Fundamentals of $\mathcal{RS}$ — Part I: The Basic $\mathcal{RS}$ Model

Damian M. Lyons

Autonomous Systems Department
Philips Laboratories
North American Philips Corporation
345 Scarborough Road
Briarcliff Manor
NY 10510

1 Introduction

This paper is the first in a series of documents standardizing the $\mathcal{RS}$(Robot Schemas) model of computation and related methods and algorithms. This model has been used for robot assembly plan representation [12, 15] and for dextrous hand control[9, 10], it will be used in a number of different components of the task planning domain[16], it is being used to study real-time robot computing issues[22, 21], and its use in sensing and representation is being explored[27]. Central to every application of $\mathcal{RS}$ is the notation and semantics of the model, a set of analysis methods and algorithms, and a computer implementation. This series of papers is intended to maintain consistency in these central issues.

This first paper presents a summary of the motivation and structure of the lowest-level of description in the $\mathcal{RS}$ model, the so-called *basic $\mathcal{RS}$ model.* The first few papers in this series will be in the same vein; informal summaries containing a minimum of new material. This paper describes the basic $\mathcal{RS}$ model; following papers will include material on the process composition operations, atomic processes, operator and connection semantics, programming languages, and proof methods and algorithms. This list will be edited as necessary to reflect the state of the research.

The paper is laid out as follows. It begins by looking at two motivation questions: why look at special models of computation for robotics, and what are the characteristics of robot computation on which such a model could be based? This provides the background for the

construction of the basic $\mathcal{RS}$ model. This is the lowest level of description possible in $\mathcal{RS}$. The basic $\mathcal{RS}$ model is first described informally, to give the reader an intuitive grasp of the subject matter. The (first installment of the) model notation is then introduced in a separate section.

It is important to distinguish the model notation from a programming language for $\mathcal{RS}$.

- The notation is used to describe computation, to define $\mathcal{RS}$ operators, and to analyze computation. The constraints on the notation are the following: how well does it express the 'intuitions' captured in the model and is it well-defined mathematically.
- A programming language is used to describe programs in the $\mathcal{RS}$ model in such a manner that they can actually be executed on a computer. The constraints on a programming language are the following: how well does it represent the notation, can it be parsed automatically, and does it contain sufficient implementation-dependent information to be executed on a computer.

The last section describes how we build on the basic $\mathcal{RS}$ model, and thus sets the stage for the material in later papers in this series.

2 Motivation

The material in this section is summarized from [18]. One common approach to robot programming is to use a general purpose programming language. 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. Whereas the general purpose approach says 'let us make cosmetic changes to a programming language to simplify robot programming,' the special purpose approach says 'let us start with a clean slate, and try to define computation in terms of what is needed in robotics.'

Why look at special models of computation? This approach offers the potential to construct task-level specification mechanisms which are easier for humans to deal with automatic planners to interface to. In addition, these mechanisms are concise, and can execute in an efficient manner because the model can be built with efficiency for specific primitives

in mind. Arguably, 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 [25] and Basic, AL [20] and Pascal). Other special purpose languages provide insight into the nature of task-level specification [7, 8] but don't address conciseness or efficiency of execution (and this applies to many AI plan representation languages also).

In order to keep our approach general and free from implementation detail, we will *not* develop yet another 'robot programming language.' Our goal in this paper is to formalize the key computational characteristics of robot programming into a single mathematical model, i.e., the basic $\mathcal{RS}$ model. This model will have two major applications: Firstly, it will provide a 'deep structure' for subsequent program or plan representation languages[1]. Secondly, a formal level at which robot behavior can be specified and analyzed is introduced. Since a robot system operates in 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.

