What is planning in the presence of sensing?*

Hector J. Levesque
Department of Computer Science
University of Toronto
Toronto, ON, M5S 3H5 Canada
hector@cs.toronto.edu

Abstract

Despite the existence of programs that are able to generate so-called conditional plans, there has yet to emerge a clear and general specification of what it is these programs are looking for: what exactly is a plan in this setting, and when is it correct? In this paper, we develop and motivate a specification within the situation calculus of conditional and iterative plans over domains that include binary sensing actions. The account is built on an existing theory of action which includes a solution to the frame problem, and an extension to it that handles sensing actions and the effect they have on the knowledge of a robot. Plans are taken to be programs in a new simple robot program language, and the planning task is to find a program that would be known by the robot at the outset to lead to a final situation where the goal is satisfied. This specification is used to analyze the correctness of a small example plan, as well as variants that have redundant or missing sensing actions. We also investigate whether the proposed robot program language is powerful enough to serve for any intuitively achievable goal.

Much of high-level symbolic AI research has been concerned with planning: specifying the behaviour of intelligent agents by providing goals to be achieved or maintained. In the simplest case, the output of a planner is a sequence of actions to be performed by the agent. However, a number of researchers are investigating the topic of *conditional planning* (see for example, [3, 9, 14, 17]) where the output, for one reason or another, is not expected to be a fixed sequence of actions, but a more general specification involving conditionals and iteration. In this paper, we will be concerned with conditional planning problems where what action to perform next in a plan may depend on the result of an earlier *sensing action*.

Consider the following motivating example:

*Thanks to the members of the University of Toronto Cognitive Robotics group (Yves Lesperance, Fangzhen Lin, Daniel Marcu, Ray Reiter, and Richard Scherl) and to Fahiem Bacchus, for discussion, comments, and suggestions. A special thanks to Yves for helping with the definition in Section 3, and to Fangzhen for asking and helping to answer the question of Section 5. This research was made possible by financial support from the Information Technology Research Center, the Institute for Robotics and Intelligent Systems, and the Natural Science and Engineering Research Council. They also had to pick up the asinine AAAI fee for extra pages.

The Airport Example The local airport has only two boarding gates, Gate A and Gate B. Every plane will be parked at one of the two gates. In the initial state, you are at home. From home, it is possible to go to the airport, and from there you can go directly to either gate. At the airport, it is also possible to check the departures screen, to find out what gate a flight will be using. Once at a gate, the only thing to do is to board the plane that is parked there. The goal is to be on the plane for Flight 123.

There clearly is no sequence of actions that can be shown to achieve the desired goal: which gate to go to depends on the (runtime) result of checking the departure screen.

Surprisingly, despite the existence of planners that are able to solve simple problems like this, there has yet to emerge a clear specification of what it is that these planners are looking for: what is a plan in this setting, and when is it correct? In this paper, we will propose a new definition, show some examples of plans that meet (and fail to meet) the specification, and argue for the utility of this specification independent of plan generation.

What we will *not* do in this paper is propose a new planning procedure. In many cases, existing procedures like the one presented in [3] will be adequate, given various representational restrictions. Moreover, our specification goes beyond what can be handled by existing planning procedures, including problems like the following:

The Omelette Example We begin with a supply of eggs, some of which may be bad, but at least 3 of which are good. We have a bowl and a saucer, which can be emptied at any time. It is possible to break a new egg into the saucer, if it is empty, or into the bowl. By smelling a container, it is possible to tell if it contains a bad egg. Also, the contents of the saucer can be transferred to the bowl. The goal is to get 3 good eggs and no bad ones into the bowl.

While it is far from clear how to automatically generate a plan to solve a problem like this,[1] our account, at least, will make clear what a solution ought to be.

[1]However, see [10] for some ideas on how to generate plans containing loops (when there is no sensing).

Classical planning

There are a number of ways of making the planning task precise, but perhaps the most appealing is to put aside all algorithmic concerns, and come up with a specification in terms of a general theory of action. In the absence of sensing actions, one candidate language for formulating such a theory is the situation calculus [12]. We will not go over the language here except to note the following components: there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol *do* where $do(a, s)$ denotes the successor situation to s resulting from performing the action a; relations whose truth values vary from situation to situation, are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

Within this language, we can formulate domain theories which describe how the world changes as the result of the available actions. One possibility is a theory of the following form [15]:

- Axioms describing the initial situation, S_0.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s. These take the place of the so-called effect axioms, but also provide a solution to the frame problem [15].
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

For any domain theory of this sort, we have a very clean specification of the planning task (in the absence of sensing actions), which dates back to the work of Green [5]:

Classical Planning: Given a domain theory *Axioms* as above, and a goal formula $\phi(s)$ with a single free-variable s, the planning task is to find a sequence of actions[2] $\vec{a}$ such that

$$Axioms \models Legal(\vec{a}, S_0) \wedge \phi(do(\vec{a}, S_0))$$

where $do([a_1, \ldots, a_n], s)$ is an abbreviation for

$$do(a_n, do(a_{n-1}, \ldots, do(a_1, s)\ldots)),$$

and where $Legal([a_1, \ldots, a_n], s)$ stands for

$$Poss(a_1, s) \wedge \ldots \wedge Poss(a_n, do([a_1, \ldots, a_{n-1}], s)).$$

In other words, the task is to find a sequence of actions that is executable (each action is executed in a context where its precondition is satisfied) and that achieves the goal (the goal formula ϕ holds in the final state that results from performing the actions in sequence).[3] A planner is *sound* if any sequence of actions it returns satisfies the entailment; it is *complete* if it is able to find such a sequence when one exists.

