For inclusion in the Proceedings of the ZiF Conference on Prerational Intelligence: Phenomenology of complexity emerging in systems of simple interacting agents. Center for Interdisciplinary Studies, University of Bielefeld, Germany 1994.

TOPIC I.4 Schema Theory as a Framework for Intelligence and Action.

TITLE:
A schema-theory approach to specifying and analyzing the behavior of robot systems.

Damian M. Lyons

Philips Research
345 Scarborough Road
Briarcliff Manor NY 10510
PH: (914) 945-6444 FX: (914) 945-6141
EM: dml@philabs.philips.com

Abstract

Schema Theory was developed by Arbib to represent systems in both Artificial Intelligence and Brain Theory. This paper presents an overview of Arbib's Schema Theory and its applications. It delves into more detail on one specific application of schema theory called *Robot Schemas* ($\mathcal{RS}$). The $\mathcal{RS}$ language is an attempt to design a special purpose model of computation for robot programming. In particular, this paper shows how $\mathcal{RS}$ can be used to address the problem of building a robot system whose reactions exploit patterns of activity in its environment.

Keywords: Schema Theory, Robot Schemas, $\mathcal{RS}$ Brain Theory, Reactive Robots, Design of Robot Controllers.

A Schema-Theory Approach to Specifying and Analyzing the Behavior of Robot Systems.

Damian M. Lyons
Philips Laboratories
Philips Electronics North America Corporation
345 Scarborough Road
Briarcliff Manor NY 10510

1 Introduction

Schema Theory was developed by Arbib [1, 2, 4, 3] to represent systems in both Artificial Intelligence and Brain Theory [1]. This paper presents an overview of Arbib's Schema Theory and its applications. It delves into more detail on one specific application of schema theory called *Robot Schemas.*

The remainder of the paper is divided into three parts. Section 2 is a broad introduction to schema theory with some reference to various applications and implementations of it. Schema theory is not a precise, mathematical theory, but rather a set of guiding principles that can be shared both by researchers building models of animal behavior and those building artificial agents. Section 3 introduces the Robot Schemas ($\mathcal{RS}$) model. This was developed as one approach for applying schema theory concepts to the design and analysis of robot behavior.

Section 4 describes an example of using $\mathcal{RS}$ to build a reactive system that can exploit its environment. Like Horswill [14], our objective is to allow the agent to take advantage of special features of the environment. Horswill's *habitat constraints* allow a designer to simplify the design of an agent so as to exploit its 'niche' in the environment. Our approach is an attempt to automate the identification and use of a class of such constraints, namely constraints that express how an agent can exploit patterns of activity in the environment. Sections 5 and 6 present the details for using $\mathcal{RS}$ to address this kind of design problem. The basic approach will be to use $\mathcal{RS}$ to design a "simulation" model of the environment in which the agent will be situated. Some techniques will then be introduced to allow us, as designers, to explore the model for patterns of activity, that we can use in the construction of a robot controller.

2 Schema Theory

This section of the paper presents an overview of Arbib's schema theory and some of its applications. It will necessarily be brief. The best reference for the 'full story' is [2]. Behavior is produced in schema theory as the result of cooperation and competition between a distributed network of schemas — what Arbib calls a concurrent control program — rather than by executive control. One of the most interesting features of the theory is the support it thereby offers to instinctual or prerational intelligence as opposed to the highly rational approach to intelligence implied by AI systems such as SOAR [3].

2.1 A Schema

A *schema* is both a store of prototypical knowledge, as in Minsky's Frames, *and also* the procedural information for applying that knowledge. Applications of schema theory will implement schemas in different ways, but a crucial common element is seeing the unit of representation as a combination of procedural and declarative. A schema *instance* is an active process for applying the information contained in the schema. A schema can be *instantiated* as a response to sensory information or by the command of other schema instances. There could be multiple instances of the same schema, each a distinct process with its own parameters.

[1] Which he defines as Cognitive Science, Neuromodeling and so on.

2.2 Perceptual and Motor Schemas

Arbib hypothesizes two kinds of schemas. *Perceptual schemas* map sensory information or other information about the agent's environment onto domains of interaction between agent and the environment. *Motor schemas* are prototypical action patterns available to the agent. The perception of some facet of the environment (e.g., an 'object') corresponds to the generation of instances of one or more perceptual schemas. These, in turn, identify potential domains of interaction between the agent and this part of the environment, and provide access to the appropriate routines for interaction (motor schemas) in each case.

A perceptual schema only pays attention to the sensory or other information that is directly relevant to its domain of interaction. Thus, for a predatory agent, the 'prey' perceptual schema would only pay attention to how well the current information supports the concept that prey is present. It would identify the sources of sensory information that indicate where the agent should attack and some important parameters of the attack, e.g., distance, height and so forth. It would give access to the motor schemas for prey pursuit and prey acquisition.

The 'predator' perceptual schema (which might be currently active with the 'prey' schema) pays attention to a different set of sensory stimuli. It attempts to interpret how well the current sensory data supports the concept that a predator is present. It reports on key facets of the sensory information such as escape routes, and makes available motor schemas for running away, hiding and so forth.

Different perceptual schema may give completely different 'segmentations' and interpretations of sensory data. Something classified as a distinct object by one perceptual schema may not even be resolved out of the mass of sensory information by another. Thus, the concept of 'object' in schema theory is task dependent.

Schemas can be abstracted away from their direct sensory and motor origins. An abstract perceptual schema might look for evidence amongst schema assemblages rather than sensory information. An abstract motor schema might be a prototypical set of 'mental' rather than physical actions.

