Fundamentals of $\mathcal{RS}$ — Part II: Automata Semantics and Process Composition

Damian M. Lyons

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

ABSTRACT

This paper is the second in a series of documents describing a standard motivation, notation and semantics for the $\mathcal{RS}$ model of computation. Standardization is critical because $\mathcal{RS}$ is being used in a number of different areas. This paper, and those which follow it, present a coherent view of the $\mathcal{RS}$ model and related methods and algorithms. This paper describes process composition operations and defines a set of basic or primitive, atomic processes. (A previous version of this paper, without the automata semantics and section on process equality, was co-authored with Indur Mandhyan).

Keywords: Robot programming, Task Planning, Plan Representation, Plans, Model of Computation, $\mathcal{RS}$, Schema, Schemas, Process Composition, Process Algebra.

Fundamentals of $\mathcal{RS}$ — Part II: Automata Semantics and Process Composition

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 second in a series of documents standardizing the $\mathcal{RS}$ model of computation and related methods and algorithms. This model has previously been used for robot assembly plan representation and for dextrous hand control. It is being used in a number of different components of the task planning domain and to study real-time robot computing issues. 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. (A previous version of this paper, without the automata semantics and section on process equality, was co-authored with Indur Mandhyan).

The first paper[15] in this series presents a summary of the motivation and structure of the lowest-level of description in $\mathcal{RS}$, the so-called *basic $\mathcal{RS}$ model.* This second paper builds a set of processes composition operators on top of the basic model. The term 'process' is used in the literature as a synonym for the term 'computing agent' [3]. The computing agents in the $\mathcal{RS}$ model are schema instances (SIs). Thus, process composition in the $\mathcal{RS}$ model is SI composition. We will stick with the term process composition, however, because it already has wide acceptance.

Informally speaking, a process composition operator maps a tuple of processes onto a single process that is composed, in some specified way particular to the operator used, from the tuple of processes. These operators can be used to express temporal ordering, decision making, and synchronization with the external world. Such expression is necessary for robot task plans that interact robustly and efficiently with the environment[11].

We will formalize process composition in an algebraic setting. It is important to have a precise semantics for process composition for the following reasons:

1. To implement it correctly on a computer.
2. To formally analyze plans, and
3. to automate reasoning about composition for autogeneration of plans.

We start with a brief introduction to the automata semantics of $\mathcal{RS}$. We then describe the first set of composition operators, *atomic* compositions. The operators are defined in terms of operations on the automata that are the sematics of $\mathcal{RS}$ processes. For continuity, some of the material from the first paper in this series[15] has been included at this point. In order to 'ground' process composition (in an inductive sense), it is necessary to postulate a set of basic or primitive, atomic processes: processes that are not themselves constructed from process composition operators. We introduce a useful set of primitive processes in this paper, and this will pave the way for a brief presentation of the algebraic properties of the atomic composition operations. The second set of composition operators that we will present here are non-atomic (i.e., they can be built from the first set). Their algebraic properties are presented briefly.

1.1 Port Automaton Definitions

The port automata definition presented here differs from the orginal port automata model of Steenstrup et al.[17] as described by Lyons&Arbib [14], and differs slightly from the Lyons&Arbib model to provide better support for defining the composition operators.

Definition 1 A *port automaton* is a collection of objects and maps

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

where,

- $L = L_x \cup L_y$ is the port set, where L_x is the set of *input ports* and L_y is the set of *output ports.*
- Q is the set of *states.*
- $\tau \in 2^Q$ is the set of *initial states.*
- $\bar{\tau} \in 2^Q$ is the set of *final 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.
- $\delta : Q \times \hat{X} \rightarrow 2^{Q \times \hat{Y}}$ 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 $\bar{x} \leq x$ on $\hat{X}$ if $x = (x_1, \ldots, x_n), \bar{x} = (\bar{x}_1, \ldots, \bar{x}_n)$ and for each i either $\bar{x}_i = x_i$ or $\bar{x}_i = \#$. The *full-state transition map* is

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

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

Definition 2 $(x', q', y') \in \varphi(x, q, y)$ just in case there exists $\bar{x} \leq x$ 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 3 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 reads and writes of an admissible sequence of full states.

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, 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 4 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 $\sum_{P^i \in M}^{C} P^i$.

The Semantic Mapping from the $\mathcal{RS}$ model to the Port Automata Model.

The semantics of an $\mathcal{RS}$ schema S is the augmented port automaton P_s constructed based on S according to the procedure outlined in [5]. (The extensions we have made here to the port automata definitions do not affect this mapping.) We write the mapping as follows:

$$[\![\mathrm{S}]\!] = P_s \tag{2}$$

The semantics of an $\mathcal{RS}$ schema instance SI of schema S is a port automaton *plus* an initial state for that automaton. We write this is

$$[\![\mathtt{SI}]\!] = (P_s, \tau_{SI}) \tag{3}$$

The semantics of an assemblage or network of processes is the network automaton built using the semantics of each component process, i.e.

$$[\![[\mathtt{A_1}, \mathtt{A_2}, \ldots, \mathtt{A_n}]^c]\!] = \sum_{i=1,\ldots,n}^{C=f(c)} [\![\mathtt{A_i}]\!] \tag{4}$$

The communication map C is built from the $\mathcal{RS}$ communication relation c by the function $f()$. This function removes fan-in and fan-out and deals with any default ports/connections[5].

