; we denote this $\mathtt{Jmot}_i$ (u)(). Computing agents such as these, whose behavior is defined only in terms of the effect they have on the robot mechanism, or the effect the state of the external world has on them, are called *primitive motor* or *sensory SIs*, respectively.

Figure 1: A Position Servo Network

These agents are the only sensory input and motor output 'channels' allowed; they could be pieces of hardware, for example. In the formal analysis of programs/plans, the sensory and motor schemas define the *plant/world model* with which the plan interacts (characteristic (C.1)).

The connection between sensing and action can be implemented by a third SI $\mathtt{Jset}\ _{i,\bar{x}}(x)(u)$, which (continually) takes the value on its input port x and writes $PC(x-\bar{x})$ to its output port u, where $\bar{x}$ is an initialized variable of `Jset` giving the desired goal position. We write this network as (Figure 1):

$$(\ \mathtt{Jpos}_i()(x),\ \mathtt{Jset}_{i,\bar{x}}(x)(u),\ \mathtt{Jmot}_i(u)()\ )^C$$

$$C:\quad (\mathtt{Jpos}, x) \mapsto (\mathtt{Jset}, x),\quad (\mathtt{Jset}, u) \mapsto (\mathtt{Jmot}, u)$$

This simple network of three connected computing agents describes the essence of task representation in $\mathcal{RS}$; how characteristics (C.2) and (C.5) are incorporated. Implicit parallelism in the task is released by 'overlapping' the control and sensing components as concurrent processes. The specific relationship between sensing and action (in this case, a rather simple one) is implemented as communication constraints between the computing agents representing sensing and those representing action.

3.2 Nested Networks: The Assemblage Construct

Constructing complex robot programs would be very difficult if it was necessary to work with primitive schemas all the time. One of the most important constructs in $\mathcal{RS}$ is, therefore, the *assemblage mechanism*, and this is where characteristic (C.3) comes in. An *assemblage* is a network of SIs that externally appears to be a single SI. An *assemblage schema* is a generic specification of a network of SIs. The position control network could be rewritten as some assemblage `Jmove` as follows:

$$\mathtt{Jmove}_{i,\bar{x}}()(x) = [\ \mathtt{Jpos}_i()(x),\ \mathtt{Jset}_{i,\bar{x}}(x)(u),\ \mathtt{Jmot}_i(u)()\ ]^{C,E}$$

The use of square brackets denotes that the network is an assemblage. Two important pieces of information need to be specified to make a network into an assemblage: how does the initialization of local variables of the assemblage schema affect the initialization in each of the component schemas, and how are the ports of the assemblage schema related to the ports of the component schemas. The effects of variable initialization are denoted by relating the subscript of the assemblage schema name (LHS of the assemblage specification) to the subscripts on the component schemas (RHS of the specification). For example, `Jmove` is constructed to have variables initialized to i and $\bar{x}$, which causes the *joint* variables of `Jpos, Jset` and `Jmot` to be initialized to i, and the goal position variable of `Jset` to be initialized to $\bar{x}$. The port equivalence map E specifies the relationship between the ports of the assemblage schema and ports of the component schemas. In our example, E for `Jmove` is: $E : (\texttt{Jpos}, x) \mapsto (x)$. `Jmove` can now be treated the same as any schema. When an instance of `Jmove` is created and initialized, it internally sets up the position control network. By our specification of `Jmove,` its output port x simply echoes the current position information available on `Jpos.`

3.3 High-Level Robot Programming

By high-level robot programming we mean the specification of robot programs at a high level of abstraction (i.e., lack of detail); humans find it easier to specify correct programs in this manner (perhaps because it is the level at which they deal with each other [5]). The basis of high-level robot programming is that actions are referenced against objects, rather than against manipulator parts[19]. The assemblage construct gives us a tool to generate complex, task-specific object models, consisting of teams of SIs which cooperate to generate a current analysis of the object state.

For example, a single entity `Bolt` can be constructed which communicates standard information about a bolt, e.g., position, orientation, length and thread size. Internal to `Bolt` , this information could be garnered from multiple sources, e.g., position and orientation might come directly from visual data, length might come from a geometric (CAD) object model, and the thread size might come from tactile data (Figure 2). In addition, the sources could interact via network connections to decided on a 'consensus' view of the object.

3.4 Robot Programs as Task Plans

A *task plan* is a set of instructions necessary to achieve some goal. A task plan corresponds to a particular kind of robot program. For example, the `Jmove` program is a task plan, whereas the `Bolt` program is not. A task plan at a high-level of abstraction corresponds to what is normally

Figure 2: An Example Object Model as a Network of SIs

considered a *task-level* robot program [18, 19, 30].

The most general way to describe a task plan is as the relationship between a task-specific object model and a set of actions[3]. The task is carried out on a particular object by applying the task-specific object model to the target object. For example, the task-specific object model for the task of opening a door might just supply the radius of the door, and the position of the handle relative to the hinge line (determined by what information the actions in the task plan need). This plan can now be applied to any object for which the task-specific object model can pull out the necessary information.

A task plan is represented in $\mathcal{RS}$ is a structured assemblage — an assemblage with the following components: a set of sensory SIs, which compose the task-specific object model; a set of motor SIs, which represent the actions available in the task; and a set of task SIs, which implement the connection between sensing and action. In $\mathcal{RS}$ notation, where `T` is the task plan assemblage, called the *task-unit*, `S` the sensory schemas, `M` the motor schemas, and `t` the task schemas:

$$\mathtt{T} = [\ \mathtt{S},\ \mathtt{t},\ \mathtt{M}\ ]^{C}$$

Clearly, the components of this task plan assemblage can, themselves, be assemblages — generating a sensorimotor task hierarchy. This structure integrates the logical sensor concept (here equivalent to the sensory model) [13] into the task it is logically part of. Notice the two main improvements to the Albus [1] sensory motor hierarchy: Since the sensory-motor division is *internal* to each task description, the number of hierarchical levels in one task does not constrain the number in any other task. Secondly, each level of the hierarchy is a network: SIs at each level can communicate directly with their siblings. This property will become relevant in the next section when we discuss action sequences. Section 5 of this paper presents an example of of task-unit programming for a standard robotic problem, the so-called centered grasp.

