An Incremental Interpreter for High-Level Programs with Sensing

Giuseppe De Giacomo[1] and Hector J. Levesque[2]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
degiacomo@dis.uniroma1.it

[2] Department of Computer Science
University of Toronto
Toronto, Canada M5S 3H5
hector@cs.toronto.edu

Abstract. Like classical planning, the execution of high-level agent programs requires a reasoner to look all the way to a final goal state before even a single action can be taken in the world. This deferral is a serious problem in practice for large programs. Furthermore, the problem is compounded in the presence of sensing actions which provide necessary information, but only after they are executed in the world. To deal with this, we propose (characterize formally in the situation calculus, and implement in Prolog) a new incremental way of interpreting such high-level programs and a new high-level language construct, which together allow much more control to be exercised over when actions can be executed. We argue that such a scheme leads to a practical way to deal with large agent programs containing both nondeterminism and sensing.

1 Introduction

The research reported in this paper is strongly based on Ray Reiter's work on the situation calculus, on the frame problem, and on cognitive robotics.

In [14] it was argued that when it comes to providing high level control to autonomous agents or robots, the notion of *high-level program execution* offers an alternative to classical planning that may be more practical in many applications. Briefly, instead of looking for a sequence of actions $\boldsymbol{a}$ such that

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

where ϕ is the goal being planned for, we look for a sequence $\boldsymbol{a}$ such that

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

where δ is a high-level program and $Do(\delta, s, s')$ is a formula stating that δ may legally terminate in state s' when started in state s. By a high-level program here, we mean one whose primitive statements are the domain-dependent actions of some agent or robot, whose tests involve domain-dependent fluents (that are caused to hold or not hold by the primitive actions), and which contains nondeterministic choice points where reasoned (non-random) choices must be made about how the execution should proceed.

What makes a high-level agent program different from a deterministic "script" is that its execution is a problem solving task, not unlike planning. An interpreter needs to use what it knows about the prerequisites and effects of actions to find a sequence with the right properties. This can involve considerable search when δ is very nondeterministic, but much less search when δ is more deterministic. The feasibility of this approach for AI purposes clearly depends on the expressive power of the programming language in question. In [14], a language called GOLOG is presented, which in addition to nondeterminism, contains facilities for sequence, iteration, and conditionals. In this paper, we extend the expressive power of this language by providing much finer control over the nondeterminism, and by making provisions for sensing actions. To do so in a way that will be practical even for very large programs requires introducing a different style of on-line program execution.

In the rest of this section, we discuss on-line and off-line execution informally, and show why sensing actions and nondeterminism together can be problematic. In the following section, we formally characterize program execution in the language of the situation calculus. Next, we describe an incremental interpreter in Prolog that is correct with respect to this specification. The final section contains discussion and conclusions.

1.1 Off-line and On-line execution

To be compatible with planning, the GOLOG interpreter presented in [14] executes in an *off-line* manner, in the sense that it must find a sequence of actions constituting an entire legal execution of a program *before* actually executing any of them in the world.[1] Consider, for example, the following program:

$$(a|b)\ ;\Delta\ ;p?$$

where a and b are primitive actions, $|$ indicates nondeterministic choice, Δ is some very large deterministic program, and $p?$ tests whether fluent p holds. A legal sequence of actions should start with either a or b, followed by a sequence for Δ, and end up in state where p holds. Before executing a or b, the agent or robot must wait until the interpreter considers all of Δ and determines which initial action eventually leads to p. Thus even a single nondeterministic choice occurring early in a large program can result in an unacceptable delay. We will see below that this problem is compounded in the presence of sensing actions.

If a small amount of nondeterminism in a program is to remain practical (as suggested by [14]), we need to be able to choose between a and b based on

[1] It is assumed that once an action is taken, it need not be undoable, and so backtracking "in the world" is not an option.

some local criterion without necessarily having to go through all of Δ. Using something like

$$(a|b)\ ; r?\ ; \Delta\ ; p?$$

here does not work, since an off-line interpreter cannot settle for a even if it leads to a state where r holds. We need to be able to *commit* to a choice that satisfies r, with the understanding that it is the responsibility of the programmer to use an appropriate local criterion, and that the program will simply fail without the option of backtracking if p does not hold at the end.

