A Situated View of Representation and Control

Stanley J. Rosenschein*
Leslie Pack Kaelbling†

October 31, 1994

For correspondence concerning this paper, please contact

Leslie Pack Kaelbling
Computer Science Department
Box 1910
Brown University
Providence, RI 02912-1910

Abstract

Intelligent agents are systems that have a complex, ongoing interaction with an environment that is dynamic and imperfectly predictable. Agents are typically difficult to program because the correctness of a program depends on the details of how the agent is situated in its environment. In this paper, we present a methodology for the design of situated agents that is based on *situated automata theory*. This approach allows designers to describe the informational content of an agent's computational states in a semantically rigorous way without requiring a commitment to conventional run-time symbolic processing. We start by outlining this situated view of representation, then show how it contributes to design methodologies for building systems that track perceptual conditions and take purposeful actions in their environments.

1 Introduction

Humans, delivery robots, and automated factories are all systems that have an intelligent, ongoing interaction with environments that are dynamic and imperfectly predictable. Such systems are often called *situated agents*. They constitute an important class of systems that are very difficult to program because of their close interaction with the environment in which they are situated. Specifications of correctness for situated agents amount to specifications of their interactions with the environment: what action should the agent take when the

*Work on this paper was supported in part by the National Aeronautics and Space Administration under contract NAS2-13326 and by the Defense Advanced Research Projects Agency under NASA contract NAS2-13229 and under TEC contract DATA76-93-C-0017.

†Work on this paper was supported in part by National Science Foundation National Young Investigator Award IRI-9257592 and in part by ONR Contract N00014-91-4052, ARPA Order 8225.

environment is in a particular configuration? Programs for situated agents must allow them to respond appropriately to diverse, rapidly changing situations.

The emphasis on an agent's connection to its environment is an important change from that of traditional theories of representation and control. In this paper, we present an informal overview of a particular methodology for the design of situated agents. This methodology, based on *situated automata theory* [14, 16], allows system designers to use high-level symbolic languages to describe the informational content of agents without requiring the symbolic structures to be implemented in the agent. It has become folk wisdom that 'situated agents' and 'representation' are incompatible concepts. In our view, this is not at all the case, and we feel there is much to gain from analyzing the semantics of representations from a situated perspective. In this spirit, we start by outlining this situated view of representation, then show how it contributes to design methodologies for building systems that track perceptual conditions and take purposeful actions in their environments.

2 Situated Representation

Our ultimate aim, in designing and building situated agents, is to have them perform a rich set of tasks correctly; that is, at each moment to carry out actions that are appropriate to their situations and goals. In order to specify correct behavior, and then to show that a particular program will satisfy that specification, we need to make precise the relationship between internal states of an agent and conditions in its environment. Once this relationship is made precise, we can give clear specifications of desired behavior for an agent in an environment and then generate a program for manipulating the internal states of the agent that satisfies those specifications.

For example, suppose we are designing an agent whose task it is to water a plant if and only if it is dry. From the outset, we must take into account the interaction between the agent and the environment: the specification of our problem contains a statement about the agent, *water the plant*, and a statement about the environment, *when it is dry.* In order to design such a controller, we must have a systematic way of talking about the relationship between the agent and its environment.

2.1 Existing Approaches

There has been a variety of approaches to describing the relationship between agents and their environments.

Many simple embedded systems are designed according to the principles of control theory. These systems usually have very little internal state, which typically consists of estimates of a set of real-valued variables that describe the state of the environment directly in parametric form. The designer of the control system chooses the real-world quantities upon which correct control responses depend, then designs machinery to estimate those quantities inside the agent based on incoming sensory signals. The control actions taken by the agent depend on the estimates of the quantities. This approach works well when the agent's interaction with the environment can be described by simple continuous functions and when the quantities that must be estimated can be sensed fairly directly. As agents and their environments become more complex, more abstract information will have to be represented, and this approach will no longer suffice.

In the artificial intelligence community, an agent's information about the state of its environment is typically represented symbolically. A formal language is developed by the system's designer, including an intended semantics that relates sentences stored in the agent's memory to the propositions about the world that they denote. This approach is applicable to arbitrarily complex relationships between states of the agent and the world, but it can be computationally intractable to maintain such a representation.

Many researchers have found it useful to provide ascriptional accounts of the relationship between states of an agent and states of the environment. McCarthy [10], for example, speaks of attributing knowledge of the temperature to a thermostat, and Newell [12] gives a definition of knowledge of an agent in terms of what would have to be true in the world in order for that agent's actions to be rational (in service of some goal). More formal notions of an agent's having knowledge about its environment are found in the work on *epistemic logic*: Moore [11] uses epistemic logic to model the knowledge of one agent about another's knowledge for the purpose of asking questions, for example. Halpern and Moses [2] provide a concrete computational model of knowledge in their applications of epistemic logic to the formalization of communication protocols in distributed systems. We use an approach similar to that of Halpern and Moses for specifying embedded agents, finding the concept of knowledge to be an effective way of describing the relationship between agent and environment.

2.2 The Situated-Automata Model

Traditional theories of computation are based on the notion of computation as *computation of a function*. The inputs are presented to the computational process, it works for some amount of time, then generates the answer and terminates. There is only one question and only one answer; of course, the question can be arbitrarily complex, potentially including many simple questions, but once the computation has started, it cannot be changed.

2.2.1 Interaction Model

It is more appropriate to think of an agent embedded in an environment as performing a *transduction*. It has a stream of inputs from the environment and generates a stream of outputs or actions to the environment. For the purposes of this work, we model the coupling between the agent and the environment as that of a pair of automata that are operating synchronously with one another; that is, their interaction can be seen as taking alternating turns, with the world generating an input (or "perception") to the agent, then the agent generating an output (or "action") to the world. In order to make this synchronous interaction a plausible model of interacting with a dynamic environment, we require the agent to generate actions, without fail, at strictly timed intervals.

In the transduction model, then, an agent is viewed as an automaton that generates a mapping from inputs to outputs, mediated by its internal state. Figure 1 shows the coupling of the agent and its environment. We note in passing that although there is only one agent in this model, most interesting environments have large numbers of agents. From the

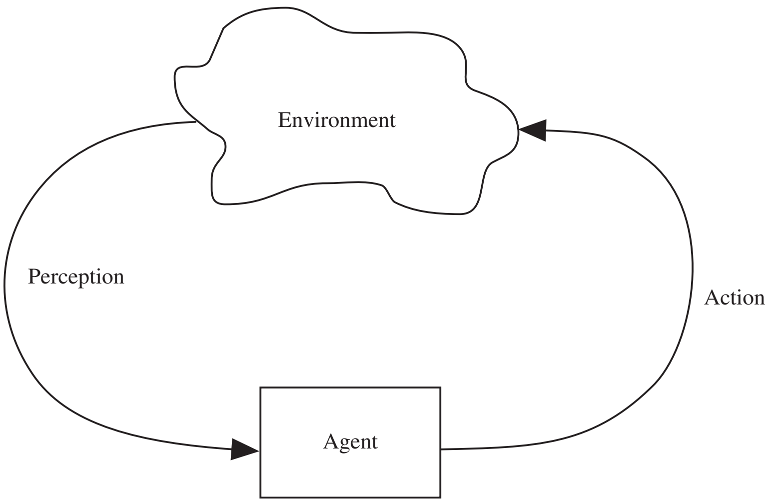

Figure 1: Interaction between agent and environment

perspective of this model, however, all of the other agents, whether robotic or biological, are taken to be part of the environment. This gives us a way to discuss the properties of a single agent from *its* perspective.