Of course in real applications, for efficiency reasons, we may need to move away from the full generality of this specification. In some circumstances, we may settle for a sound but incomplete planner. We may also impose constraints on what what sorts of domain theories or goals are allowed. For example, we might insist that S_0 be described by just a finite set of atomic formulas and a closed world assumption, or that the effect of executable actions not depend on the context of execution, as in most STRIPS-like systems.

However, it is clearly useful to understand these moves in terms of a specification that is unrelated to the limitations of any algorithm or data structure. Note, in particular, that the above account assumes nothing about the form of the preconditions or effects of actions, uses none of the terminology of STRIPS (add or delete lists *etc.*), and none of the terminology of "partial order planning" (threats, protecting links *etc.*). It is neutral but perfectly compatible with a wide range of planners. Indeed the STRIPS representation can be viewed as an *implementation strategy* for a class of planning tasks of this form [6].

Incorporating sensing actions

In classical planning, it is assumed that what conditions hold or do not hold at any point in the plan is logically determined by the background domain theory. However, agents acting in the world may require sensing for a number of reasons:[4]

- There may be incomplete knowledge of the initial state. In the Airport example above, nothing in the background theory specifies where Flight 123 is parked, and the agent needs to check the departure screen at the airport to find out. The Omelette example is similar.
- There may be exogenous actions. The agent may know everything about the initial state of the world, but the world may change as the result of actions performed by other agents or nature. For example, a robot may need to check whether or not a door is open, in case someone has closed it since the last time it checked.
- The effects of actions may be uncertain. For example, a tree-chopping robot may have to check if the tree went down the last time it hit it with the axe.

This then raises an interesting question: is there a specification of the planning task in a domain that includes sensing actions like these which, once again, is neutral with respect to the choice of algorithm and data structure?

Informally, what we expect of a plan in this setting is that it be some sort of *program* that leads to a goal state no matter how the sensing turns out. For the Airport example, an expected solution might be something like

[2]To be precise, what we need here (and similarly below for robot programs) are not actions, but ground terms of the action sort that contain no terms of the situation sort.

[3]This definition is easily augmented to accommodate maintenance goals, conditions that must remain true throughout the execution. For space reasons, we ignore them here.

[4]In this paper, we limit ourselves to the first of these.

```
go to the airport;
check the departures screen;
if Flight 123 is boarding at Gate A
    then go to Gate A
    else go to Gate B;
board the plane.
```

Similarly, in the case of the Omelette, we might expect a plan like

```
/* Assuming the bowl and saucer are empty initially */
until there are 3 eggs in the bowl do
    until there is an egg in the saucer do
        break an egg into the saucer;
        smell the saucer;
        if the saucer has a bad egg
            then discard its contents
    end until;
    transfer the contents of the saucer to the bowl
end until
```

Note that in either case, the plan would not be correct without the appropriate sensing action.

The closest candidate that I could find to a formal specification of a plan in this setting is that of Etzioni *et al* in [3]. In addition to a partial-order plan-generation procedure in the style of SNLP [11], they propose a definition of a plan, and what it means for a plan containing sensing actions to be *valid* (achieve a desired goal for a given initial state).

Unfortunately, as a specification, their account has a number of drawbacks. For one thing, it is formulated as a rather complex refinement of the STRIPS account. It deals only with atomic conditions or their negations, assumes that we will be able to "match" the effects of actions with goals to be achieved, and so on. There are also other representational limitations: it does not allow preconditions on sensing actions, and does not handle iteration (and so could not deal with the Omelette example). While limitations like these may be perfectly reasonable and even necessary when it comes to formulating efficient planning procedures, they tend to obscure the logic behind the procedure.

There are other problems as well.[5] In describing plan validity, they insist that every branch of a plan must be valid, where a branch is one of the possible executions paths through any if-then-else in the plan. But this is overly strict in one sense, and not strict enough in another. Imagine a plan like the Airport one above except that it says that if Flight 123 is at Gate A, the agent should jump off the roof of the airport. Suppose however, that the sensing happens to be *redundant* because the agent already knows that the gate for Flight 123 is Gate B. In this context, the plan should be considered to be correct, despite what appears to be a bad branch. Next, imagine a plan like the Airport one above, but without the sensing action. Even though both branches of the if-then-else are handled properly, the plan is now incorrect since an agent executing it would not know the truth value of the condition. This is not to suggest that the *planner* developed by Etzioni *et al* is buggy; they may never end up generating plans like the above. However, as a procedure independent specification, we should be able to evaluate the appropriateness of plans with extra or missing sensing actions.

[5]It may be possible to fix their definition to handle these [4].

