Action and Planning in Embedded Agents*

Leslie Pack Kaelbling
and Stanley J. Rosenschein
Teleos Research, 576 Middlefield Road, Palo Alto, CA 94301, USA

Embedded agents are computer systems that sense and act on their environments, monitoring complex dynamic conditions and affecting the environment in goal-directed ways. This paper briefly reviews the situated automata approach to agent design and explores issues of planning and action in the situated-automata framework.

Keywords: Intelligent agents; Reactive systems; Planning; Action; Situated-automata theory; Gapps; Embedded agents

Ms Kaelbling received a Philosophy A.B. from Stanford University in 1983 and is currently a Ph.D. candidate with the Department of Computer Science at that University. From 1982 to 1984 Ms Kaelbling acted as a Research and Teaching Assistance at Stanford and in 1984 she joined SRI International's Artificial Intelligence Center as a Computer Scientist, leaving to join Teleos Research as a Computer Scientist in 1988. Ms Kaelbling's research interests include artificial intelligence, machine learning, programming languages and robotics; her Ph.D. dissertation will address learning in embedded systems.

* This work was supported in part by the Air Force Office of Scientific Research under contract F49620-89-C-0055DEF and in part by the National Aeronautics and Space Administration under Cooperative Agreement NCC-2-494 through Stanford University subcontract PR-6359.

North-Holland
Robotics and Autonomous Systems 6 (1990) 35–48

1. The Design of Embedded Agents

Embedded agents are computer systems that sense and act on their environments, monitoring complex dynamic conditions and affecting the environment in goal-directed ways. Systems of this kind are extremely difficult to design and build, and without clear conceptual models and powerful programming tools, the complexities of the real world can quickly become overwhelming. In certain special cases, designs can be based on well-understood mathematical paradigms such as classical control theory. More typically, however, tractable models of this type are not available and alternative approaches must be used. One such alternative is the situated-automata framework, which models the relationship between embedded control systems and the external world in qualitative terms and provides a family of programming abstractions to aid the designer. This paper briefly reviews the situated-automata approach and then explores in greater detail one aspect of the approach, namely the design of the action-generating component of embedded agents.

1.1. The Situated-Automata Model

The theoretical foundations of the situated-automata approach are based on modeling the world

Dr. Rosenschein received his B.A. from Columbia University in 1971 and his Ph.D. in Computer and Information Sciences from the University of Pennsylvania in 1975. After a post-doctorate at the Courant Institute of Mathematics of New York University, a lectureship at the Technion in Haifa, Israel, and a research position at the Rand Corporation, Dr. Rosenschein joined the Artificial Intelligence Center at SRI International in 1980. He became Director of the Center in 1984 and held that post until 1988 at which time he left the Center to found Teleos Research, a research and development company working in artificial intelligence, robotics and advanced computing. Dr. Rosenschein's research has focused on theoretical issues in automated reasoning, knowledge representation, natural language processing, and robotics.

as a pair of interacting automata, one corresponding to the physical environment and the other to the embedded agent. Each has local state that varies as a function of signals projected from the other. The aim of the design process is to synthesize an agent, in the form of an embedded state machine, that causes the desired effects in the environment over time.

In applications of interest, it is often useful to describe the agent in terms of the information available about the environment and the goals the agent is pursuing. It is also desirable that these descriptions be expressed in language that refers to states of the environment rather than to specific internal data structures, at least during the early phases of design. Moreover, the inputs, outputs, and internal states of the state machine will be far too numerous to consider explicitly, which means the machine must be constructed out of a set of separate components acting together to generate complex patterns of behavior. These requirements highlight the need for compositional, high-level languages that compactly describe machine components in semantically meaningful terms.

Situated-automata theory provides a principled way of interpreting data values in the agent as encoding facts about the world expressed in some language whose semantics is clear to the designer. Interpretations of this sort would be of little use were it not also the case that whenever the data structure had a particular value, the condition denoted was guaranteed to hold in the environment. Such considerations motivate defining the semantics of data structures in terms of objective correlations with external reality. In this approach, a machine variable x is said to carry the information that p in world state s, written $s \models K(x, p)$, if for all world states in which x has the same value it does in s, the proposition p is true. The formal properties of this model and its usefulness for programming embedded systems have been described elsewhere [9,11,5,10].

Having established a theoretical basis for viewing a given signal or state in the agent as carrying information content by virtue of its objective correlation with the environment, one can consider languages in which this content might be expressed. In general there will be no single "best" language for expressing this information. For example, one language is the set of signals or states themselves. These can be regarded as a system of signs whose semantic interpretations are exactly the conditions with which they are correlated. However, the designer will typically wish to employ other, higher-level, languages during the design process. This theme will be expanded upon below in connection with goal-description languages.

1.2. Perception-Action Split

One way of structuring the design process for the cognitive ease of the designer is to separate the problem of acquiring information about the world from the problem of acting appropriately relative to that information. The former we shall label *perception* and the latter, *action*. In terms of the state-machine model, as shown in *Fig. 1*, the perception component corresponds to the update function and the initial state, whereas the action component corresponds to the output mapping.

The perception-action split in itself is entirely conceptual and may or may not be the basis for modularizing the actual system. Horizontal decompositions that cut across perception and action have been advocated by Brooks as a practical way of approaching agent design [2]. The horizontal approach allows the designer to consider simultaneously those limited aspects of perception and action needed to support specific behaviors. In this way, it discourages the pursuit of spurious generality that often inhibits practical progress in robotics.

These attractive features are counterbalanced, however, by the degree to which horizontal decomposition encourages linear thinking. In practice, the methodology of not separating the acquisition of information from its use tends to encourage the development of very specific behaviors rather than the identification of elements that can recombine freely to produce complex *patterns* of behavior. The alternative is a vertical strategy based on having separate system modules that

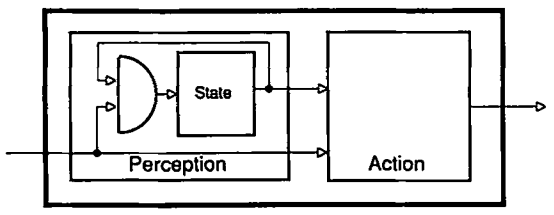

Fig. 1. Division between perception and action components.

recover broadly useful information from multiple sources and others that exploit it for multiple purposes. The inherent combinatorics of information extraction and behavior generation make the vertical approach attractive as a way of making efficient use of a programmer's effort.