2.2.2 Correlational Definition of Information

Agent tasks are often specified in the form: *when P is true of the state of the* environment*, then the agent should take action A.* This specification could only be implemented immediately if the agent had direct access to arbitrary properties of the world. In general, that is not the case. Because an agent's actions in reality can only depend on its inputs and internal state, agent programs must ultimately be expressed in the form: *when P' is true of the state of the* agent*, then the agent should take action A.* Our problem, then, is to give a systematic account of the relationship between P and P'; if we can find some P' that implies P, then taking action A when P' holds of the agent's state is sufficient to satisfy the specification.

One way to view this relationship is in terms of a correlation between states of the agent and states of the external world. We will say that when an agent x is in state v, it *carries the information that* φ if and only if whenever it is in state v, φ is true in the world. This definition was originally articulated [14] in terms of equivalence classes of strings that would leave an automaton in the same state.

Given this definition, we can specify the simple robot plant-watering task more precisely: whenever the agent carries the information that the plant is dry, it should water the plant. Of course, as it stands, this is a fairly weak specification. For example, it could be satisfied trivially by any agent whose internal state is simply uncorrelated with the state of the plant. To rule out such a consequence, we might require further that the agent *track* whether the plant is dry: that is, if the plant is dry, the agent should carry the information that it is dry, and if it is not dry, the agent should carry the information that it is not dry. These two informational requirements taken together specify an agent that will water the plant if and only if it is dry.

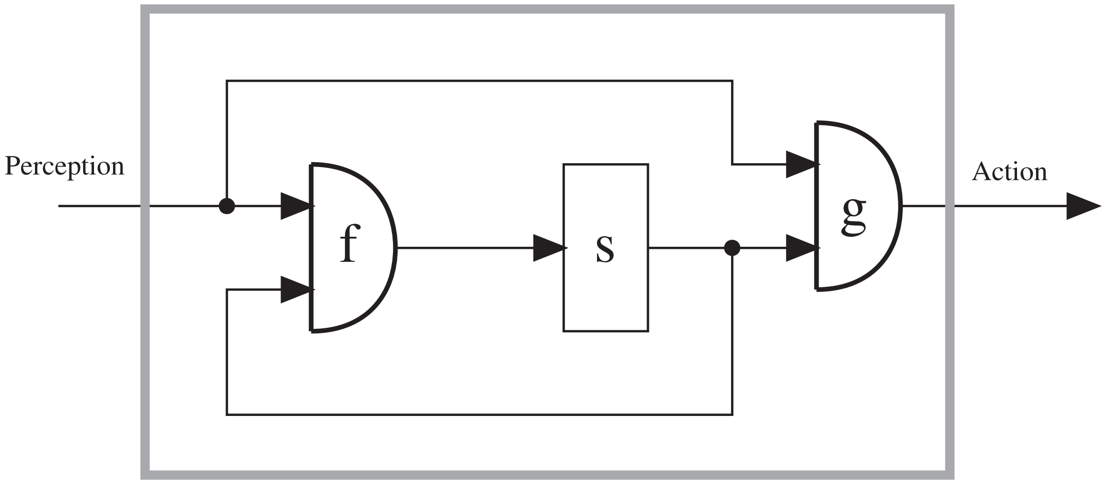

Figure 2: Circuit model of a finite-state machine

2.2.3 Circuit Model of Computation

We motivated the concept of information as a way to describe the relation between internal states of an agent and the external state of the world. In this section, we will introduce a way of describing the structure of agents and their internal states.

Considering an agent as an automaton does not, in itself, give us much help in its design or analysis. We take a finer-grained view here, considering an agent to be made up of parts, called *locations*, each of which is capable of assuming a set of local states (or "values") over time. The set of possible global states of the agent, then, will be the cross product of the local state sets of the atomic locations.

A machine is a set of locations whose values depend on one another over time. We can construct arbitrarily complex machines from machines of two primitive types: pure functions and delays. Pure function machines consist of two (possibly complex) locations and specify the values of one location (the output) as a function of the values of the other (the input). Delay machines also consist of two locations, but constrain the values of the output location to be the values that the input location had on the previous tick.[1]

Given a network of delay and function elements, complete with feedback connections, it is possible to organize it into a circuit of the form shown in figure 2, in which there is a state-update function, f, that maps the input and the old value of the internal state into a new value of the internal state, and an output function, g, that maps the input and the old value of the internal state into the output. It is often useful to think of the internal state as being a "state vector" containing the state of all the individual delay elements in the machine.

Although these definitions can be satisfied by machines made up of infinitely many locations, each of which can take on infinitely many values, we will focus our attention on machines with finite sets of atomic locations, each of which can take on only a finite set of values.

One obvious model for this abstract notion of machine exists in the form of digital hard-

[1]We use "tick" to mean one discrete time unit.

ware: locations correspond to wires, function machines correspond to logic gates, and delay machines correspond to flip-flops or registers. As is well known, any finite-state transduction can be carried out by a network of such basic components.

2.3 Consequences of the Situated-Automata Model

Although the correlational model of information can be used directly without the need for logical formalism, we have found it useful to adopt a logical framework, with a suitable syntax and semantics, for its application. In this section, we describe a logical formalization of the situated-automata model, then consider some insights it gives us into the nature of situated representation.

2.3.1 Formalization of the model

We have adopted logic as a means of describing the structure of machines, characterizing their interactions with the environment, and stating criteria for their correctness. This use of logic as a specification language is independent of its use as an implementation strategy. This is crucial because machines with complex logical descriptions often have very simple implementations and *vice versa*. Because a location x, in a certain state, carries information about any proposition that is true whenever x is in that state, there can be an infinity of propositions φ such that x carries the information that φ. Those propositions could not all be written down and manipulated symbolically. A simple example will illustrate this point. Consider a machine that consists of a single `and` gate. Behaviorally, the device is very simple: the output of the machine is 1 if and only if both its inputs are 1. Semantically, it may be more involved: if a 1 at the first input carries the information that φ and a 1 at the second input carries the conditional information that $\varphi \rightarrow \psi$, then whenever the output location has value 1, it carries the information that ψ. The inference rule *modus ponens* is not implemented in the machine, but it can be applied freely to reason about the information that locations of the machine have about the state of the world.

The correlational definition of information, introducted in section 2.2.2, can be directly formalized in epistemic logic [16], using the form $K(x, \varphi)$ to indicate that agent x carries the information that φ. This definition of information induces an equivalence relation on possible worlds, thus giving rise to an accessibility relation satisfying the S5 axioms [3]:

$K(x, \varphi) \rightarrow \varphi$	(truth)
$K(x, \varphi \rightarrow \psi) \rightarrow (K(x, \varphi) \rightarrow K(x, \psi))$	(consequential closure)
$K(x, \varphi) \rightarrow K(x, K(x, \varphi))$	(positive introspection)
$\neg K(x, \varphi) \rightarrow K(x, \neg K(x, \varphi))$	(negative introspection)

We view an agent as the union of all of its locations, so states of agents are comprised of states of component locations. We find it useful to apply the K operator not only to the agent as a whole, but also to its constituent locations. Information can be carried in simple locations or in compound locations, and aggregating locations leads us to an important corollary of the axioms, the principle of *spatial monotonicity*: an aggregate location carries the conjunction of the information carried by its constituent locations:

$$K(x, \varphi) \wedge K(y, \psi) \rightarrow K([x, y], \varphi \wedge \psi) \ .$$

The formal model can be used to prove correctness properties of agents: given a description of the circuit that makes up the agent and a description of the informational properties of the inputs to the agent, it can, for example, be shown that the outputs have certain informational properties [16]. This constitutes a correctness proof for an embedded agent.