3.5 Parallel, Sequential and Alternative Actions

Our task plan description $[\ \mathtt{S},\ \mathtt{t},\ \mathtt{M}\ ]^C$ is weak because it is very general. A stronger description can be made by explicitly including sequencing and decision information which is 'hidden' within the behavior of the task schema, `t`. The key concept used to represent sequencing information is the *precondition operation*, written ':', which takes a precondition schema (a schema that tests some condition) and, based on the result, decides whether and how to instantiate (informally, 'trigger') a specified *consequent* schema. The precondition schema can also choose values of the parameters with which the consequent schema is instantiated. This is written

$$\mathtt{Pre}_h :^C \mathtt{T}_v$$

where h is the test to be carried out, and if it succeeds $\mathtt{T}_v$ is instantiated and connected according to C. This most general statement of the precondition will not be used very often. If, by dint of name and explanation, h and C are clear, we shall omit them. A wide variety of action sequencing can be represented using the precondition schema. The key idea is that the precondition schema is a parallel, decision making mechanism, which can be used in a number of useful robot task plans.

<u>Sequential Preconditions.</u>

If two actions are to be activated sequentially (i.e., the second instantiated when the first deinstantiates) then in a sense, the second can be considered as the consequent of a special precondition schema which detects the deinstantiation of the first. This special sequential precondition will be denoted by ';'. Two task-units are established as sequential by writing

$$\mathtt{T1} ;^C \mathtt{T2}$$

where C describes the connections for the consequent schema.

<u>World Preconditions.</u>

An important form of temporal ordering in a task is the synchronization of actions with the environment, i.e., grasping a part when it appears from the feeder, starting a fine motion when the end-effector is in some defined spatial envelope. The so-called oportunistic scheduling of [7] is one example of this synchronization. Precondition schemas which detect this class of condition we call world preconditions.

Figure 3: The Box Assembly Plan

The world and sequential precondition allow the description of quite complex temporal sequences, e.g. where `T`i are task-units and `W`i world preconditions:

$$\texttt{T1 ; T2 ; [T3, T4] ; [W5 : T5 ; T6, W7 : T7]}$$

The 'activation' order above is left to right. Commas indicate concurrent task-units (as they have all along), and square brackets are used to group networks. Using these, task plans can be represented as a partially ordered sequence of actions. The partial ordering is controlled by the preconditions, which are parameterized at execution time to yield a set of sequential and parallel actions.

Consider an example of a task plan for the assembly of a box composed of a base, four sides, and a lid. In this example, the sides can be placed on the base in any order; the only part ordering constraints are that the base be first and the lid last. Let us call the precondition schema which recognizes an instance of the base in the world, `Base`, and similarly for `Side1`, `Side2`, `Side3`, `Side4`, and `Lid`. Let us call the schema which implements the placing action `Place`. We will assume it can be parameterized by the recognition preconditions to accept a particular object o and destination d, i.e., $\texttt{Place}_{o,d}$. The program implementing the assembly task plan is:

$$\begin{array}{l}
\texttt{Base : Place}_{o,d}\ ; \\
\qquad\qquad [\ \texttt{Side1 : Place}_{o,d}\ ,\ \texttt{Side2 : Place}_{o,d}, \\
\qquad\qquad\ \ \texttt{Side3 : Place}_{o,d}\ ,\ \texttt{Side4 : Place}_{o,d}\ ]\ ; \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Lid : Place}_{o,d}
\end{array}$$

The sequence in which the base and lid are placed is rigidly determined using the sequential preconditions. However, the order in which the placing actions are triggered is determined solely by the order in which these parts arrive (e.g., [7]) or are recognized (e.g., if they were in a jumbled heap). Note that all the side placement actions could be triggered simultaneously; but, of course, if there are fewer than four robot arms, they cannot all be executed concurrently. The notion of *resource precondition schemas* will be introduced to deal with the resource allocation problem.

Selection Preconditions.

A *selection precondition* is a class of precondition which selects between consequent schemas on the basis of some test. A good example of this is grasp selection for a *dextrous robot hand.* A dextrous hand can usually grip an object in more than one configuration. There are a number of attempts in the literature to determine a best configuration based on knowledge of what is to be done with the grasped object, as well as the characteristics of the object itself[3] [6, 15, 24].

Let there be n grasps available, each represented by a task-unit schema $\texttt{Grasp}^{\,i}$, for $i \in \{1, \ldots, n\}$. Each grasp has an associated test which determines when it is best to apply this grasp; let us call this $\texttt{Gtest}^{\,i}$, for $i \in \{1, \ldots, n\}$. We can write the relationship between grasp and test as

$$\texttt{Gtest}^i : \texttt{Grasp}^i \qquad i \in \{1, \ldots, n\}$$

It is important to see that this does not necessarily mean that the tests are independent of each other; the preconditions could be connected in a distributed decision network. In that case, it might make more sense to write a single common precondition: $\texttt{Gtest} : \texttt{Grasp}^i$.

Resource Preconditions.

The resources available to a robot program consist of manipulators, sensors, objects, etc. Resource allocation is a crucial notion in a distributed programming model. In the box assembly example, we had the situation where up to four actions might simultaneously compete for the robot arm. Our solution is to introduce a class of preconditions to deal with shared resources. The resource precondition schema has as its test, a query to determine if the shared resource can be safely allocated. Once the consequent schema has terminated, the resource is freed.

For example, the `Place` task-unit schema might internally have as part of its program

$$\texttt{Robot}x : \texttt{Reach}_r$$

where $\texttt{Robot}x$ is a resource precondition which triggers `Reach` when a robot of some target class x can be safely allocated. The parameter r tells `Reach` which robot has been allocated; it might, for example, be the name of a schema which when instantiated will provide a link to control the robot.

[3] [21] Chapter 2, describes a simplified example of this selection mechanism in detail.

4 The Formal Definition of $\mathcal{RS}$

In this section, a concise meaning is constructed for the components of the $\mathcal{RS}$ model. This is an important, if detailed, step for two reasons:

1. We can eliminate any inherent contradictions in our model.
2. We can use these definitions as the basis for future work in
   (a) Establishing that task plans behave in a specific way.
   (b) Reasoning about equivalent plans.
   (c) Automatically generating task plans.

Defining a formal semantics consists of specifying mathematical objects which define entities in $\mathcal{RS}$ concisely. This construction renders our model well-defined in the sense that details omitted from the informal semantics can be inferred from the mathematical semantics. Thorough analysis and exploration of the model cannot proceed satisfactorily until such a formal basis exists. But a formal basis is also necessary for the verification and concise specification of task plans, or for establishing guarantees about their behavior. It also much simplifies the construction of correct processor-level or chip-level implementations of the model itself.

There are many mathematical models of concurrency, e.g., [14, 25, 29] to name but a few. An operational semantics is constructed for $\mathcal{RS}$ based on the *Port Automaton Model* (PA) of Steenstrup et al. [29]. We chose the PA model because it offers an intuitive and elegant way to characterize the concept of an SI, and the way in which SIs may interact with each other. In addition, it provides a composition definition which facilitates the definition of the assemblage. We begin by considering the semantics of the *basic* schema.