2 Atomic Composition Operations

In $\mathcal{RS}$, a schema is a parametrizable description of a computing agent. A schema instance or SI is a computing agent. This is supported in the port automaton semantics by saying that a schema has a set of starting states; making an instance of that schema involves choosing one particular starting state. As we have noted, we will continue to use the word 'process' to denote an SI. We adopt this terminology because there is already a sizable literature in process composition and process algebra that is directly related to the composition operations of $\mathcal{RS}$(e.g., [3]).

A process composition operation is an operation on processes whose resultant is also a process. The name 'composition operation' is meant to indicate that the process arguments are composed together in some way to produce the resultant. We have already described one form of process composition in the first document in this series [15], that is, the assemblage operation. The semantics for the assemblage operation is based on the network automaton in the port automata model. The following assemblage definition and example have been summarized from [15] for continuity. The composition semantics is given by definition (4).

Assemblage Composition. A set of SIs with a port connection relation between them can be grouped together into a single SI using the assemblage operation. This is written as

$$\mathtt{Assem} = [\mathtt{A}^1, \mathtt{A}^2, \ldots]_e^c \tag{5}$$

where $\mathtt{A}^1, \mathtt{A}^2, \ldots$ are the component SIs of the assemblage, c is the port-to-port connection relationship that specifies how the SIs can communicate with each other, and e is a port renaming mapping that specifies which of the component SIs ports are visible as ports of the assemblage SI and what their names are. All the SIs in an assemblage are *concurrent* and interlinked only by

the connection relation. (In future examples, if c and e are null or obvious we shall omit them.) The external view of an assemblage is identical to the external view of a single SI; in other words, assemblage composition implements *hiding* of its internal network.

Network Composition. We will also define a version of assemblage composition that does not hide its internals. Network composition has the same effect as assemblage composition, but connected and unconnected ports ar not hidden in any way, and the external view of a network is the same as the internal view. Network composition is concurrency without hiding. This is written as

$$\texttt{Net} = (\texttt{A}^1, \texttt{A}^2, \ldots)^c \tag{6}$$

where $\texttt{A}^1, \texttt{A}^2, \ldots$ are the component SIs of the assemblage, and c is the port-to-port connection relationship that specifies how the SIs can communicate with each other. All the SIs in the network are concurrent and interlinked only by the communication relation. (In future examples, if c is null we shall omit it).

Note. Since $[(\texttt{A}^1, \texttt{A}^2, \ldots)^c]_e = [\texttt{A}^1, \texttt{A}^2, \ldots]_e^c$, we can treat $[]_e$ as a purely hiding operation.

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

$$\texttt{Joint}_i\texttt{(s)}() = [\texttt{Jpos}_i()(\texttt{x}), \texttt{Jset(s,x)(u)}, \texttt{Jmot}_i\texttt{(u)}()]_e^c$$

$$\begin{aligned} c:\ & (\texttt{Jpos},\texttt{x}) \mapsto (\texttt{Jset},\texttt{x}) \\ & (\texttt{Jset},\texttt{u}) \mapsto (\texttt{Jmot},\texttt{u}) \\ e:\ & (\texttt{Jset},\texttt{s}) \mapsto \texttt{s} \end{aligned} \tag{7}$$

For more examples of such networks, see [5, 7, 10].

The following composition operations have been developed to represent the flow of control in a robot program. Their semantics are all based on the network automaton, but with some minor variations.

Sequential Composition. The sequential composition operation describes a process `T` as a direct sequence of two other processes, `P` and `Q`. `T` behaves like `P` until `P` terminates, then it behaves like `Q`. This sequential composition is written

$$\texttt{T} = \texttt{P};\texttt{Q} \tag{8}$$

(There is a long history of using ';' to denote this form of sequential relationship.)

Semantics The semantics of sequential composition, intuitively, is that the resultant automaton behaves like the semantic image of the first argument, until that machine enters one of its final states, and the resultant automaton thereafter behaves like the semantic image of the second automaton.

The semantics of the sequential composition

$$[\![\mathtt{T} = \mathtt{Q};\mathtt{R}]\!] = [\![\mathtt{Q}]\!];||[\![\mathtt{R}]\!]$$

is defined in terms of the automaton composition ;|| as follows. Let $[\![\mathtt{T}]\!] = P^T, [\![\mathtt{Q}]\!] = P^Q, [\![\mathtt{R}]\!] = P^R$ where P^i is a port automaton. The ;|| composition automaton P^T is defined in terms of P^R and P^Q as follows:

- Ports: $L_x^T = L_x^R \cup L_x^Q$, $L_y^T = L_y^R \cup L_y^Q$, and similiarly for X and Y for the automata.
- State set: $Q^T = Q^R \cup Q^Q$.
- Initial and final states: $\tau^T = \tau^Q$, $\bar{\tau}^T = \bar{\tau}^R$.
- State transition map

$$\delta^T(q,\hat{x}) = \begin{cases} \delta^R(q,\hat{x}) & if\ q \in Q^R \\ \delta^Q(q,\hat{x}) & if\ q \in (Q^Q - \bar{\tau}^Q) \\ \delta^R(q_0,\hat{x}) & if\ q \in \bar{\tau}^Q,\ q_0 \in \tau^R \end{cases}$$

Example. If $\mathtt{T}^i$ for $i = 1 \ldots 5$ are a set of actions then the network $\mathtt{T}$ defined as

$$\mathtt{T} = [\mathtt{T}^1;\mathtt{T}^2\ ,\ \mathtt{T}^3;\mathtt{T}^4];\mathtt{T}^5 \tag{9}$$

has the following behavior. $\mathtt{T}^1$ and $\mathtt{T}^3$ are begun concurrently. Whenever $\mathtt{T}^1$ dies, then $\mathtt{T}^2$ is started; Whenever $\mathtt{T}^3$ dies, then $\mathtt{T}^4$ is started, but both 'streams' (i.e., $\mathtt{T}^1$ and $\mathtt{T}^3$) remain concurrent. When *all* of $\mathtt{T}^1$ to $\mathtt{T}^4$ have terminated, then $\mathtt{T}^5$ is started. $\mathtt{T}$ terminates when $\mathtt{T}^5$ terminates.

Precondition Composition. The precondition composition operation between SIs $\mathtt{P}$ and $\mathtt{Q}$ is written as

$$\mathtt{T} = \mathtt{P}\langle v1, v2, \ldots\rangle : \mathtt{Q}_{v1,v2,\ldots} \tag{10}$$

Informally, this means that the resultant SI $\mathtt{T}$ behaves like $\mathtt{P}$ until $\mathtt{P}$ successfully terminates, it then behaves like $\mathtt{Q}$ with parameters $v1, v2, \ldots$ chosen by $\mathtt{P}$. If $\mathtt{P}$ unsuccessfully terminates, then $\mathtt{T}$ also unsuccessfully terminates. This is called precondition composition because $\mathtt{P}$ enables the execution of $\mathtt{Q}$.

Semantics The difference between sequential and conditional (or enabling) composition is that the latter can pass a value onto its consequent process *and* that it will abort if the first argument aborts. We need to define the meaning of an abort or unsuccessful ending. The simplest way to do this is to consider $\bar{\tau}$ for any machine to be partitioned into two sets based on a function *abort*:

Definition 5 The function $abort^P : \bar{\tau}^P \rightarrow \{0,1\}$ partitions the set of end states of process P into two: if $abort(q) = 1$ we call q an abort ending. We indicate the subset of $\bar{\tau}$ for which *abort* is 1 as $\bar{\tau}_a$. Every process should have at least one abort state, even if that is it's only end state.

We now need to define a connection between the end states of the first process and the start states of the next. We call this connection the parameterization function p defined as follows:

Definition 6 The parameterization function for a conditional composition between Q and R is a function $p^{Q,R} : \bar{\tau}^Q \rightarrow \tau^R$ that captures the way in which the final state of Q influences the first state, or parameterization of R . If $q \in \bar{\tau}_a{}^Q$ then $p^{Q,R}(q) \in \bar{\tau}_a{}^R$.

The semantics of the conditional composition

$$[\![\mathtt{T} = \mathtt{Q}\langle v\rangle : \mathtt{R_v}]\!] = [\![\mathtt{Q}]\!] : \|_v [\![\mathtt{R_v}]\!]$$

is defined in terms of the automaton composition : || as follows. Let $[\![\mathtt{T}]\!] = P^T, [\![\mathtt{Q}]\!] = P^Q, [\![\mathtt{R}]\!] = P^R$ where P^i is a port automaton. The : || composition automaton P^T is defined in terms of P^R and P^Q as follows:

- Ports: $L_x^T = L_x^R \cup L_x^Q$, $L_y^T = L_y^R \cup L_y^Q$, and similiarly for X and Y for the automata.
- State set: $Q^T = Q^R \cup Q^Q$.
- Initial and final states: $\tau^T = \tau^Q$, $\bar{\tau}^T = \bar{\tau}^R$.
- State transition map

$$\delta^T(q,\hat{x}) = \begin{cases} \delta^R(q,\hat{x}) & if\ q \in Q^R \\ \delta^Q(q,\hat{x}) & if\ q \in (Q^Q - \bar{\tau}^Q) \\ \delta^R(p_v^{Q,R}(q),\hat{x}) & if\ q \in \bar{\tau}^Q \end{cases}$$

  $p_v^{Q,R}$ is the parameterization function that maps the end states to start states based on the value of the variable v, and also aborts T should Q abort.