Instead of building on a STRIPS-like definition of planning, we might again try to formulate a specification of the planning task in terms of a general theory of action, but this time including sensing actions and the effect they have on the knowledge of the agent or robot executing them.

As it turns out a theory of this sort already exists. Building on the work of Moore [13], Scherl and Levesque have provided a theory of sensing actions in the situation calculus [16]. Briefly, what they propose is a new fluent K, whose first argument is also a situation: informally, $K(s', s)$ holds when the agent in situation s, unsure of what situation it is in, thinks it could very well be in situation s'. Since different fluents hold in different situations, the agent is also implicitly thinking about what could be true. Knowledge for the agent, then, is what *must* be true because it holds in all of these so-called accessible situations: $Know(\phi, s)$ is an abbreviation for the formula $\forall s'[K(s', s) \supset \phi(s')]$. Beyond this encoding of traditional modal logic into the situation calculus, Scherl and Levesque provide a successor state axiom for K, that is an axiom which describes for any action (ordinary or sensing) the knowledge of the agent after the action as a function of its knowledge and other conditions before the action.

Assume for simplicity that we have two types of primitive actions: ordinary ones that change the world, and *binary sensing actions*, that is, sensing actions that tell the agent whether or not some condition ϕ_a holds in the current situation.[6] For each sensing action a, we assume the domain theory entails a *sensed fluent axiom* of the form

$$SF(a, s) \equiv \phi_a(s),$$

where SF is a distinguished predicate like $Poss$, relating the action to the fluent. For the Airport example, we might have

$$SF(check\_departures, s) \equiv Parked(Flight123, gateA, s)$$

which says that checking the departure screen will tell the agent whether or not Flight 123 is parked at Gate A. Similarly,

$$SF(smell(c), s) \equiv \exists e.Bad\_egg(e, s) \wedge Contains(c, e, s)$$

says that smelling a container c tells the agent whether or not c contains a bad egg e. We also assume that the domain theory entails $[SF(a, s) \equiv \mathbf{True}]$ for every ordinary non-sensing action a. Under these assumptions, we have the following successor state axiom for K:

$$\begin{array}{l} Poss(a, s) \supset \{K(s'', do(a, s)) \equiv \\ \quad \exists s'.\, s'' = do(a, s') \wedge K(s', s) \wedge Poss(a, s') \wedge \\ \qquad [SF(a, s') \equiv SF(a, s)]\} \end{array}$$

Roughly speaking, this says that after doing any action a in situation s, the agent thinks it could be in a situation s'' iff s'' is the result of performing a in some previously accessible

[6]Later we discuss other types of sensing, especially sensing that involves a sensor reading.

s', provided that action a is possible in s' and s' is identical to s in terms of what is being sensed, if anything. For example, if a is *check_departures*, this would have the effect of ensuring that any such s' would have Flight 123 parked at the same gate as in s. Assuming the successor state axiom for *Parked* is such that where a plane is parked is unaffected by sensing, any accessible s'' would also have Flight 123 parked at the same gate as in s. Thus, the result is that after *check_departures*, the agent will know whether or not Flight 123 is parked at Gate A. More generally, the set of accessible situations after performing any action is completely determined by the action, the state of the world, and the accessible situations before the action. This therefore extends Reiter's solution to the frame problem to the K fluent.

Robot programs

While the above theory provides an account of the relationship between knowledge and action, it does not allow us to use the classical definition of a plan. This is because, in general, there is no *sequence of actions* that can be shown to achieve a desired goal; typically, what actions are required depends on the runtime results of earlier sensing actions.

It is tempting to amend the classical definition of planning to say that the task is now to find a *program* (which may contain conditionals or loops) that achieves the goal, a sequence of actions being merely a special case.

But saying we need a program is not enough. We need a program that does not contain conditions whose truth value (nor terms whose denotations) would be unknown to the agent at the required time: that is, the agent needs to *know how* to execute the program. One possibility is to develop an account of what it means to know how to execute an arbitrary program, for example, as was done by Davis in [2]. While this approach is certainly workable, it does lead to some complications. There may be programs that the agent "knows how" to execute in this sense but that we do not want to consider as plans.[7] Here, we make a much simpler proposal: invent a programming language $\mathcal{R}$ whose programs include both ordinary and sensing actions, and which are all so clearly executable that an agent will trivially know how to do so.

Consider the following simple programming language, defined as the least set of terms satisfying the following:

1. *nil* and *exit* are programs;
2. If a is an ordinary action and r is a program, then $\underline{seq}(a, r)$ is a program;
3. If a is a binary sensing action and r_1 and r_2 are programs, then $\underline{branch}(a, r_1, r_2)$ is a program;
4. If r_1 and r_2 are programs, then $\underline{loop}(r_1, r_2)$ is a program.

We will call such terms *robot programs* and the resulting set of terms $\mathcal{R}$, the robot programming language.

