Integrating Planning and Reactive Control*

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

1 Introduction

Artificial intelligence research on planning is concerned with designing control systems that choose actions by manipulating explicit descriptions of the world state, the goal to be achieved, and the effects of elementary operations available to the system. Because planning shifts much of the burden of reasoning to the machine, it holds great appeal as a high-level programming method [3,10,12]. Experience shows, however, that it cannot be used indiscriminately because even moderately rich languages for describing goals, states, and the elementary operators lead to computational inefficiencies that render the approach unsuitable for realistic applications. This inadequacy has spawned a recent wave of research on "reactive control" or "situated activity" in which control systems are modeled as reacting directly to the current situation rather than as reasoning about the future effects of alternative action sequences [2,1,11]. While this research has confronted the issue of run-time tractability head on, in many cases it has done so by sacrificing the advantages of declarative planning techniques.

This paper discusses ways in which the two approaches can be unified. We begin by modeling reactive control systems as state machines that map a stream of sensory inputs to a stream of control outputs. These machines can be decomposed into two continuously active subsystems: the planner and the execution module. The planner computes a "plan," which can be seen as a set of bits that control the behavior of the execution module. An important element of this work is the formulation of a precise semantic interpretation for the inputs and outputs of the planning system. We show that the distinction between planned and reactive behavior is largely in the eye of the beholder: Systems that seem to compute explicit plans can be redescribed in situation-action terms and vice versa. We also discuss practical programming techniques that allow the advantages of declarative programming and guaranteed reactive response to be achieved simultaneously.

*This work was supported in part by NASA Cooperative Agreement #NCC-2-494 through Stanford subcontract #PR6359 and in part by a gift from the System Development Foundation.

2 Planning and Reactive Control

Classical AI views the generation of behavior as a two-step process consisting of planning and execution. Planning produces a data structure describing a course of action; execution is the step-by-step interpretation of this data structure to produce overt behavior. The planning step can be viewed as a form of stylized program synthesis in a weak logic of programs, and many formalisms have been proposed to capture the logic of planning. A common approach is to employ predicate calculus formulas as state descriptions (e.g., *on(blockA,blockB)*) and to model operators as state-transforming functions, described either axiomatically (using facts of the form $holds(p,s) \rightarrow holds(q,op(s))$) or as syntactic transformations that map state descriptions to state descriptions. Letting *ops*, *init*, and *goal* stand for formulas expressing, respectively, facts about the operators, the initial conditions, and the goal statement, we require the planner to find $plan = \text{make-plan}(ops, init, goal)$ such that

$$ops \models init \wedge plan \rightarrow goal \ .$$

In other words, it should follow from the operator descriptions that if the initial condition holds and the plan is carried out, the goal condition will be achieved. Note also that $init \wedge plan$ should be consistent; otherwise, the requirement can be trivially satisfied.

The complexity of plan synthesis obviously depends on the specific nature of the domain. For realistic domains, however, traditional planning typically requires significantly more time than the fundamental reflex cycle of the system, and controlling the rate at which planning occurs relative to changes in the environment is extremely challenging. For this reason, classical planning techniques have almost always been applied, in practice, to "static" domains, in which the only significant source of change is the agent itself and in which, therefore, the time required for planning can be safely ignored.

In an attempt to deal with more dynamic domains, some researchers have abandoned planning in favor of reactive control, which does not take a two-stage view of behavior generation. In this approach, the behavior of the agent is specified directly using situation-action rules that are evaluated at frequent intervals. A reactive control system could be implemented, for example, as a program executing a tight loop, the body of which exhibits a high degree of conditionality, for example:

```
do forever
      if tiger_approaching then
          set wheel velocities to [+30,+30],
      else if ...
```

Since the conditions can be evaluated in parallel, reactive systems can also be described as circuits or operator networks implementing a function that maps a stream of information states to a stream of output commands to the effectors. The key to reactivity is to design this function so that it can be computed quickly again and again.

Each approach has its advantages. Planning provides a convenient high-level declarative formalism and leaves much of the reasoning to the machine. In principle, this makes it possible for the control system to handle classes of situations that are too complex for the programmer to anticipate in advance but are amenable to analysis at run time, once a concrete initial state and goal state are available. In contrast, reactive control offers the advantage of guaranteed

response time and hence the ability to react quickly to a changing environment. Because neither approach clearly dominates the other and because many application domains have attributes that make each attractive, a synthesis of these two techniques is necessary.