Characteristics of the robot domain. What makes robot computation a unique subclass of general purpose programming is that the robot needs to interact directly with a (possibly noisy, dynamic and/or unpredictable) environment [2, 26]. 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. This is the most basic computational characteristic of the robot domain:

> Robot computation interacts strongly with the environment. (C.1)

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

> The central paradigm in robot computation is that sensory input is linked with knowledge of the plan/world model to produce appropriate action. (C.2)

A convenient term for this type of computation is *sensori-motor computation.* It is surprising to find that the guarded move is frequently the only structure which relates sensing and action in typical robot languages[6]. This type of connection of sensing events to motor actions is relevant *at all levels of abstraction in robot programming.*

[1]For example, both [18] and [21] contain programming languages based on the $\mathcal{RS}$ model.

Albus [1] has suggested a hierarchical model with this sensori-motor structure. That models consists of distinct hierarchies for sensing, modelling and control activities, each with the same fixed number of levels, where levels at the same 'height' are connected in the sensori-motor computation 'loop'. 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. Thus, the third characteristic is that

> Robot computation needs a hierarchical, dynamically reconfigurable, sensori-motor structure. (C.3)

At any level of abstraction, the program should be described as a *network* consisting of a sensory model for the task and the primitive motor actions of the task. In turn, these elements[2] 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. This is an *action-oriented* view of sensing. Conversely, the sensory network can be thought of as an embedded 'logical sensor' for the program. This has some similarity to Henderson's *logical sensors* [4], but with one difference: the 'sensor' is integrated directly with the set of actions which use it. This approach to sensori-motor computation provides the basis for a general and efficient way to interact robustly with an unstructured environment.

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. 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 argue that this inherent parallelism is manifest to some degree in all robot programming.

> Robot computation is inherently distributed. (C.4)

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

It is possible to identify the following ways in which this inherent distribution needs to be made explicit:

1. It is necessary to be able to represent the ways in which low-level parallelisms, e.g. degrees of freedom, can be collected into groups with common control needs[3] and controlled in parallel. It is important to be able to represent *logical groupings* that can be controlled in parallel, e.g., *Virtual Fingers* [3] and *Oppositions* [5]. 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.

2. Another major source of inherent parallelism is that which exists between the gathering of the sensations necessary to construct a task-specific model of an object, and the actions to be carried out on 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 [2].

The schema [2, 11] or skeletal procedure mechanism reoccurs often in robot programming. For example, Taylor [24] discusses an approach to robot programming based on *procedure skeletons* that represent prototypical motion strategies. Objects are also 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. This is the final characteristic:

Robot computation needs a schema or class structure (C.5)

In the next section, a model of computation based on these characteristics is introduced. The model is called *Robot Schemas* (or $\mathcal{RS}$) to emphasize the continuity of this work with the 'schema' style of computation described informally in [2, 22, 11].

[3]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.

3 The Basic $\mathcal{RS}$ Model

$\mathcal{RS}$ is a model of distributed computation. The basic $\mathcal{RS}$ model provides the lowest level of description in the $\mathcal{RS}$ model. In a Turing machine model, for example, all the computation is 'funneled' through the read/write head of the machine. Thus, the model is inherently sequential. In a distributed model, computation can occur at a possibly infinite set of 'locations' usually called locii of computation. The mechanism that carries out the computation at each locus is usually called a computing agent (agent, for short). Computation is performed in a distributed model by the *interaction* of a number of *concurrent* agents. We will use this concurrency to bring out what we have called the inherent parallelism (characteristic (C.4)).

In order for computing agents to interact with each other, they need to have some kind of communication method. All interaction in $\mathcal{RS}$ is expressed as *message passing* between connected agents. A connection between two (or more) agents sets up a message channel. Interaction occurs when agents read from or write to this channel. A set of connected agents is called a network.

The $\mathcal{RS}$ World View. The sensors and actuators of a robot system can be considered as a set of computing agents. The control program for the robot can also be considered as a network of computing agents. Furthermore, the environment containing the robot can be considered as a network of computing agents. This may be hard to visualize, since we do not normally think of the 'real world' in these terms (see Figure 1). The behavior of these environmental agents is governed by the laws of physics. Particular networks of these agents represent particular environmental scenarios, i.e., objects interacting with each other in specific ways (supporting each other, sliding, moving in free space, etc). The environment network and the control network are connected only through the set of sensor and actuator agents. Putting this another way, the control network and the environment network share the set of sensor and actuator agents. This is the 'world view' of the $\mathcal{RS}$ model. It directly supports characteristics (C.1), (C.2) & (C.4).

