On sensing and off-line interpreting in GOLOG

Gerhard Lakemeyer

Department of Computer Science, Aachen University of Technology,
D-52056 Aachen, Germany, Email: gerhard@cs.uni-bonn.de

Abstract. GOLOG is a high-level programming language for the specification of complex actions. It combines the situation calculus with control structures known from conventional programming languages. Given a suitable axiomatization of what the world is like initially and how the primitive actions change the world, the GOLOG interpreter derives for each program a corresponding linear sequence of legally executable primitive actions, if one exists. Despite its expressive power, GOLOG's applicability is severely limited because the derivation of a linear sequence of actions requires that the outcome of each action is known beforehand. Sensing actions do not meet this requirement since their outcome can only be determined by executing them and not by reasoning about them. In this paper we extend GOLOG by incorporating sensing actions. Instead of producing a linear sequence of actions, the new interpreter yields a tree of actions. The idea is that a particular path in the tree represents a legal execution of primitive actions conditioned on the possible outcome of sensing actions along the way.

Prologue

When I left Toronto in late 1990, Ray had just begun to revive the situation calculus as a serious contender among the various logics of action.[1] To be honest, I myself was rather skeptical at first whether his approach and, in particular, GOLOG would ever be more than just a specification language for dynamic domains. When Ray gave a talk at my then department at Bonn in 1994, he was met with even more skeptical questions regarding GOLOG's practicability by the "real" roboticists at Bonn. As there were no conclusive answers at the time, I decided it was time to put GOLOG to the test and, lo and behold, within a year and with the invaluable support of our robotics group, we conducted the first experiments controlling a real robot using GOLOG. Again, Ray's vision proved to be right, and I have since joined his quest to explore cognitive robotics. This paper[2] is a small contribution in this regard. Needless to say, none of this would have been possible without Ray's efforts and that of the other members of the Cognitive Robotics Group at Toronto.

---

[1] It is very fitting that Ray's first paper on the subject appeared in the Festschrift in honor of John McCarthy [17].

[2] An earlier version of this paper appeared in [3].

1 Introduction

When reasoning about action one is often faced with incomplete knowledge. For example, when trying to achieve a goal such as catching an airplane, there usually is not enough information at the outset for an agent to come up with a single course of action which would satisfy the goal. For instance, I may not know the departure gate of the plane until I actually reach the airport (or, to be more modern, until I check my airline's web-site.) What is needed are sensing actions which, when executed at the appropriate time, gather relevant information about the world and whose outcome determines what other actions need to be performed later.

Despite this obvious observation, dealing with sensing in both a principled and practical way has been surprisingly difficult. On the principled side, there has been substantial progress in understanding the connection between knowledge, sensing, and action, see for example [15,16,18,10,9]. There have also been several proposals to incorporate sensing actions into planning systems such as [5,6,1]. While planning may be workable in limited domains, we support the view of Levesque and Reiter [11] that general purpose planning is not sufficient as the main means for agents such as robots to decide how to achieve a task. The argument here is mainly one of complexity. The planning problem without sensing is already highly intractable, and adding sensing only compounds the problem.

Rather than leaving it completely up to the robot to construct a plan from a set of primitive actions, an alternative strategy would be to devise a suitable high-level programming language in which the user specifies not just a goal but also how it is to be achieved, perhaps leaving small subtasks to be handled by an automatic planner. An example of such a language is GOLOG [12], which combines the expressive power of the situation calculus with control structures known from conventional programming languages. A key property of GOLOG (or, more precisely, the GOLOG interpreter) is that it takes a program and verifies off-line whether it is legally executable. In case the verification succeeds, it also produces a plan in terms of a linear sequence of primitive actions which can then be immediately executed.