4.1 Basic Schemas

A *basic schema* description consists of the following: a list of input and output ports, an internal local variable list and a behavior section. The behavior section is a program which loops continuously, once the schema has been instantiated as an SI, until that SI *deinstantiates*. The program instructions can synchronously read from, or write to, the ports, access internal variables, instantiate other *schemas*, or deinstantiate. We formalize these statements with the programming-style syntax definitions below.

Definition 1 A basic schema description is:

$$basic - schema ::= [\begin{array}{ll} Schema - Name : & \langle N \rangle \\ Input - Port - List : & (\langle Iplist \rangle) \\ Output - Port - List : & (\langle Oplist \rangle) \\ Variable - List : & (\langle Varlist \rangle) \\ Behavior : & (\langle Behavior \rangle) \quad] \end{array}$$

where

- `N` is an identifying name for the schema.
- `Iplist, Oplist` are lists of $\langle Portname \rangle$:$\langle Porttype \rangle$ pairs for input and output ports respectively.
- `Varlist` is a list of $\langle Varname \rangle$:$\langle Vartype \rangle$ pairs for all internal variables.
- `Behavior` is a specification of computing behavior.

Any component other than the name may be absent, and when relevant the omission is indicated by ( ).

Nothing we have said constrains the way the computing behavior of the basic schema is implemented. It could a neural network, for example, or a VLSI chip. We have chosen a simple procedural language to specify the computing behavior of the basic schema (details in Appendix A).

4.2 Instantiation-Free Semantics

We can now define the instantiation-free semantics of a basic schema; i.e., the semantics of a basic schema which does not contain an instantiation or deinstantiation statement. This simple class of schemas is a good starting point. The Port Automaton model of [29] is the basis of our semantics. However, we modify the PA model to represent unidirectional ports, synchronous communication, and a network automaton definition.

Definition 2 A *port automaton* is a collection of objects and maps

$$P = (L_x, L_y, Q, X, Y, \tau, \delta) \tag{1}$$

where,

L_x is the set of *input ports*,

L_y is the set of *output ports*, and we use L to denote $L_x \cup L_y$,

Q is the set of *states*,

$\tau \subset 2^Q$ is the set of *initial states*,

$X = (X_i : i \in L_x)$, where X_i is the *input set* for port i,

$Y = (Y_i : i \in L_y)$, where Y_i is the *output set* for port i,

$\delta : Q \times \hat{X} \to 2^{Q \times \hat{Y}}$ is the *transition function*, where $\hat{X} = \Pi_{i \in L_x}(X_i \cup \{\#\})$ and $\hat{Y} = \Pi_{i \in L_y}(Y_i \cup \{\#\})$. A value of $\#$ in a tuple of $\hat{X}$ or $\hat{Y}$ indicates there is no input or output value at the designated port.

We introduce the notation $\bar{x} \le x$ on $\hat{X}$ if $x = (x_1, \dots, x_n), \bar{x} = (\bar{x}_1, \dots, \bar{x}_n)$ and for each i either $\bar{x}_i = x_i$ or $\bar{x}_i = \#$. The *full-state transition map* is

$$\varphi : \hat{X} \times Q \times \hat{Y} \to 2^{\hat{X} \times Q \times \hat{Y}}$$

This map formalizes the way an automaton can process an input port or send values to an output port.

Definition 3 $(x', q', y') \in \varphi(x, q, y)$ just in case there exists $\bar{x} \le x$ such that there exists $(q', \bar{y}) \in \delta(q, \bar{x})$ with the following conditions:

$$x'_i = \begin{cases} x_i, \ if \ \bar{x}_i = \# \\ \#, \ if \ \bar{x}_i = x_i \neq \# \end{cases} \qquad y'_i = \begin{cases} y_i, \ if \ \bar{y}_i = \# \\ \bar{y}_i, \ if \ y_i = \# \end{cases}$$

where for all i, $y_i \neq \#$ implies that $\bar{y}_i = \#$.

That is, to be a candidate successor state to any full-state, there must be a state transition on a 'reduced' input tuple which reads a subset of the non-# entries, and given that all the necessary output ports are free, the destination state has these ports written to. This formalism will represent *read-only* ($\bar{y}$ is all #), *write-only* ($\bar{x}$ is all #), and *internal* ($\bar{x}$ and $\bar{y}$ are all #) transitions. This is the formal meaning for an SI reading an input port and writing an output port. The internal transitions are useful for describing state changes not caused by communication (e.g. iteration).

Note that external agents (other PAs connected to this one) can also change the full state, either by changing a # to a non-# value in $\hat{X}$ (writing to an input port), or by changing a non-# value to a # in $\hat{Y}$ (reading from an output port). To complete our communication definitions we need to formalize how these external communications occur.

A behavior of this PA is any sequence of input reads and output writes consistent with possible changes of full state:

Say $(x', q', y') \in R(x, q, y)$ ('R' for read) if $q' = q, y' = y$ and there is a j with $x'_j \neq \#, x_j = \#$ and $x'_i = x_i$ for $i \neq j$. We call (x'_j, j) the *read* of the transition.

Say $(x', q', y') \in W(x, q, y)$ ('W' for write) if $q' = q, x' = x$ and there is a j with $y'_j = \#, y_j \neq \#$ and $y'_i = y_i$ for $i \neq j$. We call (y_j, j) the *write* of the transition.

Definition 4 A sequence ((x_t, q_t, y_t)) of full states is *admissible* if for each t, (x_t, q_t, y_t) belongs to one of $\varphi(x_{t-1}, q_{t-1}, y_{t-1})$, $R(x_{t-1}, q_{t-1}, y_{t-1})$ or $W(x_{t-1}, q_{t-1}, y_{t-1})$.

A *behavior* of the PA is then any sequence, in order, of of reads and writes of an admissible sequence of full states.

In order to satisfy ourselves that no power has been lost from [29] by the introduction of separate input and output ports, it is necessary to show that for any port automaton with bidirectional ports, it is possible to construct a port automaton with unidirectional ports which, in some defined sense, 'does the same computation', and vice versa. This is presented elsewhere [21]. The next step is to construct the mapping from the basic schema syntax on to the port automaton model developed here. We start by defining what a *state* of a basic SI is.

The State of a Basic SI.

Assuming the serial program approach to defining a basic schema given in Appendix A (the assemblage definition will then introduce concurrency), the internal state of a basic SI can be constructed using the standard approach based on statement labels and the values of internal variables. The internal variables of schema `N` can be listed by name $(V_1, V_2, \ldots, V_k)$, where `N` is called a $k-$variable schema. In an arbitrary instance of `N`, each variable V_i will have a value v_i. At any stage, the contents of all the variables of $\mathtt{N}_j$ can be written as a tuple[4] $(v_1, v_2, \ldots, v_k) \in \Re^k$.