The commitment to a decomposition based upon the perception-action split still leaves open the question of development strategy. One approach is to iteratively refine the perception-action pair, more or less in lockstep. The information objectively carried by an input signal or an internal state is relative to constraints on other parts of the system—including constraints on the action component. The more constrained the rest of the system, the more the designer can deduce about the world from a given internal signal or state, hence the more "information" it contains. As the designer refines his design, his model of the information available to the system and what the system will do in response becomes increasingly specific.

An alternative to iterative refinement, suitable in many practical design situations, is the strict divide-and-conquer strategy in which the design of the perception component is carried out in complete isolation from the development of the action component except for the specification of a common interface—the data structures that encode the information shared between the perception and action modules. Although there may be occasions when the designer needs to rely on some fact about what the agent will *do* in order to guarantee that a certain signal or state has the semantic content he intends, if these situations can be minimized or ignored, considerable simplification will result.

1.3. Goals

As we have seen, one way of semantically characterizing an agent's states is in terms of the information they embody. The perception component delivers information, and the action component maps this information to action. In many cases, however, it is more natural to describe actions as functions not only of information but of the goals the agent is pursuing at the moment [12].

Goals can be divided into two broad classes: static and dynamic. A static goal is a statement the agent's behavior is simply designed to make true. In reality, a static goal is nothing more than a specification, and as such the attribution of this "goal" to the agent is somewhat superfluous, although it may be of pragmatic use in helping the designer organize his conception of the agent's action strategy. Dynamic goals are another matter. The ability to attribute to the agent goals that change dynamically at run time opens the possibility of dramatically simplifying the designer's description of the agent's behavior.

Since we are committed to an information-based semantics for reactive systems, we seek an "objective" semantics of goals defined explicitly in informational terms. We can reformulate the notion of having a goal p as having the information that p implies a fixed top-level goal, called N for "Nirvana." Formally, we define a goal operator G as follows:

$$G(x, p) \equiv K(x, p \rightarrow N).$$

In this model, x has the goal p if x carries the information that p implies Nirvana. [1] This definition captures the notion of dynamic goals because p can be an indexical statement, such as "it is raining now," whose truth varies with time. Since this model defines goals explicitly in terms of information, the same formal tools used to study information can be applied to goals as well. In fact, under this definition, goals and information are dual concepts.

To see the duality of goals and information, consider a function f mapping values of one variable, a, to values of another variable, b. Under the information interpretation, such a function takes elements having more specific information into elements having less specific information. This is because functions generally introduce ambiguity by mapping distinct inputs to the same output. For example, if value u_1 at a is correlated with proposition p and value u_2 at a is correlated with q and if f maps both u_1 and u_2 to v at b, the value v is ambiguous as to whether it arose from u_1 or u_2, and hence the information it contains is the disjunctive information $p \vee q$, which is less specific than the information contained in either u_1 or u_2. Thus, functional mappings are a form of forgetting.

[1] We observe that under this definition *False* will always be a goal; in practice, however, we are only interested in non-trivial goals.

Under the goal interpretation, this picture is reversed. The analog to "forgetting" is committing to subgoals, which can be thought of as "forgetting" that there are other ways of achieving the condition. For instance, let the objective information at variable a be that the agent is hungry and that there is a sandwich in the right drawer and an apple in the left. If the application of a many-to-one function results in variable b's having a value compatible with the agent's being hungry and there being a sandwich in the right drawer and either an apple in the left drawer or not, we could describe this state of affairs by saying that variable b has lost the information that opening the left drawer would be a way of finding food. Alternatively, we could say that variable b had committed to the subgoal of opening the right drawer. The phenomena of forgetting and commitment are two sides of the same coin.

We can relate this observation to axioms describing information and goals. One of the formal properties satisfied by K is the deductive closure axiom, which can be written as follows:

$$K(x, p \rightarrow q) \rightarrow (K(x, p) \rightarrow K(x, q)).$$

The analogous axiom for goals is

$$K(x, p \rightarrow q) \rightarrow (G(x, q) \rightarrow G(x, p)).$$

This is precisely the subgoaling axiom. If the agent has q as a goal and carries the information that q is implied by some other, more specific, condition, p, the agent is justified in adopting p as a goal. The validity of this axiom can be established directly from the definition of G.

Given these two ways of viewing the semantics of data structures, we can revisit the state-machine model of agents introduced above. Rather than specify the action component of the machine as a function of one argument interepreted in purely "informational" terms, $f(i)$, it may be much more convenient for designers to define it as a function of two arguments, $f'(g, i)$ where the g argument is interpreted as representing the dynamic goals of the agent. Where does the g input come from? Clearly, it must ultimately be computed from the agent's current information state as well as its static goals, g_0. As such, it must be equivalent to some non-goal-dependent specification: $f(i) = f'(extract(i, g_0), i)$. Nevertheless, the decomposition into a goal-extraction module and a goal-directed action module may significantly ease the cognitive burden for the designer while leaving him secure in the knowledge that his design is semantically grounded.

1.4. Software Tools for Agent Design

Although it is conceptually important to have a formal understanding of the semantics of the data structures in an embedded agent, this understanding does not, directly, simplify the programmer's task. For this reason, it is necessary to design and implement software tools that are based on proper foundations and that make it easier to program embedded agents.

Rex [5,7] is a language that allows the programmer to use the full recursive power of Lisp at compile time to specify a synchronous digital circuit. The circuit model of computation facilitates semantic analysis in the situated-automata theory framework. However, Rex only provides, however, a low-level, operational language that is more akin to standard programming languages than to declarative AI languages. For this reason, we have designed and implemented a pair of declarative programming languages on top of the base provided by Rex. Ruler [10] is based on the "informational" semantics and is intended to be used to specify the perception component of an agent. Gapps [6] is based on the "goal" semantics and is intended to be used to specify the action component of an agent. In the rest of this paper, we will describe the Gapps language, its use in programming embedded agents, and a number of extensions that relate it to more traditional work in planning.