While GOLOG comes with an efficient Prolog implementation, its applicability in real world domains is severely limited because sensing actions are not handled properly. The problem is that in order to come up with a sequence of actions GOLOG needs to have all the relevant information beforehand to decide on a course of actions to achieve a goal, whereas the whole point of sensing is that some information becomes available only at run-time. This deficiency became very clear in a recent robotics application [2], where our group employed GOLOG to specify the actions of a robot who gives guided tours in a museum. While GOLOG provided more than enough flexibility in

terms of the available control structures, not being able to deal with sensing proved to be rather cumbersome.[3]

Despite recent arguments against it [4],[4] we believe that off-line verification of a plan is a valuable feature of GOLOG, in particular, during program development where mistakes are bound to happen, and it seems desirable to be able to take into account sensing actions as well. This paper provides a step in this direction.

To illustrate the problem and our proposal, let us consider the airport example in somewhat more detail. Suppose that the agent is already at the airport, but she does not know the gate yet. Before boarding the plane, she wants to buy a newspaper and a coffee. In case the gate number is 90 or up, it is preferable to buy coffee at the gate, otherwise it is better to buy coffee before going to the gate. Let us assume, we have the following primitive actions: *buy_paper*,*buy_coffee*, *goto_gate*, *board_plane*, and *sense_gate*, which senses the value of *gate* (perhaps by glancing at the departure information monitor).

In GOLOG one might be tempted to write the following (grossly oversimplified) procedure to catch a plane.

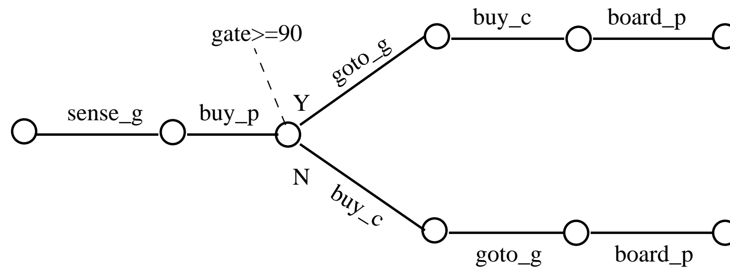
proc *catch_plane*
    sense_gate;
    buy_paper;
    if $gate \geq 90$ **then** *goto_gate*;*buy_coffee* **else** *buy_coffee*;*goto_gate*
    endif;
    board_plane
endproc

Let us assume also that we have a set of axioms which suitably characterize what the world is like initially, what the action preconditions are, and how actions change the world and the agent's knowledge about the world (see the next section for hints about how all this is done). Given these axioms, the GOLOG interpreter then tries to logically derive a linear sequence of primitive actions which are legally executable and which represent an execution trace of *catch_plane*. In our case, the only plausible candidates are *sense_gate*·*buy_paper*·*goto_gate*·*buy_coffee* and *sense_gate*·*buy_paper*·*buy_coffee*·*goto_gate*. The problem is that it will only be known at runtime and after the execution of *sense_gate* which of the two sequences is the actual one. Hence GOLOG, running off-line, is bound to fail since it cannot decide between the two.

[3] Here we do not mean sensing as it is needed for safe navigation, which was not handled at the logical level at all, but was left to lower level components of the robot. What we do mean is sensing at the abstract task level, which, in this application, involved mainly the interaction with a visitor during a guided tour.

[4] We will get back to [4] in Section 7.

If GOLOG allowed for branching in the plans it produces the problem could be overcome. In the example, we would need a plan that starts with *sense_gate* followed by *buy_paper* and then splits into two branches consisting of *goto_gate* followed by *buy_coffee* and the other way around, depending on whether $gate \geq 90$ or not. This is in fact the main modification of GOLOG we propose in this paper. We call plans with branches *conditional action trees (CAT's)*, which are binary trees whose nodes can be thought of as situations with the root representing the initial situation. Every edge is labeled with a primitive action, which indicates how a situation is obtained from its predecessor. In addition, whenever branching occurs, the corresponding node/situation is labeled by a formula, whose truth value at execution time determines which branch is taken. A CAT for the airport example could be drawn as follows:



However, we will be writing it as a term using the following notation

$$\begin{aligned} &\mathit{sense_gate}\cdot \mathit{buy_paper}\cdot [\mathit{gate} \geq 90, \\ &\qquad \mathit{goto_gate}\cdot \mathit{buy_coffee}\cdot \mathit{board_plane}, \\ &\qquad \mathit{buy_coffee}\cdot \mathit{goto_gate}\cdot \mathit{board_plane}]. \end{aligned}$$

It turns out that the GOLOG interpreter which handles CAT's has a simple specification, which is very similar to the original one given in [12]. In our extension of GOLOG we allow sensing truth values as well as the referent of terms (as in the above example). Note also that branching need not occur immediately at the time of sensing. In contrast, [10,4] only consider sensing truth values and branching happens immediately at the time of sensing.

The rest of the paper is organized as follows. In Sections 2 and 3, we give very brief introductions into the situation calculus and GOLOG. Section 4 introduces conditional action trees into the situation calculus. In Section 5, we define sGOLOG (= GOLOG + sensing) and in Section 6 we present a simple interpreter implemented in Prolog. In Section 7, we summarize our results and compare our work to [4].

2 The Situation Calculus

One increasingly popular language for representing and reasoning about the preconditions and effects of actions is the situation calculus [14]. We will only go over the language briefly here noting the following features: all terms in the language are one of three sorts, ordinary objects, actions or situations; 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; similarly, functions varying across situations are called functional fluents and are denoted analogously; finally, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s. Throughout the paper we write action and situation variables using the letters a and s, respectively, possibly with sub- and superscripts. (The same convention applies to meta-variables for terms of the respective sorts.)

Within this language, we can formulate theories which describe how the world changes as the result of the available actions. One possibility is a *basic action theory* of the following form [17]:

- 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 $F(\boldsymbol{x}, do(a, s))$ holds as a 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 [17].
- Domain closure and unique names axioms for the primitive actions.
- A collection of foundational, domain independent axioms.

In [13] the following foundational axioms are considered:[5]

1. $\forall s \forall a. S_0 \neq do(a, s)$.
2. $\forall a_1, a_2, s_1, s_2.\ do(a_1, s_1) = do(a_2, s_2) \supset \quad (a_1 = a_2 \wedge s_1 = s_2)$.
3. $\forall P.\ P(S_0) \wedge [\forall s \forall a.(P(s) \supset P(do(a, s)))] \supset \forall s P(s)$.
4. $\forall s.\ \neg(s < S_0)$.
5. $\forall s, s', a.\ (s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s'))$,
   where $s \leq s'$ is an abbreviation for $s < s' \vee s = s'$.

The first three axioms serve to characterize the space of all situations, making it isomorphic to the set of ground terms of the form $do(a_n, \cdots, do(a_1, S_0) \cdots)$, which we also abbreviate as $do(\boldsymbol{a}, S_0)$, where $\boldsymbol{a}$

[5] In addition to the standard axioms of equality.

stands for the sequence $a_1 \cdot a_2 \cdot \ldots \cdot a_n$. The third axiom ensures, by second-order induction, that there are no situations other than those accessible using *do* from S_0. The final two axioms serve to characterize a $<$ relation between situations.

Knowledge and Action So far the language allows us to talk only about how the actual world evolves starting in the initial situation S_0. With sensing, we also need an account of what the agent doing the actions knows about the world initially and in successor situations. Following [18], which in turn is based on [15], we introduce a binary fluent $K(s', s)$ which can be read as "in situation s, the agent thinks that s' is (epistemically) accessible."[6]

Given K, knowledge can then be defined in a way similar to possible-world semantics [8,7] as truth in all accessible situations. We denote knowledge using the following macro, where α may contain the special situation symbol *now*. Let α_s^{now} refer to α with all occurrences of *now* replaced by s. Then

$$\texttt{Knows}(\alpha, s) \doteq \forall s' K(s', s) \supset \alpha_{s'}^{now}$$