Informally, these programs are executed by an agent as follows: to execute *nil* the agent does nothing; to execute *exit* it must be executing a *loop*, in which case see below; to execute $\underline{seq}(a, r)$, it executes primitive action a, and then r; to execute $\underline{branch}(a, r_1, r_2)$ it executes a which is supposed to tell it whether or not some condition ϕ_a holds, and so it executes r_1 if it does, and r_2 otherwise; to execute $\underline{loop}(r_1, r_2)$, it executes the body r_1, and if it ends with *nil*, it repeats r_1 again, and continues doing so until it ends with *exit*, in which case it finishes by executing r_2.

The reason a loop-exit construct is used instead of the more "structured" while-loop, is that to ensure that an agent would always know how to execute a robot program, $\mathcal{R}$ does not include any conditions involving fluents. Thus, although we want robot programs to contain branches and loops, we cannot use the traditional if-then-else or while-loop constructs. $\mathcal{R}$is a minimal language satisfying our criteria, but other designs are certainly possible. Note that it will not be our intent to ever write programs in this language; it should be thought of as an "assembly language" into which planning goals will compile.

Here are two example robot programs. The first, R_{air}, is from the Airport domain:

```
seq(go(airport),
  branch(check_departures,
    seq(go(gateA),seq(board_plane(Flight123),nil))
    seq(go(gateB),seq(board_plane(Flight123),nil))));
```

the second, R_{egg}, is from the Omelette domain:

```
loop(body,
  seq(transfer(saucer,bowl),
    loop(body,
      seq(transfer(saucer,bowl),
        loop(body,
          seq(transfer(saucer,bowl),nil)))))),
```

where *body* stands for the program

```
seq(break_new_egg(saucer),
  branch(smell(saucer),
    seq(dump(saucer),nil),
    exit)).
```

There is an equivalent formulation of robot programs as finite directed graphs. See the regrettably tiny figures squeezed in after the references.

Intuitively at least, the following should be clear:

- An agent can always be assumed to know how to execute a robot program. These programs are completely deterministic, and do not mention any fluents. Assuming the binary sensing actions return a single bit of information to the agent, there is nothing else it should need to know.
- The example robot programs above, when executed, result in final situations where the goals of the above planning problems are satisfied: R_{air} gets the agent on Flight 123, and R_{egg} gets 3 good eggs into the bowl.

In this sense, the programs above constitute a solution to the earlier planning problems.

To be precise about this, we need to first define what situation is the final one resulting from executing a robot program r in an initial situation s. Because a robot program

[7]Consider, for example, the program that says (the equivalent of) "find a plan and then execute it." While this program is easy enough to generate, figuring out how to execute it sensibly is as hard as the original planning problem.

could conceivably loop forever (e.g. $\underline{loop}(\underline{nil},\underline{nil})$), we will use a formula $Rdo(r, s, s')$ to mean that r terminates legally when started in s, and s' is the final situation. Formally, Rdo is an abbreviation for the following second-order formula:

$$Rdo(r, s_1, s_2) \overset{def}{=} \forall P[\ldots \supset P(r, s_1, s_2, 1)]$$

where the ellipsis is (the conjunction of the universal closure of) the following:

1. Termination, normal case:
   $P(\underline{nil}, s, s, 1)$;
2. Termination, loop body:
   $P(\underline{exit}, s, s, 0)$;
3. Ordinary actions:
   $Poss(a, s) \wedge P(r', do(a, s), s', x) \supset$
   $P(\underline{seq}(a, r'), s, s', x)$;
4. Sensing actions, true case:
   $Poss(a, s) \wedge SF(a, s) \wedge P(r', do(a, s), s', x) \supset$
   $P(\underline{branch}(a, r', r''), s, s', x)$;
5. Sensing actions, false case:
   $Poss(a, s) \wedge \neg SF(a, s) \wedge P(r'', do(a, s), s', x) \supset$
   $P(\underline{branch}(a, r', r''), s, s', x)$;
6. Loops, exit case:
   $P(r', s, s'', 0) \wedge P(r'', s'', s', x) \supset$
   $P(\underline{loop}(r', r''), s, s', x)$;
7. Loops, repeat case:
   $P(r', s, s'', 1) \wedge P(\underline{loop}(r', r''), s'', s', x) \supset$
   $P(\underline{loop}(r', r''), s, s', x)$.

By using second-order quantification in this way, we are defining *Rdo* recursively as the *least* predicate P satisfying the constraints in the ellipsis. Second-order logic is necessary here since there is no way to characterize the transitive closure implicit in unbounded iteration in first-order terms.

Within this definition, the relation $P(r, s, s', 0)$ is intended to hold when executing r starting in s terminates at s' with $\underline{exit}$; $P(r, s, s', 1)$ is the same but terminating with $\underline{nil}$. The difference shows up when executing $\underline{loop}(r, r')$: in the former case, we exit the loop and continue with r'; in the latter, we continue the iteration by repeating $\underline{loop}(r, r')$ once more.

