From: AAAI-88 Proceedings. Copyright ©1988, AAAI (www.aaai.org). All rights reserved.

# Goals as Parallel Program Specifications*

**Leslie Pack Kaelbling**
Artificial Intelligence Center
SRI International
and Center for the Study of Language and Information
Stanford University

## Abstract

Classical planning is inappropriate for generating actions in a dynamic world. This paper presents a formalism, called Gapps, that allows a programmer to specify an agent's behavior using symbolic goal-reduction rules that are compiled into an efficient parallel program. Gapps is designed for use in domains that require real-time response, that cannot be completely characterized by operator descriptions, and that allow multiple actions to be carried out in parallel.

## 1 Introduction

It has been standard practice in the field of artificial intelligence to use planning as the method of action selection in such computer agents as robots. Classical planning consists of encoding the domain in terms of atomic actions, their preconditions, and their effects in the world, and then, given an initial situation and a goal situation, searching through the space of operator sequences until one is found that will transform the initial state into the goal state. This method of action generation is attractive for two reasons. First, the style of programming is highly declarative, making it easy to modify incrementally the characterization of the domain. Second, the method is quite general, allowing the agent to solve a wide range of problems in its domain.

Unfortunately, many properties of classical planning make it inappropriate for use in real domains in which time is critical, the world is unpredictable, actions are fine-grained, and an agent can perform many actions in parallel. Moreover, planning is undecidable in the general case, and highly computationally intractable, even in its simpler forms [Chapman, 1987]. When an agent must generate prompt responses to events in its environment, it will not, in general, have enough time to devote to planning. The standard model of planning requires that the effects of actions be completely known at plan-time, but this is an unrealistic assumption for most real domains. A contributing factor to this lack of knowledge about actions is that in many cases the actions that the agent must reason about in the process of plan formation are at a very low level, more like "set the left wheel velocity to 200" than "go through door4." Finally, planning seldom recognizes that some agents can perform different actions, such as moving and talking, in parallel.

*This work was supported in part by a gift from the System Development Foundation and in part by DARPA and NASA under NASA grant PR5671 (SRI Project 4099).

These limitations of classical planning have been known for some time, and many researchers have sought to develop new action-generation methods that surmount them. Systems that interleave planning and execution [Wilkins, 1985] allow for the failure of actions and make it possible to take unanticipated events into account. They still use fundamentally intractable algorithms for the planning phases, however. Georgeff and Lansky's *reactive planning* [Georgeff and Lansky, 1987] is really not planning at all, but run-time interpretation of highly conditional user-specified plans. Schoppers [Schoppers, 1987] has proposed a system in which planning is carried out automatically at programming time, generating a tree of plans to achieve a goal from all possible initial situations, thus affording a high degree of robustness and the ability to recover from unexpected events. This approach is interesting, but the plans generated are too large for use in practical domains. Lansky [Lansky, 1987] has done interesting work in generating synchronized concurrent plans that uses the structure of the domain to make the reasoning process more tractable, but her approach still relies on complete information and plans completely in advance of acting.

In this paper 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 does not provide for classical run-time planning, but we find this acceptable because we believe that the vast majority of the activity of any agent is routine and requires none of the sophistication of general planning systems [Agre and Chapman, 1987]. In the final section we will explore methods for integrating classical planning into Gapps programs.

## 2 Gapps

Gapps is based on a model of computation in which an agent is seen to perform a finite transduction from a stream of input into a stream of output. A program, in this model, is a function that maps an input and the current value of the state into an output and a new value of the state. We require of this function that there be a small, finite upper bound on its computation time. This guarantees that the agent can react quickly to external events by having a fixed delay between the arrival of any given input and the generation of an output that depends on that input.

Rosenschein and Kaelbling have developed a language, called Rex, for programming in this model [Rosenschein and Kaelbling, 1986; Kaelbling, 1987b]. Rex takes a Lisp-like program specification and generates the description of a synchronous digital circuit with delay components that

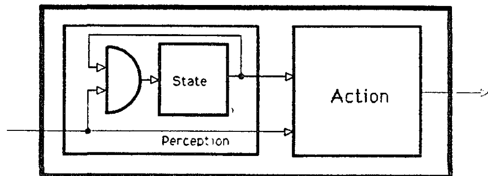


Figure 1: Decomposition into perception and action components.

satisfies the specification. Because the computation is described in terms of a finite circuit that delivers an output on every cycle, it can be carried out in constant time. This language has been used for programming the SRI mobile robot, with state updates being computed many times per second.

One convenient way to view computations of this sort is to divide them into two components: perception and action (as shown in Figure 1.) The perception component contains all of the state and the computation required to update it. This allows the action component to be state-free, taking its input from the output of the perception component.