2.3 Schema Assemblages and Interaction

The behavior of the agent is thus produced as the result of the interactions between a network of perceptual and motor schemas. Such a network is called an *assemblage* of schemas. The principle interaction mechanism provided by Schema Theory is the *activity level* associated with each schema instance. Cooperating schemas increase each others activity level; competing schemas decrease each others level. For example, 'prey' and 'predator' schema instances might be competitive when they attempt to interpret the same piece of sensory information.

The activity level of a perceptual schema represents a 'confidence level' that the domain of interaction represented by that schema actually exists in the environment. The activity level of a motor schema indicates its 'degree of readiness' to influence the agent's behavior.

The VISIONS project [11] is a natural-scene interpretation system based on schema theory. It emphasizes the perceptual side of the theory. The long-term memory component of VISIONS is a set of perceptual schemas for natural scene interpretation. The short-term memory component is an assemblage of schema instances that have been generated to interpret current visual information. Schema instances in this assemblage represent hypotheses that particular objects are present in the scene. A typical visual input for the VISIONS system is an outdoor scene of a house surrounded by various natural object such as trees, fenceposts, people and so on. The 'roof' schema locates any part of the visual image that indicates the potential presence of a house roof, and will attach itself to that part of the image (and there may be more than one instance of the schema attached to different parts of the scene). If a 'wall' schema attaches itself onto the adjacent, lower part of the image, then these two schemas will cooperate – enhance each others activity level. On the other hand, if a 'sky' schema attaches itself onto that spot, they will compete. In this fashion, mutually supporting interpretations of the scene reinforce each other. No centralized executive or scheduler exists; the interpretation of the scene emerges from the interaction of the set of perceptual schemas.

2.4 Learning

It is fundamental to schema theory that the set of perceptual and motor schemas that control an agent's behavior does not necessarily constitute a logically correct or complete system for achieving goals. Schemas can be fallible. New schemas may be formed as assemblages of existing schemas and 'tuned' over time so that they in turn become 'primitive' or atomic schemas. A perceptual schema that is activated by the sensory signals from seeing a car might, over time, spawn perceptual schemas that trigger on specific makes of car, or indeed on "my" car even though the make of "my" car will also change over time.

2.5 Applications and Implementations

Schema theory is not a concise mathematical theory, but rather a set of guide-lines that can be employed by researchers engaged in modeling animal behavior and those involved in building artificial agents. A schema theory model of depth-sensing and 'path planning' behavior in the Frog is described in [6]. A high-level model for primate reaching and grasping is proposed in [7]. Arkin [8] describes an approach to controlling mobile robots based on schema theory. Lyons [17, 16] proposes a schema-based language for generating and analyzing robot behavior. VISIONS [11], a computer vision application of schema theory, has already been discussed. An approach to natural language based on schema theory is presented in [5].

In the case of schema models of neural systems, the underlying implementation is one or more neural networks. In the case of the robot models, the underlying implementation is some computer programs and robot hardware. Nonetheless, because they share the general tenets of schema theory, there can be mutually profitable 'give and take' between, for example, the neuromodeling and robotics applications [9].

3 The Robot Schemas ($\mathcal{RS}$) Model

I have used schema theory to address the problem of building industrial robotic systems. The key constraints from this domain are, first, that it must be possible to take schema theory concepts down to the level of implementation on robot hardware, and second, that it be possible to give formal guarantees about the kind of behaviors exhibited by the robot system. The latter is very important in a robot system that must operate twenty-four hours a day, seven days a week.

In [17] we used schema theoretic concepts and the computational constraints of the robot domain to build a computational model for sensory-based robot programming — *Robot Schemas* ($\mathcal{RS}$). This work mapped schemas and schema instances to process descriptions and processes with a mathematical semantics based on port automata [22]. Subsequent work [15, 16] enriched this model by developing a process-algebra language to describe schema assemblages and patterns of actions in schemas.

3.1 Processes in $\mathcal{RS}$

A description of a process, or network of processes, is called a *schema*. For example, $\texttt{Joint}(v)\langle x\rangle$ denotes a process that is an instance of the schema `Joint` with one ingoing parameter v and one outcoming result x. A process can terminate either successfully (stop) or unsuccessfully (aborts). Networks are built by composing processes together using several kinds of *process composition operators*. These allow processes to be ordered in various ways, including concurrent, conditional and iterative orderings. At the bottom of this hierarchy, every network must be composed from a set of atomic, pre-defined processes. The set of *basic schemas* defines what processes are atomic.

$\mathcal{RS}$ notation has a *process algebra* [13] style: processes are specified as algebraic compositions of other processes. The relationship between this and state-machine models of computation is well understood. A process is essentially a state-machine; for $\mathcal{RS}$ a process is a special kind of automaton called a *port automaton* [22]. A process composition operation is an instruction on how to combine automata: more specifically, how to combine state-transition graphs. Thus, $\mathcal{RS}$ provides a way to build processes without ever explicitly mentioning states. This allows us to avoid some of the problems of state-based approaches to representing and reasoning about actions [21].

3.2 Composing Processes

We will make use of four atomic composition operations, sequential, concurrent, conditional and disabling, and two non-atomic composition operations, synchronous recurrent and asynchronous recurrent.

Sequential composition is used to enforce a strict ordering on operations. For example, ensuring that $part2$ is always placed after $part1$, $\mathtt{T} = \mathtt{Place}(part1);\mathtt{Place}(part2)$.

Concurrent composition indicates that two or more processes should be carried out concurrently. This allows us to represent a lack of ordering between activities, e.g., $\mathtt{T} = \mathtt{Place}(part1) \mid \mathtt{Place}(part2)$, or actions which need to be done simultaneously, e.g., squeezing an object obj with two fingers $f1$ and $f2$, $\mathtt{T} = \mathtt{ApplyForce}(f1, obj) \mid \mathtt{ApplyForce}(f2, obj)$. The process `T` terminates once all its arguments have terminated.