2. Gapps

In this section we describe Gapps, a language for specifying behaviors of computer agents that retains the advantage of declarative specification, but generates run-time programs that are reactive, do parallel actions, and carry out strategies made up of very low-level actions.

Gapps is intended to be used to specify the action component of an agent. The Gapps compiler takes as input a declarative specification of the agent's top-level goal and a set of goal-reduction rules, and transforms them into the description of a circuit that has the output of the percep-

tion component as its input, and the output of the agent as a whole as its output. The output of the agent may be divided into a number of separately controllable actions, so that we can independently specify procedures that allow an agent to move and talk at the same time. A sample action vector declaration is:

```
(declare-action-vector
    (left-wheel-velocity int)
    (right-wheel-velocity int)
    (speech string))
```

This states that the agent has three independently controllable effectors and declares the types of the output values that control them.

In the following sections, we shall present a formal description of Gapps and its goal evaluation algorithm, and explain how Gapps specifications can be instantiated as circuit descriptions.

2.1. Goals and Programs

The Gapps compiler maps a top-level goal and a set of goal-reduction rules into a program. In this section we shall clarify the concepts of goal, goal-reduction rule, and program.

There are three primitive goal types: goals of execution, achievement, and maintenance. Goals of execution are of the form $\mathrm{do}(a)$, with a specifying an instantaneous action that can be taken by the agent in the world—the agent's goal is simply to perform that action. If an agent has a goal of maintenance, notated $\mathrm{maint}(p)$, then if the proposition p is true, the agent should strive to maintain the truth of p for as long as it can. The goal $\mathrm{ach}(p)$ is a goal of achievement, for which the agent should try to bring about the truth of proposition p as soon as possible. The set of goals is made up of the primitive goal types, closed under the Boolean operators. The notions of achievement and maintenance are dual, so we have $\neg\mathrm{ach}(p) \equiv \mathrm{maint}(\neg p)$ and $\neg\mathrm{maint}(p) \equiv \mathrm{ach}(\neg p)$.

In order to characterize the correctness of programs with respect to the goals that specify them, we must have a notion of an action *leading to* a goal. Informally, an action a leads to a goal G (notated $a \rightsquigarrow G$) if it constitutes a correct step toward the satisfaction of the goal. For a goal of achievement, the action must be consistent with the goal condition's eventually being true; for a goal of maintenance, if the condition is already true, the action must imply that it will be true at the next instant of time. The *leads to* operator must also have the following formal properties:

$$a \rightsquigarrow \mathrm{do}(a)$$

$$(a \rightsquigarrow G) \wedge (a \rightsquigarrow G') \Rightarrow a \rightsquigarrow (G \wedge G')$$

$$(a \rightsquigarrow G) \vee (a \rightsquigarrow G') \Rightarrow a \rightsquigarrow (G \vee G')$$

$$\mathrm{cond}(p, a \rightsquigarrow G, a \rightsquigarrow G') \Rightarrow a \rightsquigarrow \mathrm{cond}(p, G, G')$$

$$(a \rightsquigarrow G) \wedge (G \rightarrow G') \Rightarrow a \rightsquigarrow G'.$$

This definition captures a weak intuition of what it means for an action to lead to a goal. The goal of doing an action is immediately satisfied by doing that action. If an action leads to each of two goals, it leads to their conjunction; similarly for disjunction and conditionals. The definition of *leads to* for goals of achievement may seem too weak—rather than saying that doing the action is consistent with achieving the goal, we would like somehow to say that the action actually constitutes *progress* toward the goal condition. Unfortunately, it is difficult to formalize this notion in a domain-independent way. In fact, any definition of *leads to* that satisfies this definition is compatible with the goal reduction algorithm used by Gapps, so the definition may be strengthened for a particular domain.

Goal reduction rules are of the form (defgoalr G G') and have the semantics that the goal G can be reduced to the goal G'; that is, that G' is a specialization of G, and therefore implies G. By the definition of "leads to", any action that leads to G' will also lead to G.

A program is a finite set of condition-action pairs, in which the condition is a run-time expression (actually a piece of Rex circuitry with a Boolean-valued output) and an action is a vector of run-time expressions, one corresponding to each primitive output field. These actions are run-time mappings from the perceptual inputs into output values, and can be viewed as strategies, in which the particular output to be generated depends on the external state of the world via the internal state of the agent. Allowing the actions to be entire strategies is very flexible, but makes it impossible to enumerate the possible values of an output field. In order to specify a program that controls only the speech field of an action vector,

we need to be able to describe a program that requires the speech field to have a certain value, but makes no constraints on the values of the other fields. One way to do this would be to enumerate a set of action vectors with the specified speech value, each of which has different values for the other action vector components. Instead of doing this, we allow elements of an action vector to contain the value $\emptyset$, which stands for all possible instantiations of that field.

A program Π, consisting of the condition-action pairs $\{\langle c_1, a_1\rangle, \ldots, \langle c_n, a_n\rangle\}$, is said to *weakly satisfy* a goal G if, for every condition c_i, if that condition is true, the corresponding action a_i leads to G. That is,

Π weakly satisfies $G \Leftrightarrow \forall i. c_i \rightarrow (a_i \rightsquigarrow G)$.

Note that the conditions in a program need not be exhaustive—satisfaction does not require that there be an action that leads to the goal in every situation, since this is impossible in general. We will refer to the class of situations in which a program does specify an action as the *domain* of the program. We define the domain of Π as

$$\mathrm{dom}(\Pi) = \bigvee_i c_i.$$

A goal G is *strongly satisfied* by program Π if it is weakly satisfied by Π and $\mathrm{dom}(\Pi) = true$; that is, if for every situation, Π supplies an action that leads to G. The conditions in a program need not be mutually exclusive. When more than one condition of a program is true, the action associated with each of them leads to the goal, and an execution of the program may choose among these actions nondeterministically.