It is convenient to handle this type of commitment by changing the execution style from off-line to on-line, but including a special off-line search operator. In a *on-line* execution, nondeterministic choices are treated like random ones, and any action selected is executed immediately. So if the program

$$(a|b)\ ; \Delta\ ; p?$$

is executed on-line, one of a or b is selected and executed immediately, and the process continues with Δ; in the end, if p happens not to hold, the entire program fails. We use a new operator Σ for search, so that $\Sigma\delta$, where δ is any program, means "consider δ off-line, searching for a successful termination state". With this operator, we can control how nondeterminism will be handled. To execute

$$\Sigma\{(a|b)\ ; r?\}\ ; \Delta\ ; p?$$

on-line, we would search for an a or b that successfully leads to r, execute it immediately, and then continue boldly with Δ. In this scheme, it is left to the programmer to decide how cautious to be. If the programmer drops the search operator completely thus writing

$$(a|b)\ ; r?; \Delta\ ; p?$$

then the choice between a and b will be random. If the programmer puts the entire program within a Σ operator, thus writing

$$\Sigma\{(a|b)\ ; r?; \Delta\ ; p?\}$$

then the choice between a and b will be based on full lookahead to the end of the program, *i.e.*, the program will be executed essentially in the old way.

1.2 Sensing actions

This on-line style of execution is well-suited to programs containing sensing actions. As described in [9,13,20], sensing actions are actions that can be taken by the agent or robot to obtain information about the state of certain fluents, rather than to change them. The motivation for sensing actions involves applications where because the initial state of the world is incompletely specified or because of hidden exogenous actions, the agent must use sensors of some sort to determine the value of certain fluents.

Suppose, for example, that nothing is known about the state of some fluent q, but that there is a binary sensing action $read_q$ which uses a sensor to tell the robot whether or not q holds. To execute the program

$$a\ ;read_q\ ;\mathbf{if}\ q\ \mathbf{then}\ \Delta_1\ \mathbf{else}\ \Delta_2\ \mathbf{endIf}\ ;p?$$

the interpreter would get the robot to execute a in the world, get it to execute $read_q$, then use the information returned to decide whether to continue with Δ_1 or Δ_2. But consider the program

$$(a|b)\ ;read_q\ ;\mathbf{if}\ q\ \mathbf{then}\ \Delta_1\ \mathbf{else}\ \Delta_2\ \mathbf{endIf}\ ;p?.$$

An off-line interpreter cannot commit to a or b in advance, and because of that, cannot use $read_q$ to determine if q would hold after the action. The only option available is to see if one or a or b would lead to p for *both* values of q. This requires considering both Δ_1 and Δ_2, even though in the end, only one of them will be executed. Similarly, if we attempt to generate a low-level robot program (as suggested in [13] for planning in the presence of sensing), we end up having to consider both Δ_1 and Δ_2.

The situation is even worse with loops. Consider

$$(a|b)\ ;read_q\ ;\mathbf{while}\ q\ \mathbf{do}\ \Delta\ ;read_q\ \mathbf{endWhile}\ ;p?.$$

Since an off-line interpreter has no way of knowing in advance how many iterations of the loop will be required to make q false, to decide between a and b, it would be necessary to reason about the effect of performing Δ an *arbitrary* number of times (by discovering loop invariants *etc.*). But if a commitment could be made to one of them on local grounds we could modify the program as follows

$$\Sigma\{(a|b);r?\}\ ;read_q\ ;\mathbf{while}\ q\ \mathbf{do}\ \Delta\ ;read_q\ \mathbf{endWhile}\ ;p?.$$

and then use $read_q$ to determine the actual value of q, and it would not be necessary to reason about the deterministic loop. It therefore appears that an on-line execution style is often more practical for large programs containing nondeterminism and sensing actions.

2 Preliminaries

The technical machinery we use to define on-line program execution in the presence of sensing is based on that of [4], *i.e.*, we use the predicates *Trans* and *Final* to define a single step semantics of programs [10,17]. However some adaptation is necessary to deal with on-line execution, sensing results, and the Σ operator.

2.1 Situation calculus

The starting point in the definition is the situation calculus [16]. 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; there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s; finally, following [13], there is a special predicate $SF(a, s)$ used to state that action a would return the binary sensing result 1 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 an action theory of the following form [18]:

- Axioms describing the initial situation, S_0. Note that there can be fluents like q about which nothing is known in the initial state.
- Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.
- Successor state axioms, one for each fluent F,[2] stating under what conditions $F(\boldsymbol{x}, do(a, s))$ 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 [18].
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

Finally, as in [13], we include

- Sensed fluent axioms, one for each primitive action a of the form $SF(a, s) \equiv \phi_a(s)$, characterizing SF.

For the sensing action $read_q$ used above, we would have $[SF(read_q, s) \equiv q(s)]$, and for any ordinary action a that did not involve sensing, we would use $[SF(a, s) \equiv \mathbf{true}]$.

2.2 Histories

To describe a run which includes both actions and their sensing results, we use the notion of a history. By a *history* we mean a sequence of pairs (a, x) where a is a primitive action and x is 1 or 0, a sensing result. Intuitively, the history $(a_1, x_1) \cdot \ldots \cdot (a_n, x_n)$ is one where actions $a_1, \ldots, a_n$ happen starting

[2] A fluent whose current value could only be determined by sensing would normally not have a successor state axiom. However, see [7] for a proposal that overcomes this limitation.

in some initial situation, and each action a_i returns sensing value x_i. The assumption is that if a_i is an ordinary action with no sensing, then $x_i = 1$. Notice that the empty sequence ϵ is a history.