We assign a unique label, or index, to all the statements in the program description part of `N` from the set of natural numbers $\mathcal{N}$, by scanning the statements of `N` in sequential fashion, starting at the first statement after the opening parenthesis of the behavior specification, and finishing with the last statement before the closing parenthesis. A *state of computation* for an instance of a $k-$variable schema is a $k+1$ dimensional tuple over $\mathcal{N} \times \Re$. We shall also call it a *state vector*. The definition of assignment in the basic schema allows statements of the form $V := EXPR(i_1, \ldots, i_l, v_1, \ldots, v_m)$ where V is a variable or output port name, the i's are input port names, the v's are variable names, and $EXPR()$ denotes an arithmetic expression. If there are l input ports in the $EXPR()$, then we have to model the evaluation of $EXPR()$ as a multiple state transition comprising any necessary reads to the input ports, followed by $EXPR()$ evaluation. There is *non-determinism* in the ordering of the reads, since it is impossible to foretell what order the inputs will arrive in.

[4]Since all data types can be mapped onto computer memory, for technical convenience and without loss of generality, $\mathcal{T}$, the set of port data-types, can be restricted to $\Re$, the set of computable reals[17].

The Semantic Mapping

Taking into consideration those issues discussed above, we can now specify the *semantic mapping* which takes a basic schema onto that port automaton which is its instantiation-free semantics.

Definition 5 The semantic mapping from the components of a schema `N` to those of a port automaton $P = (L_x, L_y, Q, X, Y, \tau, \delta)$ is defined:

- The sets of input ports L_x, and output ports L_y, of P can be constructed as follows: For each name in the input port list of `N`, add an input port with this name to L_x. Repeat the process with the output port list of `N` and L_y.
- $X_i = \Re$ for all input ports. $Y_i = \Re$ for all output ports.
- The set of states Q of P is $Q = \mathcal{N} \times \Re^k$; where the state vector of `N` provides the element of $\Re^k$, and the statement label in `N` encodes the current statement in the behavior section. • The set of initial states of P, τ, is defined: $\tau \subset 2^{\{1\}\times\Re^k}$ It denotes any assignment of initial variable values in which the next instruction to be executed is the *first* instruction.
- The transition map δ is constructed from the behavior section of `N` in a straightforward way which need not be detailed here[5]

This concludes our definition of the instantiation-free semantics of basic schemas. Most other schemas will, however, need to use instantiation. The instantiation-free semantics gives us an appropriate starting point to consider these more complex schemas.

4.3 The Full Semantics

An *augmented* port automaton is a port automaton which can execute a state-transition which is an instantiation operation, i.e., connecting in a copy of another specified port automaton to specified ports, and a state-transition which is a deinstantiation, i.e., the removal of the connections of a port automaton to specified ports.

The Network Automaton

Steenstrup et al. demonstrate that a port connection automaton (PCA), two port automata with some of their ports connected together, is also a port automaton. If P^1 and P^2 are the component automata, and they are connected by map c, then the PCA is written $P^1\|_c P^2$. Their proof consists of constructing the PCA in terms of its two component automata. All ports of the component automata which do not have connections appear on the resultant PCA. Internally,

[5]See [17] for the general methodology, and [21] for a more detailed, if slightly different, version of the present problem. There are many examples of state-transition definitions of programming statements in the literature.

the set of states of the PCA is the Cartesian product of the states of its component automata. The network map of the PCA is constructed by considering how a message across a port to port connection causes transitions in each of the participating automata.

The state transitions in the component schema happen *concurrently and asynchronously* once a communication has occurred. For example, one component automaton may be waiting for communication on some port, while another automaton is executing state-transitions. Another key point about the PCA is that its behavior can be *non-deterministic* due to non-determinism in the ordering of its internal communications (if more than one internal connection is 'activated', then it is impossible to pre-determine which communication will occur next, and hence, which states each of the component automata will go to next).

Definition 6 We extend the definition of the port connection automaton to define the *network automaton* as follows. The network automaton of a set of m port automata $M = \{P^j : j = 1, \ldots, m\}$ and a set of port connection maps $C = \{c_k : k = 1, \ldots, m-1\}$ is a single port connection automaton constructed in the following manner: Select P^1 and P^2, let $\tilde{P}^1$ be the port connection automaton $P^1 \|_{c_1} P^2$. Continue constructing port connection automata in the ordered sequence; $\tilde{P}^j = \tilde{P}^{j-1} \|_{c_j} P^{j+1}$ until $j = m-1$. This final port connection automaton $\tilde{P}^{m-1}$ is the network automaton, and is denoted $\sum\limits_{P^i \in M}^{C} P^i$.

Full Semantics

We can now give the full semantics. A state now comprises a specified connection of port automata, together with a full state for each automaton of the network. Note that the state must be *pairwise port-compatible*, i.e., if output port y_i of one automaton is connected to input port x_j of another, then the values (possibly $\#$) on the two ports must be equal.

An instantiation operation then adds a new copy of the port automaton of the designated schema, makes the designated connections, and uses the initialization of variables and the prior values on the port to which the new automaton is connected, to establish its initial full state. The full-states of all other automata in the network remain unchanged.

For deinstantiation, we need to make explicit that the port automaton semantics P is currently a network automaton $\sum\limits_{M} P$. Deinstantiation results in a state-transition which yields a new network automaton identical to the original except that one component automaton has been removed. If P^N is the port automaton that is deinstantiated, then the next-state is

$$(\bar{q}, \sum_{M \setminus \{P^N\}} P) \tag{2}$$

where $\bar{q}$ comprises the full-state vector of all the automata in $M \setminus \{P^N\}$.

The semantics of instantiation and deinstantiation are constructed by considering the behavior of the network automaton composed of all port automata connected together. However, the schemas themselves, of which the port automata are the semantics, are specified as single concurrent computing agents. It is also very useful to consider aggregating a network of schemas into a single computing agent description; this is what we call the *assemblage*.

4.4 The Assemblage Schema

An *Assemblage SI* is a computing agent whose behavior is defined as the interaction of a number of communicating SIs. This aggregate SI can be considered the instance of a single *assemblage schema*, which must contain information on how the individual SIs are created and connected, and how the ports of the component SIs appear as the ports of the assemblage SI. An assemblage schema describes a generic network of schemas, where a schema is now defined to be either a basic schema or an assemblage schema; hence, assemblages can be nested.