Concurrent processes can communicate with each other via *communication ports.* Two special basic processes are defined for this purpose: $\mathtt{IN}(p)\langle v\rangle$ reads a value from port p and can pass it on through its result v, $\mathtt{OUT}(p, v)$ writes the value v out on port p. All port communication is synchronous and there can be fan-in and fan-out on ports. A connection between ports on two or more processes is indicated by specifying the same *port variable* name in the parameters to each processes[2]. For example, if $\mathtt{X}(p) = \mathtt{IN}(p)$ and $\mathtt{Y}(q) = \mathtt{OUT}(q, n)$ then $\mathtt{N} = \mathtt{X}(a) \mid \mathtt{Y}(a)$ is a concurrent network of two processes, where the output port on one process (`Y`) is connected to the input port on the other (`X`).

Conditional composition allows the construction of networks whose behavior is conditional. The network of `T = P:Q` behaves like `P;Q` iff `P` terminates successfully. If `P` aborts, then `Q` is not carried out, and `T` aborts. For example, in $\mathtt{LidOpen}(box) : \mathtt{Place}(x, box)$ whether `Place` is carried out or not depends on whether `LidOpen` terminates successfully or not. Conditional composition can be used to pass results between processes in a 'pipelined' fashion. If $\mathtt{X}(p) = \mathtt{IN}(p)\langle v\rangle : \mathtt{A}(v)$ the value read in on the port p is used as a parameter to the process `A`.

Disabling composition allows one process to terminate another. The network `T = A#B` behaves like `A|B` except that it terminates whenever *any* of its arguments terminates.

Finally, two useful non-atomic composition operations are defined in terms of these four: *synchronous recurrent* $\mathtt{A} :; \mathtt{B} = \mathtt{A} : (\mathtt{B}; (\mathtt{A} :; \mathtt{B}))$, and *asynchronous recurrent* $\mathtt{A} :: \mathtt{B} = \mathtt{A} : (\mathtt{B} \mid (\mathtt{A} :: \mathtt{B}))$. Synchronous recurrent composition is similar to *while-loop* iteration. Asynchronous recurrent composition does not iterate, but rather 'spawns' off a set of concurrent processes every time its 'condition' is satisfied. These special forms of guarded recursion are a useful shorthand for specifying iteration (and can be implemented efficiently).

3.3 Building Agents and Analyzing their Behavior

If what we wish to do is construct robot agents, then $\mathcal{RS}$ gives us a programming language for this that retains the distributed, cooperation and competition style of schema theory. Examples of the use of $\mathcal{RS}$ for robot programming are found in [17, 15]. However, if we wish to use some kind of formal method to design or analyze the behavior of robot agents, then we need to go a little further.

[2]The concept of a *port variable* a variable referring to a port, is being used here in place of the port connection relation used in previous $\mathcal{RS}$ work, e.g., [17].

If we build robot programs that react to conditions in the environment, then to explore the behavior of these programs we have to consider not *just* the robot program, but rather the combination of robot program *and* the environment in which it will operate. This is very necessary if we are to discover/exploit so-called *emergent* properties.

We take the approach of modeling *both* the robot controller *and* the environment model using $\mathcal{RS}$. Employing the same 'language' for both simplifies the exploration of their interaction. The use of $\mathcal{RS}$ for specifying both reactive controller and environment model is discussed in more detail in [16].

Rather than go into detail now on the use of $\mathcal{RS}$ to analyze behavior, I will instead present a motivating example of why such work is necessary, and then present the detail on how such analysis can be accomplished.

4 Exploiting Patterns of Activity

Our objective is to construct agents that *reactively exploit* their environment to achieve their objectives. That is, in building a reactive agent, we want to look for useful inherent patterns that an agent can capitalize on. Chapman [10] calls such a pattern *a dynamic,* and we adopt that usage here. Dynamics are structural or temporal regularities of the environment. They exist independent of the agent and are essentially acausal.

4.1 Dynamics in the Kitting Domain

Our principal application is an industrial robot system called a *kitting workcell.* A kitting robot is a robot system that puts together assembly kits — trays containing all the necessary parts from which to build a specific product. Simpler and cheaper automation can construct the assemblies once they have been placed in the kits and routed appropriately. The key characteristic of the kitting robot is not so much its ability to reason about assembly, but rather its ability to choose timely and effective actions to suit an uncertain and changing environment.

A simple example of a dynamic in the kitting robot domain is of a conveyor belt carrying parts past the robot. A part on the belt will start by being drawn inexorably towards the robot; after a certain time, it will be within the robots workspace; and, if it remains on the belt, a certain time after that it will be irrecoverably gone.

Some dynamics admit participation by the agent. That is, the actions available to the agent can be affected by and/or can affect the course of the dynamic. This is a much weaker concept than the notion of an event being within or partially within an agent's control [21]. For example, the motion of the belt may indeed be beyond the control of an agent. Nonetheless, an agent can participate in this dynamic to acquire an object. To participate in a dynamic, an agent must be capable of knowing about or sensing components of the dynamic and of producing actions that affect or are affected by the dynamic.

For example to exploit the belt dynamic to acquire an object, the kitting robot needs to be able to tell when the part is in its workspace and it needs to be able to take the part from the belt. We say an agent *participates in a dynamic to achieve an objective* when a reaction of the agent is triggered by some portion of the dynamic, and the effects caused by that reaction in addition to the effects of the dynamic, cause the agent's objective to be achieved. In terms of the belt example, the agent can exploit the motion of the belt to carry the object to within the agent's reach (the dynamic's contribution to the objective) and once the object is within reach (the sensory trigger) the agent reaches out and picks up the object (the reaction's contribution to the objective). Note that this allows us to utilize actions whose effects are context dependent, since the dynamic resolves the context.