Histories are not terms of the situation calculus. It is convenient, however, to use $end[\sigma]$ as an abbreviation for the situation term called the *end situation* of history σ on the initial situation S_0, and defined by: $end[\epsilon] = S_0$; and inductively, $end[\sigma \cdot (a,x)] = do(a, end[\sigma])$.

It is also useful to use $Sensed[\sigma]$ as an abbreviation for a formula of the situation calculus, the *sensing results* of a history, and defined by: $Sensed[\epsilon] =$ **true**; and inductively, $Sensed[\sigma \cdot (a,1)] = Sensed[\sigma] \wedge SF(a, end[\sigma])$, and $Sensed[\sigma \cdot (a,0)] = Sensed[\sigma] \wedge \neg SF(a, end[\sigma])$. This formula uses SF to tell us what must be true for the sensing to come out as specified by σ starting in the initial situation S_0.

2.3 The *Trans* and *Final* predicates

In [4] two special predicates *Trans* and *Final* were introduced. $Trans(\delta, s, \delta', s')$ was intended to say that by executing program δ starting in situation s, one can get to situation s' in one elementary step with the program δ' remaining to be executed, that is, there is a possible transition from the configuration (δ, s) to the configuration (δ', s'). $Final(\delta, s)$, instead, was intended to say that program δ may successfully terminate in situation s, that is, the configuration (δ, s) is final.

For example, the transition requirements for sequence is

$$\begin{array}{l} Trans([\delta_1;\delta_2], s, \delta', s') \quad \equiv \\ \quad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s') \quad \vee \\ \quad\quad \exists \gamma'.\ Trans(\delta_1, s, \gamma', s') \wedge \delta' = (\gamma'; \delta_2). \end{array}$$

This says that to single-step the program $(\delta_1; \delta_2)$, either δ_1 terminates and we single-step δ_2, or we single-step δ_1 leaving some γ', and $(\gamma'; \delta_2)$ is what is left of the sequence.

For our account here, we adopt the definitions of *Trans* and *Final* in [4] (the details of which we omit).[3]

3 Off-line lookahead

On-line executions are characterized by the fact that the robot at each step makes a transition chosen among those that are legal (we'll see later in which precise sense) and execute it in the real world. In executing it the robots commits to the transition chosen since there is no possibility of undoing it. This is to be contrasted with the off-line execution mode where we commit

[3] For an in-depth study of *Trans* and *Final*, including a suitable treatment of procedures and constructs for concurrency, can be found in [5,6].

to a sequence of actions to be executed only after having shown that the sequence is guaranteed to terminate successfully.

Since when we execute a program on-line there is no possibility of backtracking, how to select among the possible transitions the one to execute is a critical step. How should this selection be done? As discussed in Section 1, one extreme possibility is to make a random choice. Obviously this selection mechanism is very efficient, however the choice made may compromise the successful termination of the program. On the other extreme we may require the robot to be very cautious and do full lookahead to the end of the program to make a transition only if it is guaranteed to lead to a successful termination of the program. Naturally, this is quite heavy computationally. [4]

Here we introduce a mechanism to have a controlled form of lookahead, so that the amount of lookahead to be performed is under the control of the programmer. Namely, we introduce a *search operator* in the programming language.

We define *Final* and *Trans* for the new operator as follows. For *Final*, we simply have that $(\Sigma\delta, s)$ is a final configuration of the program if (δ, s) itself is, and so we get the requirement

$$\mathit{Final}(\Sigma\delta, s) \quad\equiv\quad \mathit{Final}(\delta, s).$$

For *Trans*, we have that the configuration $(\Sigma\delta, s)$ can evolve to $(\Sigma\gamma', s')$ provided that (δ, s) can evolve to (γ', s') and from (γ', s') it is possible to reach a final configuration in a finite number of transitions. Thus, we get the requirement

$$\begin{array}{l}\mathit{Trans}(\Sigma\delta, s, \delta', s') \quad\equiv \\ \quad \exists\gamma'.\ \delta' = \Sigma\gamma' \wedge \mathit{Trans}(\delta, s, \gamma', s') \wedge \\ \qquad \exists\gamma'', s''.\ \mathit{Trans}^*(\gamma', s', \gamma'', s'') \wedge \mathit{Final}(\gamma'', s'').\end{array}$$