One method for achieving such a synthesis is to embed a reactive controller in a classical planner-based architecture. In a sense, this is what the term "execution monitoring" is often taken to mean in classical planning: The planner sends a data structure to the execution module, which in turn reacts to changing world conditions under the control of the plan. The execution module is also able to detect conditions in the world that violate the assumptions upon which the plan's correctness depends. Unfortunately, the mathematical framework of classical planning, based on atemporal state transformations, offers little guidance as to how the passage of time during the planning process ought to be handled.

Since reactive control is based on a model of time-bounded computation, it is more natural to incorporate planning by extending the reactive-control architecture rather than vice versa, and this is the approach we shall take. In order to do this, however, we must first characterize the semantics of the data structures produced by the planner in a way that makes sense in the reactive control model.

3 Semantics for Planning and Control

We shall model a control system as a state machine that transduces inputs carrying information about the environment to outputs that affect the environment. In the simplest case, this machine has no state and simply computes a pure function from inputs to outputs. In more complex cases, including cases in which significant planning occurs, the computation requires internal state. A major challenge in designing control systems is to provide a clear semantic model of the information available to the control system, of the goals achieved by the chosen actions, and of the mapping between the two.

Let M be a control system with input variable *in*, output variable *out*, and an internal state vector a. The inputs carry information about the world, the outputs are commands to the effectors, and the internal state allows the computation of outputs to depend on past inputs and to be extended in time. To introduce a planner into this model, we decompose the machine into components, introducing three subsidiary variables, *init*, *goal*, and *plan*, and four sub-machines: E_{init}, E_{goal}, *Planner*, and *Exec*. We assume that *ops* is fixed in advance. The inputs and outputs of these modules are as follows:

- E_{init}: input *in*, output *init*
- E_{goal}: input *in*, output *goal*
- *Planner*: input *init*, *goal*, output *plan*
- *Exec*: input *in*, *plan*, output *out*

The overall structure of the machine is illustrated in Figure 1. Informally, the E_{init} and E_{goal} machines operate on the input, extracting values representing the initial conditions and goal condition, respectively. These are transduced by *Planner*, in a way that may involve internal state and computation over time, to a continuously available *plan* output. Note, however,

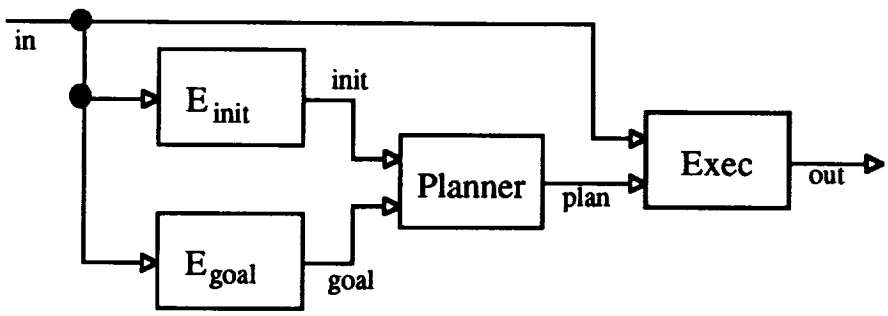

Figure 1: Embedding a planner in a reactive control system.

that the output may be vacuous, indicating that the final plan has not yet been computed [4].

We are interested in characterizing the semantics of the inputs of *Planner* and of its result, but must first consider the more general question of where semantic "interpretations" for data values come from.

For data structures like *init*, the classical view is that the data value is a description of facts about the world expressed in some language whose semantics is clear to the designer of the system. This description would be of little use were it not also the case that when the data structure had a particular value, the condition denoted was guaranteed to hold in the environment. Such semantic considerations form the foundation of the situated-automata model in which the semantics of data structures are characterized in terms of objective correlations with external reality rather than in terms of designer-stipulated interpretations. In this approach, one says a machine variable x carries 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 [7,8,5,9].

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] Since this 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 this, 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

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

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.

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.

Formally 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 *Planner* module with inputs *init* and *goal* and output *plan*. The most natural way to interpret the values of these variables is to apply the information interpretation to the values of *init* and the goal interpretation to the values of *goal* and *plan*. However, as observed above, since the goal interpretation is derived directly from the informational model, we could have applied either interpretation to any of the values.

In summary, one need not think of "planning" as an essentially different kind of function performed by the system. Rather, it can be thought of as a *perspective* one takes on certain data structures when one thinks of them—for design convenience—as encoding goals rather than information.

4 Current Research Directions

In this section we list several efforts currently underway that are aimed at exploring the practical consequences of our approach toward integrating planning and reactive control.