2.3.2 Properties of Representations

In both standard programming methods and AI inference systems, the semantics of machine states is typically *stipulated*: the designer of the machine has a particular meaning in mind for values in designated locations and strives to construct the machine so that those stipulated semantics will hold. The situated-automata view allows us to attribute semantics to values in locations more "objectively" based on the correlational definition of knowledge. In the following sections, we examine aspects of the relationship between these two kinds of semantics.

Time is Meaning The correlational definition of knowledge assigns information content to locations *at a point in time* as a function of the value they contain at that time and the world states with which that value can co-occur. One immediate consequence of this definition is that a location containing a value at time t and continuing to hold that value until time $t+k$ will be assigned as its information content the disjunction of world conditions satisfied at any time instant in the interval $[t, t+k]$. Unless the designer has taken special care to design mechanisms for updating values in time to track external change, the objective information at a location may not be what he intended at all. For example, consider a robot that senses an object, stores the sensor reading, which the designer takes to be a representation of the distance to that object, and only updates the state every 10 seconds. The actual information content of the stored representation *all along* will in all likelihood not be what was intended but some weaker (though not necessarily vacuous) proposition that bounds the distance to the object, depending on the possible maximum relative velocity between the robot and the object.

Given the situated view of knowledge, this is not very surprising; but standard AI systems often operate in a way that does not take seriously the degradation of information over time. They get certain sensory inputs that are written down symbolically in the memory of the machine and manipulated over time. Even if the stipulated semantics of the formulae were true when the process began, it is entirely possible that the world will change enough during the course of the inference process that the data on which the conclusions were based are no longer valid.

Machines that Manipulate Symbols The situated-automata framework can also be applied to computations that perform symbol manipulation. The requirement that intended semantics match real ones is emphasized in symbolic systems. It is possible, in theory, to design a system that manipulates symbolic representations of propositional information about the world in such a way that a proposition is only symbolically written in memory when it is justified by the correlational theory; that is, when those locations carry that same propositional information. This can be achieved through careful use of time stamps and axioms that describe the dynamics of the world, but it has proven very difficult in practice and is rarely done.

It is interesting to consider, in more detail, an example illustrating two different ways of encoding information. In the first case, we dedicate a particular bit in a machine to the representation of whether there is an obstacle within two meters of the agent; if it is on, then there is an obstacle and if it is off, it is not known whether there is an obstacle. In the second approach, the symbols `distance(robot,obstacle) < 2` (or symbols in some other language with compositional semantics) can be written anywhere in memory and, if the system is designed properly, those locations will carry the same information as the single bit of the first example being on. In the first case, we say that the information is encoded "by location"; that particular bit is devoted to the representation of a single condition, so the location of the encoding of the bit is crucial. In the second case, the information is encoded "by value"; the values of the locations involved in representing the information are crucial, but that same combination of values occurring at any location would carry the same information.

To see how a correctly functioning symbolic representation system yields the correct correlational semantics, consider a very simple language with symbols of two types: unary predicates (e.g., `fully-charged`, `red`, `tired`, `distant`) and individual constants (e.g., `robot1`, `ball37`, `john`, `wall2`). Sentences in this language could be represented by simple juxtaposition of symbols as in this sequence:

```
[fully-charged robot1 red ball37 tired john distant wall2]
```

While the correlational account of information assigns time-varying propositions as the meanings of a value at a location, under a more standard Tarskian account, these symbols might be interpreted differently: the unary predicates might denote mappings from individuals to temporally-indexical propositions (e.g., mappings from time to truth values), while individual constants might denote individuals (or perhaps temporally-indexical individuals). Note that if the representation is uniformly veridical, that is, if the sentence is in memory only when it is true, then at the level of propositions, the two semantics agree. Importantly, however, the situated-automata framework would have attached that meaning to the symbol sequence whether or not the designer had that Tarskian semantics in mind. In other words, the attribution of semantic content need not be externally stipulated.

There is clearly a trade-off between using these two kinds of representation (and a whole spectrum of intermediate cases). When information is represented largely by location, the number of bits or atomic locations used can be very small, making this form of representation quite efficient. Symbolic representations are notoriously space-inefficient, requiring many bits to encode syntactically the propositional content to be represented. However, these representations are also very flexible. When storing information by value, any proposition expressible in the language can be represented; when storing information by location, on the other hand, each individual proposition must be thought of in advance by the designer of the machine (or, as we will see, by a compilation system).

As observed above, when a location carries the information that P, so does every super-location; there is a minimal location that carries the information that P. We will say that information is *localized* to a minimal location that carries it. An especially important consideration in the decision between encoding by value and by location is the amount of

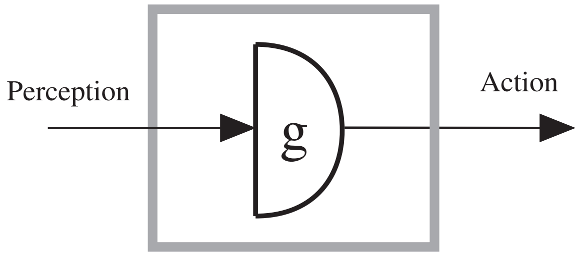

Figure 3: Pure action case

work that must be done to localize important information. In order to take a particular action when P is true, the locations that control the effector must eventually carry the information that P; it is insufficient for that information to be encoded in some collection of locations in a uniformly-interpreted memory. The amount of work that has to be done to take the information that P and localize it into the effector bits, or any other location where it might be further combined with other information, can crucially depend on how it was initially represented. When information is encoded by value, it can often take many complex operations of indexing and pattern matching to localize; information that is encoded by location can often be used directly. Such considerations should guide the representational choices that are made in designing and building agents.

3 Designing Agents

Given the situated view of agents and environments as interacting automata and the circuit model of computation, we can build a design methodology for agents situated in dynamic environments. We first consider the case in which the agent has no internal state, then the case in which the agent monitors, but does not affect, the environment. We conclude by combining our design methodologies for an entire agent.

3.1 Pure Action

We begin by considering control in a very simple setting, namely stimulus-response systems that map current inputs to outputs without any dependence on prior inputs. At each instant, the inputs carry information about the immediate state of the environment, but the agent has no internal memory by which to distinguish otherwise similar states through residues of past experiences. In the automaton model, the state set of a stimulus-response automaton contains only one state, and inputs are simply passed on to the output relation. This is illustrated schematically in Figure 3. Although stimulus-response agents are extremely limited, they are complete agents, nonetheless, and constitute a relatively easy-to-analyze starting point.

By what criteria can a stimulus-response system, or any action-selection system, be judged successful? A natural way to answer this question is to relativize success to some stated goal specification that is taken to be part of the problem statement. Different families of control problems arise, depending on what is meant by the term “goal.”

One dimension of variability in defining goals is whether the goals are fixed or dynamic. With fixed goals (or design-time goals), action-selection mappings are evaluated relative to the entire trajectory of states they engender. To model an agent as pursuing dynamic goals, on the other hand, assumes some method for defining moment-to-moment variations in what the agent seeks. Even with dynamic goals, the agent is seen as having at least the fixed goal of acting rationally, that is, at each moment selecting actions consistent with its current goals and information. (Interesting subtleties arise in pathological situations where agents might preserve rationality by choosing not to want or not to know.)

Another dimension of variability in goal definition arises from the complexity of the goal and the form in which it is expressed. Goals can vary in complexity from that of maintaining simple environmental invariants, to satisfying arbitrary temporal predicates, to optimizing complex numerical evaluation criteria (e.g., maximize throughput while minimizing energy, with complex tradeoffs). Regarding the form of presentation, complexity can vary from a simple enumeration of states, for goals of maintenance, to complex formulas in expressive logical languages, closed under Boolean operations, quantification, and rich temporal operators.