In this assertion, *Trans** is the reflexive transitive closure of *Trans*, defined by

$$\mathit{Trans}^*(\delta, s, \delta', s') \stackrel{def}{=} \forall T\,[\ldots \supset T(\delta, s, \delta', s')]$$

where the ellipsis stands for the conjunction of (the universal closure of)

$$\begin{array}{c}T(\gamma, s, \gamma, s)\\ T(\gamma, s, \gamma', s') \wedge \mathit{Trans}(\gamma', s', \gamma'', s'') \;\supset\; T(\gamma, s, \gamma'', s'').\end{array}$$

The semantics of Σ can be understood as follows: (1) $(\Sigma\delta, s)$ selects from all possible transitions of (δ, s) those from which there exists a sequence of further transitions leading to a final configuration; (2) the Σ operator is propagated through the chosen transition, so that this restriction is also

[4] In [8] on-line executions were considered to allow execution monitoring and recovery. There a brave and a cautions interpreter were defined which corresponded to exactly to these two extreme approaches.

performed on successive transitions. In other words, within a Σ operator, we only take a transition from δ to γ', if γ' is on a path that will eventually terminate successfully, and from γ' we do the same. As desired, Σ does an off-line search before committing to even the first transition.

It is not too hard to prove that Σ has some intuitively plausible properties. In particular we have the following ones:

Property 1.

$$Trans(\Sigma\delta, s, \delta', s') \supset \exists\gamma.\delta' = \Sigma\gamma$$

i.e., a program of the form $\Sigma\delta$ can evolve only to programs of the form $\Sigma\delta'$. In other words, the search operator is indeed propagated through the transition.

Property 2.

$$Trans(\Sigma\delta, s, \Sigma\delta', s') \equiv Trans(\delta, s, \delta', s') \wedge \exists s''.Do(\delta', s', s'')$$

i.e., in performing a transition step for a configuration $(\Sigma\delta, s)$ we are in fact performing a transition from (δ, s) and verifying that such transition leads to successful termination.

Property 3.

$$\begin{array}{l} Trans(\Sigma\Sigma\delta, s, \Sigma\Sigma\delta', s') \equiv Trans(\Sigma\delta, s, \Sigma\delta', s') \\ Final(\Sigma\Sigma\delta, s) \equiv Final(\Sigma\delta, s) \end{array}$$

i.e., nesting search operators is equivalent to apply the search operator only once.

4 Characterizing on-line executions

The on-line execution of a program consists of a suitable sequence of *legal* single-step transitions. We distinguish the case where we do not have sensing from the one in which we do.

4.1 Without sensing

In the absence of sensing, we say that a δ can be executed in the current situation s leading to the new situation s' with program δ' that remains to be executed, only when

$$Axioms \models Trans(\delta, s, \delta', s')$$

i.e., a transition step is legal if only if is logically implied by $Axioms$. In this way we capture the intuition that a transition is legal if on the base of our knowledge (as expressed by $Axioms$) we are certain that the transition can be executed. Analogously, we are allowed to successfully terminate a program δ in s when

$$Axioms \models \mathit{Final}(\delta, s)$$

i.e., δ can legally terminate in s if and only if it is logically implied by $Axioms$.

Hence, without sensing an on-line execution of a program δ starting from a situation s is a sequence $(\delta_0 = \delta, s_0 = s), \ldots, (\delta_n, s_n)$ such that for $i = 0, \ldots, n-1$:

$$Axioms \models \mathit{Trans}(\delta_i, s_i, \delta_{i+1}, s_{i+1}).$$

An on-line execution is successful if

$$Axioms \models \mathit{Final}(\delta_n, s_n).$$

It is possible to prove the following theorem.

Theorem 1. *If there exists a successful on-line execution* $(\delta_0 = \delta, s_0 = s), \ldots, (\delta_n, s_n = s')$ *of a program* δ *in the situation* s *leading to* s'*, then there exists a successful off-line execution of* δ *in* s *leading to* s'*,* i.e.*,*

$$Axioms \models Do(\delta, s, s')$$

The converse of this theorem does not hold, since an on-line execution requires all transition steps to be logically implied by $Axioms$, while an off-line execution does not. For example, consider the program $\delta = \phi; a \mid \neg\phi; a$, where $Axioms \not\models \phi[S_0]$ and $Axioms \not\models \neg\phi[S_0]$. δ executed in S_0 has a successful off-line execution, namely $Axioms \models Do(\delta, S_0, do(a, S_0))$. But it has no successful on-line executions, since there are no transitions logically implied by $Axioms$.