4.2 Procedural Models

Horswill's habitat constraints allow a designer to simplify the design of an agent so as to exploit its 'niche' in the environment. Our approach is an attempt to automate the identification and use of a

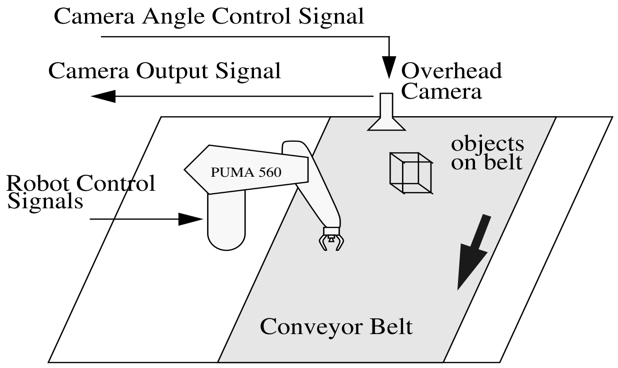

Figure 1: The Tracking Environment.

class of such constraints, namely constraints that express how an agent can exploit processes at work in the environment. We address the problem as follows: We will use $\mathcal{RS}$ to build a *procedural model* of the environment. This environment model needs to capture the interaction topology between parts of the environment as well as between the environment and the agent. We will then examine this model formally to identify candidate useful modes of interaction given some goal.

This approach has the advantage that the designer's focus is now on building an accurate environment model rather than on devising the agent's view of the environment as seen through a set of action descriptions, i.e., interpreting the environment for the agent. It also has the advantage that the devised actions do not need to be consistent across different tasks, since they are extracted from the environment model on a per task basis, or even in the same task over time, since the processes at work in the environment may change over time.

We will represent the environment model as a set of concurrent, communicating processes. The key information that this procedural model captures is the interaction topology: what parts of the environment 'are connected to' other parts and how 'messages' are passed along those communication links. This kind of world model does *not* contain information about the current state of the world such as e.g., (at-position object26 0.25 0.34 0.29); rather it contains information about how objects interact with each other, or how an object interacts with the agent's visual input sensors. In other words, the model describes the structural and temporal regularities, the patterns of interaction in the environment and how they evolve over time.

Modeling a dynamic environment as a set of processes is analogous to the approach taken in Qualitative Process Theory [12]. The sole mechanism assumption of QP-theory states that all changes in (physical) systems are caused directly or indirectly by processes. Our modeling of the environment also subscribes to this assumption. This modeling only restricts the amount of interaction. It does not necessarily prescribe it, since the processes may contain uncertainty, e.g., random motion of objects, or the unpredictable occurrence of events such as an unknown object at an unexpected location.

4.3 Example: The Tracking Reaction

To explain the concept of exploiting dynamics, consider the simple tracking environment shown in figure 1. This environment contains a number of objects. The objects may be moving (e.g., on the conveyor belt) and that motion is beyond the control of the agent. The environment also contains a camera, the

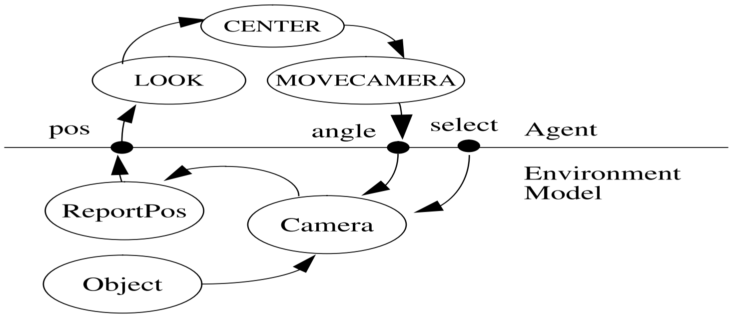

Figure 2: Modeling the Tracking Environment

orientation of which can be controlled. If an object is in view, the camera will report the position of that object. We will assume that the agent has an object of type T in view initially, and that its objective is to know the position of this (possibly moving) object at all times.

The $\mathcal{RS}$ model of this tracking environment `TEM` is shown in figure 2. (The formal representation for this model will be described in a later section.) The central line denotes the sensory-motor interface of the agent. Above the line are the agents reactions. Below the line is the model of the environment. Reactions and environment model interact only through the sensory-motor interface. This does not imply that the agent needs to know about the model in this figure — only that the designer does.

The tracking environment model consists of a process representing the agent's visual sensor `Camera,` and some number of object processes, `Object.` The `Camera` process accepts communication on a port, *angle*, which selects the pose of the camera and *select*, which tells the camera the type of object of which we're interested in getting the position. If an object of the selected type is in view at the angle the camera is currently set to, then the position of that object is returned on the port *pos*. `Camera` determines what objects are in view as follows: It has port connections to all existing `Object` processes. These processes represent objects in the environment. Each `Object` process continually reports its position and type to `Camera,` which applies a function `InView` based on the *select* and *angle* values to determine if a selected object is in view. The position of any object in view is echoed back on *pos*.

The agent can acquire position information via its sensory interface port *pos*. To find out what kind of information this port contains, we have to trace back the connections of this port through the processes in the environment, and possibly back to the motor interface ports (in this case *angle* and *select*). As we trace back, we can annotate the connections with the conditions under which they actually transmit information (The formalization of this analysis in our process model will be explained presently). For now, say that the connection from the `Camera` to the *pos* port is only transmitting whenever an object of type T is in the field of view of the camera, as determined by the condition process $\texttt{InView}(angle, posn)$.