An *assemblage schema* description consists of a schema name and input and output port lists and a set of internal variables, the same as a basic schema. These describe how the assemblage SI appears to other SIs. Unlike a basic schema, the assemblage schema does not have a behavior section with s sequence of programming statements. Instead, it has a *Network* section, which contains the SI network description. The syntax of the assemblage network description should be, and is, completely different from the syntax of the behavior section of a basic schema — their purposes are different, the assemblage facilitates the description of (possibly dynamic) networks of computing agents.

The assemblage schema is more than a basic schema with different syntax because it *hides* the details of its internal network of SIs. From the view of another SI, an assemblage and a basic SI are identical. The assemblage is also a natural scoping mechanism: Statements about a schema or SI within an assemblage affect only the SIs which are *local* to that assemblage, i.e., the SIs composing the assemblage's network. An assemblage deinstantiates if and only if all the SIs of its internal network deinstantiate.

Definition 7 We use the following syntax for assemblage definition:

$$\begin{array}{lll} [& Assemblage-Name: & \langle N \rangle \\ & Input-Port-List: & (\langle Iplist \rangle) \\ & Output-Port-List: & (\langle Oplist \rangle) \\ & Variable-List: & (\langle Varlist \rangle) \\ & Network: & (\langle Network \rangle) \quad] \end{array}$$

where

- $\langle N\rangle, \langle Iplist\rangle$, and $\langle Oplist\rangle$ are the assemblage name and the lists of its input and output ports respectively (these determine how the assemblage appears to other SIs). $\langle Varlist\rangle$ is an internal variable list.
- $\langle Network\rangle$ initializes the network of SIs; it creates and connects the SIs which form this assemblage.

The $\langle Network\rangle$ specification is essentially a more programming-oriented version of the notation we used in Section 3. Rather than use the C and E maps, for brevity a version of the $\langle Instn\rangle$ command of the basic schema is used. In addition, some iterative statments are defined to simplify the creation of large networks — these are the $\langle For\rangle$ and $\langle Forall\rangle$ statements.

Definition 8

$$\begin{array}{lcl} \texttt{Network} & ::= & \wedge \mid \langle Nstat\rangle, \langle Network\rangle \mid \langle Nstat\rangle \\ \texttt{Nstat} & ::= & \langle For\rangle \mid \langle Forall\rangle \mid \langle Instlist\rangle \\ \texttt{Instlist} & ::= & \wedge \mid \langle Instn\rangle\langle Instlist\rangle \end{array}$$

The $\langle Instn\rangle$ is the instantiation command of Definition 16 in Appendix A. Within $\langle Network\rangle$, an equivalence (the E map) between an assemblage port and a port on a component SI is established by placing the assemblage port name in the port connection lists at the appropriate place and preceded by the $\equiv$ symbol in the instantiation command. The $\langle For\rangle$ statement specifies a definite iteration, to simplify multiple schema instantiation. The $\langle Forall\rangle$ statement is a definite iteration across *schema instances* (once for each SI of the index schema). For notational convenience we borrow $\langle For\rangle$ from Definition 15 in Appendix A.

<u>The `Forall` Operation.</u>

The `Forall` operation is a special schema operation which uses a schema name as an *index* for iteration. The commands in the loop-body $\langle Instlist\rangle$ will be executed *once for each instance of* `S` *which exists* for as long as the assemblage exists. This special operation facilitates the construction of the various types of precondition schemas discussed previously (e.g., see [22]). Within the body of `Forall,` any reference to `S` is taken to refer to the instance of the current iteration.

Definition 9

$$\texttt{Forall} \ ::= \ \texttt{Forall}\, S \ : \ \langle Instlist\rangle \ \texttt{Endforall}$$

As an example, consider the problem of classifying objects seen by the robot. Let us assume a visual subsystem adept enough to segment the world into a set of candidate objects each tagged with a set of visual features. Each candidate object will be represented by an instance of the `EOB` schema (Environmental OBject), the primitive visual schema, which has a set of output ports on

Figure 4: Object classification using `Forall`

which can be read the values of the features. The classification problem now becomes one of testing the ports of each `EOB` instance to determine if it represents an object of the designated class.

The `Forall` operation provides a nice way to do this classification. We build a schema `Xtest` which when instantiated and connected to the ports of an `EOB` SI will determine if the values on those ports qualify the object to be member of the class X of objects. Its action on success is to create a corresponding instance of a schema `Xobject` connected to each `EOB` which passes the test. The `Forall` operation is used to create (and connect) one instance of `Xtest` *for each* instance of `EOB` as follows (assuming `EOB` ports f_1, f_2, f_3):

`Forall EOB` :
    `Xtest(EOB(`f_1`),EOB(`f_2`),EOB(`f_3`))();`
`Endforall`

This sets up a network to classify each `EOB` concurrently. A more sequential version of `Forall` can be constructed from this version [21].

The Semantics of the Assemblage

The semantics of the assemblage is easily constructed — the network automaton, where the semantics of the component SIs of the assemblage supply the set of port automata. The only difficulty concerns the assemblage port names. $\sum_{M} P$ has as its ports the unconnected ports of each of the automata in M. In an assemblage definition, the component SI names can be changed, or hidden, from the outside world. Unconnected component SI ports not declared equivalent with an assemblage port (i.e., hidden) do not cause problems; they simply cannot occur in more than one state-transition.

The equivalence connections in $\langle Network \rangle$ can be used to generate a renaming map E to determine how unconnected (and unhidden) component SI port names are renamed as assemblage port names. Finally, semantics of the assemblage is given by the network automaton $\sum_{M \cup \{PP\}} P$ with its ports renamed according to the map E. For a treatment of why $\langle For \rangle$ and $\langle Forall \rangle$ are valid in $\langle Network \rangle$, see [21].

4.5 Port Connection Fan-in and out

Fan-out occurs when the output port of some SI is connected to more than one input port on other SIs — if there are n such connections, the fan-out is said to be of degree n. For example, to instantiate schema `A` with its first output port (called a, say) connected to two other (input) ports b and c, we write $\mathtt{A}()(b \mid c)$. Any value written to a by `A` will be passed *both* to b and c. However, it is necessary to consider whether the write at a terminates when *one* of b or c is read, or whether both must be read before the write will terminate. The former is called OR-semantics (either read), and the latter, AND-semantics (both read). In general, there are uses for both kinds. We use $\mid$ to denote OR-semantics and $\mid^+$ to denote AND-semantics.

Fan-In occurs when an input port on some SI has more than one output port connected to it; again, if there are n such connections, we say the fan-in has degree n. For example, to instantiate `A` with its first input port (called $\bar{a}$, say) connected to two other (output) ports $\bar{b}$ and $\bar{c}$, we write $\mathtt{A}(\bar{b} \mid \bar{c})()$. Any value written by *either* $\bar{b}$ or $\bar{c}$ will be received at $\bar{a}$. Again, $\mid$ denotes OR-semantics and $\mid^+$ denotes AND-semantics.