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 perception 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 $a \leadsto G$) if it constitutes a correct step toward the satisfaction of a 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:

$$\begin{array}{rcl} & a \leadsto \mathrm{do}(a) & \\ (a \leadsto G) \wedge (a \leadsto G') & \Rightarrow & a \leadsto (G \wedge G') \\ (a \leadsto G) \vee (a \leadsto G') & \Rightarrow & a \leadsto (G \vee G') \\ \mathrm{cond}(p, a \leadsto G, a \leadsto G') & \Rightarrow & a \leadsto \mathrm{cond}(p, G, G'). \end{array}$$

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, there seems to be no good way 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 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 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 and 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 create 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 generate 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 $\Pi$, 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,

$$\Pi \text{ weakly satisfies } G \iff \forall i. c_i \rightarrow (a_i \rightsquigarrow G).$$

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 $\Pi$ as

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

A goal $G$ is *strongly satisfied* by program $\Pi$ if it is weakly satisfied by $\Pi$ and $\text{dom}(\Pi) = true$; that is, if for every situation, $\Pi$ 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.

## 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 [Kaelbling, 1987b; Kaelbling and Wilson, 1988]. 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 an integer, *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 $\Gamma$, $\text{eval}(G, \Gamma)$ 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

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

with the *rex-expr* in the $i$th 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

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

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

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

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

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

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:

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

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

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,

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

**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.

**If**

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

$$\texttt{conjoin-cond}\,(p, \Pi) = \{\langle p \wedge c_1, a_1\rangle, \ldots, \langle p \wedge c_n, a_n\rangle\}.$$

Thus,

$$\texttt{disjoin-programs}\,(\texttt{conjoin-cond}\,(p, \Pi'), \texttt{conjoin-cond}\,(\neg p, \Pi'')) = \{\langle p \wedge c_1', a_1'\rangle, \ldots, \langle p \wedge c_n', a_n'\rangle, \langle \neg p \wedge c_1'', a_1''\rangle, \ldots, \langle \neg p \wedge c_m'', a_m''\rangle\}.$$

**Ach and Maint**

Goals of maintenance and achievement are evaluated by disjoining the results of all applicable reduction rules in the rulebase $\Gamma$. 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 variable, dist-err and angle-err are Rex parameters, 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 upon rule-invocation, 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 chose at compile time from among a class of available run-time functions.

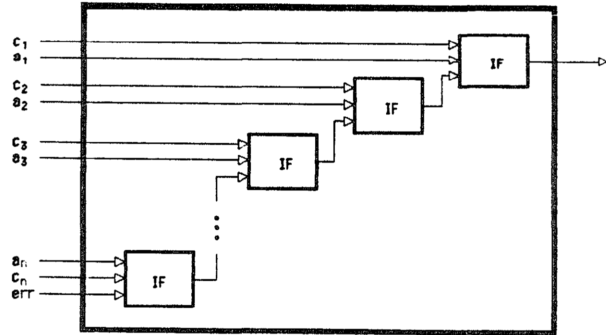


Figure 2: Circuit generated from Gapps program

## 2.3 Generating a Circuit

Once a goal expression has been evaluated, yielding a program, a circuit, similar to the one shown in Figure 2, that instantiates that program is generated.[1] The output of the circuit is the action corresponding to the first condition that is true. The conditions are tested in an arbitrary order that is chosen at compile-time. Because any action whose associated condition is true is sufficient for correctness, the order in which they are arranged is unimportant. 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 computed by the Rex expressions supplied in the **if** and **do** forms. The outputs of the circuit are used to control the agent.

# 3 Additional Features

For the sake of exposition, the previous section presented a somewhat simplified version of Gapps. In the following sections we shall explain additional features that make Gapps more effective for use in practical applications.

## 3.1 Reducing Conjunctive Goal Expressions

In many cases, an effective behavior for achieving $G' \wedge G''$ cannot be generated simply by conjoining programs that achieve $G'$ and $G''$ individually. A program for the goal (and (ach have hammer) (ach have saw)) will almost certainly generate errors when the two tools are in different rooms, because there will be no actions available that are consistent with the standard programs for achieving the each of the subgoals. Because of this, we allow reduction rules of the form (defgoalr (and (*ach-or-maint* $pat_1$ $\textit{rex-params}_1$) (*ach-or-maint* $pat_2$ $\textit{rex-params}_2$)) *goal-expr*) so that special behaviors can be generated in the face of a conjunctive goal. In this case, we might write a rule like

```
(defgoalr (and (ach have hammer) (ach have saw))
  (if (have hammer)
      (and (maint have hammer) (ach have saw))
```

