An Incremental Interpreter for High-Level Programs with Sensing

Giuseppe De Giacomo
Dipartimento di Informatica e Sistemistica
Universita di Roma "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
degiacomo@dis.uniroma1.it

Hector Levesque
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, and without loss of generality, allow much more control to be exercised over when actions can be executed. We argue that such a scheme is the only practical way to deal with large agent programs containing both nondeterminism and sensing.

Introduction

In [4] 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 $\vec{a}$ such that

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

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

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

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 [4], a language called ConGolog is presented, which in addition to nondeterminism, contains facilities for sequence, iteration, conditionals, concurrency, and prioritized interrupts. 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.

Off-line and On-line execution

To be compatible with planning, the ConGolog interpreter presented in [4] 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.

[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.

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 [4]), we need to be able to choose between a and b based on 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 globally 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. There is no loss of expressive power here since to execute a program the old way, we need only put the entire program within a Σ operator.

Sensing actions

This on-line style of execution is well-suited to programs containing sensing actions. As described in [5, 9, 15], 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

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

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

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

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 [9] 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

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

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 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 only an on-line execution style is practical for large programs containing nondeterminism and sensing actions.

Characterizing program execution

The technical machinery we use to define on-line program execution in the presence of sensing is essentially that of [4], *i.e.* we use the predicates *Trans* and *Final* to define a single step semantics of programs [6, 13]. However some adaptation is necessary to deal with on-line execution, sensing results, and the Σ operator.

Situation calculus

The starting point in the definition 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; there is a special predicate *Poss*(a,s) used to state that action a is executable in situation s; finally, following [9], 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 [14]:

- 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 $F(\vec{x}, do(a, s))$ 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 [14].
- Unique names axioms for the primitive actions.
- Some foundational, domain independent axioms.

Finally, as in [9], 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 $[SF(a, s) \equiv \mathbf{true}]$.

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 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 $end[\sigma, s]$ as an abbreviation for a situation term called the *end situation* of history σ on s, and defined by: $end[\epsilon, s] = s$; and inductively, $end[\sigma\cdot(a, x), s] = do(a, end[\sigma, s])$.

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

The *Trans* and *Final* predicates

The on-line execution of a program consists of a sequence of legal single-step transitions. In [4], two special predicates, *Final* and *Trans* were axiomatized, where $Final(\delta, s)$ was intended to say that program δ may legally terminate in situation s, and where $Trans(\delta, s, \delta', s')$ was intended to say that program δ in situation s may legally execute one step, ending in situation s' with program δ' remaining. For example, the transition axiom for sequence is

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

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 δ', and $(\delta';\delta_2)$ is what is left of the sequence.

For our account here, we include all the axioms for *Trans* and *Final* from [4] (the details of which we omit), and add two new ones below for the Σ operator. However, instead of using these axioms to characterize a Do formula for off-line execution, we will use them together with sensing values to define on-line execution.

In the absence of sensing, we have that an action a is a legal next step for program δ in situation s only when

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

for some remaining program δ'. With sensing however, the existence of such an a may depend on the values sensed so far. That is, if s is $end[\sigma, S_0]$ where σ is the history of actions and sensing values, a should be such that

$$Axioms \cup \{Sensed[\sigma, S_0]\} \models Trans(\delta, s, \delta', do(a, s)).$$

In general, given history σ starting in situation s, we look for a next action a satisfying

$$\begin{aligned}&Axioms \cup \{Sensed[\sigma, s]\} \models\\ &\quad Trans(\delta, end[\sigma, s], \delta', do(a, end[\sigma, s])).\end{aligned}$$

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

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

where again the history σ can be taken into account.

How do we know that this specification is appropriate? It is easy to see that if every sensing result in a history happens to be 1 (*i.e.* there is no sensing information), then the specification correctly reduces to the specification of a legal single step from before. Moreover, we can see that it corresponds intuitively to on-line execution, in that we get to take into account the sensing information returned by the current action before deciding on the next one. So if a happened to be the sensing action $read_q$ from above, and it returned the value 0 in situation s, then in looking for the next legal action, we would assume that $\neg SF(read_q, s)$ was true, and thus, that $\neg q$ held in situation s. So if δ' above were [**if** q **then** ... **else** ...], the correct **else** branch would be taken for the next action.