If fan-in/out occurs on a port due to multiple separate connection lists, we apply *default semantics* to the connections. For simplicity, we shall assume that the default semantics is equivalent to OR-semantics.

<u>Formal Semantics of Fan-in/out</u>

Port automaton connections are always of the form one output port to one input port. The challenge of defining fan-in/out semantics is in devising a way to represent multiple connections to a port. We introduce a class of port automata called *connectors*. Any instance of fan-in/out on an SI port will be modeled by having a connector automaton which takes the multiple connections, one at a time, on a set of ports, and maps them to a single port, and by connecting this port to the designated port on the port automaton which is the semantics of the SI.

We will refer to a fan-out connector, which takes a output port to a set of output ports, as a *split connector*, and a fan-in connector, which takes a set of input ports to s ingle input port, as a *join connector* (Figure 5).

Definition 10 A split connector is a port automaton P^{split}, with a set of output ports $L_y = \{1, \ldots, n\}$,

Figure 5: Split and Join Connectors

an input port $L_x = \{1\}$, where $Y_1 = Y_2 = \cdots = Y_n = X_1$, a single state $Q = \{q\}$ and a full-state transition function defined by: $(x', q, y') \in \varphi(x, q, y)$ iff either

AND-semantics split:

$x_1 \neq \#$, and $x'_1 = x_1$, and for all i, $y'_i = x_1$.

OR-semantics split:

$x_1 \neq \#$, and $x'_1 = x_1$, and there is an i, $y'_i = x_1$ where for all $k \neq i$, $y'_k = \#$.

Definition 11 A join connector is a port automaton P^{join}, with a single output port $L_y = \{1\}$, a set of input ports $L_x = \{1, \ldots, n\}$, where $X_1 = X_2 = \cdots = X_n = Y_1$, a single state $Q = \{q\}$ and a full-state transition function defined by: $(x', q, y') \in \varphi(x, q, y)$ iff either

AND-semantics join:

For all i, $x_i \neq \#$ and for all i, $x'_i = x_i$, and $y'_1 = x_j$ for some j.

OR-semantics join:

There is an i, $x_i \neq \#$, and for all $k \neq i$, $x'_k = \#$, $x'_i = x_i$, and $y'_1 = x_i$.

The following two observations result directly from the fan-in/out semantics (for more detail see [21]):

Observation 12 A connection can be constructed between an input port on one SI and an output port on another SI, which has the property that a read to the input port will *always* terminate, even if the output port is never written to (time-limit schema).

Proof Outline: A schema can be constructed which simply cycles through some number of internal states and then transmits a message on its one port. An SI of this schema can be made and fan-in connected to the receiving port with OR-semantics. Since this SI is guaranteed to always write its port, a guarantee can be made that eventually any read to the receiving port will always terminate. This is a weak form of time-out, since the 'clock' duration is unpredictable.

Observation 13 Using the synchronous communication operations and the instantiation operation, it is possible to duplicate completely an asynchronous communication operation (messenger schema).
Proof Outline: When an SI wishes to deliver a message asynchronously to a given port, it creates a `messenger` SI using the instantiation parameters to pass the text of the message to it, and connects it with OR-semantics fan-in to the port. The `messenger` SI uses the standard synchronous send primitive. OR-semantics fan-in guarantees that every message sent to this port will eventually be received, but does not guarantee their order.

4.6 Semantics of Task-Units and Preconditions

The $\mathcal{RS}$ task-unit is simply a structured assemblage; an assemblage that agrees with the task unit 'formula' in Section 3.4. As such it doesn't need any computational semantics other than already provided for the assemblage.

The general form of a precondition from Section 3.5 is $\mathtt{Pre}_h \ :^C \ \mathtt{T}_v$, meaning that `Pre` applies test h to its environment, and if it is successful, it creates $\mathtt{T}_v$ connected as described by C. Any of the preconditions in Section 3.5 can be reduced to this description. Applying our semantics of instantiation and deinstantiation, where P^p is the port automaton semantics of the precondition schema, P^T that of its consequent schema, and $\sum_M P$ the semantics of all other SIs, then the network goes through the stages: $\sum_M P$, then $\sum_{M \cup \{P^p\}} P$, and the test h is evaluated, and when and if h becomes true, $\sum_{M \cup \{P^T\}} P$. Let H be defined as

$$H(h, A, B) = \begin{cases} \{B\} & if\ h\ has\ evaluated\ true \\ \{A\} & if\ h\ has\ not\ yet\ evaluated\ true \end{cases}$$

Determining if h was true boils down to determining when P^p has entered some 'h is true' state. So the precondition operation can be defined more concisely as *a special kind of automata composition:*

$$\sum_{M \cup H(h, P^p, P^T)} P$$

Figure 6: The Centered Grasp Problem

This puts the emphasis on the test h carried out by the precondition, and shifts emphasis from P^p which is the implementation of the test. Varieties of this precondition semantics can be used to express each of the preconditions introduced already.

5 Task-Unit Definition Example

Now that we have introduced $\mathcal{RS}$ syntax and semantics, we give a more detailed example of task-unit programming for a standard robotics problem: the centered grasp.

The centered grasp problem consists of centering a gripper over an object to be grasped based on contact feedback from the fingers. The problem arises whenever gripper fingers need to be moved together in opposing pairs, and it is not limited to two-fingered grippers; the following example was tested on the four-fingered Philips Multi-Functional Gripper[9] (Figure 6).

We divide the problem into two concurrent activities; moving the fingers in to contact, and moving the wrist to eliminated contact. Let us assume the following primitive sensory and motor schema:

`FTact` $_i()(l, r)$ is parameterized to report contact on finger-pair i on its two output ports l and r as a bit value 1 for contact, 0 for no contact.

`FClose` $_i(sg)()$ is parameterized to control the separation between fingers in finger-pair i. If a 0 is written to sg, the fingers close at some fixed rate until a 1 is written to sg at which point they stop.

`WristPos` $(p)()$ controls the position of the wrist. A desired wrist offset from the current position is written to p, and the wrist moves from its initial position, to the initial position plus the offset vector.

We shall describe the centered grasp problem in a modular way that can be applied to grippers

Figure 7: The `Grasp` and `Compensate` Task-Units

with multiple finger-pairs. We will build a task-unit $\texttt{Grip}_i$ which will close finger-pair i as long as there is no contact on either finger, and a task-unit $\texttt{Compensate}_i$ which moves the wrist whenever there is contact on one finger pair in such a way as to eliminated that contact. A network $(\texttt{Grip}_i, \texttt{Compensate}_i)$, $i \in \{1, \ldots, n\}$ implements the task for a gripper comprising of n finger-pairs.