Schemas and Schema Instances (SIs). A *schema* is a parameterizable description of a computing agent (characteristic (C.5)). A computing agent is created from a schema by the *instantiation* operation, and the term *schema instance*, SI, or process, denotes a computing agent. Instantiation creates an SI, sets some parameter values in the SI, and connects the SI to other SIs so that interaction can occur. A schema that directly represents an actuator is called a primitive motor schema. A schema that directly represents a sensor is called a primitive sensory schema.

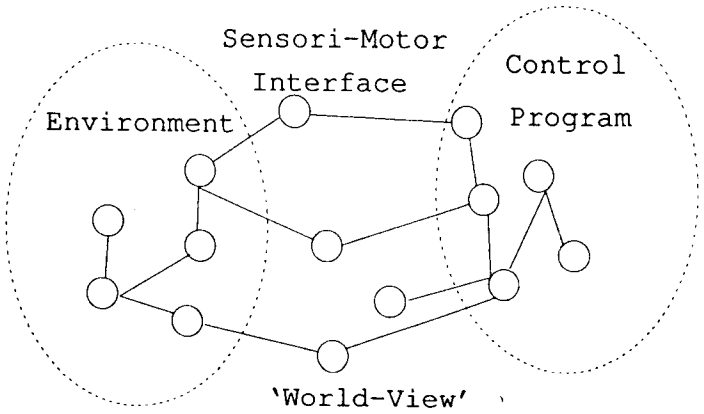

Figure 1: The $\mathcal{RS}$ World View.

Communication. In order to build the sensori-motor 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. 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.

Assemblage Schemas. The final piece of the basic $\mathcal{RS}$ model structure we need is a mechanism to group connected SIs together into a network in such a manner that the network appears to other SIs to be a single computing agent (characteristic (C.3)). A network grouped together in this way is called an *assemblage* SI, and its definition is called an assemblage schema. Some of the ports on individual SIs within the assemblage may appear as ports of the assemblage. The general rule is that if a port on a component SI has not been connected to anything, then it will appear as a port on the assemblage (perhaps under a new name). It is also convenient to be able to hide unconnected ports on occasion.

Task-Unit Schemas. The most important structure is $\mathcal{RS}$ is called the *task-unit*. This is a group of three types of computing agents that captures characteristic (C.2). A task-unit schema is a 'structured' assemblage schema, a network consisting of a set of sensory schemas, a set of motor schemas and a set of task schemas (this is the 'control' program for the robot in the $\mathcal{RS}$ world view). This captures the robot controller structures as a sensory process pulling out important sensory data from the environment via the sensors, a motor process capable of controlling the appropriate robot actuators, and a task process that is using the sensory data to effect motions of the robot with the objective of achieving some goal.

While any set of connected computing agents is a 'program' in the $\mathcal{RS}$ model, task-units typify that sort of program usually called a robot *task plan*: a set of instructions for

manipulating the world to achieve some goal. The sensory schemas defines the task-specific object model or world view, the motor schema defines a task-specific robot model, and the task schema defines how the task is implemented using the object/world and robot models.

The components of a task-unit can themselves be assemblages. This generates a sensori-motor hierarchy. 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.

4 Model Notation

A schema name will always be written in teletype font, e.g., `Joint`, $\texttt{Locate}_{obj=bolt}$. In general, the first letter of a schema name will be capitalized. An SI is written as a schema name subscripted by parameter values, e.g., $\texttt{Joint}_{i=2}$, and this denotes the SI made by instantiating the named schema with its parameters set to the indicated values. Parameters to a schema are analogous to arguments in a subroutine call or to the slots in an AI frame. It is convenient to think of parameter setting as the assignment of values to some local variable within the schema. Actually, for semantic reasons, it is easier to model parameter setting as a special kind of port communication than as variable assignment. Later papers in this series will detail the semantic modelling of parameters; we need not concern ourselves with it here.