Given the non-deterministic execution model, we can give programs a declarative semantics, as well. A program $\Pi = \{\langle c_1, a_1\rangle, \ldots, \langle c_n, a_n\rangle\}$, can be thought of has having the logical interpretation

$$\left(\bigwedge_i (a_i \rightarrow c_i) \wedge \bigvee_i a_i\right) \vee \neg \bigvee_i c_i.$$

Either the domain of the program is false (the second clause) or there is some action that is executed and the condition associated with that action is true.

2.2. Recursive Goal Evaluation Procedure

Gapps is implemented on top of Rex, and makes use of constructs from the Rex language to provide perceptual tests. There is not room here to describe the details of the Rex language, so we refer the interested reader to other papers [5,7]. Gapps programs are made up of a set of goal reduction rules and a top-level goal-expression. The general form of a goal-reduction rule is

(defgoalr *goal-pat goal-expr*),

where

goal-pat:: = (ach *pat rex-params*)
(maint *pat rex-params*)

goal-expr:: = (do *index rex-expr*)
(and *goal-expr goal-expr*)
(or *goal-expr goal-expr*)
(not *goal-expr*)
(if *rex-expr goal-expr goal-expr*)
(ach *pat rex-expr*)
(maint *pat rex-expr*)

index is a keyword, *pat* is a compile-time pattern with unifiable variables, *rex-expr* is a Rex expression specifying a run-time function of input variables, and *rex-params* is a structure of variables that becomes bound to the result of a *rex-expr*. The details of these constructs will be discussed in the following sections.

The Gapps compiler is an implementation of an evaluation function that maps goal expressions into programs, using a set of goal reduction rules supplied by the programmer. In this section we shall present the evaluation procedure; we have shown that it is correct; that is, that given a goal G and a set of reduction rules Γ, eval(G, Γ) weakly satisfies G.

Given a reduction-rule set Gamma, we define the evaluation procedure as follows:

```
define eval(G)

case first(G)
  do  : make-primitive-program(second(G),
        third(G))
  and : conjoin-programs(eval(second(G)),
        eval(third(G)))
  or  : disjoin-programs(eval(second(G)),
        eval(third(G)))
  not : eval (negate-goal-expr(second(G)))
  if  : disjoin-programs
          (conjoin-cond(second(G),
           eval(third(G))),
```

```
        conjoin-cond(negate-cond(G),
        eval(fourth(G))))
maint,
ach : for all R in Gamma such that
      match(G,head (R))
      disjoin-programs(eval(body(R))
```

We shall now consider each of these cases in turn.

Do

The function make-primitive-program takes an index and a Rex expression and returns a program. The index indicates which of the fields of the action vector is being assigned, and the Rex expression denotes a function from the input to values for that action field. It is formally defined as

$$\text{make-primitive-program}(i, \textit{rex-expr}) = \{\langle \textit{true}, \langle \emptyset, \ldots, \textit{rex-expr}, \ldots, \emptyset \rangle \rangle\},$$

with the *rex-expr* in the ith component of the action vector. This program allows any action so long as component i of the action is the strategy described by *rex-expr*.

And

Programs are conjoined by taking the cross-product of their condition-action pairs and merging each of elements of the cross-product together. In conjoining two programs, the merged action vector is associated with the conjunction of the conditions of the original pairs, together with the condition that the two actions are mergeable. The conjunction procedure simply finds the pairs in each program that share an action and conjoins their conditions. We can define the operation formally as

$$\text{conjoin-programs}(\Pi', \Pi'') = \left\{ \left\langle \left(c_i' \wedge c_j'' \wedge \text{mergeable}(a_i', a_j'')\right), \text{merge}(a_i', a_j'') \right\rangle \right\}$$

for $1 \le i \le m$, $1 \le j \le n$ where

$$\Pi' = \{\langle c_1', a_1' \rangle, \ldots, \langle c_m', a_m' \rangle\}$$

$$\Pi'' = \{\langle c_1'', a_1'' \rangle, \ldots, \langle c_n'', a_n'' \rangle\}.$$

The conjunction operation preserves the declarative semantics of programs; that is, the semantic interpretation of the conjoined program is implied by the conjunction of the semantic interpretations of the individual programs.

Two action vectors are *mergeable* if, for each component, at least one of them is unspecified or they are equal.

$$\text{mergeable}(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_n \rangle) \equiv \forall i. (a_i = \emptyset) \vee (b_i = \emptyset) \vee (a_i = b_i).$$

If either component is unspecified, the test can be completed at compile time and no additional circuitry is generated. Otherwise, an equality test is conjoined in with the conditions to be tested at run time.

Action vectors are merged at the component level, taking the defined element if one is available. If the vectors are unequally defined on a component, the result is undefined:

$$\text{merge}(\langle a_1, \ldots, a_n \rangle, \langle b_1, \ldots, b_n \rangle) = \langle c_1, \ldots, c_n \rangle, \text{ where}$$

$$c_i = \begin{cases} a_i & \text{if } b_i = \emptyset \text{ or } b_i = a_i \\ b_i & \text{if } a_i = \emptyset \\ \perp & \text{otherwise.} \end{cases}$$

The merger of two action vectors results in an action vector that allows the intersection of the actions allowed by the original ones.

Or

The disjunction of two programs is simply the union of their sets of condition-action pairs. Stated formally,

$$\text{disjoin-programs}(\Pi', \Pi'') = \Pi' \cup \Pi''.$$

Not

In Gapps, negation is driven into an expression as far as possible, using DeMorgan's laws and the duality of ach and maint, until the only expressions containing not are those of the form (ach (not *pat*)), (maint (not *pat*)), and (not (do *index rex-expr*)). In the first two cases, there must be explicit reduction rules for the goal; in the last case we simply return the empty program. The handling of negation could be much stronger if we provided for the enumeration of all possible values of any action vector component and required them to be known constants at compile time. Then (not (do left-velocity 6)) would be the same as $\vee_{i \neq 6}$ make-primitive-program (left-velocity, i); that

is, license to go at any velocity but 6. As we noted before, these limitations are too severe for use in controlling a complex agent that has large numbers of possible outputs.