We do not impose any selection criteria on on-line executions. The robot at each step makes a legal transition that is randomly chosen. Thus we cannot guarantee that the robot follows a successful on-line execution a priori. We can however make use of the search operator for giving the robot the possibility, under the control of the programmer, of doing some lookahead and avoid dead-end executions. Indeed by Property 2 above we have that:

$$Axioms \models \mathit{Trans}(\Sigma\delta, s, \Sigma\delta', s')$$

if and only if

$$Axioms \models \mathit{Trans}(\delta, s, \delta', s') \text{ and } Axioms \models \exists s''.Do(\delta', s', s'')$$

i.e., there is a legal transition from $(\Sigma\delta, s)$ to $(\Sigma\delta', s')$ if and only if (i) there is a legal transition from (δ, s) to (δ', s'), and (ii) there exists an execution of δ' in s' that successfully terminates.

By Theorem 1, if $Axioms \not\models \exists s''.Do(\delta', s', s'')$ then there are no successful on-line executions of δ' in s'. It follows that applying the search operator to a program δ we prune potential on-line executions that are bound to be unsuccessful.[5]

[5] Note that, although we can guarantee the existence of a successful execution, it is not always the case that we can actually find an successful *on-line execution*,

4.2 With sensing

First we observe that we did not require special axioms for *Trans* and *Final* in order to deal with sensing. Sensing actions are just like ordinary actions in all respects except for what specified by the sensed fluent axioms involving SF. However, the existence of a given legal transition may now depend on the values sensed so far. That is, if s is $end[\sigma]$ where σ is the history of actions and sensing values starting from the initial situation S_0, then δ in s can make a legal transition leading to s' with program δ' that remains to be executed when

$$Axioms \cup \{Sensed[\sigma]\} \models Trans(\delta, s, \delta', s').$$

In other words, now we are looking for a transition that is logically implied by *Axioms* together with *the values sensed so far.*

In executing the next step we can take into account that the transition may have result in getting some new information from the sensors. Specifically, if the transition did not result in any action, *i.e.*, $s' = s$,[6] then we still consider logical implication from $Axioms \cup \{Sensed[\sigma]\}$. If, instead, an action a was performed and the value x returned, then we consider logical implication from $Axioms \cup \{Sensed[\sigma']\}$ where $\sigma' = \sigma \cdot (a, x)$, *i.e.*, we consider the value returned by action a as well.

Similarly, we are allowed to terminate the program δ successfully if

$$Axioms \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma]),$$

where again the history σ is taken into account.

Thus, in presence of sensing, an on-line execution of a program δ starting from a situation $end[\sigma]$ is a sequence $(\delta_0 = \delta, \sigma_0 = \sigma), \ldots, (\delta_n, \sigma_n)$ such that for $i = 0, \ldots, n-1$:

$$Axioms \cup \{Sensed[\sigma_i] \models Trans(\delta_i, end[\sigma_i], \delta_{i+1}, end[\sigma_{i+1}])$$

$$\sigma_{i+1} = \begin{cases} \sigma_i & \text{if } end[\sigma_{i+1}] = end[\sigma_i] \\ \sigma_i \cdot (a, x) & \text{if } end[\sigma_{i+1}] = do(a, end[\sigma_i]) \text{ and } a \text{ returns } x \end{cases}$$

An on-line execution is successful if

$$Axioms \cup \{Sensed[\sigma_n]\} \models Final(\delta_n, end[\sigma_n]).$$

Note that, if no sensing action is performed then $Sensed[\sigma]$ becomes equivalent to **true**, and hence the specification correctly reduces to the specification from before.

Finally, let us focus on the meaning the search operator in the context of on-line executions in presence of sensing. By Property 2 above we have that:

which in fact requires the existence of a sequence of transitions that are logically implied by *Axioms*.

[6] Such "null transitions" arise from tests in the program.

$$Axioms \cup \{Sensed[\sigma]\} \models Trans(\Sigma\delta, s, \Sigma\delta', s')$$

if and only if

$$\begin{array}{l} Axioms \cup \{Sensed[\sigma]\} \models Trans(\delta, end[\sigma], \delta', s') \text{ and} \\ Axioms \cup \{Sensed[\sigma]\} \models \exists s''.Do(\delta', s', s'') \end{array}$$

i.e., in looking for the existence of a successful execution of δ' in s', we obviously do not take into account how the sensing values will turn out to be (we will know these values only when we actually execute the actions in the transitions). Hence now Theorem 1 implies that if $Axioms \cup \{Sensed[\sigma]\} \not\models \exists s''.Do(\delta', s', s'')$ then there, are no successful on-line executions of δ' in s' *that do not gather new information by sensing.* It follows that applying the search operator to a program δ we prune potential on-line executions that depend on how sensing turns out in order to be successful.

5 An incremental interpreter

Next we present a simple incremental interpreter in Prolog. Although the on-line execution task characterized above no longer requires search to a final state, it remains fundamentally a theorem-proving task: does a certain *Trans* or *Final* formula follow logically from the axioms of the action theory together with assertions about sensing results?