A full schema or SI name is the schema or SI name augmented with a list of input and output ports. For example, `Joint(x)(y)` refers to the schema `Joint` with one input port `x` and one output port `y`. Port names will usually be written in lower case teletype font. $\texttt{Joint}_{i=2}\texttt{(x)(y)}$ denotes an SI of schema `Joint`, with ports as before, and with parameter i set to the value 2.

A network schema composed of SIs `A1, A2, ..., An` connected together by a *port-to-port connection relation* c is written as follows.

$$(\texttt{A1}, \texttt{A2}, \ldots, \texttt{An})^c \tag{1}$$

The parentheses in this description simply group the schema instances together. In later papers in this series, parentheses will be used in this same fashion to disambiguate composition operator priority. A connection relation is a set of couples of the following form:

$$(\texttt{SSIname}, \texttt{sportname}) \mapsto (\texttt{DSIname}, \texttt{dportname}) \tag{2}$$

This denotes that output port `sportname` on `SSIname` is connected to input port `dportname` on `DSIname` (input ports will always be on the right).

Example. Let $\texttt{Jpos}_i\texttt{()(x)}$ be a primitive schema that continually reports the current position of the robot joint i on its output port `x`. Let $\texttt{Jmot}_i\texttt{(u)()}$ be a primitive schema that continually accepts a motor control signal on its input port `u`, and applies it to the actuator of joint i on the robot. Let `Jset(s,x)(u)` be a schema that accepts a set-point on port `s`, and iteratively then inputs a joint position on port `x` and outputs a motor signal on port `u` in such a fashion that the joint position is driven to the setpoint on `s`. We can construct a position servo network for joint i as follows

$$\begin{gathered}(\texttt{Jpos}_i\texttt{()(x)}, \texttt{Jset(s,x)(u)}, \texttt{Jmot}_i\texttt{(u)()})^c \\ c:\ (\texttt{Jpos}_i, \texttt{x}) \mapsto (\texttt{Jset}, \texttt{x}) \\ (\texttt{Jset}, \texttt{u}) \mapsto (\texttt{Jmot}_i, \texttt{u})\end{gathered} \tag{3}$$

(For many examples of such networks, see [10, 12, 15].)

An assemblage schema composed of SIs `A1, A2, ..., An` connected together by a port-to-port connection relation c is written as follows.

$$\texttt{Asm}_v = [\texttt{A1}, \texttt{A2}, \ldots, \texttt{An}]_e^c \tag{4}$$

`Asm` is the single SI that 'contains' the network. The *port equivalence* function e maps unconnected component SI port names onto either null (if the port is to be hidden, and not appear as a port of `Asm`) or to a new name (if the port is to appear on `Asm`). e is also written as a set of couples in the same form as c, with the new name appearing on the right of the $\mapsto$ symbol. Any ports in c are prohibited from also being in e. This constraint arises from the use of the *port automata* model as a mathematical semantics for $\mathcal{RS}$[23]. The parameters to `Asm`, v, can be used as parameter values to component SIs within the assemblage.

Example. The position servo network can be phrased as an assemblage `Joint` as follows:

$$\begin{gathered}\texttt{Joint}_i\texttt{(s)()} = [\texttt{Jpos}_i\texttt{()(x)}, \texttt{Jset(s,x)(u)}, \texttt{Jmot}_i\texttt{(u)()}]_e^c \\ c:\ (\texttt{Jpos}_i, \texttt{x}) \mapsto (\texttt{Jset}, \texttt{x}) \\ (\texttt{Jset}, \texttt{u}) \mapsto (\texttt{Jmot}_i, \texttt{u}) \\ e:\ (\texttt{Jset}, \texttt{s}) \mapsto \texttt{s}\end{gathered} \tag{5}$$