The procedure negate-goal-expression rewrites goal expressions as follows:

(not (and G_1 G_2)) ⇒ (or (not G_1) (not G_2))

(not (or G_1 G_2)) ⇒ (and (not G_1) (not G_2))

(not (not G)) ⇒ G

(not (if c G_1 G_2)) ⇒ (if c (not G_1) (not G_2))

(not (ach p)) ⇒ (maint (not p))

(not (maint p)) ⇒ (ach (not p))

If

The evaluation procedure for conditional programs hinges on the definition of the conditional operator cond(p, q, r) as $(p \wedge q) \vee (\neg p \wedge r)$. The procedure for conjoining a condition and a program is defined as follows:

conjoin-cond(p, Π)

$$= \{\langle p \wedge c_1, a_1\rangle, \ldots, \langle p \wedge c_n, a_n\rangle\}.$$

Thus,

disjoin-programs(conjoin-cond(p, Π'), conjoin-cond($\neg p$, Π''))

$$= \{\langle p \wedge c'_1, a'_1\rangle, \ldots, \langle p \wedge c'_n, a'_n\rangle, \langle \neg p \wedge c''_1, a''_1\rangle, \ldots, \langle \neg p \wedge c''_m, a''_m\rangle\}.$$

Ach and Maint

Goals of maintenance and achievement are evaluated by disjoining the results of all applicable reduction rules in the rulebase Γ. A reduction rule whose head is the expression (ach pat_1 *rex-params*) matches the goal expression (ach pat_2 *rex-expr*) if pat_1 and pat_2 can be unified in the current binding environment. The patterns are s-expressions with compile-time variables that are marked by a leading ?. The Rex expression and parameter arguments may be omitted if they are null. The binding environment consists of other bindings of compile-time variables within the goal expression being evaluated. Thus, when evaluating the (ach (go ?p)) subgoal of the goal (and (ach (drive ?q ?p)) (ach (go ?p))), we may already have a binding for ?p. As in Prolog, evaluation of this goal will backtrack through all possible bindings of ?p and ?q.

Once a pattern has been matched, Gapps sets up a new compile-time binding environment for evaluating the body of the rule. This is necessary in case variables in the body are bound by the invocation, as in

```
(defgoalr (ach (at ?p) [dist-err angle-err])
  (if (not-facing ?p angle-err)
      (ach (facing ?p) angle-err)
      (ach (moved-toward ?p) dist-err))).
```

In the rule above, (at ?p) is a pattern, ?p is a compile-time parameter, dist-err and angle-err are Rex variables, and (not-facing ?p angle-err) will be a Rex expression once a binding is substituted for ?p. A possible invocation of this rule would be:

```
(ach (at (office-of stan)) [*distance-eps* 10]).
```

Gapps also creates a new Rex-variable binding environment when the rule is invoked, binding the Rex variables in the head to the evaluated Rex expressions in the invocation. These variables may appear in Rex expressions in the body of the rule. Note that compile-time variables may also be used in Rex expressions, in order to choose at compile time from among a class of available run-time functions.

2.3. Generating a Circuit

Once a goal expression has been evaluated, yielding a program, a circuit similar to the one shown in *Fig. 2*, that instantiates the program is generated. [2] Because any action whose associated condition is true is sufficient for correctness, the conditions are tested in an arbitrary order that is chosen at compile time. The output of the circuit is the action corresponding to the first condition that is true. If no condition is satisfied, an error action is output to signal the programmer that he has made an error. If, at the final stage of circuit generation, there are still $\emptyset$ components in an action vector, they must be instantiated with an arbitrary value. The inputs to the circuit are com-

[2] An equivalent, but more confusing, circuit with $\log(n)$ depth can be generated for improved performance on parallel machines.

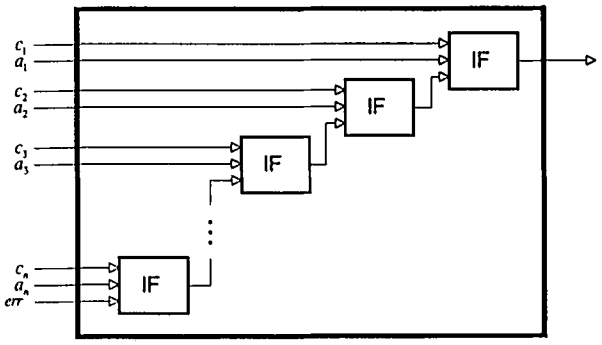

Fig. 2. Circuit generated from Gapps program.

puted by the Rex expressions supplied in the if and do forms. The outputs of the circuit are used to control the agent.

2.4. Reducing Conjunctive Goal Expressions

Conjunctive goal expressions can have two forms: (*ach-or-maint* (and p_1 p_2)) and (and (*ach-or-maint* p_1) (*ach-or-maint* p_2)). Because of the properties of maintainance, the goals (maint (and p_1 p_2)) and (and (maint p_1) (maint p_2)) are semantically equivalent. This is not true, however, for goals of achievement. The goal (ach (and p_1 p_2)) requires that p_1 and p_2 be true simultaneously, whereas the goal (and (ach G_1) (ach G_2)) requires only that they each be true at some time in the future.

Goals of the form (*ach-or-maint* (and p_1 p_2)) can only be reduced using reduction rules whose pattern matches this conjunctive pattern. Goals of the form (and (*ach-or-maint* p_1) (*ach-or-maint* p_2)) can be reduced in two ways: using the standard evaluation procedure for conjunctive goals and using special reduction rules. It is often the case that an effective behavior for achieving G_1 and achieving G_2 cannot be generated simply by conjoining programs that achieve G_1 and G_2 individually. A program for the goal (and (ach have hammer) (ach have saw)) will almost certainly be incomplete when the two tools are in different rooms, because there will be no actions available that are consistent with the standard programs for achieving each of the subgoals. Because of this, we allow reduction rules of the form (defgoalr (and (*ach-or-maint* pat_1 *rex-params*$_1$) (*ach-or-maint* pat_2 *rex-params*$_2$)) *goal-expr*) so that special behaviors can be generated in the face of a conjunctive goal.