Example. Let $\mathtt{Locate}_m\langle si\rangle$ be a schema that causes the sensory system on a robot to search its environment for an object that matches the model m, and that maintains a pointer to this object in si. Let $\mathtt{Grasp}_i$ be a schema that when given a pointer to an object recognized by the sensory system causes the robot to reach out and grasp that object. We can couple these schemas into a simple network

$$\mathtt{Acquire}_m = \mathtt{Locate}_m\langle si\rangle : \mathtt{Grasp}_{si} \tag{11}$$

This network has the behavior that it will cause the robot to search for an object of model m in its environment and then reach out and grasp that object.

Transcondition Composition. While precondition composition effectively allows one SI to enable the execution of another, transcondition composition allows one SI to *terminate* the execution of another. This is written

$$\mathtt{T} = \mathtt{P}\#\#\mathtt{Q} \tag{12}$$

T behaves like the network (P,Q) until either Q terminates, in which case it behaves like P , or until P terminates, in which case T terminates. Note that P is unaffected if Q dies, but if P dies, Q is forced to die.

Semantics Transcondition composition results in a machine that is very like a concurrent composition, except with respect to the end states: if either machine enters an end state, then the other machine is forced to enter an abort state. (This is why every machine must have at least one abort state). In the following definition, let $P^{|}$ refer to the machine that is the semantics of the concurrent composition of the two process Q and R with some connection map c.

The semantics of the transcondition composition

$$[\![\mathtt{T} = (\mathtt{Q}\#\mathtt{R})^{c}]\!] = [\![\mathtt{Q}]\!]\#\|_{c}[\![\mathtt{R}]\!]$$

is defined as following in terms of the semantics of the concurrent composition $P^{|} = [\![(\mathtt{Q},\mathtt{R})^{c}]\!]$. All the components of $[\![\mathtt{T}]\!]$ are the same as those of $P^{|}$ *except* for the transition function. That function is defined as follows in terms of the transition function of $P^{|}$ such that whenever its $[\![\mathtt{Q}]\!]$ state component enters an abort state, $[\![\mathtt{T}]\!]$ immediately transitions to an abort state (i.e., a state in which both the $[\![\mathtt{Q}]\!]$ and $[\![\mathtt{R}]\!]$ components of its state are in abort states).

$$\delta^{T}((q^{Q},q^{R}),\bar{x}) = \begin{cases} \delta^{|}((q^{Q},q^{R}),\bar{x}) & if\ q^{Q} \notin \bar{\tau}^{Q}\ AND\ q^{R} \notin \bar{\tau}^{R} \\ \delta^{|}((q^{Q},qa^{R}),\bar{x}) & if\ q^{Q} \in \bar{\tau}^{Q}\ where\ qa^{R} \in \bar{\tau_{a}}^{R} \end{cases}$$

Example. Let GT be a schema that tests the environment of the robot and determines if the goal of task plan T has been accomplished. If T was a plan to finish an assembly, then GT can detect when the assembly is actually finished. The relationship between these processes can be expressed as

$$\mathtt{Plan} = \mathtt{GT}\#\#\mathtt{T} \tag{13}$$

Now, let IT be a schema that can test the environment to determine if the task plan T is still an appropriate way to achieve the desired goal. For example, let GT test if the robot end-effector is in contact with a surface, and T implement a compliant move. In this case, if an obstacle is detected between the robot end-effector and the surface, then a compliant move will no longer result in the end-effector contacting the surface. Hence the move should be aborted. The relationship between these three processes is

$$\mathtt{Plan} = \mathtt{IT}\#\#(\mathtt{GT}\#\#\mathtt{T}) \tag{14}$$

3 Primitive Processes

Schemas can be described only as networks of other schemas. To ground this recursion we need to specify a set of primitive, atomic schemas. In [14], primitive schemas were called *basic* schemas, and a special procedural language was used to define them. Here, we shall informally define the semantics of the primitive schemas, but clearly the language of [14] could be used to define them more concisely.

Strictly speaking, it is necessary to define primitive schemas that can evaluate expressions, print or read data, etc. This is quite straightforward[1], and is a chore similar to the construction of the procedural language in [14]. Here we shall bypass this tedious job and assume that arbitrary expressions can be written for the parameter values in an SI. For example, we shall allow all the following forms: $\mathtt{P}_{i=2x+x^2}$, $\mathtt{Q}_{sin(\theta)-1}$, $\mathtt{R}_{v1=ln(x),v2=e^{2x}}$.

This allows us to concentrate on those primitive processes that are novel to $\mathcal{RS}$. Our convention will be to write primitive schema/SI names as uppercase schema names. Note that semantics of each of these primitive schemas can be expressed as specific port automata. This step is not difficult and it is not included in this paper.

1. NULL terminates successfully immediately on creation. It has no ports and takes no parameters.

2. ABORT terminates unsuccessfully immediately on creation. It has no ports and takes no parameters.

3. INF never terminates once created. It has no ports and takes no parameters.

4. $\mathtt{IN}_p\langle v\rangle$ reads its input port p and can pass on that value in v if used in a precondition composition. It terminates as soon as the value is read. This schema implements full synchronous reception of messages on port p. See the next item for an example of the use of this schema.