A task-unit schema is simply a structured assemblage schema. It is usually written as follows.

$$\texttt{TU} = [\texttt{S}, \texttt{T}, \texttt{M}]_e^c \tag{6}$$

where S is the set of sensory schemas, M is the set of motor schemas and T is the set of task schemas. The assemblage mechanism could be used to group together SIs in each of these categories so that there was just one sensory schema, one motor schema and one task schema. However, it is convenient when using the process composition operations (the topic of a later paper in this series) to have the more general form. A hierarchy of task-units (e.g., Fig. 2) is called a sensori-motor hierarchy, because it structures a robot program into a *tree* with branches representing sensory and motor components. The leaves of the sensori-motor tree are primitive sensory and motor schemas.

Example. The assemblage Joint is a task-unit assemblage. We can construct a sensori-motor hierarchy by using Joint within a second task-unit as follows:

$$\begin{aligned} \texttt{Joint}_i(\texttt{s})() &= [\texttt{Jpos}_i()(\texttt{x}), \texttt{Jset}(\texttt{s},\texttt{x})(\texttt{u}), \texttt{Jmot}_i(\texttt{u})()]_{e0}^{c0} \\ \texttt{Touch}_i &= [\texttt{Tact}_i()(\texttt{v}), \texttt{Gmove}(\texttt{v})(\texttt{y}), \texttt{Joint}_i(\texttt{y})()]_{e1}^{c1} \\ c1: &\ (\texttt{Tact}_i, \texttt{v}) \mapsto (\texttt{Gmove}, \texttt{v}) \\ &\ (\texttt{Gmove}, \texttt{y}) \mapsto (\texttt{Joint}_i, \texttt{y}) \end{aligned} \tag{7}$$

$\texttt{Tact}_i$ reports on tactile contact on the *ith* joint on its port v. Gmove increments the setpoint of the joint actuator as long as it gets a no-contact signal on its port v. The task-unit $\texttt{Touch}_i$ implements a *guarded-move* of the *ith* link.

5 Working in the Model

So far, we have concentrated on program construction. In the next few paragraphs we will explore some straightforward examples of analysis of $\mathcal{RS}$ programs. Two results will be derived: Firstly, a way to extract the interface between a task-unit and the world model (we call this the world view of the task-unit) and secondly, a way to 'collapse' a hierarchy of networks into a single network. Although both of these results have some importance, the method of analysis is the key concept being imparted.

World View Extraction. A schema function $f()$ maps one schema to another. A schema set function $F()$ maps a schema to a set of schemas. The schema set function $S()$ maps a task-unit onto its sensory schemas. The schema set function $M()$ maps a task-unit onto its motor schemas. If TU is a task-unit, then

$$\texttt{TU} = [S(\texttt{TU}), T, M(\texttt{TU})] \tag{8}$$

Both of these functions need to recognize when they are applied to a primitive sensory or motor schema. With this in mind, we designate the following interesting subsets of SCH, the set of all schemas:

- $TSCH \subset SCH$ is the set of all task-unit schemas.
- $SSCH \subset SCH$ is the set of all primitive sensory schemas.
- $MSCH \subset SCH$ is the set of all primitive motor schemas.
- $MSCH \cap SSCH = \emptyset$, the two sets of primitive schemas are disjoint[4].
- $MSCH \cup SSCH \subset TSCH \subset SCH$.