It is not hard to show that these robot programs are *deterministic*, in that there is at most a single s' such that $Rdo(r, s, s')$ holds. Less trivially, we also get:

Theorem 1: *The following formulas are logically valid:*

1. $Rdo(\underline{nil}, s, s') \equiv (s = s')$.
2. $Rdo(\underline{seq}(a, r), s, s') \equiv Poss(a, s) \wedge Rdo(r, do(a, s), s')$.
3. $Rdo(\underline{branch}(a, r, r'), s, s') \equiv Poss(a, s) \wedge$
   $[SF(a, s) \supset Rdo(r, do(a, s), s')] \wedge$
   $[\neg SF(a, s) \supset Rdo(r', do(a, s), s')]$.
4. $Rdo(\underline{loop}(r, r'), s, s') \equiv$
   $Rdo(\text{unwind}(r, r', \underline{loop}(r, r')), s, s')$
   where $\text{unwind}(r, r', r'')$ *is defined recursively by*
   (a) $\text{unwind}(\underline{exit}, r', r'') = r'$
   (b) $\text{unwind}(\underline{nil}, r', r'') = r''$
   (c) $\text{unwind}(\underline{seq}(a, r), r', r'') = \underline{seq}(a, \text{unwind}(r, r', r''))$
   (d) $\text{unwind}(\underline{branch}(a, r_1, r_2), r', r'') =$
   $\underline{branch}(a, \text{unwind}(r_1, r', r''), \text{unwind}(r_2, r', r''))$
   (e) $\text{unwind}(\underline{loop}(r_1, r_2), r', r'') =$
   $\underline{loop}(r_1, \text{unwind}(r_2, r', r''))$

This theorem tells us how to build an interpreter for robot programs. For example, to execute

$$\underline{loop}(\underline{branch}(a, \underline{exit}, \underline{seq}(b, \underline{nil})), r),$$

we can unwind the loop and execute

$$\underline{branch}(a, r, \underline{seq}(b, \underline{loop}(\underline{branch}(a, \underline{exit}, \underline{seq}(b, \underline{nil})), r))).$$

Note that we should not try to define *Rdo* "axiomatically" using axioms like these (as in [13], for example) since they are first-order, and not strong enough to characterize loop termination.

The revised planning task

With the definition of a plan as a robot program, we are now ready to generalize the classical planning task:

Revised Planning: Given a domain theory *Axioms* and a goal formula $\phi(s')$ with a single free-variable s', the planning task is to find a robot program r in the language $\mathcal{R}$ such that:

$$Axioms \models \forall s.K(s, S_0) \supset \exists s'[Rdo(r, s, s') \wedge \phi(s')]$$

where *Axioms* can be similar to what it was, but now covering sensing actions and the K fluent.

To paraphrase: we are looking for a robot program r such that it is known in the initial situation that the program will terminate in a goal state.[8] This reduces to the classical definition when there are no sensing actions, and $K(s, S_0)$ holds iff $(s = S_0)$. In this case, it is sufficient to find an r of the form $\underline{seq}(a_1, \underline{seq}(a_2, \ldots \underline{nil}))$.

Note that we are requiring that the program lead to a goal state s' starting in any s such that $K(s, S_0)$; in different s, r may produce very different sequences of actions.

To show this definition in action, we will formalize a version of the Airport problem and establish the correctness of the above robot program and a few variants.

For our purposes, there are two ordinary actions $go(x)$ and *board_plane*(p), one sensing action *check_departures*, and three relational fluents $At(x, s)$, *On_plane*(p, s) and *Parked*(p, x, s), where x is a location, either *home*, *airport*, *gateA*, or *gateB*, and p is a plane. We have the following domain theory:[9]

[8]We are requiring the agent to *know how* to achieve the goal, in that the desired r must be known initially to achieve ϕ. A variant would require an r that achieved ϕ starting in S_0, but perhaps unbeknownst to the agent. A third variant might require not merely ϕ, but that the agent know that ϕ at the end. So many variants; so little space.

[9]We omit here unique name axioms for constants, as well as domain closure axioms, including one saying that Gate A and Gate B are the only gates.

- Precondition axioms:
  $Poss(board\_plane(p), s) \equiv \exists x.Parked(p, x, s) \wedge At(x, s)$
  $Poss(check\_departures) \equiv \neg At(home, s)$
  $Poss(go(x), s) \equiv x = airport \vee At(airport, s)$;
- Successor state axioms: the one above for K and
  $Poss(a, s) \supset \{At(x, do(a, s)) \equiv$
  $a = go(x) \vee (At(x, s) \wedge \neg\exists y.a = go(y))\}$
  $Poss(a, s) \supset \{On\_plane(p, do(a, s) \equiv$
  $a = board\_plane(p) \vee On\_plane(p, s)\}$
  $Poss(a, s) \supset \{Parked(p, x, do(a, s)) \equiv Parked(p, x, s)\}$;
- Sensed fluent axiom:
  $SF(go(x), s) \wedge SF(board\_plane(p), s) \wedge$
  $[SF(check\_departures, s) \equiv$
  $Parked(Flight123, gateA, s)]$.

The goal $\phi(s')$ to be satisfied is $On\_plane(Flight123, s')$. We claim that a solution to this planning problem is the earlier robot program R_{air}. Using the above theorem, the proof is straightforward:[10] We need to show

$$K(s, S_0) \supset \exists s' [Rdo(R_{\text{air}}, s, s') \wedge On\_plane(Flight123, s')].$$

So let us imagine that $K(s, S_0)$ and show that there is an appropriate s'. There are two cases: first suppose that $Parked(Flight123, gateA, s)$.