Because, in extreme cases, agent synthesis can be intractable, it is not a reasonable objective to try to develop universal solution methods. Rather, it is preferable to develop a methodology for specifying particular action strategies, and to develop an inventory of solved special cases that can expedite agent construction in practical situations. As with goal specification, the specification of action-selection mappings can take many forms, direct and indirect. One family of direct methods includes notations for defining functions of one or more input variables in a suitable language, such as look-up tables (only in very simple cases), functional expressions, circuit descriptions, or data-flow graphs. A related family of methods uses the calculus of relations rather than functional expressions, in some cases with a determinization operator applied as the last step, after the output relation has been composed by applying operations like union, intersection, and restriction to primitive relations. The base-level relations can be represented either enumeratively or in more compact form.

Rex [4, 8] is a language for specifying action mappings (as well as machines with internal state) as abstract circuit descriptions. Rex served as a substrate for Gapps [5, 7], which takes a symbolic specification of an action mapping and compiles it into fixed runtime circuitry. A Gapps program consists of a set of goal reduction rules, which specify how a high-level goal is transformed to a more specific and simpler low-level goal. When given a fixed goal to satisfy, the Gapps compiler generates a provably correct (though possibly partial) reactive program (input-to-output map) for that goal. A Gapps program is guranteed to be corrrect, but not necessarily complete; if it outputs an action, it is appropriate to the situation, but it will not necessarily output an action in every situation. One of the main strengths of Gapps is its least-commitment approach: if many low-level actions satisfy a particular goal, they are all returned as part of the result, which allows non-deterministic choice of action; this greatly simplifies the compositional construction of programs that satisfy multiple goals.

Although Gapps programs are specified using symbolic rules, they still require the programmer to do a great deal of the work, especially in determining what chains of actions will result in desired outcomes. Gapps does not support the reduction of goals into sequences of action, so the programmer has to maintain any such committments himself based on perceived conditions. In many cases, we would like programs to be derived from much more abstract specifications.

3.1.1 Maintaining Invariants Through Goal Regression

Indirect methods define the action-selection mapping by deriving it from some description of the environment and the goal, whether in the form of an explicit combinatorial object like a graph, or in the form of declarative assertions, such as operator descriptions found in classical AI planning systems. To illustrate how a stimulus-response agent can be constructed algorithmically from an explicit description of an environment and goal, we consider the special case of agents that maintain invariants. Although the method illustrated does not scale well with large state sets, it does introduce important concepts and build up intuitions about properties of action strategies.

A stimulus-response agent that maintains invariants can be synthesized as follows. Let the environment be represented as a nondeterministic automaton $\langle S, P, A, \mathit{init}, \nu, \mathit{out}\rangle$, where

- S is a finite set of states of the environment;
- P is a finite set of outputs (these are usefully viewed as *percepts* from the agent's perspective);
- A is a finite set of actions that the agent can generate as input to the environment;
- *init* is a set of states containing the one that the environment is known to be in initially;
- ν is a relation on $S \times A \times S$ where $\nu(s_1, a, s_2)$ holds if it is possible for the world to make a transition from state s_1 to state s_2 when action a is generated by the agent; and
- *out* is a function mapping S to P

For the simple pure-action case, we assume that the environment automaton outputs its full state as output. In other words, the percept set P is identical to the state set S and *out* is the identity function on states. Let the goal be represented by G, a subset of S, that the agent is to maintain as an invariant condition.

A solution to this problem is G^*, a subset of G within which the environment can be made to stay indefinitely, and a mapping from G^* to A, specifying the actions the agent should take in order to stay within G^*. The set G^* can be computed iteratively, as follows:

Let $G_0 := G$
For $n = 0$ to ...
    Let $G_{n+1} = \emptyset$
    For all $g \in G_n$
        If $\exists a. \forall g'. \ \nu(g, a, g') \rightarrow g' \in G_n$ then
            add g to G_{n+1}
    When $G_n = G_{n+1}$, terminate and return G_n.

Each intermediate set G_n is the set of states from which G can be maintained for at least n steps. For any state $g \in G$, if there exists an action such that from every possible successor state g', G can be maintained for n steps, then in state g, G can be maintained for $n + 1$ steps. This step is called "goal regression," because G_{n+1} is the weakest precondition under which G_n can be made true on the next step (see Rosenschein [13] or Waldinger [17] for a more complete description of regression-based planning). When this process reaches a fixed point, then we have determined the set, G^* of states from which G can be maintained indefinitely. In order to maintain G^* from some state g, the agent can do any action a such that $\forall g'.\nu(g, a, g') \rightarrow g' \in G^*$.

3.1.2 Goal Regression Example

Consider a simple domain in which a robot must keep plants alive by watering them. The action set of the robot contains actions to water each plant and *no-op*, the action that does nothing. The state of the world can be expressed as a vector describing the moisture level of each plant, where 4 is wet (just watered) and 0 is dead. For example, the vector `(4 3 0)` describes a situation in which the first plant is wet, the second slightly drier, and the third is dead. Moisture decreases by one every time step on which the plant is not watered. Plants that die (reach moisture level 0) stay dead forever.

We consider a situation in which there are three plants and the goal is to maintain the condition "no plants are dead"; G is enumerated below (with equivalent states under different orderings of plants deleted, because the identity of the individuals is irrelevant to the maintenance of this goal):

```
(4 4 4) (4 4 3) (4 4 2) (4 4 1) (4 3 3) (4 3 2) (4 3 1) (4 2 2) (4 2 1)
(4 1 1) (3 3 3) (3 3 2) (3 3 1) (3 2 2) (3 2 1) (3 1 1) (2 2 2) (2 2 1)
(2 1 1) (1 1 1)
```

The G^* resulting from the goal regression algorithm is:

```
(4 4 4) (4 4 3) (4 4 2) (4 4 1) (4 3 3) (4 3 2) (4 3 1) (4 2 2) (4 2 1)
(3 3 3) (3 3 2) (3 3 1) (3 2 2) (3 2 1)
```

The most constrained state in G^* is `(3 2 1)`. By watering the plant at moisture level 1, it is changed to `(4 2 1)`, and then to `(4 3 1)`, and then to `(4 3 2)`. At this point, the *no-op* action is allowed, and the robot can rest, leading to the original starting state of `(3 2 1)`. Note that there are no states in G^* with two plants at level 1 or three at level 2; although these states are in G (no plants are currently dead), they are not in G^*, because it is not possible to keep all the plants from dying in the future.

3.1.3 Discussion

As mentioned above, this construction does not scale well as the number of environment states increases, and this motivates the use of other representations. Although ordinarily used to handle run-time goals of achievement, the declarative operator descriptions used in AI planning systems encode the same information as state-transition graphs, and can be used to drive the construction above. Operator descriptions provide a more intuitively

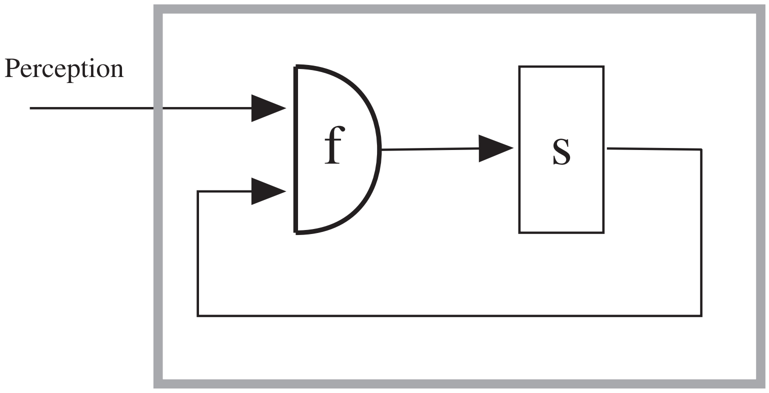

Figure 4: The pure perception case

interpretable form of expression and can often be manipulated more efficiently because they refer to large subspaces of the state space with terse symbolic labels. Rather than calculating G^* through enumeration, operator descriptions allow it to be calculated through symbolic regression. This may be more or less efficient that the alternative, depending on specifics of the problem domain. This technique has been implemented and explored as an extension to the Gapps programming system [6].