We can now define a function SEN that 'extracts' all the sensory schemas within a task-unit:

$$SEN(\texttt{TU}) = \begin{cases} \{\texttt{S}\} & \textit{if } \texttt{S} \textit{ is a primitive sensory schema} \\ \emptyset & \textit{if } \texttt{M} \textit{ is a primitive motor schema} \\ SEN(S(\texttt{TU})) \cup SEN(M(\texttt{TU})) & \textit{otherwise} \end{cases} \tag{9}$$

The range of this function is the set of primitive sensory schemas. Its recursive effect is to extract all sensory schemas from the task-unit hierarchy (e.g., the one in Fig. 2). In similar fashion we can define a function MOT that extracts all the primitive motor components of the task-unit.

$$\begin{aligned} SEN &: TSCH \longrightarrow SSCH \\ MOT &: TSCH \longrightarrow MSCH \end{aligned} \tag{10}$$

The set $MOT(\texttt{TU})$ describes how the task-unit TU exerts control on the world. This is analogous to the concept of a *control (enable) vector* in discrete-event control theory. The set $SEN(\texttt{TU})$ describes the observable component of the world. It is now possible to define the world (plant or environment) view associated with a task-unit TU as[5]

$$WORLD(\texttt{TU}) = [SEN(\texttt{TU}) \cup MOT(\texttt{TU})]^{\emptyset}_{\emptyset} \tag{11}$$

The function $WORLD()$ applied to a task-unit 'peels-off' the interface layer between that task-unit and the environment.

[4]This constraint simplifies recursive definitions using TSCH. In addition, if there was a schema that was in both SSCH and MSCH, it could be constructed as an assemblage from a number of more primitive schemas that were only in SSCH or MSCH.

[5]Where for convenience we treat $[\{A, \ldots\}]$ as $[A, \ldots]$.