Thus, in order to keep receiving information on port *pos* the agent has to keep this condition true. In other words, we have extracted a condition that should be the loop invariant[3] for the tracking reaction. An example of a tracking reaction that maintains this invariant is shown above the agent/environment dividing line in figure 2 and works as follows: The agent continually reads in the position of an object on port *pos*, transforms this position to a camera angle at which the object is centered in the camera and transmits that angle to the motor interface port *angle*.

In the subsequent sections of this paper, we will adopt this tracking reflex as our working example. This is principally for purposes of illustration - we wouldn't expect a designer to need this kind of

[3] A term from the field of program analysis that means a condition that remains invariant across iterations of a program loop.

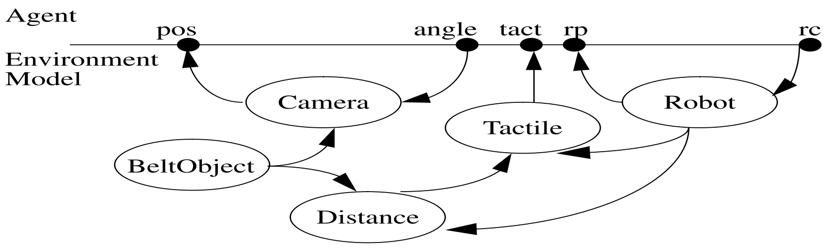

Figure 3: The Belt Environment BEM

analysis to suggest tracking, although it does make explicit some of the constraints on the reflex such as actuator and sensor timing and potential object motions. On the other hand, for even a slightly more complicated model of the kitting robot environment, the analysis can identify some highly novel candidates for building reflexes [19].

Figure 3 shows a environment model in which both tactile information and knowledge about potential belt motions are available. Identifying the dynamics in this model yields such interesting candidate behaviors as acquiring objects by lying in wait for them and ambushing them as they pass by.

In the next section, we lay the groundwork for automating this kind of design procedure.

5 Process Evolution

To analyze how process networks (world models and/or robot controllers) execute over time, it is important to be able to derive how networks evolve as component processes dynamically terminate. To this end, we introduce the *evolves* operator. We say that process P evolves into process Q under condition Ω if P possibly becomes equal to Q when condition Ω occurs; we write this as $\mathtt{P} \xrightarrow{\Omega} \mathtt{Q}$. If there is a set of processes that P can possibly become equal to on condition c, then P necessarily becomes equal to exactly one of these when c occurs.

To define the evolves operator, we need two pieces of information: First, we need to know for each basic process under what conditions it terminates. Second, we need to know for each composition operator, how the composition is affected by the termination conditions of each argument process.

The conditions under which the basic process P terminates successfully will be written $\Omega\mathtt{P}$. In some cases, we will also need to use the conditions under which P terminates unsuccessfully, i.e., *aborts* itself; these necessary (but not sufficient[4]) conditions we write as $\mho P$.

We start by defining *single-step* evolution; this captures the concept of the very next single process creation/termination change that a network can undergo. Consider a sequential composition of processes T = P1;P2 , where P1 and P2 are basic. When T executes, it begins with P1, which can either never terminate, can abort, or can terminate successfully. We introduce a mapping $ev()$ that takes a network as an argument, and maps it to couple consisting of a new network and a condition. This mapping denotes that when the argument network is executed, then under the conditions given in the first element of the image couple, the execution will produce the network given in the second image of the couple. For example

$$ev(\mathtt{P1;P2}) \;=\; (\Omega\mathtt{P1} \vee \mho\mathtt{P1},\ \mathtt{P2})$$

I.e., P1;P2 always evolves into P2 whether P1 terminates successfully (condition Ωp1) or aborts

[4]A process can also be forced to abort in disabling composition '#'.

(condition $\mho$`p1`). In a similar fashion, we can define the way in which evolves interacts with each one of the composition operators using the $ev()$ mapping[16].

A specific network `P` might be capable of more than one possible next evolution. We define the set of next possible evolutions as

$$poss(\mathtt{P}) = \{(c, \mathtt{Q}) \text{ such that } ev(\mathtt{P}) = (c, \mathtt{Q})\} \tag{1}$$

For example, the network `Cond1 : A` evolves to `A` if Ω`Cond1` but aborts if $\mho$`Cond1`.

In general we will be interested in the ultimate effect of a chain of successive one-step evolutions — an "execution path" through the network. Thus, we define the evolves operator as the transitive closure of a chain of one-step evolutions:

$$ev(\mathtt{P}) = (c1, \mathtt{P1}) \ \& \ ev(\mathtt{P1}) = (c2, \mathtt{P2}) \ \& \ \ldots \ \& \ ev(\mathtt{Pn}) = (cn, \mathtt{Q}) \iff \mathtt{P} \xrightarrow{c} \mathtt{Q} \qquad with\ c = c1.c2.c3\ldots.cn. \tag{2}$$

We write temporal ordering of conditions using a period, e.g., $A.B$ is read A then B.