The challenge in writing a practical interpreter is to find cases where this theorem-proving can be done using something like ordinary Prolog evaluation. The interpreter in [4] as well as in earlier work on which it was based [14] was designed to handle cases where what was known about the initial situation S_0 could be represented by a set of atomic formulas together with a closed-world assumption. In the presence of sensing, however, we cannot simply apply a closed-world assumption blindly. As we will see, we can still avoid full theorem-proving if we are willing to assume that a program executes appropriate sensing actions prior to any testing it performs. In other words, our interpreter depends on a *just-in-time history assumption*[7] where it is assumed that *whenever a test is required, the on-line interpreter* at that point *has complete knowledge of the fluents in question to evaluate the test without having to reason by cases* etc.

5.1 The main loop

As it turns out, most of the subtlety in writing such an interpreter concerns the evaluation of tests in a program. The rest of the interpreter derives almost

[7] The notion of just-in-time histories is investigated further in [7].

directly from the axioms for *Final*, and *Trans* described above. It is convenient, however, to use an implementation of these predicates defined over encodings of histories (with most recent actions first) rather than situations. We get

```
/*  P is a program                */
/*  H is a history, initially []  */
/*    H ::= [] | [(Act,1/0)|H]    */

incrInterpret(P,H) :- final(P,H).
incrInterpret(P,H) :-
   trans(P,H,P1,[(Act,_)|H]), !, execute(Act,Sv),
   incrInterpret(P1,[(Act,Sv)|H]).
incrInterpret(P,H) :-
   trans(P,H,P1,H), !,
   incrInterpret(P1,H).
```

So to incrementally interpret a program on-line, we either terminate successfully, or we find a transition involving some action, commit to that action, execute it in the world to obtain a sensing result, and then continue the interpretation with the remaining program and the updated history.[8] In looking for the next action, we skip over transitions involving successful tests where no action is required and the history does not change. To execute an action in the world, we connect to the sensors and effectors of the robot or agent. Here for simplicity, we just write the action, and read back a sensing result.

```
execute(Act,Sv) :-
   write(Act),
   (senses(Act,_) ->
      (write(':'), read(Sv)) ; (nl, Sv=1)).
```

We assume the user has declared using `senses` (described below) which actions are used for sensing, and for any action with no such declaration, we immediately return the value 1.

5.2 Implementing *Trans* and *Final*

Clauses for `trans` and `final` are needed for each of the program constructs. For example, for sequence, we have

```
trans(seq(P1,P2),H,P,H1) :-
   final(P1,H), trans(P2,H,P,H1).
trans(seq(P1,P2),H,seq(P3,P2),H1) :-
   trans(P1,H,P3,H1).
```

[8] In practice, we would not want the history list to get too long, and would use some form of “rolling forward” [15].

which corresponds to the axiom given earlier except for the use of histories instead of situations. We omit the details for the other constructs, except for Σ (search):

```
final(search(P),H) :- final(P,H).

trans(search(P),H,search(P1),H1) :-
   trans(P,H,P1,H1), ok(P1,H1).

ok(P,H) :- final(P,H).
ok(P,H) :- trans(P,H,P1,H), ok(P1,H).
ok(P,H) :- trans(P,H,P1,[(Act,_)|H]),
   (senses(Act,_) ->
       ( ok(P1,[(Act,0)|H]) ,
         ok(P1,[(Act,1)|H]) ) ;
       ok(P1,[(Act,1)|H])).
```

The auxiliary predicate `ok` here is used to handle the *Trans** and *Final* part of the axiom by searching forward for a final configuration.[9] Note that when a future transition involves an action that has a sensing result, we need the program to terminate successfully for *both* sensing values. This is clearly explosive in general: sensing and off-line search do not mix well. It is precisely to deal with this issue in a flexible way that we have taken an on-line approach, putting the control in the hands of the programmer.

5.3 Handling test conditions

The rest of the interpreter is concerned with the evaluation of test conditions involving fluents, given some history of actions and sensing results. We assume the programmer provides the following clauses:

- `poss(`*Act*`,`*Cond*`)`: the action is possible when the condition holds;
- `senses(`*Act*`,`*Fluent*`)`: the action can be used to determine the truth of the fluent;[10]
- `initially(`*Fluent*`)`: the fluent holds in the initial situation S_0;
- `causesTrue(`*Act*`,`*Fluent*`,`*Cond*`)`: if the condition holds, performing the action causes the fluent to hold;
- `causesFalse(`*Act*`,`*Fluent*`,`*Cond*`)`: if the condition holds, performing the action causes the fluent to not hold.

[9] In practice, a breadth-first search may be preferable. Also, we would want to cache the results of the search to possibly avoid repeating it at the next transition.

[10] The specification allows a sensor to be linked to an arbitrary formula using *SF*; the implementation insists it be a fluent.