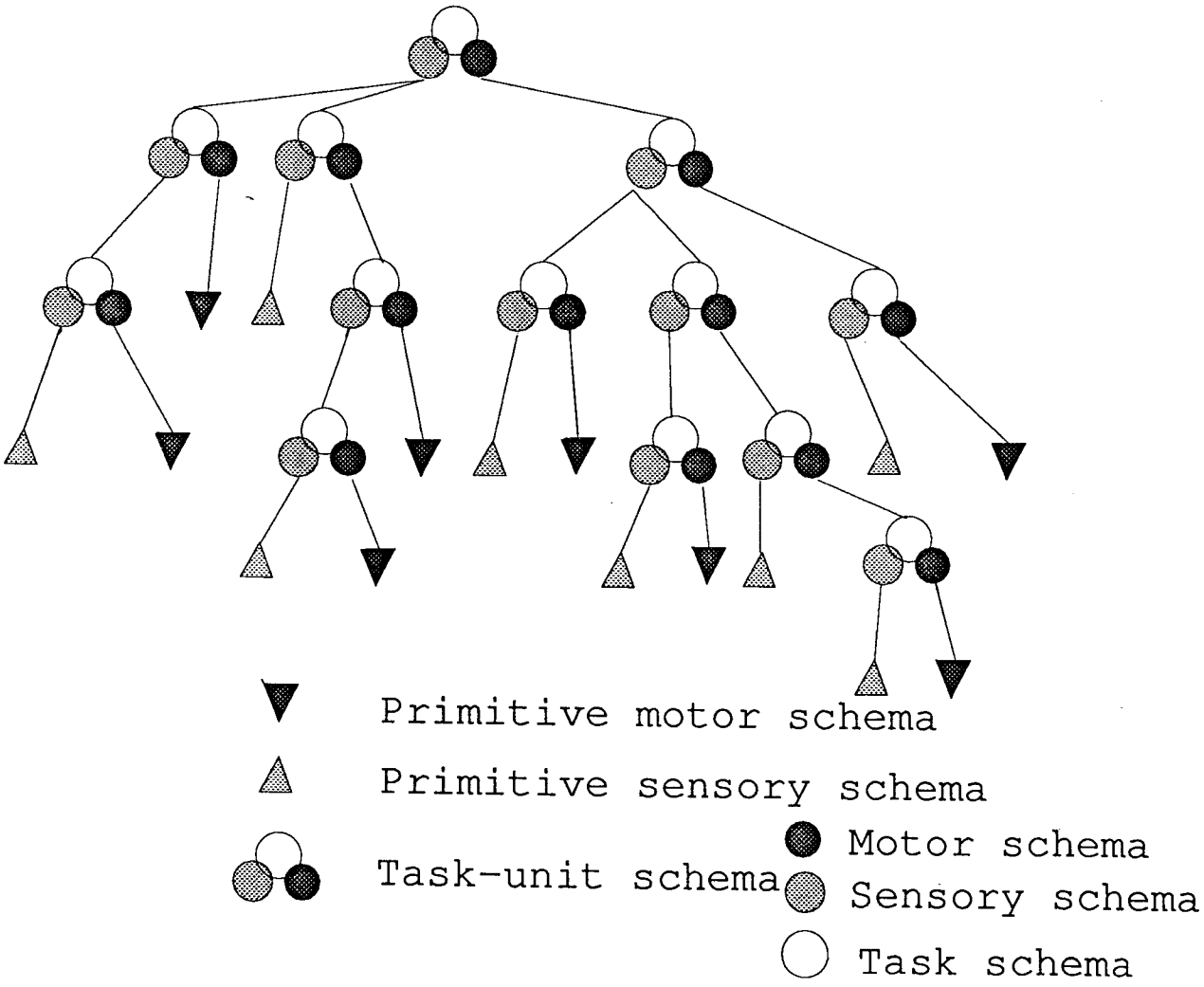

Figure 2: An example of a task-unit hierarchy.

Collapsing Hierarchical Networks. Sometimes it is useful to be able to 'collapse' hierarchies of assemblages. The following lemma defines how this can occur. The intuition here is that ultimately any $\mathcal{RS}$ plan is a network of processes, and the assemblage construct simply 'partitions-off' certain parts of the network. These partitions can be removed under the proper conditions.

The Flat Lemma.

$$\begin{array}{lcl} \textit{Given } \mathtt{Asm} & = & [\mathtt{A}^1,\ldots,\mathtt{A}^{i-1},\mathtt{A}^i,\mathtt{A}^{i+1},\ldots,\mathtt{A}^n]^c_e \\ \textit{and } \mathtt{A}^i & = & [\mathtt{B}^1,\ldots,\mathtt{B}^n]^{ci}_{ei} \\ \textit{then } \mathtt{Asm} & = & [\mathtt{A}^1,\ldots,\mathtt{A}^{i-1},\mathtt{B}^1/ei,\ldots,\mathtt{B}^n/ei,\mathtt{A}^{i+1},\ldots,\mathtt{A}^n]^{c\cup ci/ei}_e \\ & & \textit{where '/' denotes application of a renaming function.} \end{array}$$

Proof. $\mathtt{A}^i$ is an assemblage internally connected by ci and its ports on the component SIs renamed by ei. We can 'pop' the hidden components of the $\mathtt{A}^i$ assemblage out into the Asm assemblage only if we update the connection relations c and ci.

One way to view ei is as renaming function: it takes certain port names, and maps them to other port names (including null names, in order to hide ports). We can apply the renaming function to a schema or to a connection relation. In each case, its effect is to update some of the port names in the schema or connection relation. It is quite standard in the literature to write a 'renaming' function in the form $\mathtt{B}^1/ei$ rather than $ei(\mathtt{B}^1)$. The schema $\mathtt{B}^1/ei$ is the 'external' view of what the schema $\mathtt{B}^1$ contributes to the assemblage.

Now, ci needs to be updated to reflect the new port names after ei has been applied. ci/ei is the updated connection relation. When this is done, the two relations can simply be unioned to get the 'flattened' assemblage relation, $c \cup ci/ei$.

Examples. There are a number of sources of examples of $\mathcal{RS}$ notation for specification and for analysis:

1. [12, 15] contain easily understandable specification examples. [11] contains good examples, but the notation diverges from that discussed here.
2. [10, 18] contain a more detailled discussion of the motivation and mathematical structure of the model.
3. [10, 14, 17] contain examples of program analysis.