The set of all possible evolutions is the transitive closure of $poss(\mathtt{P})$ over the one-step evolves operator, $ev()$ (analogous to the concept of 'reachability' and the 'reachable set' in linear systems theory). We can define this recursively in terms of one-step evolution as 'every network that can be reached in one evolution plus everything that can be reached from there',

$$poss^*(\mathtt{P}) = poss(\mathtt{P}) \cup (\bigcup_{(c,\mathtt{Q}) \in poss(\mathtt{P})} \{(c.\hat{c}, \hat{\mathtt{Q}}) \text{ such that } (\hat{c}, \hat{\mathtt{Q}}) \in poss^*(\mathtt{Q})\}) \tag{3}$$

or, equivalently, in terms of the transitive evolves operator as:

$$poss^*(\mathtt{P}) = \{(c, \mathtt{Q}) \text{ such that } \mathtt{P} \xrightarrow{c} \mathtt{Q}\} \tag{4}$$

5.1 Process Limits

To pull out dynamics from an environment model, we need to understand how that environment behaves in the following sense. We need to understand how sensory input reflects change in the model, and ultimately how that change is related to motor output (if it is). That is, we need to construct something equivalent to the usual transfer function of linear systems theory: the result of transitively closing all the internal evolution steps between the motor and sensory interfaces. For a set of reactions, this gives us a sensory input to motor output mapping; for an environment model, it gives us a motor output to sensory input mapping. The concept of process limits and bound processes is introduced below as an approach to pulling out dynamics from an environment model expressed in $\mathcal{RS}$.

A *process limit* is a process that has evolved from the initial process, but cannot evolve any further. The significance of the limit is that it prevents any further change of behavior by the process. Any process may have zero or more limits; we call the set of limits of a process `P`, $limitset(\mathtt{P})$, and it is a subset of $poss^*(\mathtt{P})$:

$$\begin{array}{lll} (c, \mathtt{Q}) \in limitset(\mathtt{P}) & & \\ & i) & \mathtt{P} \xrightarrow{c} \mathtt{Q},\ and \\ & ii) & there\ exists\ no\ (cr, \mathtt{R}) \quad s.t.\ \mathtt{Q} \xrightarrow{cr} \mathtt{R} \end{array} \tag{5}$$

The evolves operator provides us a way to determine process limits. In the case that a process does have limits, then we can use evolves to 'simulate' the execution of the network until we reach a limit. To get *all* the limits of a network we need to produce $poss^*$ for the network. Therein lies a problem: If the process is in an endless loop, however, then the 'simulated' execution will also be in a loop! This can be addressed by stopping the production of $poss^*$ whenever a process recursion is detected. In the remainder of the paper, $limitset(\mathtt{P})$ will refer to this augmented concept: all the process limits *plus* any recursive process invocation by `P`.

A process, such as `Q` in the above definition, that does not evolve to anything, we call a *bound*, since it may bound the evolution of any process it occurs in. Any basic process that never terminates is a bound. The basic processes `STOP` and `ABORT` are also bounds. Strictly speaking, whether or not a process is a bound is just a question of its definition. However, we will also find it convenient to also designate specific processes as bounds for the purpose and duration of some analysis. Such processes will be referred to as pseudo-bounds.

Let `EM` be an $\mathcal{RS}$ model of the environment, i.e., a process network. First, we treat any sensory processes in `EM` as pseudo-bounds. That is, whenever one of those processes is produced in the evolution of `EM`, then the evolution will go no further, and this bound and the conditions under which it was evolved, will become part of the limitset. This will tell us the ultimate sensory process(es) produced due to any motor input or any environment changed. This will also tell us about internal cycles in the environment, e.g., the motion of an autonomous object, since recursive process invocations are now also contained in the limitset.

6 Example

Recall that the purpose of building the environment model is to allow us, as designers, to explore the model for patterns of activity, dynamics, that we can use in the construction of a robot controller. We wish to build a controller that exploits these dynamics to achieve its objectives.

The $\mathcal{RS}$ network to implement the environment for the tracking example discussed previously is described below. We will explore this model to identify its dynamics. We will then use these dynamics to posit candidate actions for building a robot controller with a tracking reflex.

Tracking Environment Model. The model is comprised of a set of mutually recursive process equations. The topmost process is the `TEM` (for tracking environment model) network. Its parameters are the initial position of the object to be tracked p, and the camera angle a. For simplicity, the *select* port has been omitted from the camera model; this allows us to focus more directly on the reactive portion of the problem. We will assume for now that the camera only recognizes objects of the desired type.

$$
\begin{aligned}
\texttt{TEM}(p,a) &= \texttt{Object}(p) \mid \texttt{Camera}(a) \\
\texttt{Object}(p) &= (\texttt{STREAM}(op,p)\#\texttt{Delay}(t0))\ ;\ \texttt{Ran}(-\epsilon,\epsilon)\langle\gamma\rangle\ \ :\ \texttt{Object}(p+\gamma) \\
\texttt{Camera}(a) &= (\texttt{Monitor}(a)\#\texttt{IN}(angle)\langle na\rangle)\ :\ \texttt{Camera}(na) \\
\texttt{Monitor}(ca) &= (\texttt{Delay}(ts0);\texttt{IN}(op)\langle n\rangle;\texttt{Delay}(ts1))\ ::\ (\texttt{InView}(n,ca):\texttt{ReportPos}(n)) \\
\texttt{ReportPos}(q) &= \texttt{STREAM}(posn,q)\#\texttt{Delay}(ts0) \\
\texttt{STREAM}(y,v) &= \texttt{OUT}(y,v)\ ::;\texttt{STOP}
\end{aligned}
\tag{6}
$$

The process $\texttt{Ran}(l,h)$ produces an arbitrary result value between l and h. `Ran` is being used here to model the uncertainty in object motion. Thus, $\texttt{Object}(p)$ continually repeats the following behavior: it offers its current position p for a time $t0$ on the port op, and then 'moves' (updates its position) some arbitrary amount between $+\epsilon$ and $-\epsilon$ relative to p. The process $\texttt{STREAM}(y,v)$ offers to transmit the value v on the port y infinitely often. $\texttt{Camera}(a)$ is continually checking (via `Monitor`) for any object in view at the current position, while concurrently being ready to accept a new position and repeat the behavior for that position. `Monitor` polls the position data from `Object` processes and spawns off a subnetwork to determine if the object is in view and passes on the position data if it is. The process `InView` carries out the calculations to determine if the object is in view. If it is, then the `ReportPos` process continually offers the position of the object on port *pos*. If the object is not in view, the subnetwork just aborts. The values of $t0, ts0$ and $ts1$ are constants and they model timing characteristics of the environment.