In the absence of sensing, the last two clauses provide a convenient specification of a successor state axiom for a fluent F, as if we had (very roughly)

$$
\begin{aligned}
&F(do(a,s)) \;\equiv \\
&\quad \exists\phi(\mathit{causesTrue}(a,F,\phi) \wedge \phi[s]) \quad \vee \\
&\quad\; F(s) \;\wedge\; \neg\exists\phi(\mathit{causesFalse}(a,F,\phi) \wedge \phi[s]).
\end{aligned}
$$

In other words, F holds after a if a causes it to hold, or it held before and a did not cause it not to hold. With sensing, we have some additional possibilities. We can handle fluents that are completely unaffected by the given primitive actions by leaving out these two clauses, and just using sensing. We can also handle fluents that are partially affected. For example, in an elevator controller, it may be necessary to use sensing to determine if a button has been pushed, but once it has been pushed, we can assume the corresponding light stays on until we perform a reset action causing it to go off. We can also handle cases where some initial value of the fluent needs to be determined by sensing, but from then on, the value only changes as the result of actions, *etc.* Note that an action can provide information for one fluent and also cause another fluent to change values.

With these clauses, the transitions for primitive actions and tests would be specified as follows:

```
trans(prim(Act),H,nil,[(Act,_)|H]) :-
   poss(Act,Cond), holds(Cond,H).

trans(test(Cond),H,nil,H) :- holds(Cond,H).
```

where `nil` is the empty program. The `holds` predicate is used to evaluate arbitrary conditions. Because of the just-in-time histories assumption, the problem reduces to `holdsf` for fluents (we omit the reduction). For fluents, we have the following:

```
holdsf(F,[]) :- initially(F).

holdsf(F,[(Act,X)|H]) :-
   senses(Act,F),!, X=1. /* mind the cut */

holdsf(F,[(Act,X)|H]) :-
   causesTrue(Act,F,Cond), holds(Cond,H).

holdsf(F,[(Act,X)|H]) :-
   not ( causesFalse(Act,F,Cond), holds(Cond,H) ),
   holdsf(F,H).
```

Observe that if the final action in the history is not a sensing action, and not an action that causes the fluent to hold or not hold, we regress the test to the previous situation. This is where the just-in-time histories assumption:

for this scheme to work properly, the programmer must ensure that a sensing action and its result appear in the history as necessary to establish the current value of a fluent.

5.4 Correctness

This completes the incremental interpreter. The interpreter presented above is correct under suitable hypotheses. In particular, apart from the usual assumption required when we encoding an action theory in Prolog (see [19]), we make the hypothesis that the predicate `holds` satisfies the following properties.[11] Let δ and σ contain free variables only on objects and actions:

1. If a goal $\texttt{holds}(\phi, \sigma)$ succeeds with computed answer θ, then (by $\forall\psi$, we mean the universal closure of ψ)
$$Axioms \cup \{Sensed[\sigma]\} \models \forall\phi(end[\sigma])\theta.$$
2. If a goal $\texttt{holds}(\phi, \sigma)$ finitely fails, then
$$Axioms \cup \{Sensed[\sigma]\} \models \forall\neg\phi(end[\sigma]).$$

Although we do not attempt to show it formally here, it should be intuitively clear that our definition for `holds` under the just-in-time histories assumption does actually satisfy the requirements above.

We can formally state the correctness of the incremental interpreter as follows.

Theorem 2. *Let δ and σ contain free variables only on objects and actions. Then under the hypotheses above:*

1. *If a goal $\texttt{trans}(\delta, \sigma, \delta', \sigma')$ succeeds with computed answer θ, then*
$$Axioms \cup \{Sensed[\sigma]\} \models \forall\, Trans(\delta, end[\sigma]\delta', end[\sigma'])\theta$$
moreover $\delta'\theta$ and $\sigma'\theta$ contain free variables only on objects and actions.
2. *If a goal $\texttt{trans}(\delta, \sigma, \delta', \sigma')$ finitely fails, then*
$$Axioms \cup \{Sensed[\sigma]\} \models \forall\neg\, Trans(\delta, end[\sigma], \delta', end[\sigma']).$$
3. *If a goal $\texttt{final}(\delta, \sigma)$ succeeds with computed answer θ, then*
$$Axioms \cup \{Sensed[\sigma]\} \models \forall Final(\delta, \sigma)\theta.$$
4. *If a goal $\texttt{final}(\delta, \sigma)$ finitely fails, then*
$$Axioms \cup \{Sensed[\sigma]\} \models \forall\neg Final(\delta, end[\sigma]).$$

Notably, because of the assumption above on `holds` –and hence because of the just-in-time histories assumption– we have that if `trans` succeeds for a program of the form $(\Sigma\delta, s)$ then an on-line successful execution exists indeed.