5. $\mathtt{OUT}_{p,v}$ writes the value v to its output port p. It terminates once the write has completed. This schema implements full synchronous reception. In conjunction with the last item we can write a simple program that reads the input on a port p and writes it out again on another port q as follows:

$$\mathtt{IN}_p\langle v\rangle : \mathtt{OUT}_{q,v} \tag{15}$$

6. $\mathtt{DELAY}_t$ terminates after a period of duration t measured according to some universal clock. (We never intend to implement this schema, but it does play a useful role in analysis). It has no ports and takes no parameters.

$$\begin{aligned} \mathtt{T1} &= \mathtt{DELAY}_{t1}; \mathtt{DELAY}_{t2}; \mathtt{A} \\ \mathtt{T2} &= \mathtt{DELAY}_{t1+t2}; \mathtt{B} \end{aligned} \tag{16}$$

[1]It is interesting to note that such a set of atomic, primitive schemas is very much in the data-flow[16] style.

If T1 and T2 are started concurrently, then A and B will also start concurrently sometime later.

7. RAN terminates after waiting period of random duration. It has no ports and takes no parameters. The process RAN;A behaves like A started at some arbitrary time in the future. (As an alternative, we could have defined RAN = $\texttt{DELAY}_{t=random}$).

8. $\texttt{EXISTS}_s\langle si\rangle$ will terminate if an instance of schema s exists in the current assemblage scope, and the parameter si is a pointer to the instance located. We will give an example of the use of this schema presently.

Defining restrictions of a schema.

If $\texttt{A}_{v1,v2,v3}$ is a schema with 3 parameters, then we can define a more restricted version of this schema in which $v1 = x$, where x is some constant value, by writing

$$\texttt{AX}_{v1,v2} \equiv \texttt{A}_{x,v1,v2} \tag{17}$$

The validity of this restriction has the same basis as the *s-m-n* theorem of computability theory[2] by which a more restricted version of a program can always be defined by setting some of the inputs of the program equal to a constant. In automata theory, this boils down to the start set of the semantics of AX being a subset of the start set of the semantics of A . Note that both left and right sides of $\equiv$ refer to schemas not SIs.

The following useful schemas can be constructed almost directly from the primitive schemas.

1. STOP is sometimes a more intuitive name for NULL , so we define

$$\texttt{STOP} \equiv \texttt{NULL} \tag{18}$$

2. $\texttt{LOCATE}_m\langle i\rangle$ terminates if an object of model type m is recognized in the environment and i is a pointer to an SI representing that instance. If we assume that every instance of an object in the environment is represented by an instance of a schema with the same name as the model of the object, then

$$\texttt{LOCATE}_m\langle i\rangle \equiv \texttt{EXISTS}_m\langle i\rangle \tag{19}$$

4 When Are Processes Equal?

We say that two $\mathcal{RS}$ processes are equal if their semantics is the same port automaton, i.e.,

$$\texttt{A} = \texttt{B} \;\; iff \;\; [\![\texttt{A}]\!] = [\![\texttt{B}]\!] = P \tag{20}$$

To determine if two machines are equal, one must construct an isomorphic mapping from the components of one machine to the components of the other. Of course, one machine might have a number of no-ops or redundant transitions that would confuse this mapping. Thus we demand that both machines be reduced to some minimal canonical form using techniques such as the Myhill-Nerode theorem, and then compared. Now understand that it is not our intention to actually deduce such minimal machines, but rather to formalize the definition of equality. Our objective is to derive an approach so that two processes can be determined equal if, say, they reduce to an identical formula.

To this objective, the composition operator definitions can be treated as axioms of equality. If two processes can be reduced to identical formula in the algebra of composition operators and basic proceses, then they are equal to each other. We can distance ourselves even further from explicit machine equality by looking at the algebraic properties of the composition operators, and adding these into our store of 'axioms' of equality. Of course, the properties of the composition operators are not axioms, but rather theorems of the automaton semantics of the compositions. We do not present the proofs of these theorems here; with the exception of one case, they are all straightforward. That case has to do with the idempotance and the network composition.

Two processes are equal if they have the same semantic image. An network consisting of two equal processes cannot be isomorphically mapped to one of the processes without one extra constraint; that is, the only states used in the network be the subset of $Q \times Q$ such that $qn = (q, q)$. However, idempotance of network composition turns out to be a desirable property, because it allows us to frame distribution 'axioms' such as `A:(B,C)=(A:B,A:C)`. For deterministic machines, this can be achieved by forcing the equivalent ports on the network components to be tied together (whatever one receives, the other receives; whatever one transmits, the other transmits). In this way, they will always take the same transitions, and the state space is collapsed from $Q \times Q$ to Q. For nondeterministic machines, this is not sufficient, since even with identical input, the machines might choose to take different transitions. A port automaton is a non-deterministic machine, and there are several places in the $\mathcal{RS}$ model where non-determinism is supported; for example, the primitive process `RAN`. Note that `RAN:(A,B)` will not behave like `(RAN:A,RAN:B)`. In the first case, `A` and `B` start concurrently; in the second case, they *may* start concurrently or they may not.