1. Let $a_1 = go(airport)$ and $s_1 = s$. The program R_{air} is of the form $\underline{seq}(a_1, R_1)$. By a precondition axiom, we have $Poss(a_1, s_1)$. So by the Theorem above, $Rdo(R_{\text{air}}, s, s')$ if $Rdo(R_1, do(a_1, s_1), s')$.
2. Let $a_2 = check\_departures$ and $s_2 = do(a_1, s_1)$. R_1 is of the form $\underline{branch}(a_2, R_{2a}, R_{2b})$. By the successor state axiom for At, we have $At(airport, s_2)$, and so by a precondition axiom, we have $Poss(a_2, s_2)$. By the successor state axiom for $Parked$, we have $Parked(Flight123, gateA, s_2)$, and so $SF(check\_departures, s_2)$. So by the Theorem, $Rdo(R_1, s_2, s')$ if $Rdo(R_{2a}, do(a_2, s_2), s')$.
3. Let $a_3 = go(gateA)$ and $s_3 = do(a_2, s_2)$. R_{2a} is of the form $\underline{seq}(a_3, R_3)$. By the successor state axiom for At, we have $At(airport, s_3)$, and so by a precondition axiom, we have $Poss(a_3, s_3)$. By the successor state axiom for $Parked$, we have $Parked(Flight123, gateA, s_3)$. So by the Theorem, $Rdo(R_{2a}, s_3, s')$ if $Rdo(R_3, do(a_3, s_3), s')$.
4. Let $a_4 = board\_plane(Flight123)$ and $s_4 = do(a_3, s_3)$. R_3 is the robot program $\underline{seq}(a_4, \underline{nil})$. By the successor state axiom for At, we have $At(gateA, s_4)$, and by the successor state axiom for $Parked$, we have $Parked(Flight123, gateA, s_4)$. Thus, by a precondition axiom, we have $Poss(a_4, s_4)$. So by the Theorem, $Rdo(R_3, s_4, s')$ if $s' = do(a_4, s_4)$. Moreover, for this s' we have by the successor state axiom for On_plane that $On\_plane(Flight123, s')$.

[10]In the following, for convenience, we will systematically be confusing use with mention: we will be saying that p where p is a logical sentence, meaning that it is true in any interpretation satisfying the above axioms.

Putting all the pieces together, we can see that for any s such that $Parked(Flight123, gateA, s)$, there is an s' such that $Rdo(R_{\text{air}}, s, s')$ and $On\_plane(Flight123, s')$, namely

$$s' = do([go(airport), check\_departures, go(gateA), board\_plane(Flight123)], s).$$

The case where $Parked(Flight123, gateB, s)$ is completely analogous, but leads to

$$s' = do([go(airport), check\_departures, go(gateB), board\_plane(Flight123)], s).$$

Note that in each case there also exists a sequence of actions not containing *check_departures* that puts the agent on Flight 123. However, no robot program without sensing would be able to generate *both* cases.

We can also consider what happens if the agent knows initially where the plane is parked:

- Initial State:
  $\forall s.K(s, S_0) \supset Parked(Flight123, gateB, s)$.

The argument above shows that R_{air} continues to work in this context even with the redundant sensing (there is only one case to consider now). The same argument also shows that if we replace the $\underline{seq}(go(gateA), \ldots)$ part in R_{air} by anything at all, the program still works. Of course, the program with no sensing would work here too.

Observe that the derivation above does not make use of the successor state axiom for K. This is because the agent was not required to know anything in the final state. It is not hard to prove that not only does R_{air} achieve the goal $On\_plane(Flight123, s')$, it also achieves $Know(On\_plane(Flight123), s')$. We can also imagine new primitive actions that depend on knowledge preconditions, such as "going to the gate of a flight," which can only be executed if the agent knows where the plane is parked:

$$Poss(go\_gate(p), s) \equiv At(airport, s) \wedge \exists x.Know(Parked(p, x), s).$$

With a suitable modification to the successor state axiom for At to accommodate this new action, an argument like the one above shows that the robot program

$$\underline{seq}(go(airport), \underline{seq}(check\_departures, \underline{seq}(go\_gate(Flight123), \underline{seq}(board\_plane(Flight123), \underline{nil})))),$$

with no conditional branching, achieves the goal. This shows that whether a plan containing sensing needs to be conditional depends on the primitive actions available.

One clear advantage of a specification like ours is that in being independent of any planner, it gives us the freedom to look at plans like these that might never be generated by a planner. This is especially useful if we are using a *plan critic* of some sort to modify an existing plan to reduce cost, or risk, or perhaps just to make sensing actions happen as early as possible. Plan correctness is not tied to any assumptions about how the plan was produced.

Are robot programs enough?

Given the extreme simplicity of the robot program language $\mathcal{R}$, and given that the planning task is defined in terms of the existence of robot programs, one might reasonably wonder if the restriction to $\mathcal{R}$ rules out goals that are intuitively achievable. Consider the following two examples:

The Odd Good Eggs Example The setup is exactly like the Omelette example, except that there is an additional sensing action, which tells you when the supply of eggs is exhausted. The goal is to have a single good egg in the bowl, but only if the supply contains an odd number of good eggs; otherwise, the bowl should remain empty.