4.1 Embedding Planning in Gapps

Gapps [6] is a declarative language for programming reactive systems. The Gapps compiler takes as input a top-level goal and a set of goal reduction rules and produces as output a

program for achieving the top-level goal. The program is guaranteed by construction to map information states to actions in constant time. By using Gapps, the programmer can gain many of the benefits of declarative programming without sacrificing real-time response. One direction of research is to embed planning in Gapps by converting operator descriptions into goal reduction rules, which in turn are transformed by Gapps into real-time programs. A typical rule schema might be:

```
(defgoalr (ach P)
  (if (regress P a)
      (do a)
      (ach (regress P a))))
```

Because Gapps produces a fixed-size circuit at compile time, a compile-time bound must be placed on the *depth* of the regression, although in principle the actual calculation of the regressed condition can be deferred to run time.

4.2 Temporally Extended Planning Processes

Traditional planners operate by carrying out a guided search through a space of plans. Depending on the combinatorics of the search, this process may or may not succeed within a single cycle of the reactive system. If it does not, the search must proceed in parallel with the execution of a more reactive, though perhaps less effective, behavior. Since the passage of time affects whether or not a data value will continue to be correlated with the environment, it is clear that the semantics of temporally-extended planning will be time-dependent. A simple solution to this problem is for the planner to monitor world conditions that would invalidate the current plan and to output the vacuous plan when those conditions arise [4]. While correct, this approach is not maximally information-preserving and more subtle methods are possible. In the case of informational data structures, we have explored declarative programming techniques to control the updating of the machine's information state so that maximal correlation with the environment is maintained [9], and similar methods might be applied to planning over time as well.

4.3 Trading Flexibility for Performance

As in conventional programming, some information required for action selection might be available at compile time, while other information may become available only at run time. Ease of programming would be enhanced by minimizing syntactic and semantic distinctions based only on differences as to when information becomes available. In traditional compilers, for instance, constant-folding optimizations take advantage of compile-time information about the values of expressions in a way that is entirely transparent to the programmer. For planning and control applications, this transparency is more difficult to achieve because without sufficient compile-time information, the symbolic synthesis procedure may not terminate, and without a clear compile-time versus run-time model in mind, the programmer may lack sufficient insight to adequately control the compilation process. Nevertheless, our ultimate goal is to make it as easy as possible to trade off flexibility against performance by conveniently moving the boundary between compile-time and run-time processing.

Acknowledgments

We have benefited greatly from discussions with Mark Drummond and Monte Zweben.

References

[1] Agre, Philip E. and David Chapman. "Pengi: An Implementation of a Theory of Activity." *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, Washington (July 1987).

[2] Brooks, Rodney A. "A Robust Layered Control System for a Mobile Robot." Technical Report 864, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (1985).

[3] Fikes, Richard and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence*, Vol. 2, Nos. 3,4 (1971).

[4] Kaelbling, Leslie P. "An Architecture for Intelligent Reactive Systems." In Michael P. Georgeff and Amy L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*. Morgan Kaufmann, Los Altos, California (1987).

[5] Kaelbling, Leslie P. "Rex: A Symbolic Language for the Design and Parallel Implementation of Embedded Systems." *Proceedings of the AIAA Conference on Computers in Aerospace*, Wakefield, Massachusetts (1987).

[6] Kaelbling, Leslie P. "Goals as Parallel Program Specifications." *Proceedings of the Seventh National Conference on Artificial Intelligence*, Morgan Kaufmann, St. Paul, Minnesota (August 1988).

[7] Rosenschein, Stanley J. "Formal Theories of Knowledge in AI and Robotics". In *New Generation Computing*, Vol. 3, No. 4, (special issue on Knowledge Representation), Ohmsha, Ltd., Tokyo, Japan (1985).

[8] Rosenschein, Stanley J. and Leslie P. Kaelbling. "The Synthesis of Digital Machines with Provable Epistemic Properties," *Proceedings of Workshop on Theoretical Aspects of Reasoning About Knowledge*, Monterey, California (1986).

[9] Rosenschein, Stanley J. "Synthesizing Information-Tracking Automata from Environment Descriptions." *Proceedings of the First Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada (to appear).

[10] Sacerdoti, Earl. *A Structure for Plans and Behavior*. Elsevier Noth-Holland, Inc., New York (1977).

[11] Schoppers, Marcel J. "Universal Plans for Reactive Robots in Unpredictable Environments." *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Morgan Kauffman, Milan (1987).

[12] Wilkins, David E. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann Publishers, Inc. San Mateo, California (1988).