Sensor and Actuator Models. The `TEM` model describes the environment itself. However, the agent interacts with this environment through its sensors and actuators. These sensors and actuators may impose limitations on this interaction, so we need to model them also. We will assume the sensory-motor interface is defined by the two processes `sense1` and `motor1` defined as follows:

$$\begin{aligned} \texttt{sense1} &= (\texttt{IN}(pos)\langle v\rangle ; \texttt{Delay}(gl)) :; (\texttt{STREAM}(s1, v)\#\texttt{Delay}(tl)) \\ \texttt{motor1} &= (\texttt{IN}(m1)\langle v\rangle ; \texttt{Delay}(gm)) :; (\texttt{STREAM}(angle, v)\#\texttt{Delay}(tm)) \end{aligned} \tag{7}$$

These primitive 'actions' affect or are affected by parts of the environment model, i.e., `sense1` is affected by *pos* and `motor1` affects *angle*. (Both tm and gm are simply timing constants, specific to any actual sensors and actuators being modeled.) There is an important issue of style here that needs to be emphasized: the agent does not invoke whatever effect `motor1` has on the environment by creating an instance of `motor1` . The action is invoked by writing a message on `motor1` 's input port $m1$. A single instance of each of `sense1` and `motor1` is always present and these constitute the sensory-motor interface. Each is in a loop, continually reading a value and passing that value to/from the environment.

The timing delays in these actions, gl, tl, gm, and tm, model the timing delays in real sensor and effector hardware. These actions have been given neutral names, rather than names such as 'movecamera' and so on, to reinforce the fact that they do not constitute actions in the classical planning sense of the word. Rather they are simply signals to and from the environment the meaning of which can only be derived by looking at the environment model.

Extracting Dynamics. The next step is to proceed with extracting dynamics from the environment model. As discussed in section 5, we do this by looking at the limitset of the environment model network assuming that the sensory input processes (and any recursive calls) are pseudo-bounds. This gives us the following set:

$$\begin{aligned} limitset(\texttt{TEM}(p, a)) = \{ \quad & (time(t0).\gamma \in [-\epsilon, \epsilon] &&, \ \texttt{TEM}(p+\gamma, a)), \\ & (\Omega\texttt{IN}(angle)\langle na\rangle &&, \ \texttt{TEM}(p, na)), \\ & (inview(p) &&, \ \texttt{TEM}(p, a) \mid \texttt{ReportPos}(p)) \\ \} \end{aligned} \tag{8}$$

This set lists the dynamics that a designer can exploit in the tracking environment. The first element of the limitset tells us that left on its own (no conditions) the object process may move. The second element tells us that changing the value on the *angle* port changes the camera position. Let the termination condition of the process `InView` be written as $inview(p)$. The final element tells us that if the function $inview(p)$ holds then the port *pos* holds the value of the object position.

To illustrate *how* the limitset is derived, let us look at the first element. The process `TEM` has two parallel components `Object` and `Camera.` Thus, the limitset of `TEM` will contain limits contributed to by both. Let us confine our attention to `Object` – this process is defined by the second equation in (6). This process evolves as follows:

$$\begin{aligned} \texttt{Object}(p) &= (\texttt{STREAM}(op, p)\#\texttt{Delay}(t0)) \ ; \ \texttt{Ran}(-\epsilon, \epsilon)\langle\gamma\rangle \ : \ \texttt{Object}(p+\gamma) \\ &\xrightarrow{time(t0)} \texttt{Ran}(-\epsilon, \epsilon)\langle\gamma\rangle \ : \ \texttt{Object}(p+\gamma) \\ &\xrightarrow{\gamma\in[-\epsilon,\epsilon]} \texttt{Object}(p+\gamma) \end{aligned}$$

The final process produced is a recursive instance of `Object` (and the evolution is pruned here, as per our discussion at the end of the previous section). When this procedure is repeated for the remaining process in `TEM` , i.e., `Camera,` the other two limitset elements emerge.

The purpose of extracting dynamics from the environment is to present a candidate set of actions from which to build a robot controller that 'fits its niche' in the environment. That is, a robot controller that exploits the dynamics of the environment to achieve its objectives.

If we assume that tm and gm are small, then we can posit a candidate action $\texttt{MoveCamera}(m1)$. This action writes a value onto the port *angle* which we can now see from (8) causes the camera to move to this position. Similarly, if we assume tl and gl are small, then we can postulate an action $\texttt{Look}(s1)$.

This action will produce the position of a viewed object on $s1$ if that object is in view. A designer might then choose to exploit these actions by building a tracking reaction consisting of the three concurrent processes:

$$\texttt{Track} \quad = \quad (\texttt{Look}(p)\#\texttt{Center}(p,q)\#\texttt{MoveCamera}(q)) \tag{9}$$

where $\texttt{Center}(a,b)$ produces a camera angle on port b at which position a is centered in the camera view or else aborts (and hence, by # composition, aborts `Track`) if none such is possible.

7 Conclusion

This paper has presented an overview of Arbib's Schema Theory and briefly discussed some applications of that theory. One application in particular was discussed in detail – the $\mathcal{RS}$ model.