Given Knows we introduce further abbreviations which tell us whether the truth value of a formula or the value of a term is known:

$$\texttt{Kwhether}(\alpha, s) \doteq \texttt{Knows}(\alpha, s) \lor \texttt{Knows}(\neg\alpha, s).$$
$$\texttt{Kref}(\tau, s) \doteq \exists x \texttt{Knows}(\tau = x, s).$$

To specify how actions and, in particular, sensing actions change what is known, we follow [10] and introduce a special function *SF* with two arguments, an action and a situation. As in the case of *Poss* it is assumed that *SF* is user-defined, that is, the user writes down *sensed fluent axioms*, one for each action type. The idea is that $SF(a, s)$ gives the value sensed by action a in situation s. So we might have, for example,

$$SF(sense_gate, s) = gate(s).$$

In case the action a has no sensing component (as in simple physical actions, like moving), the axiom should state that $SF(a, s)$ is some fixed value. If the action serves to sense whether or not some fluent $\phi(s)$ holds, two fixed values can be used such as 0 and 1.

In [18], Scherl and Levesque formulate a solution to the frame problem for knowledge by proposing a successor state axiom for K. Here we use the variant given in [10]:

Definition 1. $\forall a, s, s'.Poss(a, s) \supset K(s', do(a, s)) \equiv$
$$\exists s''.s' = do(a, s'') \land K(s'', s) \land Poss(a, s'') \land [SF(a, s) = SF(a, s'')].$$

[6] This view requires, in general, that there are initial situations other than S_0, which also means that the above induction axiom no longer holds. See [9] for a way to handle many initial situations axiomatically.

Roughly, after doing action a the agent thinks it could be in situation s' just in case s' results from doing a in some previously accessible situation s'', provided a is possible in s'' and both s and s'' agree on the value being sensed.

3 GOLOG

GOLOG [12] is a logic-programming language which, in addition to the primitive actions of the situation calculus, allows the definition of complex actions using programming constructs which are very much like those known from conventional programming languages. The procedure *catch_plane* introduced earlier is an example of such a complex action. Here is a list of the constructs available in GOLOG:

A	primitive action
$\phi?$	test a condition
$(\rho_1;\rho_2)$	sequence
$(\rho_1\vert\rho_2)$	nondeterministic action choice
$(\pi x.\rho)$	nondeterministic argument choice
ρ^*	nondeterministic iteration
if ϕ **then** ρ_1 **else** ρ_2 **endif**	conditional
while ϕ **do** ρ **endwhile**	loop
proc $\rho(\boldsymbol{x})$ **endproc**	procedure

What is special about GOLOG is that the meaning of these constructs is completely defined by sentences in the situation calculus. For this purpose, a macro $Do(\rho, s, s')$ is introduced whose intuitive meaning is that executing the program ρ in situation s leads to situation s'. Here we provide some of the definitions needed for Do. See [12] for the complete list.

$Do(A, s, s') \doteq Poss(A, s) \wedge s' = do(A, s)$, where A is a primitive action.
$Do((\rho_1;\rho_2), s, s') \doteq \exists s'' Do(\rho_1, s, s'') \wedge Do(\rho_2, s'', s')$
$Do((\rho_1|\rho_2), s, s') \doteq Do(\rho_1, s, s') \vee Do(\rho_2, s, s')$
$Do(\phi?, s, s') \doteq \phi(s) \wedge s = s'$
$Do(\textbf{if } \phi \textbf{ then } \rho_1 \textbf{ else } \rho_2 \textbf{ endif}, s, s') \doteq Do([(\phi?;\rho_1)|(\neg\phi?;\rho_2)], s, s')$

Here ϕ is a formula of the situation calculus with all situation arguments suppressed, which we also call a situation-free formula. $\phi(s)$ is then obtained from ϕ by reinserting s as the situation argument at the appropriate places. For example, if $\phi = (gate = A) \wedge am_at(x, airport)$ with fluents *gate* and *am_at*, then $\phi(s) = (gate(s) = A) \wedge am_at(x, airport, s)$.