The `Grip` task unit can be specified:

$$\texttt{Grip}_i = [\texttt{FTact}_i()(l,r), \texttt{tGrip}(l,r)(fs), \texttt{FClose}_i(s_d)()]^{C,E}$$

with C as show in Figure 7. The task schema `tGrip` can be specified as a basic schema:

```
[  tGrip
   Input-Port-List:    ( l,r:Bit )
   Output-Port-List:   ( fs:Integer )
   Behavior:           ( If (l=1)OR(r=1) Then fs=0; Else fs=1;
                         Endif; )                                ]
```

The `Compensate` task-unit can be specified as:

$$\texttt{Compensate}_i = [\texttt{FTact}_i()(l,r), \texttt{tCompensate}_{f_i}(l,r)(wp), \texttt{WristPos}(p_d)()]^{C,E}$$

with C as show in Figure 7. The task schema `tCompensate` can be specified as a basic schema which is parameterized by f_i, the unit direction vector of finger-pair i (see Figure 6):

```
[  tCompensate
   Input-Port-List:    ( l,r:Bit )
   Output-Port-List:   ( wp:Vector )
   Variables:          ( tl,tr:Bit )
   Behavior:           ( tl = l;  tr = r;
                         If (tl=1)AND(tr=0) wp = δ ∗ f_i;  Endif;
                         If (tl=0)AND(tr=1) wp = −δ ∗ f_i; Endif;
                         )                                      ]
```

where δ is some built-in small position increment. We can stop `tCompensate` as soon as we get contact on both fingers. However, it may be better to let `Grip` continue to run to combat unforeseen disturbances until the object needs to be released.

6 Conclusion

This paper is one of the few attempts in the literature to say 'what is special' about robotics in computational terms. The goal of this research was to come up with a model of computation for robots — a way of describing and analyzing computation from a robotics point of view — rather than a particular syntax for robot programming. What we have accomplished is simply the start.

$\mathcal{RS}$ is a model of distributed computation, specifically constructed to express and analyze sensory-based robot programs. We presented a set of 5 characteristics of the robot programming domain to fix an appropriate structure for the model. Among the properties that make our work unique are that it uses *nested networks* for representation and efficiency, it is formally defined to facilitate plan verification and auto-generation, and it can support the use of a plant model to determine the behavior of plan in response to the world. Future work will proceed along the lines of using the formal basis we have established here to reason about sensory-interaction and resource usage in plans. A key point to be addressed is the nature of the plant/world model, a topic minimally addressed in this paper.

This paper describes the motivation, structure, and formal semantics of $\mathcal{RS}$. [21] presents a linear-time time temporal logic specification and verification methodology for $\mathcal{RS}$, having the automata semantics as its interpretation (and also presents a rudimentary plan/world model). A *distributed robot control environment* has been implemented on a VAX^{TM}11/780. The environment consisted of a *compiler* and *emulator* for $\mathcal{RS}$, and a robot simulation package. Current work involves implementing an updated version of this system on an in-house multi-processor system to control a Puma 560 equipped with a multi-functional gripper[23, 11, 9].

References

[1] Albus, J., MacLean, C., Barbera, A., and Fitzgerald, M., "Hierarchical Control for Robots in an Automated Factory," *Proceedings, 13th ISIR*, Chicago, Ill., Apr. 1983, pp.13.29–13.43.

[2] Andrews, G.R., and Schneider, F.B., "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol.15, No.1, Mar. 1983, pp.3–43.

[3] Arbib, M.A., "Perceptual Structures and Distributed Motor Control," in (V.B.Brooks, Ed.) *Handbook of Physiology : The Nervous System, II. Motor Control*, American Physiological Society: Bethesda, MD, 1981, pp.1449-1480.

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

[5] Barber, K.S., Agin, G.J., "Analysis of Human Communication During Assembly Tasks" *Technical Report* CMU-RI-TR-86-13, Carnegie-Mellon University, Pittsburg, PA, Jun. 1986.

[6] Cutkosky, M., "Grasping and Fine Manipulation for Automated Manufacturing," *Ph.D. Dissertation*, Dept. of Mech. Eng., Carnegie-Mellon University, Pittsburg, PA, Sep. 1985.

[7] Fox, B.R., and Kempf, K.G., "Opportunistic Scheduling for Robotic Assembly" *Proceedings, IEEE Robotics & Automation* St.Louis MO 1985. pp880–889.

[8] 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.

[9] van Gerwen, J., van der Kruk, R., and Vredenbregt, J., "A Multi-Functional Gripper" *Philips Technical Note* 053/84E, 1984.

[10] Gauthier, D., Freedman, P., Carayannis, G., Malowany, A., "Interprocess Communication in Distributed Robotics," *IEEE Int. J. Robotics & Automation* Vol. RA-3, No.6, Dec. 1987, pp.493-504.

[11] Guida, F., Harosia, T., and Lyons, D.M., "The Eindhoven Multi-Functional Gripper: Installation and Evaluation" *Philips Technical Note* TN-87-145, Nov. 1987.

[12] Hayward, V., Paul, R., "Robot Manipulator Control under Unix, RCCL: A Robot Control "C" Library" *Int. J. Robotics Research* Vol.6, No. 4, 1986, pp 94–111.

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

[14] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice-Hall International Series in Computer Science, 1985.

[15] Iberall, T., "The Nature of Human Prehension: Three Dextrous Hands in One" *Proceedings, IEEE Int. Conf. Robotics and Automation*, Raleigh NC, Apr. 1987, pp396-401.

[16] Korein, J.U., Maier, G.E., Taylor, R.H., Durfee, L.F., "A Configurable System for Automation Programming and Control" *Proceedings, IEEE Int. Conf. Robotics and Automation* San Francisco, CA 1986, pp.1871–1877.

[17] Kfoury, A.J., Moll, R.N., and Arbib, M.A., *A Programming Approach to Computability*, Texts and Monographs in Computer Science, Springer-Verlag: New York, Heidelberg, Berlin, 1982.

[18] Lieberman, L.I., Wesley, M.A., "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly" *IBM J. Res Dev.* Jul. 1977, pp.321–333.

[19] Lozano-Perez, T., "Task Planning," in (Brady, M., Hollerbach, J., Johnson, T., Lozano-Perez, T., Mason, M., Eds.) *Robot Motion Planning and Control*, MIT Press, Cambridge, MA, and London, England, 1983.

[20] Lozano-Perez, T., and Brooks, R., "An Approach to Automatic Robot Programming," *AI Memo 842*, MIT, Cambridge, MA, Apr. 1985.