It is because we want to support both non-determinism *and* idempotance of network composition, that we make the following definition:

Definition 7 *Any identically named and parameterized process terms in a network composition refer to the* same *process (SI), and* not *to distinct but identically parameterized instances of the same schema.*

Thus by this definition `(RAN:A,RAN:B=RAN:(A,B))` because both `RAN` terms on the right-hand side refer to the *same* process. However, there are cases when we specifically do not want terms to

be identified. Consider a process $\texttt{Select}_{\phi,N}\langle x\rangle$ that chooses a value $x \in N$ according to probability distribution ϕ over N. In that case, we couldn't get different random values by writing

$$\texttt{T} = (\texttt{Select}_{\phi,N}\langle x\rangle : \texttt{A}_x, \texttt{Select}_{\phi,N}\langle y\rangle : \texttt{B}_y) \tag{21}$$

In that case, we would prefer each `Select` to refer to a different process. For this we revert back to using our $\equiv$ operator to define different schemas from which to coin the identically parameterized and concurrent processes.

$$\begin{array}{lcl} \texttt{Select1}\langle x\rangle & \equiv & \texttt{Select}_{\phi,N}\langle x\rangle \\ \texttt{Select2}\langle x\rangle & \equiv & \texttt{Select}_{\phi,N}\langle x\rangle \\ \textit{and now} & & \\ \texttt{T} & = & (\texttt{Select1}\langle x\rangle : \texttt{A}_x, \texttt{Select2}\langle y\rangle : \texttt{B}_y) \end{array} \tag{22}$$

5 Some Algebraic Properties of the Composition Operators

The following table summarizes the most important properties of the simple composition operators. Communication is omitted from network composition for technical convenience (the flat-lemma of [15] describes how to 'rearrange' communication to provide associativity).

Notice that since no composition operators have inverses, none will form groups[2]. The strongest structure formed is a commutative monoid[3] (the assemblage operation). All the rest are monoids.

The transcondition operation has an interesting transitivity property when used in conjunction with network composition.

Property	Network	Precond./Seq.	Transcond.
Closed	By definition.	By definition.	By definition.
Associative	(A,(B,C))=((A,B),C)	A:(B:C)=(A:B):C	A##(B##C)=(A##B)##C
Commutative	(A,B)=(B,A)	A:B≠B:A	A##B≠B##A
Idempotent	(A,A)=A	A:A≠A	A##A=A
Identity	(A,NULL)=A	A:NULL=NULL:A=A	INF##A=A=A##INF
Inverse	None	None	None
Transitive	No	No	(A##B,B##C)=A##B##C

It is also important to understand how the operations interact with each other. One important such interaction is distribution. The following table captures what operations distribute over other operations (the row distributes over the column).

[2] I am not convinced that useful inverses don't exist.

[3] A monoid is a set operation which is closed, associative and has an identity element.

Left-Distributes	Assemblage	Precond./Seq.	Transcond.
Assemblage	—	No	No
Precond./ Seq.	`A:[B,C]=[A:B,A:C]`	—	`A:(B##C)=(A:B)##(A:C)`
Transcond.	`A##[B,C]=[A##B,A##C]`	No	—

Note that each of the non-commutative monoids left-distributes over the commutative monoid (assemblage). This means that precondition/seq. composition and transcondition composition each form semi-rings[4] with assemblage composition. Precondition/seq. composition left-distributes over transcondition composition, but since neither is commutative, we can't form a semi-ring.

Recall that our justification for looking at these algebraic properties was that these properties will be useful in testing implementations, as mechanisms for analysis of plans, and as the basis of algorithms for reasoning about plans. In the next paragraph we give some examples of reasoning using these properties.

Examples. Consider the issue of combining and simplifying plan segments: If we have some plan segment $\texttt{Robot}_R\langle r\rangle : \texttt{DoX}_r$ for allocating a robot r from some set R to a process that carries out an action `DoX` , and a similar plan segment $\texttt{Robot}_R\langle r\rangle : \texttt{DoY}_r$, and we want to join them sequentially, we can simplify

$$\begin{aligned} \texttt{Plan} &= \texttt{Robot}_R\langle r\rangle : \texttt{DoY}_r; \texttt{Robot}_R\langle r\rangle : \texttt{DoX}_r \\ &= \texttt{Robot}_R\langle r\rangle : (\texttt{DoY}_r; \texttt{DoX}_r) \end{aligned} \tag{23}$$

Now consider the problem of deducing the behavior of some agent of change in the world (also called a *process* in the literature, but to avoid confusion we won't use that name here). Consider a mechanism that works as follows: When a button is pressed and released, a bulb lights up for a fixed time period, proportional to how long the button was pressed. We model the button and the resultant light as processes as follows:

$$\texttt{Machine} = \texttt{Button}\langle t\rangle : (\texttt{Delay}_t \#\# \texttt{Light}) \tag{24}$$

`Button` dies with a value t when the button is released after having been pressed down for time t. $\texttt{Delay}_t$ is then created and dies again after time period t. `Light` represents the resulting light from the bulb. `Light` is forced to die when $\texttt{Delay}_t$ dies. `Machine` is a process network that captures the relevant part of the behavior of the machine we have described. Some simple deductions can be made from this network. If our goal is to achieve the process `Light` , then we can use the distributive property to get

$$\texttt{Machine} = (\texttt{Button}\langle t\rangle : \texttt{Delay}_t \#\# \texttt{Button}\langle t\rangle : \texttt{Light}) \tag{25}$$

This brings out the relationship between the button and the light as one operation, but it doesn't lose the information about the delay. This sort of reasoning can be done by implementing the algebraic rules using a graph formalism or a language like Prolog.