$\mathcal{RS}$ is a computational view of some aspects of Schema Theory as applied to robotics. The paper has presented two views of $\mathcal{RS}$. The first is as a schema-theory style programming language with which to built robot controllers. The second is as a formal system for modeling both robot controller and the environment in which that controller is embedded and exploring their interaction. One example of the combination of these views is our recent work on integrating deliberation and reaction by casting deliberation as the ongoing, incremental adaptation of a reactive system [18].

The use of $\mathcal{RS}$ as a formal design tool was emphasized in this paper. This was presented via an example of designing robot controllers that exploit existing patterns of activity in the environment to achieve their objectives Specifically, the example of a simple tracking reflex for an industrial kitting robot workcell was used. An $\mathcal{RS}$ model of the environment was constructed and this model was explored for useful patterns of activity – dynamics – that could be exploited in designing the tracking reflex. The *limitset* concept was used to identify the dynamics of the environment model. In [20] we investigate the automation of the entire process of designing reactions, from evaluating the limitset to actually building the reaction, and for more challenging environment models.

This concept of building robot controllers that exploit existing patterns of activity in the environment to achieve their objectives has a number of advantages. Firstly, exploiting dynamics can lead to a much simpler and more robust controller. Secondly, exploiting a dynamic is a much weaker concept that the classical notion of whether something is within an agent's control, thus it demands less from both robot and environment. Finally, the approach of expressing knowledge about the environment as a procedural model or 'simulation' demands less of the designer than the classical approach of devising the agent's view of the environment as seen through a set of action descriptions.

References

[1] M.A. Arbib. Perceptual structures and distributed motor control. In V.B. Brooks, editor, *Handbook of Physiology – The Nervous System II. Motor Control*, pages 1449–1480. Amer Physiological Society, Bethesda, MD, 1981.

[2] M.A. Arbib. Schema theory. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*. Wiley-Interscience, 1992. Also USC Center for Neural Engineering TR-91-03.

[3] M.A. Arbib. Book Review: Allen Newell's Unified Theory of Cognition. *Artificial Intelligence*, 59:265–283, 1993.

[4] M.A. Arbib and J. Buhmann. Neural networks. In S.C. Shapiro, editor, *Encyclopedia of Artificial Intelligence (2nd Edition)*. Wiley-Interscience, 1992. Also USC Center for Neural Engineering TR-91-02.

[5] M.A. Arbib, E.J. Conklin, and J.C. Hill. *From Schema Theory to Language*. Oxford University Press, Oxford, UK, 1987.

[6] M.A. Arbib and D.H. House. Depth and detours: An essay on visually-guided behavior. In M.A. Arbib and A.R. Hanson, editors, *Vision, Brain and Cooperative Computation*, pages 129–163. MIT Press, Cambridge MA, 1987.

[7] M.A. Arbib, T. Iberall, and D. Lyons. Schemas that integrate vision and touch for hand control. In M.A. Arbib and A.R. Hanson, editors, *Vision, Brain and Cooperative Computation*, pages 489–510. MIT Press, Cambridge MA, 1987.

[8] R. C. Arkin. Motor schema-based mobile robot navigation. *Int. J. Rob. Res.*, 8(4):92–112, Aug 1989.

[9] Center for Neural Engineering. *Workshop on Neural Architectures and Distributed AI: From Schema Assemblages to Neural Networks*, Univ. of South. Calif. Los Angeles CA 90089-2520, October 19-20th 1993.

[10] David Chapman. *Vision, Instruction, and Action.* MIT Press, Cambridge, 1992.

[11] R. Draper, B. Collins, J. Brolio, A. Hanson, and E. Riseman. The Schema System. *Int. J. Comp. Vis.*, 2:209–250, 1989.

[12] K.D. Forbus. Qualitative process theory. *Artificial Intelligence*, (24):85–168, 1984.

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

[14] I. Horswill. Analysis of adaptation and environment. *Artificial Intelligence, Special Issue on Comp. Theories of Interaction*, TBA(TBA), 1993.

[15] D.M. Lyons. A formal model for reactive robot plans. In *2nd International Conference on Computer Integrated Manufacturing*, RPI, Troy, NY, May 21–23 1990.

[16] D.M. Lyons. Representing and analysing action plans as networks of concurrent processes. *IEEE Trans. Rob. & Aut.*, 9(3), June 1993.

[17] D.M. Lyons and M.A. Arbib. A Formal Model of Computation for Sensory-Based Robotics. *IEEE Trans. Rob. & Aut.*, 5(3):280–293, June 1989.

[18] D.M. Lyons and A.J. Hendriks. Planning for reactive robot behavior. In *IEEE Int. Conf. Rob. & Aut.*, Apr. 7–12th 1992. (See also video proceedings).

[19] D.M. Lyons and A.J. Hendriks. Exploiting patterns of interaction to achieve reactive behavior. *Artificial Intelligence*, Submitted to: Special Issue on Computational Models of Interaction and Agency (Eds. P.Agre, S.Rosenschien), 1993. Also Philips Technical Note TN-93-052.

[20] D.M. Lyons and A.J. Hendriks. Exploiting patterns of interaction to achieve reactive behavior. Technical Note PLB-TN-93-052 (Public), Philips Laboratories, Briarcliff Manor, NY 10510, June 1993. Submitted to: Special Issue of *Artificial Intelligence* on Computational Models of Interaction and Agency (Eds. P.Agre, S.Rosenschien).

[21] R. Pelavin. *A Formal Approach to Planning With Concurrent Actions and External Events.* PhD thesis, Univ. of Rochester, Dept. of Computer SC., Rochester, NY, 1988.

[22] M. Steenstrup, M.A. Arbib, and E.G. Manes. Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences*, 27(1):29–50, Aug. 1983.