[21] Lyons, D.M., "$\mathcal{RS}$: A Formal Model of Distributed Computation for Sensory-Based Robot Control" *Ph.D. Dissertation* and *COINS Technical Report* # 86-43, University of Massachusetts at Amherst, Amherst, MA 01003, 1985.

[22] Lyons, D.M., and Arbib, M.A., "A Task-Level Model of Distributed Computation for Sensory-Based Control of Complex Robot Systems" *IFAC Symposium on Robot Control*, Barcelona, Spain, Nov. 6-8th 1985.

[23] Lyons, D.M., "Implementing a Distributed Programming Environment for Task-Oriented Robot Control" *Philips Technical Note* TN-87-054, Apr. 1987.

[24] Li, Z., and Sastry, S., "Task Oriented Grasping by Multifingered Robot Hands" ERL Memo UCB/ERL M86/43, University of California, Berkeley, CA 94720, 1986.

[25] Milne, G., and Milner, R., "Concurrent Processes and Their Syntax" *JACM* Vol. 26, No. 2, Apr. 1979, pp.302–321.

[26] Mujtaba, S., and Goldman,R., "AL Users' Manual," *Technical Report* STAN-CS-81-889, Department of Computer Science, Stanford University, Stanford, CA, Dec. 1981.

[27] Ramamritham, K., Pocock, G., Lyons, D.M., and Arbib, M.A., "Towards Distributed Robot Control Systems," *IFAC Symposium on Robot Control*, Barcelona, Spain, Nov. 6-8th 1985, pp.209-213.

[28] Shin, K.G., Epstein, M.E., "Intertask Communications in an Integrated Multirobot System," *Int. J. Robotics & Automation* Vol. RA-3, No. 2, Apr. 1987, pp.90-100.

[29] 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.

[30] Taylor, R., "A Synthesis of Manipulator Control Programs from Task-Level Specifications," *Technical Report* STAN-CS-76-560, Department of Computer Science, Stanford University, Stanford, CA, Jul. 1976.

[31] Unimation Inc. *The VAL Reference Manual*. 1978.

[32] Volz, R.A., "Report of the Robot Programming language Working Group: NATO Workshop on Robot Programming Languages," IEEE Int. J. Robotics & Automation Vol. 4, No. 1, Feb. 1988, pp.86–90.

A Basic Schema Specification Syntax

Our main goal in this paper is to construct a formal model of computation directly based on the characteristics of the robot domain. We are not so concerned at this stage with the details of what programming syntax should be used – that has more to do with humans than with robots. However, in order to reduce our ideas to practice we do need to use some form of program notation. From that perspective, we present the following schema specification syntax.

Definition 14

$$\begin{array}{lcl}
\texttt{Behavior} & ::= & \wedge \mid \langle Stat\rangle ; \langle Behavior\rangle \\
\texttt{Stat} & ::= & \langle Assign\rangle \mid \langle IfElse\rangle \mid \langle For\rangle \\
 & & \mid \langle Instn\rangle \mid \langle Dinstn\rangle \\
\texttt{Assign} & ::= & \langle Var\rangle{:=}\langle Expression\rangle \mid \\
 & & \langle OutputPort\rangle{:=}\langle Expression\rangle
\end{array}$$

We embed reading and writing into the syntax of `Assign;` an input portname occurring in `Expression` is a read from 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 and semantics for an `Expression,` with the addition of certain vector and matrix operations (which will be specified when needed).

Definition 15

$$\begin{array}{lcl}
\texttt{IfElse} & ::= & \texttt{If } \langle Condition\rangle \texttt{ Then } \langle Behavior\rangle \\
 & & \quad \texttt{Else } \langle Behavior\rangle \texttt{ Endif} \\
\texttt{For} & ::= & \texttt{For } \langle Index\rangle = lb \ldots ub \\
 & & \quad \langle Behavior\rangle \texttt{ Endfor}
\end{array}$$

We assume standard syntax for `Condition,` and constrain `Index` to be an internal integer variable for simplicity. Apart from this, the semantics of `If` and `For` are as one would expect.

The *instantiation operation* takes as input a schema name, some variable initializations, and a connection map, and produces a new schema instance, while *deinstantiation* effectively removes an SI from the network.

Definition 16

$$\begin{array}{lcl}
\texttt{Instn} & ::= & \langle Schemaname\rangle_{\langle V\rangle}(\langle CI\rangle)(\langle CO\rangle) \\
\texttt{Dinstn} & ::= & \texttt{Stop}
\end{array}$$

where

- `CI` is a list of input port names, denoting port connections *between the named ports and the corresponding (by position in* $\langle Iplist\rangle$*) port names of the created SI*; the *types* of connected ports must match.
- `CO` is a list of output port names, denoting connections between the named ports, and the corresponding (by position in $\langle Oplist\rangle$) ports of the created SI; the *types* of connected ports must match.
- `V` is a list of initial variable values; these are assigned *by position* to the internal variables of *Schemaname* as they are specified in `Varlist` .

Figure 8: Recursive Network of `Factorial`

Example: The `Factorial` Schema

To illustrate the use of this syntax, we recast the usual recursive definition in terms of a single schema `Factorial.` Note the way in which ports are named to ensure proper communication between adjacent instances.

When instantiated and passed a value n on port x, an SI of `Factorial` creates a network of SIs recursively to calculate $n!$ (see Figure 8).

```
[  FACTORIAL
   Input-Port-List:    ( x:Integer pfx:Integer )
   Output-Port-List:   ( xm:Integer fx:Integer )
   Variable-List:      ( temp:Integer )
   Behavior:           ( temp := x
                         If ( temp ≤ 1 ) Then fx := 1;
                           Else FACTORIAL(xm)(pfx)();
                                xm := temp - 1;
                                fx := pfx * temp;
                                Stop;
                         Endif; )                     ]
```

Upon instantiation, a value is read from the port x and stored in the variable *temp*. If the value returned was less than or equal to one, then (by definition) one is returned as its factorial on port fx. Otherwise, this `Factorial` SI creates another instance of `Factorial` connected as follows: The xm port on the 'old' SI is connected to the x port of the 'new' SI (remember, connection is made by positional correlation in the port list). Similarly, the fx port of the 'new' SI is connected to the pfx port of the 'old' SI. The ports pfx and xm function as the *continuation path* for the evaluation of the factorial. The schema is programmed so that when this SI can at last read from its input port pfx, it will be when the factorial of $temp-1$ is passed back to the creator SI through the pfx port, and this (by definition) multiplied by *temp* is the value of *temp*! to be written to port fx.

FIGURE CAPTIONS