6 Summary and Future Work

This paper has motivated and introduced the basic $\mathcal{RS}$ model. This model is the lowest-level of description in the $\mathcal{RS}$ model, and all other levels of description and analysis are built on top of it. Future papers in this series will detail the following parts of the $\mathcal{RS}$ model:

1. Model Semantics: In order to ensure that the basic $\mathcal{RS}$ model it is well defined and to facilitate program analysis in the model, it is necessary to construct a mathematical semantics. That is, the concepts in the basic $\mathcal{RS}$ model need to be mapped onto mathematical objects. The semantics of the model is constructed using a particular kind of state machine called a *port automaton* develoedp by Steenstrup, Arbib and Manes[23].

2. Process Composition: Process composition is the operation of taking a network of SIs and grouping them to appear as a single SI. The basic $\mathcal{RS}$ model has one process composition operation: the assemblage. This is the most general composition possible. A number of other important composition operations can be constructed. These facilitate expressive and analytical work, but don't add to the power of the basic $\mathcal{RS}$ model, which is already a full model of computation.

3. Logical Analysis: One way to analyze programs in $\mathcal{RS}$ is to treat the model as an interpretation for a logic. A temporal logic was introduced for this purpose in [10], and later improved to an interval temporal logic in [17].

4. Algebraic Analysis: Looking at the algebraic properties of the notation opens up another avenue to study program properties. This approach is akin to a mixture of the discrete-event control field with the process algebra field.

5. Implementation: Constructing a computer emulation of the $\mathcal{RS}$ model involves first defining an appropriate programming language [6], and constructing a multiprocessor kernel to interpret the language. [13] is an initial specification of such a kernel that executes the procedural programming language described in [10]. A uniprocessor version of this kernel was built to work both under the SPINE system and Unix.

Later papers in this series will deal with many of these topics. The bibliography lists all the current works on $\mathcal{RS}$.

[6] Based, perhaps but not necessarily, on the model notation.

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] 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.

[3] 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.

[4] 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.

[5] Iberall, T., "The Nature of Human Prehension: Three Dextrous Hands in One" *Proceedings, IEEE Int. Conf. Robotics and Automation*, Raleigh NC, Apr. 1987, pp.396-401.

[6] 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.

[7] 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.

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

[9] Lyons, D.M., "A Generalization of: A Simple Set of Grasps for A Dextrous Hand" *COINS Tech. Rep. 085-37* University of Massachusetts at Amherst, 1986. *Proceedings of the 1985 International Conference on Robotics and Automation*, St. Louis, MO, Mar. 25–28, 1985, pp.588–593.

[10] 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, 1986.

[11] 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.

[12] Lyons, D.M., "A Novel Approach to High-Level Robot Programming" *Philips Technical Note TN-87-039* April, 1987. *Proceedings, Workshop on Languages for Automation,* Vienna Austria, 1987.

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

[14] Lyons, D.M., "The Task Criterion in Grasping and Manipulation" *Philips Technical Note* TN-87-163, December 1987. Updated TN-88-045, April 1988.

[15] Lyons, D.M., "On-line allocation of robot resources to task plans" *1988 SPIE Symposium on Advances in Intelligent Robotics Systems* 6-11 Nov., Cambridge, MA, 1988.

[16] Lyons, D., et al. "Adaptive Task Planning" Research proposal March 1989.

[17] Lyons, D.M., and Pelavin, R.N., "An Analysis of Robot Task Plans using a Logic with Temporal Intervals" *Philips Technical Note TN-88-160*, November, 1988.

[18] Lyons, D.M., and Arbib, M.A., "A Formal Model of Computation for Sensory-Based Robotics" To be published: *IEEE Trans. on Robotics & Automation*, June 1989.

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

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

[21] Pocock, G., "A Distributed Real-Time Programming Language for Robotics" *Proceedings, IEEE Int. Conf. on Rob. & Aut.*, May 14–19, 1989, Scottsdale Arizona.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] Wu, H.W., et al, "Purposive Sensing" Research proposal, March 1989.