[4] A semi-ring is a pair in which the first element is a commutative monoid and the second is a monoid that distributes over the first.

As another example, consider a sleeper alarm on a radio: A timer dial can be used to regulate how long a radio will play before automatically turning itself off. We model the button, the dial and the radio as processes:

$$\texttt{Sleeper} = (\texttt{Button}\#\#\#\texttt{Dial}\langle v\rangle) : (\texttt{Delay}_v\#\#\#\texttt{Radio}) \tag{26}$$

When then button is pressed the process `Button` dies. When `Dial` dies the current value of the dial is captured in v. In this way, then the button is pressed the timer dial value is contained in v. The radio is playing as long as the process `Radio` lives. The value v is used to parameterize a `Delay` schema so that the radio is killed after time v has elapsed.

6 Non-atomic Composition Operators

The unidirectional transcondition has the unfortunate effect that it can leave 'dangling' processes. For example `A##B` will disappear if `A` dies, but if `B` dies, `A` is left dangling. We introduce a commutative version of the operation to solve this problem.

Bidirectional transcondition composition The bidirectional transcondition operation between SIs `P` and `Q` is written

$$\texttt{T} = \texttt{P}\#\texttt{Q} \tag{27}$$

The SI `T` behaves like the network `(P,Q)` until either `P` or `Q` dies, in which case `T` dies.

Semantics. Even though the bidirectional composition operation has relatively pleasant algebraic properties, we can still construct it from the atomic operation if we want. We can define this operation in terms of the unidirectional transcondition as

$$\texttt{A}\#\texttt{B} = \texttt{A}\#\#\texttt{B}\#\#\texttt{A} \tag{28}$$

Examples. We can rephrase the scenario we used with the unidirectional transcondition as follows (recall that `T` is the task, `GT` is the goal condition and `IT` is a task invariant):

$$\texttt{Plan} = (\texttt{GT}\#\texttt{T})\#\texttt{IT} \tag{29}$$

Note that whenever any of these processes die, then all die; no processes are left dangling.

Continuing the example of the sleeper alarm from the previous section (equation (26)): To cater for the case when a favorite program comes on the radio and the alarm needs to be reset, we augment the second network:

$$\texttt{Sleeper} = (\texttt{Button}\#\texttt{Dial}\langle v\rangle) : ((\texttt{Radio}\#\texttt{Delay}_v\#\texttt{Button}); \texttt{Sleeper}) \tag{30}$$

So now if the button is pressed a second time, the timer can be reset.

None of the composition operations we have introduced until now involve any repetition or iteration. The following two operations are defined to recursively repeat themselves. Note that this means they will never terminate unless they are the arguments to a transcondition operation. We call these operations the *recurring precondition* composition operations. We will introduce two recurring precondition operations: one synchronous in its recurrence, and one asynchronous. Because these operations have very unpleasant algebraic properties (they are not even associative!) we will not define them as atomic operations, but rather as recursively constructed from our atomic composition operations.

Asynchronous Recurring Precondition Composition The asynchronous recurring precondition operation between SIs P and Q is written as

$$\mathtt{T} = \mathtt{P}\langle v \rangle :: \mathtt{Q}_v \tag{31}$$

Informally, when P terminates successfully, Q is created in the standard way, but then a recursive copy of P::Q is also created. Thus process P recurs as long as T is alive, and P asynchronously triggers instances of Q . If P aborts, then P::Q aborts; this grounds the recursion.

Semantics. We can define this operation in terms of the precondition operation as

$$\mathtt{P} :: \mathtt{Q} = \mathtt{P} : (\mathtt{Q}, \mathtt{P} :: \mathtt{Q}) \tag{32}$$

Synchronous Recurring Precondition Composition The synchronous recurring precondition operation between SIs P and Q is written as

$$\mathtt{T} = \mathtt{P}\langle v \rangle :; \mathtt{Q}_v \tag{33}$$

Informally this means that Q is created when P terminates successfully, in the standard way, but when Q terminates, then a recursive copy of P:;Q is created. This is the synchronous version of the previous operation. Rather than continuously firing off copies of Q , a new copy of Q can only fire if every previous copy has already terminated. If P aborts, then P:;Q aborts; this grounds the recursion.