The More Good Eggs Example The setup is as above. The goal now is to have a single good egg in the bowl, but only if the supply contains more good eggs than bad; otherwise, the bowl should be empty.

These are unusual goals, admittedly. But they do show that it is possible to encode language-recognition problems (over strings of eggs!) in a robotic setting. Informally, both goals are achievable in that we can imagine physical devices that are able to do so. The formal claim here is this: there is a robot program that achieves the first goal (which we omit for space reasons), but there is provably none that achieves the second. The proof is essentially the proof that a finite automaton cannot recognize the language consisting of binary strings with more 1's than 0's. To do so, you need the equivalent of a counter.

To preserve the simple structure of $\mathcal{R}$, we augment our set of primitive actions to give the robot a *memory*. Thus, we assume that apart from those of the background theory, we have 5 special actions, *left*, *right*, *mark*, *erase*, *read_mark*, and two special fluents *Marked*, *pos*, characterized by the following axioms:

1. Precondition: the 5 actions are always possible
$Poss(left, s) \wedge Poss(right, s) \wedge Poss(mark, s)$
$\quad \wedge Poss(erase, s) \wedge Poss(read\_mark, s)$;
2. Successor state: only *erase* and *mark* change *Marked*
$Poss(a, s) \supset \{Marked(n, do(a, s)) \equiv$
$\quad a = mark \wedge pos(s) = n \quad \vee$
$\quad Marked(n, s) \wedge \neg[a = erase \wedge pos(s) = n]\}$;
3. Successor state: only *left* and *right* change the *pos* fluent
$Poss(a, s) \supset \{pos(do(a, s)) = n \equiv$
$\quad a = left \wedge pos(s) = n + 1 \quad \vee$
$\quad a = right \wedge pos(s) = n - 1 \quad \vee$
$\quad pos(s) = n \wedge a \neq left \wedge a \neq right\}$;
4. Sensed fluent: *read_mark* tells the agent whether the current position is marked
$SF(left, s) \wedge SF(right, s) \wedge SF(erase, s) \wedge SF(mark, s) \wedge$
$\quad [SF(read\_mark, s) \equiv Marked(pos(s), s)]$.

These axioms ensure that the 5 special actions provide the robot with what amounts to a Turing machine tape. The idea is that when solving a planning task wrt a background theory Σ, we look for a robot program that works wrt $(\Sigma \cup TM)$, where *TM* is the set of axioms above. We can then prove that the More Good Eggs example is now solvable (again, we omit the program).

We believe that no further extensions to $\mathcal{R}$ will be needed. However, to prove this, we would want to show that any "effectively achievable" goal can be achieved by some robot program. But this requires an independent account of effective achievability, that is, an analogue of computability for robots over a domain-dependent set of actions whose effects are characterized by a set of axioms. To our knowledge, no such account yet exists, so we are developing one.

Conclusion

One limitation of the work presented here is that it offers no suggestions about how to automatically generate plans like those above in a reasonable way. Of course, our specification does provide us with a planning procedure (of sorts):

```
**Planning Procedure** ($\phi$) {
    **repeat with** $r \in \mathcal{R}$ {
        **if** $Axioms \models \forall s.K(s, S_0) \supset$
                $\exists s' [Rdo(r, s, s') \wedge \phi(s')]$
            **then** return $r$ }}
```

We can also think of the r as being returned by answer extraction [5] from an attempt to prove the following:

$$Axioms \models \exists r \forall s.K(s, S_0) \supset \exists s' [Rdo(r, s, s') \wedge \phi(s')]$$

Either way, the procedure would be problematic: we are searching blindly through the space of all possible robot programs, and for each one, the constraint to check involves using the K fluent explicitly as well as the (second-order!) *Rdo* formula. However, we do not want to suggest that a specification of the planning task ought to be used this way to generate plans. Indeed, our criticism of earlier accounts was precisely that they were overly tied up with specific planning procedures.

In our own work in Cognitive Robotics, we take a slightly different approach. Instead of planning tasks, we focus on the execution of high-level programs written in the GOLOG programming language [7]. GOLOG programs look like ordinary block-structured imperative programs except that they are nondeterministic, and they use the primitive actions and fluents of a user-supplied domain theory. There is a formula of the situation calculus $Do(\delta, s, s')$, analogous to *Rdo*, which says that s' is one of potentially many terminating situations of GOLOG program δ when started in initial situation s. To execute δ (when there are no sensing actions), a GOLOG processor must first find a legal sequence of primitive actions $\vec{a}$ such that

$$Axioms \models Do(\delta, S_0, do(\vec{a}, S_0)),$$

which it can then pass to a robot for actual execution. This is obviously a special case of planning. Furthermore, when δ contains sensing actions, an argument analogous to the one presented here suggests that instead of $\vec{a}$, the GOLOG processor would need to find a robot program r [8].