3.2 Pure Perception

Until now we have assumed that inputs from the environment are sufficiently informative, in that they encode all the world-state information needed to drive action. In cases where less information is available, the inputs to action selection must be derived by accumulating partial information over time, and for this purpose additional machinery is necessary. We refer to this additional machinery as the "perception system" and explore its properties in this section.

As in the case of action selection, it will be useful to approach perception by beginning with a study of the *pure* phenomenon. By pure perception we mean agent-environment systems in which the outputs of the agent have no influence on the environment at all, and the agent is simply a tracking system, or monitor: a passive observer, seeing, but not seen by, the environment. This special type of agent, again, will be of limited practical use but does illustrates the essential features of information extraction. The setup for pure perception is illustrated in figure 4. The lack of influence of the agent on the environment cannot be depicted graphically; the environment's next state function is independent of the output of the agent.

The focus in analyzing the perception system is on the kind of correspondence maintained between its internal states and states of the environment. This correspondence, in fact, is a form of invariant of exactly the type investigated in the previous section, but over the states of the agent-environment pair rather than just the environment. Even when the environment is indifferent to the actions of the agent, it makes sense to ask how the perception component might be designed to maximize the degree of correlation between its states and those of environment, hence maximizing its information.

To see this most clearly, consider an environment, modeled once again as a nondeterministic automaton $\langle S, P, A, init, \nu, out\rangle$. What is the maximum amount of information encoded in an instantaneous percept? In general, the best we can do is to associate with each percept p the set of environment states with which it is compatible (i.e., those s such that $out(s) = p$). What is the maximum amount of information about the environment that could be accumulated by the agent automaton over time? Given a rich enough inventory of internal states, a pure perception agent could optimally track the environment by having states isomorphic to the powerset of environment's states. Let $\Sigma = \text{powerset}(S)$ be the set of internal states of the agent, with the agent in state σ_i if and only if every world state $s \in \sigma_i$ is consistent with the agent's perceptual history so far. The agent's initial state is the set of possible initial states of the environment, *init*, and its transition function $N(\sigma, a, p)$ which maps the previous internal state σ, the last action a, and the last percept p, into a new internal state, is given by

$$N(\sigma, p, a) = \{s' | \exists s \in \sigma . \nu(s, a, s') \wedge out(s) = p\} \quad .$$

This *powerset automaton* might be cumbersome, indeed, but its tracking behavior would be optimal.

Although as the number of environment states rises, the powerset construction quickly becomes infeasible, it is useful as a thought experiment because much of its value can be preserved through efficient but information-rich approximations. Mathematically, these approximations are homomorphic images of the ideal powerset automaton, and thus are consistent with, but not as complete as, that ideal, or optimal, tracker. Nevertheless, these homomorphic images allow useful information to be monitored, while carefully trading off computational space and time, under the designer's control. One simple approach to constructing homomorphic projections of the powerset automaton is to choose a set of *interesting* or *significant* states in the powerset automaton, and close these under union and intersection. The result is a lattice, which will be a sub-lattice of the powerset Boolean algebra. The construction of the initial state and transition-function of the perception system then proceeds as in the case of the powerset automaton above, but with the true powerset elements approximated by least upper bounds in the sub-lattice. For example, if in the original powerset automaton the transition function maps a state to a successor state that is not an element of the homomorphic-image lattice, the element of that lattice which best approximates the successor state will be returned instead. Thus the lattice transition function *approximates* the optimal transition function and degrades gracefully with the precision of the representation. The lattices themselves would typically be cartesian products of simpler lattices, with elements that could be represented compactly as parameter vectors.

This technique forms the basis of the RULER system [15]. RULER takes an approach analogous in many ways to AI planning systems. In RULER, the environment is described by a set of assertions, including temporal assertions that describe conditions that are either true initially or that will be true in the next state, depending on current conditions. The RULER compiler synthesizes perceptual machinery (an initial state and next-state function) by chaining together these individual assertions, not with a view toward constructing action sequences, but rather with a view toward computing descriptive parameters in the next state's world model. The use of lattices as the semantic domain of interpretation of the model parameters, along with effectively closing the parameter space under intersection, allows incremental information to be folded in nicely and leads to a compositional methodology for constructing perceptual update mechanisms.

3.2.1 Ruler Compilation

RULER's compilation method works as follows. The compiler takes as input a description of information carried by the run-time inputs to the program and the internal state variables, as well as a background theory containing temporal facts about the world. The compiler operates by deriving theorems about what is true initially and about what will be true at any time, given what was true at the previous time. In the course of the derivation, free variables are instantiated in the manner of logic programming systems. From the instantiated formulas, the compiler extracts a program for initializing and updating a state vector with the desired informational properties.

More precisely, the compiler's inputs consist of the following:

- a list $[a_1, ..., a_n]$ of input locations
- a list $[b_1, ..., b_m]$ of internal state locations
- for each input location a, a formula $P_a(U)$ with free variable U
- for each internal location b, a formula $P_b(U)$ with free variable U and a function $rconj_b$
- a finite set Γ of facts.

The formulas $P_x(U)$ express propositions parameterized by U, where U ranges over run-time values of location x; for example, $P_{b4}(6)$ might denote "current soil mosture level $\geq$ 6." These values are drawn from a lattice so that degrees of partial information can be represented. The $rconj$ operations are binary functions that take a pair of lattice values and combine them into a single lattice value summarizing their conjunctive content as precisely as possible. (The $rconj^*$ operation extends $rconj$ to sets of lattice values in the natural way.) Using formulas in this way, the propositions that were merely implicit in the information of the machine can be made explicit and manipulated by the compiler.

For each internal location b, the compiler computes two sets of runtime value terms I_b and N_b defined as follows (the $\Box$ symbol is the temporal logic operator representing "necessarily always"):

$$I_b = \{e \mid \Gamma \vdash \Box init\ P_b(e)\}$$

$$N_b = \{e' \mid \Gamma \vdash \Box(P_{a_1}(a_1) \wedge \ldots \wedge P_{b_n}(b_n) \rightarrow next\ P_b(e'))\}\ ,$$

where $e' = f([a_1, \ldots, a_n], [b_1, \ldots, b_m])$. If we are initially ignorant of soil moisture, we might have only $\Box P_{b4}(0)$, so $I_{b4} = \{0\}$. If our lower bound on moisture decreases 1 per time step, then we might have $\Box(P_{b4}(n) \rightarrow next P_{b4}(n-1))$. Each set I_b contains terms representing properties that can be proved from the background theory Γ to hold initially in the world. Each set N_b contains terms for properties that can be proved to hold "next," given the properties that hold now as represented by the values of the input and state locations. If these sets are infinite, they can be generated and used incrementally. This is discussed more fully below.