Following is an example that illustrates both kinds of conjunctive goals. At the top level, the goal is to have the hammer and saw simultaneously, but this reduces to conjunctions of ach and maint goals.

```
(defgoalr (ach (and (have hammer) (have saw))
  (if (have hammer)
    (and (maint have hammer)
         (ach have saw))
    (if (have saw)
      (and (maint have saw)
           (ach have hammer))
      (if (closer-than hammer saw)
        (ach have hammer)
        (ach have saw)))))
```

The agent will pursue the closer object until he has it, then pursue the second while maintaining posession of the first. We might need a similar rule for reducing the conjunctions of goals of achievement and maintenance. Instead of the specific rule above, we could write a more generic sequencing rule, like the following:

```
(defgoalr (ach (and ?g1 ?g2)
           [g1-params g2-params])
  (if (holds ?g1 g1-params)
    (and (maint ?g1 g1-params)
         (ach ?g2 g2-params))
    (if (holds ?g2 g2-params)
      (and (maint ?g2 g2-params)
           (ach ?g1 g1-params))
      (if (better-to-pursue ?g1 g1-params
                            ?g2 g2-params)
        (ach ?g1 g1-params)
        (ach ?g2 g2-params))))).
```

The generic form of the rule assumes that there is a Rex function, holds, that takes a compile-time parameter and generates a circuit that tests to see whether the predicate encoded by the compile-time parameter and the run-time variables is true in the world.

2.5. Prioritized Goal Lists

It is often convenient to be able to specify a prioritized list of goals. In Gapps, we can do this

with a goal expression of the form (prio *goal-expr*$_1$ ... *goal-expr*$_n$). The semantics of this is

$$\text{cond}(\text{dom}(\Pi_1), \Pi_1, \text{cond}(\text{dom}(\Pi_2), \Pi_2, \ldots, \text{cond}(\text{dom}(\Pi_{n-1}), \Pi_{n-1}, \Pi_n)\ldots)),$$

where $\Pi_i = \text{eval}(\textit{goal-expr}_i)$. The domain of a program (true in a situation if the program has an applicable action in that situation) is the disjunction of the conditions in the program. A program for a prio goal executes the first program, unless it has no applicable action, in which case it executes the second program, and so on. At circuit-generation time, this construct can be implemented simply by concatenating the programs in priority order, and executing the first action whose corresponding condition is satisfied.

An example of the use of the prio construct comes about when there is more than one way of achieving a particular goal and one is preferable to the other for some reason, but is not always applicable. We might have the rule

```
(defgoalr (ach in-room r)
  (prio (ach follow-planned-route-to r)
        (ach use-local-navigation-to r))).
```

This rule states that the agent should travel to rooms by following planned paths, but if for some reason it is impossible to do that, it should do so through local navigation. The same effect could be achieved with an if expression, but this rule does not require the higher-level construct to know the exact conditions under which the higher-priority goal will fail.

2.6. Prioritized Conjunctions

An interesting special case of a prioritized set of goals is a prioritized conjunction of goals, in which the most preferred goal is the entire conjunction, and the less preferred goals are the conjunctions of shorter and shorter prefixes of the goal sequence. We define (prio-and G_1 G_2 ... G_n) to be

$$(\text{prio } (\text{and } G_1\ G_2 \ldots G_n) \; (\text{and } G_1\ G_2 \ldots G_{n-1}) \ldots \; (\text{and } G_1\ G_2) \; G_1).$$

Isaac Asimov's three laws of robotics [1] are a well-known example of this type of goal structure. As another example, consider a robot that can talk and push blocks. It has as its top-level goal

```
(prio-and (maint not-crashed)
          (ach (in block1 room3))
          (maint humans-not-bothered)).
```

It also has rules that say that any action with the null string in the talking field will maintain humans-not-bothered; that (in ?x ?y) can be achieved by pushing ?x or by asking a human to pick it up and move it; and that any action that keeps the robot from coming into contact with a wall will maintain not-crashed. As long as the robot can push the block, it can satisfy all three conditions. If, however, the block is in a corner, getting in a position to push it would require sharing space with a wall, thus violating the first subgoal. The most preferred goal cannot be achieved, so we consider the next-most-preferred goal, obtained by dropping the last condition from the conjunction. Since it is now allowed to bother humans, the robot can satisfy its goal by asking someone to move the block for it. As soon as the human complies, moving the block out of the corner, the robot will automatically revert to its former pushing behavior. This is a convenient high-level construct for programming flexible reactive behavior without the need for the programmer to explicitly envision every combination of conditions in the world. It is important to remember that all of the symbolic manipulation of the goals happens at compile time; at run time, the agent simply executes the action associated with the first condition that evaluates to true.

3. Extending Gapps

Gapps is an appropriate language for specifying action maps that can be hard-wired at the compile time of the agent. In this section, we will consider ways of extending and augmenting Gapps to do exhaustive planning at compile time, to do run-time planning, and to do run-time goal reduction.

3.1. Universal Planning with Goal-Reduction Schema

Schoppers [13] has introduced the notion of a *universal plan.* A universal plan is a function that, for a given goal, maps every possible input situation of the agent into an action that leads to (in an informal sense) that goal. The program resulting from the Gapps-evaluation of a goal can be thought of as a universal plan, mapping situations to actions in service of the top-level goal.

Schoppers' approach differs from Gapps in that the user specifies the capabilities of the agent in an operator-description language. This language allows the user to specify a set of atomic capabilities of the agent, called operators, and the expected effect that executing each of the operators will have on the world, depending possibly on the state of the world in which the operator was executed.

Another way to characterize operators is through the use of a *regression function* [8]. The relation $q = \text{regress}(\alpha, p)$ holds if, whenever q holds in the world, the agent's performing action α will cause p to hold in the world as a result. In general, the regression function will return the weakest such q. Regression is usually used to look backwards from a goal-situation p; the proposition q describes a set of situations that are only one "step" or operator application away from the set of situations satsifying p. We know that if the agent can get to a situation satisfying q, it can easily get to a situation satisfying p.