With or without sensing, considerable searching may be required to do this type of processing. To illustrate an extreme case, the GOLOG program

$$\textbf{while } \neg Goal \textbf{ do } (\pi\, a)[Appropriate(a)?;\ a] \textbf{ end},$$

repeatedly selects an appropriate action and performs it until some goal is achieved. Finding a sequence of actions in this case is simply a reformulation of the planning problem. However, the key point here is that at the other extreme, when the GOLOG program is fully deterministic, execution can be extremely efficient since little or no searching is required. The hope is that many useful cases of high-level agent control will lie somewhere between these two extremes.

A major representational limitation of the approach presented here concerns the binary sensing actions and the desire to avoid mentioning fluents in a robot program. Sensing actions that return one of a small set of values (such as reading a digit on a piece of paper, or detecting the colour of an object) can be handled readily by a case-like construct. Even a large or infinite set might be handled, if the values can be ordered in a natural way.

But suppose that sensing involves reading from a noisy sensor, so that instead of returning (say) the distance to the nearest wall, we get a number from a sensor that is only correlated with that distance. An account already exists of how to characterize in the situation calculus such sensing actions, and the effect they have not on knowledge now, but on degrees of belief [1]. However, how robot programs or planning could be defined in terms of this account still remains to be seen.

References

[1] F. Bacchus, J. Halpern, and H. Levesque. Reasoning about noisy sensors in the situation calculus. In *Proc. of IJCAI-95*, pp. 1933–1940, Montréal, August 1995. Morgan Kaufmann Publishing.

[2] E. Davis. Knowledge preconditions for plans. Technical Report 637, Computer Science Department, New York University, 1993.

[3] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pp. 115–125, Cambridge, MA, 1992. Morgan Kaufmann Publishing.

[4] O. Etzioni and S. Hanks. Personal comm., 1995.

[5] Cordell C. Green. Theorem proving by resolution as a basis for question-answering systems. In *Machine Intelligence 4*, pp. 183–205. Edinburgh University Press, 1969.

[6] F. Lin and R. Reiter. How to progress a database II: The STRIPS connection. In *Proceedings of IJCAI-95*, pp. 2001–2007, Montreal, Aug. 20-25, 1995.

[7] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. To appear in the *Journal of Logic Programming*, 1996.

[8] Hector J. Levesque. The execution of high-level robot programs with sensing actions: theory and implementation. In preparation, 1996.

[9] K. Krebsbach, D. Olawsky, and M. Gini. An empirical study of sensing and defaulting in planning. In *Proc. of 1st Conference on AI Planning Systems*, pp. 136–144, San Mateo CA, 1992.

[10] Z. Manna and R. Waldinger. How to clear a block: A theory of plans. *Journal of Automated Reasoning*, 3:343–377, 1987.

[11] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proc. of AAAI-91*, pp. 634–639, Menlo Park, CA, July 1991.

[12] J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press, 1969.

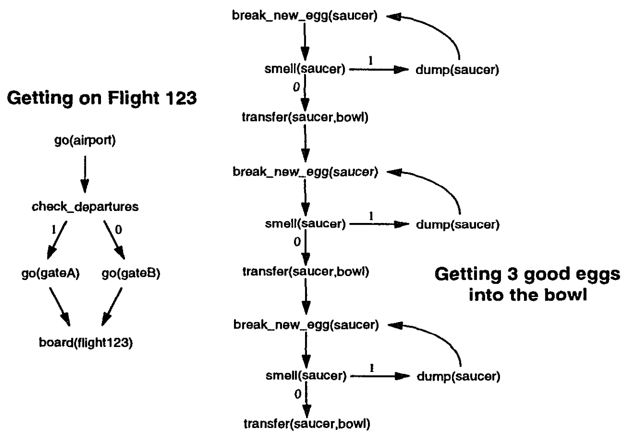
[13] R. C. Moore. A formal theory of knowledge and action. In J. R. Hobbs and R. C. Moore, editors, *Formal theories of the common sense world*, pp. 319–358. Ablex Publishing, Norwood, NJ, 1985.

[14] M. Peot and D. Smith. Conditional nonlinear planning. In *Proc. of 1st Conference on AI Planning Systems*, pp. 189–197, San Mateo CA, 1992.

[15] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pp. 359–380. Academic Press, San Diego, CA, 1991.

[16] R. Scherl and H. Levesque. The frame problem and knowledge-producing actions. In *Proc. of AAAI-93*, pp. 689–695, Washington, DC, July 1993. AAAI Press/The MIT Press.

[17] M. Schoppers. Building plans to monitor and exploit open-loop and closed-loop dynamics. In *Proc. of 1st Conference on AI Planning Systems*, pp. 204–213, San Mateo CA, 1992.


Getting on Flight 123
go(airport)
check_departures
1
0
go(gateA)
go(gateB)
board(flight123)
break_new_egg(saucer)
smell(saucer)
1
dump(saucer)
0
transfer(saucer,bowl)
break_new_egg(saucer)
smell(saucer)
1
dump(saucer)
0
transfer(saucer,bowl)
break_new_egg(saucer)
smell(saucer)
1
dump(saucer)
0
transfer(saucer,bowl)
Getting 3 good eggs into the bowl