[11] We keep implicit the translation between Prolog terms and the programs, histories, and terms of the situation calculus

6 Discussion

The framework presented here has a number of limitations beyond those already noted: it only deals with sensors that are binary and noise-free; no explicit mention is made of how the sensing influences the knowledge of the agent, as in [20]; the interaction between off-line search and concurrency is left unexplored; finally, the implementation has no finite way of dealing with search over a program with loops.

One of the main advantages of a high-level agent language containing nondeterminism is that it allows limited versions of (runtime) planning to be included within a program. Indeed, a simple planner can be written directly:[12]

$$\textbf{while } \neg\phi \textbf{ do } \pi a.\ (\mathit{Acceptable}(a)?\ ;\ a)\ \textbf{endWhile}.$$

Ignoring *Acceptable*, this program says to repeatedly perform some nondeterministically selected action until condition ϕ holds. An off-line execution would search for a legal sequence of actions leading to a situation where ϕ holds. This is precisely the planning problem, with *Acceptable* being used as a forward filter, in the style of [2].

However, in the presence of sensing, it is not clear how even limited forms of planning like this can be handled by an off-line interpreter, since a *single* nondeterministic choice can cause problems, as we saw earlier. The formalism presented here has more chances of being practical for large programs containing both nondeterministic action selection and sensing.

One concern one might have is that once we move to on-line execution where nondeterministic choice defaults to being random, we have given up reasoning about courses of action, and that our programs are now just like the pre-packaged "plans" found in RAP [3] or PRS [11]. Indeed in those systems, one normally does not search off-line for a sequence of actions that would eventually lead to some future goal; execution relies instead on a user-supplied "plan library" to achieve goals. In our case, with Σ, we get the advantages of both worlds: we can write agent programs that span the spectrum from scripts where no lookahead search is done and little needs to be known about the properties of the primitive actions being executed, all the way to full planners like the above. Moreover, our formal framework allows considerable generality in the formulation of the action theory itself, allowing disjunctions, existential quantifiers, *etc.* Even the Prolog implementation described here is considerably more general than many STRIPS-like systems, in allowing the value of fluents to be determined by sensing intermingled with the context-dependent effects of actions.

A more serious concern, perhaps, involves how to build an effective program to be executed on-line. There is a difficult tradeoff here that also shows up in the work on so-called *incremental planning* [1,12]. Even if we have an important goal that needs to be achieved in some distant place or time, we

[12] The π operator is used for a nondeterministic choice of value.

want to make choices here and now without worrying about it. How should I decide what travel agent to use given that I have to pick up a car at an airport in Amsterdam a month from now? The answer in practice is clear: decide locally and cross other bridges when you get to them, exactly the motivation for the approach presented here. It pays large dividends to assume by default that routine choices will not have distant consequences, chaos and the flapping of butterfly wings notwithstanding. But as far as we know, it remains an open problem to characterize formally what an agent would have to know (test/sense) to be able to quickly confirm that some action can be used immediately as a first step towards some challenging but distant goal.

References

1. J. A. Ambros-Ingerson and S. Steel. Integrating planning, execution and monitoring. In *Proc. AAAI-88*, 1988.
2. F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proc. AAAI-96*, 1996.
3. R. J. Firby. An investigation in reactive planning in complex domains. In *Proc. AAAI-87*, 1987.
4. G. De Giacomo, Y. Lespérance, and H. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proc. IJCAI-97*, 1997.
5. G. De Giacomo, Y. Lespérance, and H. Levesque. CONGOLOG, a concurrent programming language based on the situation calculus: language and implementation. submitted, 1999.
6. G. De Giacomo, Y. Lespérance, and H. Levesque. CONGOLOG, a concurrent programming language based on the situation calculus: foundations. submitted, 1999.
7. G. De Giacomo and H. Levesque. Progression and regression using sensors. In *Proc. IJCAI-99*, 1999.
8. G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *Proc. of KR-98*, pages 453–465, 1998.
9. K. Golden and D. Weld. Representing sensing actions: the middle ground revisited. In *Proc. KR-96*, 1996.
10. M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.
11. F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6), 1992.
12. P. Jonsson and C. Backstrom. Incremental planning. In *Proc. 3rd European Workshop on Planning*, 1995.
13. H. Levesque. What is planning in the presence of sensing? In *Proc. AAAI-96*, 1996.
14. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming: special issue on actions*, 31(1–3):59–83, 1997.
15. F. Lin and R. Reiter. How to progress a database. *Artificial Intelligence*, 92:131–167, 1997.

16. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4, 1969.
17. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department Aarhus University Denmark, 1981.
18. R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.
19. R. Reiter. Knowledge in action: Logical foundation for describing and implementing dynamical systems. In preparation., 1999.
20. R. Scherl and H. Levesque. The frame problem and knowledge producing actions. In *Proc. of AAAI-93*, pages 689–695, 1993.