The following schematic Gapps rule allows it to do the exhaustive backward-chaining search that is typically done by a planner, in order to construct a universal plan. The Gapps compiler must be augmented slightly by giving it a depth-bound for its backward chaining, because this rule would, by default, cause infinite backward chaining.

```
(defgoalr (ach (before ?p ?q))
  (if (holds ?q)
    fail
    (if (holds ?p)
      (do anything)
      (if (holds (regress ?a ?p))
        (do ?a)
        (ach (before (regress ?a ?p)
                     (regress ?a ?q)))))))
```

The reduction rule is for goals of the form (ach (before ?p ?q)); that is, the goal is to achieve some condition ?p before some other condition ?q obtains. This form of achievement goal is, we think, typical—it is rare that an agent has a goal of achieving something no matter how long it takes. The rule works as follows: if ?q is true in the world, the agent fails; if ?p holds in the world, then the agent can do anything because it has achieved its goal; otherwise, if, for any action ?a, (regress ?a ?p) holds (that is, performing action ?a will cause ?p to hold next time) then this goal reduces to the goal (do ?a); finally, this goal can be reduced to achieving, for any action ?a, (before (regress ?a ?p) (regress ?a ?q). The final reduction says that it is good for the agent to get into a state from which action ?a achieves the goal ?p before the agent gets into a state from which action ?a achieves the releasing condition ?q, because once that has been done, all the agent must do is do action ?a.

Consider the application of this process to the standard 3-block blocks-world problem. The actions are named atoms, like *pab*, which signifies "put a on b." The world is described by predicates like *ca*, which signifies "clear a" and *obt*, which signifies "on b table." An additional predicate, $time(i)$, is true if the time on some global clock, which starts at 0, is i. We will use the abbreviation t_i to stand for $time(i)$. Given the goal (ach (before (and oab obc) (time 2))), the evaluation procedure returns a program that is described propositionally as follows:

$$\{\langle(\neg t_2 \wedge obc \wedge ca \wedge cb),\ pab\rangle,$$
$$\langle(\neg t_2 \wedge \neg t_1 \wedge obc \wedge ca \wedge cb),\ pat\rangle,$$
$$\langle(\neg t_2 \wedge \neg t_1 \wedge obc \wedge oab \wedge ca),\ pat\rangle,$$
$$\langle(\neg t_2 \wedge \neg t_1 \wedge ca \wedge cb \wedge cc),\ pbc\rangle,$$
$$\langle(\neg t_2 \wedge \neg t_1 \wedge oba \wedge cb \wedge cc),\ pbc\rangle\}.$$

According to this program, if b is on c, a and b are clear, and it is not time 2, then the agent can put a on b; otherwise, if it is neither time 1 nor time 2, the agent can do a variety of other things. For instance, if b is on c and a and b are clear, the agent can put a on the table. This illustrates the generality of the program. Because it is not yet time 1, it is acceptable to undo progress (we might have some other reason for wanting to do this), because there is time to put a back on b before

time 2. Notice that this program is not complete. There are situations for which it has no action, because there are block configurations that cannot be made to satisfy the goal in two actions. Notice also that, because this is a program of the standard form used by Gapps, it can be conjoined in with programs arising from other goals, such as global maintenance goals. Its generality, in allowing any sequence of actions that achieves the first condition before the second, makes it more likely that conjoining it in with a program expressing some other constraint will result in a non-null program.

3.2. Working in Parallel with an Anytime Planner

When the size of the state space is so large that doing exhaustive planning at compile time is impractical, it is possible to solve problems described as planning problems by integrating a run-time planning system with the Gapps framework.

We can express the planning process as an incremental computation, one step of which is done on each tick. On each tick the process generates an output, but it may be one that means "I don't have an answer yet." After some number of ticks, depending on the size of the planning problem, the planner will generate a real result. This result could be cached and executed as in a traditional system, or the agent could just take the first action and wait for the planner to generate a new plan.

Because time may have passed since the planner began its task, we must take care that the plan it generates is appropriate for the situation the agent finds itself in when the planner is finished. This can be guaranteed if the planner monitors the conditions in the world upon which the correctness of its plan depends. If any of these conditions becomes false, the planner can begin again. This behavior will be correct, though not always optimal. In the worst case, the planner will continuously emit the "I don't know" output and the agent will react reflexively to its environment without the benefit of a plan.

The kind of planner discussed above is a degenerate form of an anytime algorithm [3]. An anytime algorithm always has an answer, but the answer improves over time. In the example given above, the answer is useless for a while, then improves dramatically in one step. It might be useful to have planning algorithms that improve more gradually. Such algorithms exist for certain kinds of path planning, for instance, in which some path is returned at the beginning, but the algorithm works to make the path shorter or more efficient. There is still a difficult decision to be made, however, about whether to take the first step of a plan that is known to be non-optimal or to spend more time planning. For many everyday activities, optimality is not crucial, and it will be sufficient to act on the basis of a simple plan, if a plan is required at all.

From the perspective of Gapps, the anytime planner is just a perceptual process that has state. It is "perceiving" conditions of the form: "the world is in a state such that if I do action α followed by action β, followed by action γ, my goal will be achieved." The following Gapps program makes use of such a planner, but also has the potential for reacting to emergency situations:

```
(defgoalr (ach (in room) [r t])
  (if (know-plan-for-getting-to-room r t)
    (ach execute-first-step
         (plan-for-getting-to-room r t))
    (if (time-is-critical-for-getting-to-room r t)
      (ach drive-in-the-direction-of-room r)
      (maint sit-still)))).
```

If the agent has the goal of being in room r at time t, and he knows a plan for getting there, then he should execute the first step of that plan; otherwise, if it looks like time is running out, the agent should do the best action he can think of at the moment; if there is no problem with time, his best course of action is to sit still and wait until the perception component has produced a plan. These issues of combining planning and reactive action are explored more fully by Kaelbling [4].

3.3. Run-Time Goals

So far, we have only addressed the case in which the agent's top-level goal is specified at compile time. It will often be the case that it is useful to think of the agent as acquiring goals at run time.