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

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

in which the agent pursues the closer object until he has it, then pursues the second while maintaining the first. We might need a similar rule for reducing the conjunctions of goals of achievement and maintenance. Alternatively, we could write a more generic sequencing rule, like the following:

```
(defgoalr (and (ach ?g1 g1-params) (ach ?g2 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 parameters is true in the world.

## 3.2 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 $\Pi_i = \text{eval}(goal - expr_i)$. 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.

## 3.3 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

(prio (and $G_1$ $G_2$ ... $G_n$)
      (and $G_1$ $G_2$ ... $G_{n-1}$) ...
      (and $G_1$ $G_2$)
      $G_1$).

Isaac Asimov's three laws of robotics [Asimov, 1950] 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 requires the robot and a wall to share the same space will not 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. It is important to remember that all of the symbolic manipulation of the goals happens at compile-time; at run-time, we simply execute the action associated with the first condition that evaluates to true.

## 3.4 Merger Functions

In Gapps, as described so far, the only method for combining actions is switching among them, and they may be conjoined only if they are equal. To allow more flexibility, the user can declare, for each action field, a Rex function that tests two values for compatibility, and one that merges two compatible values. If such a declaration is made, these functions are used in the conjunction of programs in place of `mergeable` and `merge` as defined above.

Merger functions can be used to implement many low-level behavior combination schemes. As an example, consider a robot with the dual goals of arriving at a destination and avoiding crashes, whose control output is a desired velocity vector. The program satisfying the goal of arriving at the destination constructs a vector pointing toward the destination, proportional to its distance away from the robot; the program satisfying the goal of avoiding crashes constructs a vector that points away from the nearest obstacle with a length inversely proportional to the square of the distance. We can define the mergeability function to be always true with the form

```
(defmergeability velocity (v1 v2) !1)
```

and define the merger of the two velocities to be their average

```
(defmerger velocity (v1 v2) (vector-average v1 v2)) .
```

A safer version of the mergeability definition might allow two vectors to be merged only if they were in the same half-plane. Another extension would be for the velocity vectors to have weights associated with them, and have the merger function perform a weighted average, or choose the one with the higher weight.

## 4 Using Gapps

Gapps has been implemented in CommonLisp, in conjunction with an existing implementation of Rex. It has proven very useful for writing navigation programs for the SRI mobile robot. This domain exercises all of the important features of Gapps programs, including low-level actions whose results are unpredictable, time-criticality, and parallel actions. This method of specifying behaviors is especially convenient because, by virtue of its compositionality, each subbehavior can be implemented and tested separately.

Although Gapps' method of specifying goals at compile time may seem inflexible, it easily handles cases in which external goals are given to the agent at run-time. In such a case, we can give the agent the standing goal of following orders and rules of the 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)
          ...))) ,
```

which will cause it to carry out requests as it perceives them.

It is also straightforward to integrate planning into a Gapps program. Planning can be seen as the perceptual process of coming to know which sequence of actions will lead to the satisfaction of a goal. The following Gapps program makes use of planning, 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 in another paper [Kaelbling, 1987a].

## Acknowledgments

This work was inspired by Stan Rosenschein, who knew there had to be a better way to write robot programs. Thanks to Stan, Martha Pollack, David Chapman, Phil Agre, and Tom Dean for helpful comments on previous drafts.

## References

[Agre and Chapman, 1987] Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272, Morgan Kauffman, Seattle, Washington, 1987.

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

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–378, 1987.

[Georgeff and Lansky, 1987] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682, Morgan Kauffman, Seattle, Washington, 1987.

[Kaelbling, 1987a] Leslie Pack Kaelbling. An architecture for intelligent reactive systems. In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning About Actions and Plans*, pages 395–410, Morgan Kauffman, 1987.

[Kaelbling, 1987b] 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.

[Kaelbling and Wilson, 1988] Leslie Pack Kaelbling and Nathan J. Wilson. *Rex Programmer's Manual.* Technical Report 381R, Artificial Intelligence Center, SRI International, Menlo Park, California, 1988.

[Lansky, 1987] Amy L. Lansky. Localized representation and planning methods for parallel domains. In *Proceedings of the National Conference on Artificial Intelligence*, Morgan Kauffman, Seattle,Washington, 1987.

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

[Schoppers, 1987] Marcel J. Schoppers. Universal plans for reactive robots in unpredictable environments. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 1039–1046, Morgan Kauffman, Milan, 1987.

[Wilkins, 1985] David E. Wilkins. Recovering from execution errors in SIPE. *Computational Intelligence*, 1:33–45, 1985.