As noted above, the only change we require to the axioms for *Trans* and *Final* is for the Σ operator. For *Final*, we have that $(\Sigma\delta, s)$ is a final configuration of the program if (δ, σ) itself is, and so we get the axiom

$$Final(\Sigma\delta, s) \equiv Final(\delta, s).$$

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 axiom[3]

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

[2] A fluent whose current value could only be determined by sensing would normally not have a successor state axiom.

[3] We do not attempt to deal with the subtleties that arise when a search is performed with other programs executing concurrently.

In this axiom, *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

$$\begin{array}{l} \forall s, \gamma.\ T(\gamma, s, \gamma, s) \quad \wedge \\ \forall s, \gamma, s', \gamma', s'', \gamma''.\ T(\gamma, s, \gamma', s') \wedge \\ \qquad \mathit{Trans}(\gamma', s', \gamma'', s'') \ \supset\ T(\gamma, s, \gamma'', s''). \end{array}$$

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 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.

An incremental interpreter

In this section 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 [10] 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 *dynamic closed-world assumption* 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.* We emphasize, however, that while this assumption is important for the Prolog implementation, it is not required by the formal specification.

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

nextAct(P,H,Act,P1) :-
   trans(P,H,P1,[(Act,_)|H]).

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

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.[4] 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. 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.

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).
```

which corresponds to the axiom given earlier. 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

[4] In practice, we would not want the history list to get too long, and would use some form of "rolling forward" [11].

configuration.[5] 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.

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;[6]
- `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.

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)

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

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 we are making a (dynamic) closed-world 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 dynamic closed-world assumption comes in: 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.

Correctness

This completes the incremental interpreter. The interpreter is correct in the sense that[7]:

- if the goal `final(`δ`,`σ`)` succeeds, then

  $$Axioms \cup \{Sensed[\sigma, S_0]\} \models Final(\delta, end[\sigma, S_0])$$

- if the goal `nextAct(`δ`,`σ`,`a`,`δ'`)` succeeds, then

  $$\begin{aligned} &Axioms \cup \{Sensed[\sigma, S_0]\} \models \\ &\quad Trans(\delta, end[\sigma, S_0], \delta', do(a, end[\sigma, S_0])) \end{aligned}$$

But despite the very close correspondence between the axioms for *Trans* and *Final* and the clauses for `trans` and `final`, actually *proving* this correctness is not trivial: we need to show how the axioms of the background action theory derive from the user-supplied Prolog clauses listed above given our dynamic closed-world assumption. We leave this to future research.

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 [15]; 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

[5] 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.

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

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

versions of (runtime) planning to be included within a program. Indeed, a simple planner can be written directly:[8]

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

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 is, as far as we know, the only one that has a chance 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 prepackaged "plans" found in RAP [3] or PRS [7]. 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 look-ahead 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 what we can *guarantee* about the on-line execution of an agent program. On-line execution may fail, for instance, even when a proper sequence of actions provably exists. There is a difficult tradeoff here that also shows up in the work on so-called *incremental planning* [1, 8]. Even if we have an important goal that needs to be achieved in some distant place or time, we 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 to be able to quickly confirm that some action can be used immediately as a first step towards some challenging but distant goal.

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

References

[1] J. A. Ambros-Ingerson and S. Steel. Integrating Planning, Execution and Monitoring. In *Proc. AAAI-88*, Saint Paul, Minnesota, 1988.

[2] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proc. AAAI-96*, Portland, Oregon, 1996.

[3] R. J. Firby. An investigation in reactive planning in complex domains. In *AAAI-87*, Seattle, Washington, 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*, Nagoya, Japan, 1997.

[5] K. Golden and D. Weld. Representing sensing actions: the middle ground revisited. In *Proc. KR-96*, Cambridge, Massachusetts, 1996.

[6] M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

[7] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, **7**, *6*, 1992,

[8] P. Jonsson and C. Backstrom. Incremental planning. In *Proc. 3rd European Workshop on Planning*, 1995.

[9] H. Levesque. What is planning in the presence of sensing? In *Proc. AAAI-96*, Portland, Oregon, 1996.

[10] 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*, pp. 59–83, 1997.

[11] F. Lin and R. Reiter. How to progress a database. In *Artificial Intelligence*, **92**, pp. 131–167, 1997.

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

[13] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept. Aarhus Univ. Denmark, 1981.

[14] 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.

[15] 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.