3.3.1. Dispatching

The simplest case of responding to run-time goals is to consider them to be another type of

perceived information and write goal-reduction rules that are conditional on the given goal. As an example of this, an agent could be given the static compile-time goal of following orders and reduction rules of the following form:

```
(defgoalr (maint follow-orders)
  (if (current-request-pending)
    (ach goal-encoded-by
         (perceived-command))
    (do twiddle-thumbs)))
(defgoalr (ach goal-encoded-by params)
  (if (move-command params)
    (ach do-move-command
         (get-destination params))
    (if (stop-command params)
      (ach stopped)
      ...))).
```

The agent will carry out requests as it perceives them by dispatching to the right goal-reductions based on the nature of the request. This method is sufficient for many cases, but requires the run-time goals to be of a few limited types, because the different types must be tested for and dispatched to directly.

3.3.2. Run Time Goal Reduction

An alternative to explicit dispatching on the types of goals is to interpret Gapps-style goal-reduction rules at run time. An interpreter for Gapps is very similar to the evaluation procedure, except that the result at each step is a set of possible actions, rather than a set of condition-action pairs. This is because the interpretation is taking place at run time, which allows all of the conditions to be evaluated during the interpretation process, rather than combined into a program that is to be evaluated later. Any action can be chosen from the set resulting from interpreting the top-level goal in the current situation.

Given a reduction-rule set Gamma, we define the interpretation procedure as follows:

```
define interp(G)

case first(G)
  do  : make-action-set(second(G),
        rex-eval(third(G)))
  and : conjoin-action-sets(interp(second(G)),
        interp(third(G)))
  or  : disjoin-action-sets(interp(second(G)),
        interp(third(G)))
  not : interp(negate-goal-expr(second(G)))
  if  : if rex-eval(second(G)) then
          interp(third(G)) else
          interp(fourth(G))
  maint,
  ach : for all R in Gamma such that
        match(G,head(R))
        disjoin-action-sets(interp(body(R))
```

The function make-action-vector takes an index and a value and returns the singleton set containing the action vector with the field specified by the index set to the indicated value. That is,

$$\text{make-action-vector}(i, v) = \{\langle \emptyset, \ldots, v, \ldots, \emptyset \rangle\}.$$

The value is calculated by evaluating, in the current state of the world, the Rex expression specifying the primitive action. Using the functions mergeable and merge described in Section 2.2, the conjunction of action sets can be defined as

$$\text{conjoin-action-sets}(A', A'') = \{\text{merge}(a_i', a_j'') \mid \text{mergeable}(a_i', a_j'')\}$$

for $1 \le i \le m$, $1 \le j \le n$ where

$$A' = \{a_1', \ldots, a_m'\}$$

$$A'' = \{a_1'', \ldots, a_n''\}.$$

The disjunction of two action sets is simply the union of the sets:

$$\text{disjoin-action-sets}(A', A'') = A' \cup A''.$$

The crucial difference between the interpretation procedure and the evaluation procedure is in the if case. When the interpreter encounters an if goal, it can simply test the condition in the current state of the world and go on to interpret the subgoal corresponding to the result of the test. This obviates the need for manipulating formal descriptions of conditions during the goal-interpretation process.

If the rule set is fixed at compile time and is not recursive, interpretation can be done by a fixed circuit (written, perhaps, in Rex) whose depth is equal to the length of the maximum-length chain of rules in the rule set. If the rule set is recursive, a depth bound will have to be imposed in order to guarantee real-time response. Another possiblity would be to make this into an anytime algorithm by using iterative-deepening search over

the course of a number of ticks, and being careful that conditions that have already been evaluated do not change their values during the search process.

If the agent acquires goal reduction rules at run time, perhaps through learning, then the interpretation process can by carried out by general-purpose goal-reduction machinery. It can either be done in real time by a fixed circuit or over time by an anytime search procedure. If interpretation is to happen in real time, there must be a limit on the number of reduction rules that can be applied, in order to make the circuitry be of fixed size.

4. Conclusions

The Gapps goal-reduction formalism provides a flexible, declarative method for describing the action component of agents that must operate in real-time in dynamic worlds. It has a formal semantic grounding and has been implemented and used in a variety of robotic applications. In addition, it can be extended in a number of ways for use in domains with different types of complexity.

References

[1] Isaac Asimov, *I, Robot* (Fawcett Crest, New York, 1950).

[2] Rodney A. Brooks, A robust layered control system for a mobile robot, Technical Report AIM-864, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts (1985).

[3] Thomas Dean and Mark Boddy, An analysis of time-dependent planning, in *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota (1988).

[4] Leslie Pack Kaelbling, An architecture for intelligent reactive systems, In Michael P. Georgeff and Amy L. Lansky (eds), *Reasoning About Actions and Plans*, (Morgan Kaufmann, 1987) 395–410.

[5] Leslie Pack 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).

[6] Leslie Pack Kaelbling, Goals as parallel program specifications, in *Proceedings of the Seventh National Conference on Artificial Intelligence*, Minneapolis-St. Paul, Minnesota (1988).

[7] Leslie Pack Kaelbling and Nathan J. Wilson, Rex programmer's manual, Technical Report 381R, Artificial Intelligence Center, SRI International, Menlo Park, California (1988).

[8] Stanley J. Rosenschein, Plan synthesis: A logical perspective, in *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, Vancouver, British Columbia (1981).

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

[10] Stanley J. Rosenschein, Synthesizing information-tracking automata from environment descriptions, in *Proceedings of Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada (1989).

[11] Stanley J. Rosenschein and Leslie Pack Kaelbling, The synthesis of digital machines with provable epistemic properties, in Joseph Halpern (ed.) *Proceedings of the Conference on Theoretical Aspects of Reasoning About Knowledge* (Morgan Kaufmann, 1986) 83–98. An updated version appears as Technical Note 412, Artificial Intelligence Center, SRI International, Menlo Park, California.

[12] Stanley J. Rosenschein and Leslie Pack Kaelbling, Integrating planning and reactive control, in *Proceedings of NASA/JPL Conference on Space Telerobotics*, Pasadena, California (1989).

[13] Marcel J. Schoppers, Universal plans for reactive robots in unpredictable environments, in *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, vol. 2, Milan (Morgan Kaufmann, 1987) 1039–1046.