From these collections of sets the compiler computes the initial value of the state vector, v_0, and its update function, f. The initial value is computed as follows:

$$v_0 = [rconj^*_{b_1}(I_{b_1}), \ldots, rconj^*_{b_m}(I_{b_m})],$$

In other words, the initial value of the state vector is the vector of values derived by *rconj*-ing values representing the strongest propositions that can be inferred by the compiler about the initial state of the environment in the "language" of each of the state components. Similarly, for the next-state function:

$$f([a_1, \ldots, a_n], [b_1, \ldots, b_m]) = [rconj^*_{b_1}(N_{b_1}), \ldots, rconj^*_{b_m}(N_{b_m})].$$

Here the compiler constructs a vector of expressions that denote the strongest propositions about what will be true next, again in the language of the state components.

In the case of the initial value, the *rconj* values can be computed at compile time because the values of all the arguments are available. In the case of the next-state function, however, the *rconj* terms will not denote values known at compile time. Rather, they will generally be nested expressions containing operators that will be used to compute values at run time. Assuming the execution time of these operators is bounded, the depth of the expressions will provide a bound on the update time of the state vector.

Without restricting the background theory, we cannot guarantee that the sets I_b and N_b will be finite. However, even in the unrestricted case the finiteness of terms in the language guarantees that whichever elements we *can* derive at compile time can be computed in bounded time at run time. Furthermore, the synthesis procedure exhibits strongly monotonic behavior: the more elements of I_b and N_b we compute, the more information we can ascribe to run-time locations regarding the environment. This allows incremental improvements to be achieved simply by running the compiler longer; stopping the procedure at any stage will still yield a correct program, although not necessarily the program attuned to the most specific information available. Since, in general, additional *rconj* operations consume run-time resources, one reasonable approach would be to have the compiler keep track of run-time resources consumed and halt when some resource limit is reached.

As we have observed, without placing restrictions on the symbolic language used to specify the background theory Γ, the synthesis method described above would hardly be practical; it is obvious that environment-description languages exist that make the synthesis problem not only intractable but undecidable. However, as with Gapps and other formalisms in the logic programming style, by restricting ourselves to certain stylized languages, practical synthesis techniques can be developed.

We have experimented with a restriction of the logical language that seems to offer a good compromise between expressiveness and tractability. This restriction is to a weak temporal Horn-clause language resembling Prolog but with the addition of *init* and *next* operators. The derivation process proceeds as described above using backward-chaining deduction techniques as the specific form of inference. A prototype system has been built implementing the Horn-clause version of the synthesis algorithm. One of the ways the language differs from Prolog is in the strong distinction between compile-time and run-time expressions. Compile-time expressions undergo unification in the ordinary manner; run-time expressions, by contrast, are simply accumulated and used to generate the final program. The

RULER system was run on several small examples involving object tracking and aggregation, and the synthesis procedure has proved tractable in our test implementation.

Using off-line synthesis techniques, conditions that are semantically complex can still be recognized with limited machinery, and for this reason it is entirely consistent with the "reactive" bias to admit sophisticated semantic information and models. With some care, the designer can have the best of both worlds: declarative forms can be used to clarify the semantics of the domain representation, and finite parametric representations can be generated by the compiler to guarantee bounded-time updates and real-time response.

3.2.2 Ruler Example

This section sketches out a simple example of a pure-perception system synthesized by the RULER system. Imagine that we again have a plant-watering robot, but we are now concerned with constructing its perceptual system so that it maintains, at all times, as much information as it can about the moisture level of a collection of plants. The representation used by the system must be able to accommodate uncertainty, so we use an interval, representing known lower and upper bounds on the true moisture level of the plant. This gives us our first rule,

```
moisture(p,[0, max]).
```

which states that the moisture of plant p is always between 0 and some maximum level. Additionally, if the robot is at the plant, it can get an approximate reading of the moisture level from its sensor:

```
moisture(p,[v-1,v+1]) :-
    at_plant(p,1),
    moisture_sensor(v).
```

The `at_plant(p,1)` term requires that the robot know that it is at plant p at the time the moisture is being sensed. In this case, there is an input bit, `a`, such that `at_plant(n,a)`. The robot has been constructed in such a way that if `a` has value 1, then the robot is known to be at plant `n`; if it has value 0, the robot is not known to be at that plant. We will treat other propositions similarly.

The dynamics of the world are specified in terms of `next` rules. If we know that the robot's last action was not to water the plant (either because we know it didn't water or because we know it wasn't at the plant), then the moisture may either increase (perhaps due to rain) or decrease by 1:

```
next moisture(p,[l-1,h+1]) :-
    not_watering(1),
    moisture(p,[l,h]).

next moisture(p,[l-1,h+1]) :-
    not_at_plant(p,1),
    moisture(p,[l,h]).
```

If we know that the robot did just water the plant, then the moisture will increase to its maximum level:

```
next moisture(p,[max,max]) :-
    at_plant(p,1),
    watering(1).
```

If we don't know whether the robot watered the plant (either because we don't know whether it watered or because we don't know whether it was at the plant), the bounds spread quickly:

```
next moisture(p,[l-1,max]) :-
    moisture(p,[l,h]).
```

Note that the last rule does not conflict with other rules that provide tighter bounds on the moisture. We combine the results of these rules by specifying an *rconj* rule for `moisture`. In this case, it is simply to intersect the intervals. Running RULER on this set of rules results in a circuit that retains as much information as possible about the moisture of the plant, given its inputs and the specified representation.

3.2.3 Objects, Properties, and Relations

While conceptually adequate for generating provably correct perceptual subsystems, at least for nonprobabilistic domain models, RULER is limited in that it makes no special provision for modeling worlds in which objects and their properties and relations are of special importance. This is the case, for example, in visual perception where objects move in and out of view, and a prime form of information to be extracted from the scene concerns the identity of objects and their spatial relations to one another and to the observer. To begin to address domains of this type, we developed an information-update schema we named Percm.

The Percm schema can be thought of as a specialized form of RULER in which a finite, but shifting, set of objects is being tracked and described. The descriptions are represented as labeled graphs, with node labels representing unary properties of objects, and edge labels representing binary relations between objects. One of the objects is the agent, and the rest of the objects can vary, moving in and out of attentional focus. This scheme bears some relationship to the indexical-functional representations developed by Chapman and Agre for Pengi, but with rigorous correlation-based semantics. The node and edge labels are drawn from a space of data values representing lattice elements, just as in the RULER case, only now the propositional matrix is fixed (i.e., a fixed conjunction of properties and relations) and the lattice elements are constrained to be of semantic type *property* or *relation*, or to be coercible to such values.

Figure 5 shows the basic runtime data structures that underlie a Percm with n elements. There is a vector of length n, each of whose elements contains the unary properties of the ith element being tracked. Often, index 1 is reserved for the agent. In addition, there is an $n \times n$ matrix in which cell $\langle i, j \rangle$ contains the strongest representable information available about the relation between objects i and j. In many cases, the relations will be symmetric (or canonicalizable) so that only the upper triangle of the matrix needs to be explicitly represented.