Semantics. We can define this operation in terms of the precondition and sequential operations as

$$\mathtt{P} :; \mathtt{Q} = \mathtt{P} : (\mathtt{Q}; \mathtt{P} :; \mathtt{Q}) \tag{34}$$

Example. If $\mathtt{Tray}\langle t \rangle$ is a schema that can determine when a tray of assembly components for assembly model t arrives at the robot station, and $\mathtt{Assemble}_i$ is the assembly plan for assembly model i, then

$$\mathtt{Tray}\langle t \rangle :; \mathtt{Assemble}_t \tag{35}$$

is a network that repeatedly triggers an appropriate assembly plan when a tray arrives, and when the previous tray has been finished. If there is more than one robot available to carry out the assembly, then we can rewrite this as

$$(\texttt{Tray}\langle t\rangle, \texttt{Robot}\langle r\rangle) :: \texttt{Assemble}_{t,r} \tag{36}$$

where `Robot` will only terminate if there is a robot r available to do the assembly. The asynchronous recurring precondition is used here because more than one assembly can be concurrently in progress in the case when more than one robot is available.

As another example, consider the following mechanism: An alarm clock rings for a short time when first triggered, it then sleeps for a while and re-triggers again periodically until a cancel button is pressed.

$$\texttt{Repeater} = \texttt{Button}\#(\texttt{Delay}_{t1} :; (\texttt{Ring}\#\texttt{Delay}_{t2})) \tag{37}$$

The alarm rings as longs as `Ring` is alive. The synchronous recurring precondition is used to sequence two delays: one to kill `Ring` after time $t2$ and one to interspace alarm rings $t1$. The whole network is killed when the button is pressed and `Button` dies.

7 Some More Algebraic Properties

The algebraic properties of these new operations are shown in the table below. Note that bidirectional transcondition composition is a commutative monoid, and as we mentioned, the recurring operations have very ugly properties. It is important to note that these recurring precondition operations don't form a monoid. This is not too important, since the definitions of these non-atomic operations can be used to rephrase networks containing them into a more tractable form. The bidirectional transcondition operation also satisfies an interesting transitivity condition.

Property	Bi. Trans.	Async. Rec. Pre.	Synch. Rec. Pre.
Closed	By definition.	By definition.	By definition.
Associative	`(A#B)#C=A#(B#C)`	No	No
Commutative	`A#B=B#A`	No	No
Idempotent	`A#A=A`	No	No
Identity	`A#INF=A`	No	No
Inverse	No	No	No
Transitive	`[A#B,B#C]=A#B#C`	No	No

As before, we can explore what operations distribute over what other operations. We include assemblage composition in the table, and consider the two recurring operations as one case.

Distributes	Network	Sync/Async. Precond.	Bi. Transcond.
Network	—	No	No
Sync/Async. Precond.	`A::(B,C)=(A::B,A::C)`	—	No
Bi. Transcond.	No	No	—

Note that the bidirectional transcondition behaves much like the undirectional version, except that now it forms a semi-ring with the precondition/sequential composition operations (*not* the recurring precondition operations, since they don't form a monoid).

8 Summary and Future Work

The basic $\mathcal{RS}$ model is described in the first paper in this series [15]. This present paper has described the process composition operations, a set of atomic, primitive processes, and the macro operation, where 'process' is synonymous with schema instance (SI). This additional structure has enriched the ways in which temporal ordering and synchronization with external events can be expressed in $\mathcal{RS}$. Since these additions have been characterized algebraically, they also now improve the way in which plans and process models can be analyzed or reasoned about formally.

Together, the basic $\mathcal{RS}$ model and the composition operations form the major part of the $\mathcal{RS}$ model. The following are the other components:

1. Model Semantics: The basic model is explained informally in [15]. In order to ensure that it is well defined and to facilitate program analysis in the model, it needs to have a mathematical semantics. That is, the concepts in the basic 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.* Both [5] and [14] contain a formal semantics for the basic $\mathcal{RS}$ model.
2. 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 [5], and later improved to an interval temporal logic in [13].
3. 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[3].
4. Implementation: Constructing a computer emulation of the $\mathcal{RS}$ model involves first defining an appropriate programming language [5], and constructing a multiprocessor kernel to interpret the language. [8] is an initial specification of such a kernel that executes the procedural programming language described in [5]. A uniprocessor version of this kernel was built to work both under the SPINE system and under Unix.

Later papers in this series will deal with many of these topics. The bibliography in this paper contains a full list of material relevant to $\mathcal{RS}$.

Acknowledgements. We would like to thank the various reviewers of this paper for their comments. Also thanks to Teun Hendricks for suggesting the alarm clock and sleeper examples.

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

References

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

[2] 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.

[3] Hennessy, M., Algebraic Theory of Processes. MIT Press, 1988.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] Lyons, D., et al. "Adaptive Task Planning" Research Proposal, March 1989.

[12] 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.

[13] 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.

[14] Lyons, D.M., and Arbib, M.A., "A Formal Model of Computation for Sensory-Based Robotics" *IEEE Trans. on Robotics & Automation*, Vol.5 No.3, June 1989, pp.280–293.

[15] Lyons, D.M., "The Fundamentals of $\mathcal{RS}$ — Part I: The Basic $\mathcal{RS}$ Model" *Philips Technical Report* TR-89-031, May 1989.

[16] Misunas, D.P., "A Computer Architecture for Data-Flow Computation," *Technical Report* MIT/LCS/TM-100, Department of Computer Science and Electrical Engineering, MIT, Cambridge, MA, 1978.

[17] 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.