The update cycle for this data structure is similar to RULER's, but in the Percm context, fixed background descriptions of the environment are provided not in the form of propositional assertions about world-state transitions, but rather as rules, both temporal and

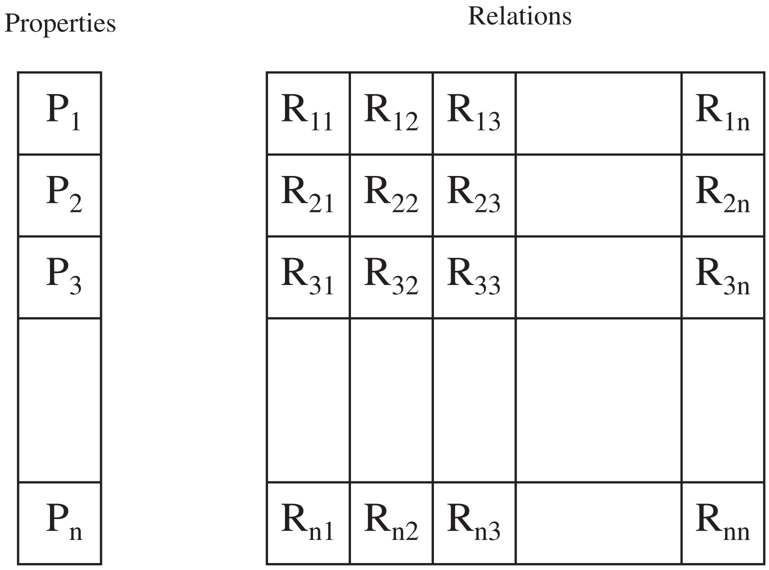

Figure 5: Data structures supporting an instance of the Percm schema with n objects

atemporal, for computing object properties and relations. This information is built into a set of operations used to update the data structures. These operations are:

create: maps an input value to initial object properties and relations inferable from that input;

propagate: strengthens properties and relations among objects x and y by deriving what can be inferred from existing properties and relations between each of x and y and some third object, z;

merge: combines descriptions of objects x and y if their properties and relations imply that they are identical;

aggregate: creates a new object y whose existence can be inferred from the existence of constituent objects $x_1, ..., x_n$ with appropriate properties and relations, and initializes y's description based on descriptions of constituents.

degrade: maps properties and relations at time t to new values inferable for time $t + 1$;

The perceptual system is synthesized by composing and iterating these operations to update the object descriptions, with values again drawn from lattices to obtain gracefully degrading approximations. Because Percm is a finite schema of bounded size, to complete the specification of an instance of the Percm schema, the designer must also define how, in the case of object overflow, objects are to be discarded or withdrawn from active attention. Circuitry to keep the data structures updated can be large, but is of bounded size. Operations like finding an empty cell for a new object can be done in a very shallow circuit with size $O(n)$.

3.2.4 Percm Example

In order to illustrate the ideas behind the Percm schema, we present a simple example of its operation. A mobile robot, traveling through a new environment, needs to construct a representation of the salient objects and their spatial relations.

The robot might begin by perceiving, instantaneously, that there are two objects in front of it: a chair and a person. It *creates* two objects, assigning them indices 2 and 3, stores some of their unary properties (such as type and color of the chair, gender and hair-color of the person) in cells P_2 and P_3, and stores bounds on the spatial relations between each one and the robot in $R_{1,2}$ and $R_{1,3}$.

Immediately, a *propagate* operation can compute bounds on the spatial relations between objects 2 and 3 and store it in cell $R_{2,3}$. These objects can be neither *merged* nor *aggregated.*

Finally, in the *degrade* step, knowledge about the generic motion abilities of chairs and people, as well as the current motion of the robot, is used to degrade the spatial relation information. The robot typically has good local odometry (motion information), so it knows how much it has moved relative to the position it was in when it first perceived these objects and can update $R_{1,2}$ and $R_{1,3}$ accordingly. If both of these objects were static, the robot could wander away and become confused about its relation to the objects, but still retain precise information about the relation of the objects to each other. However, in this case, people are far from static, so the degrade step will increase the bounds on all spatial relations between the person and other objects, because the person could potentially move in any direction.

On the next cycle, the robot again sees the person, but because of its changed perspective, is able to measure the person's height. This person gets *created* as object 4 in the Percm data structures. This time, on the *merge* step, the robot is able to infer that, because of their close spatial positions (and perhaps because two people were not seen simultaneously), that objects 3 and 4 must really be the same. They are merged by conjoining their properties and their relations to other objects and storing them in a single index. The other index is marked as free. Now, the height and hair color are both known about a single person.

The *aggregate* operation is useful when entire complex objects cannot be perceived instantaneously. Thus, a robot attempting to identify a large truck might individually identify wheels, a cab, and a flat bed, then aggregate them into a truck object.

As the data structures begin to get full, it will be important to purge items in a useful way. Objects may be purged because their information is weak, or they are superseded by a complex object, or for a variety of attentional reasons based on the robot's current goals.

3.3 Combined Perception and Action

The techniques illustrated in the two previous sections can be combined directly to synthesize control systems containing both perception and action components. For instance, using the Gapps approach, one could develop mappings from information states to actions, where the information states are the output of a perceptual subsystem synthesized using the RULER or Percm methodologies. If there were no interactions among design decisions needed for the two subsystems, the definition of the *information state* of the agent would act as a clean interface, and the combined system would exhibit the intended behavior. In general, however, there are interactions, and in this section we explore the nature of those interactions and potential methods of dealing with them.

The first problem is the specification of the information-state interface between the two modules. This problem exists even when the perceptual mechanism is degenerate. It is possible that the perceptual inputs from the environment do not provide enough information for the goal to be satisfied. Another difficulty arises if the information is available, but is encoded in such a way that the localization machinery is of intractable complexity.

The design of the system becomes much more complex when the actions taken by the agent in the environment affect the information that will be available to it. When choosing action strategies, attention must be given to how actions chosen now will maintain the flow of information necessary for distinguishing among future states to be acted on. In AI, this problem often goes under the label of the *knowledge precondition* problem [11]: it is not enough to *be* in an environmental state when a certain action is appropriate; the agent must *know* that it is in an environmental state in which that action is appropriate.

The problem grows more complex when perceptual machinery distills information contained in the sensory input stream, and still more complex when the goal itself pertains to affecting the agent's own information state. In these cases, the internal structure of the perception module is, from the point of view of the action-selection module, part of some external environment whose dynamic properties are critical to the success or failure of its strategy. Unfortunately, without elaborating the internal structure of the perception module first, statements of fact about this environment cannot be made, and hence no valid action strategy can be chosen. In general, action strategies intended to satisfy information goals can only coherently be developed in the context of fixed perceptual machinery, or, at least, in the context of articulated assumptions about the perceptual machinery.

A natural development methodology, then, would be to design the perception module first, choosing conditions to be tracked and defining update circuitry that tracks these conditions in the passive sense introduced in the previous section, but does not guarantee the input streams that will force it to the right state. After defining this fixed machinery, an action strategy can be defined, relying on the definition of the perception component as if it were part of the environment. This strategy is designed to cause input streams flowing into the perception component to drive it into the appropriate states and actively makes use of constraints imposed by the previously chosen structure of the perception module. In principle, when perception and action modules are generated from declarative domain descriptions, a single set of facts about the environment should suffice to generate both modules. In other words, RULER-like state transition rules, combined with operator-description-like action descriptions, contain enough constraints to generate systems that seek information. The RULER rules generate a perceptual system that maintains, as an invariant, correlations with conditions that the action system needs to test. This approach can involve a search, albeit at design time, for suitable conditions that can be effectively tracked.

In all of these approaches, the result is an automaton with an objective informational relation to its environments. This is unlike the usual case in AI, in which knowledge pre- and post-conditions have been analyzed using theories that link internal states of agents to their environment only through stipulated semantic-denotation relations attributed by designers somewhat arbitrarily to symbolic data. This distinction is substantial, and it is encouraging that many of the same semantic desiderata that have been pursued in traditional AI planning and representation systems can be achieved in a more mechanistic, and potentially far more efficient, control-theoretic setting.

A final area of complexity is the pervasive uncertainty found in natural environments. Throughout this work, we have modeled uncertainty using simple nondeterminism. While this allows designers and machines to avoid committing to information they do not possess, these models are extremely conservative in that they regard all alternative states that are not ruled out by hard constraints to be of equal importance. In real task domains, however, some of those alternatives are far more likely than others, and this fact is essential to the proper exploitation of the information. A model that is midway between deterministic and nondeterministic models is the probabilistic model in which state transitions, under a given input, are described by probability distributions. A natural mathematical model for such systems is the Markov process, which has been studied extensively by applied mathematicians.

The difficulty in using probabilistic models together with the symbolic techniques described above is the nonmonotonicity of probabilities, which leads to noncompositionality of the design technique. By conditioning on further evidence, the probability of a proposition can either be reduced or increased. This means that a designer cannot, in general, define a module of the perceptual component, prove a strong statement about the semantics of its outputs, and then proceed to use that module together with other modules; conditioning on the joint states of the modules may completely undermine the intended semantics of the first module. Furthermore, the action strategy embodied in the action-selection component is integral to the definition of the probabilistic state-transition matrix of the entire system. Just as before when we could not, in principle, define an action strategy before providing a fixed definition of the perception component, here we cannot define the perception component without constraining action first. The apparent circularity only points to the fundamental need to consider the agent as an integrated whole; the behavior of the entire system, agent plus environment, is determined only when all the boundary conditions have been specified. Interim constraints and incremental refinement may be useful, but must be used cautiously, especially when modeling domains probabilistically. The theory of partially observable Markov decision processes [9, 1] provides a theoretically well-founded methodology for deriving controllers in stochastic domains, but it seems to be computationally very intractable.

4 Conclusions

The aim of situated-automata theory is to provide a new semantic perspective on intelligent agents. Traditional AI has been dominated by "reasoning" metaphors drawn from folk psychology in which programs are seen as actors manipulating linguistic elements, drawing conclusions from premises, and constructing representations of action. The semantics of these systems have been made rigorous, but are almost always imposed by their designers. Moreover, traditional models have often failed to explain how so much "reasoning" can get done so fast with so little hardware. Reactive-agent architectures have been proposed as an alternative to traditional AI, but to date theoretical foundations for this work have been less developed.

Situated-automata theory provides a semantic analysis of information processing that is intended to apply to all embedded control systems without requiring designer-conceived interpretations of machine state or computational models based on run-time inference. It is based on a direct mathematical model of how the states of natural processes, in the ways they unfold over time, reflect one another through intricate cross dependencies and correlations that give rise to semantically meaningful states. The theory brings the semantic precision associated with traditional logic-based AI to the analysis of systems that are not structured as conventional reasoning systems at all. Nor are systems that *do* seem to "reason" excluded from this style of analysis; they are simply a special case.

Note that none of this analysis is inconsistent with the construction of agents as symbolic systems; it simply makes explicit the constraints that must hold for their intended interpretation to be valid and provides methods for using symbolic characterizations as program specifications rather than as an implementation strategy.

The shift from the traditional AI view to the situated view brings us to an outlook reminiscent of early cybernetic feedback models, but with more semantic subtlety and sophistication (derived from traditional symbolic AI) in describing conditions being tracked and controlled by an agent. In this view, the fundamental phenomenon to be explained is not "reasoning" but the mutual constraint exhibited between parts of a physical system over time. The key lies in understanding how a process can naturally mirror in its states subtle conditions in its environment and how these mirroring states ripple out to overt actions that eventually achieve goals. The fundamental questions include how the enormous set of discernible conditions can be modeled and grasped, how computational elements can be arranged to preserve distinctions that matter for controlling the environment while perhaps blurring others, and how can this be done in real time with high reliability using relatively modest computational resources.

While the analytical approach presented here is very general in scope, its application to synthesis problems and to the design of particular systems remains quite challenging. In this paper we have attempted to sketch directions we regard as promising, primarily involving the use of stylized off-line symbolic reasoning to generate tractable run-time machinery with desired properties of information and control. Work remains to be done on the integration of automated learning techniques as well as the modeling and exploitation of statistical covariance in ways analogous to the discrete logic-based techniques presented here.

Acknowledgments

We derived a great deal of help and inspiration from our colleagues over the years of this project. Stanley Reifel built Flakey, an experimental mobile robot platform, and constantly challenged us to match in working software what we derived in elegant formulas. Sandy Wells brought a knowledge of computer vision, hardware, and hacking that was invaluable. Nathan Wilson implemented endless versions of and variations on Rex and wrote some crucial navigational code for Flakey. Stuart Shieber was a valuable adjunct to the group and implemented natural language modules for the robot programs. Fernando Pereira was an important influence on the early development of situated-automata theory. David Chapman spent some summers with us and helped make Rex a much better language, through both ideas and implementation. He also worked on Ruler and some of its precursors. We are generally indebted to and appreciative of our colleagues at the Artificial Intelligence Center of SRI International, at Stanford University's Center for the Study of Language and Information, and at Teleos Research. We gratefully acknowledge financial support from these institutions as well as from sponsors at the Defense Advance Research Projects Agency, the National Aeronautics and Space Administration, the Air Force Office of Scientific Research, General Motors Research, and FMC.

References

[1] A. R. Cassandra, L. P. Kaelbling, and M. L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, WA, 1994.

[2] J. Halpern and Y. Moses. Knowledge and common knowledge in an distributed environment. In *Proceedings of the Third ACM Conference on Principles of Distributed Computing*, pages 50–61, 1984. A revised version appears as IBM RJ 4421.

[3] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic*. Methuen and Company, London, 1968.

[4] L. P. Kaelbling. Rex: A symbolic language for the design and parallel implementation of embedded systems. In *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts, 1987.

[5] L. P. Kaelbling. Goals as parallel program specifications. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota, 1988.

[6] L. P. Kaelbling. Compiling operator descriptions into reactive strategies using goal regression. Technical report, Teleos Research, Palo Alto, California, 1991.

[7] L. P. Kaelbling and S. J. Rosenschein. Action and planning in embedded agents. *Robotics and Autonomous Systems*, 6(1):35–48, 1990. Also published in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, Pattie Maes, editor, The MIT Press/Elsevier, 1991.

[8] L. P. Kaelbling and N. J. Wilson. Rex programmer's manual. Technical Report 381R, Artificial Intelligence Center, SRI International, Menlo Park, California, 1988.

[9] W. S. Lovejoy. A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 28(1):47–65, 1991.

[10] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, 1969.

[11] R. C. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, editors, *Formal Theories of the Commonsense World*. Ablex Publishing Company, Norwood, New Jersey, 1985.

[12] A. Newell. The knowledge level. *Artificial Intelligence*, 18:87–127, 1982.

[13] S. J. Rosenschein. Plan synthesis: A logical perspective. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, British Columbia, 1981. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds., Morgan Kaufmann, 1990.

[14] S. J. Rosenschein. Formal theories of knowledge in AI and robotics. *New Generation Computing*, 3(4):345–357, 1985.

[15] S. J. Rosenschein. Synthesizing information-tracking automata from environment descriptions. In *Proceedings of Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, 1989.

[16] S. J. Rosenschein and L. P. Kaelbling. The synthesis of digital machines with provable epistemic properties. In J. Halpern, editor, *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge*, pages 83–98. Morgan Kaufmann, 1986. An updated version appears as Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, California.

[17] R. Waldinger. Achieving several goals simultaneously. In *Machine Intelligence 8*. Ellis Horwood Limited, Chichester, 1977. Reprinted in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate, eds., Morgan Kaufmann, 1990.