Given a situation calculus theory AX of the domain in question as sketched in the previous section, executing a program ρ means to first find a sequence of primitive actions $\boldsymbol{a}$ such that

$$AX \models Do(\rho, S_0, do(\boldsymbol{a}, S_0))$$

and then handing the sequence $\boldsymbol{a}$ to an appropriate module that takes care of actually performing those actions in the real world.

4 Conditional action trees

In this section, we augment the situation calculus with conditional action trees. The idea is that, instead of having only linear action histories (i.e. situations), we have a tree of actions, where each path represents a situation.

We begin by introducing two new sorts, a sort formula for situation-free formulas and a sort CAT for conditional action trees. For each sort we add infinitely many variables to the language. We write ϕ and c with possible sub- or superscripts for variables of sort formula and CAT, respectively. Since there is a standard way of doing this, we gloss over the details of how to reify formulas as terms in the language. Given a term of sort formula, we even take the liberty to write $\phi(s)$ in place of a formula, where in fact we would need to say $Holds(\phi, s)$, where $Holds$ is appropriately axiomatized.

CAT terms are made up of a special constant ϵ, denoting the empty CAT, the primitive actions, and two constructors $a \cdot c$ and $[\phi, c_1, c_2]$, where a is an action, ϕ is a term of sort formula and c, c_1, and c_2 are themselves CAT's.[7] ϕ is also called a *branch-formula*. c_1 and c_2 are called the true- and false-branch (for ϕ), respectively. We saw an example CAT already in Section 1.

We can define CAT's within the situation calculus by adding the following foundational axioms, which are analogues of those introduced earlier in Section 2 for situations. (In the following, free variables in a formula are considered to be universally quantified.) The first five axioms make sure that CAT's are all distinct. The role of Axiom 6 is the same as the induction axiom for situations and minimizes the set of CAT's.

1. $a \cdot c \neq \epsilon$.
2. $[\phi, c_1, c_2] \neq \epsilon$.
3. $a \cdot c = a' \cdot c' \supset a = a' \wedge c = c'$
4. $[\phi, c_1, c_2] = [\phi', c_1', c_2'] \supset \phi = \phi' \wedge c_1 = c_1' \wedge c_2 = c_2'$.
5. $[\phi, c_1, c_2] \neq a \cdot c$.
6. $\forall P.P(\epsilon) \wedge \forall a P(a) \wedge [\forall a, c.P(c) \wedge c \neq \epsilon \supset P(a \cdot c)] \wedge$ $[\forall \phi, c_1, c_2.P(c_1) \wedge P(c_2) \supset P([\phi, c_1, c_2])] \supset \forall c.P(c)$.

It is also convenient to define the following predicate ext, which will be needed later on to define how GOLOG, having already produced a CAT c in situation s, extends c by a CAT c^* to produce c'. Informally, $ext(c', c, c^*, s)$ holds if c' is a CAT which contains a path p from c extended by the CAT c^*. p is obtained by starting at situation s and then moving down the tree replacing s by successor situations according to the actions encountered along the path. p follows a particular branch in the tree depending on the truth value of the corresponding branch-formula relative to the current situation. Formally:

[7] Logically, $\cdot$ and $[_, _, _]$ are binary and ternary functions, respectively. We write them this way for better readability of CAT's.

$$
\begin{aligned}
ext(c', c, c^*, s) \equiv\ & (c = \epsilon \supset c' = c^*) \wedge \\
& (c = a \supset c' = a \cdot c^*) \wedge \\
& (c = a \cdot c_1 \supset \exists c_1'.c' = a \cdot c_1' \wedge ext(c_1', c_1, c^*, do(a, s))) \wedge \\
& (c = [\phi, c_2, c_3] \supset \exists c_2', c_3'.c' = [\phi, c_2', c_3'] \wedge \\
& \qquad [\phi(s) \supset ext(c_2', c_2, c^*, s) \wedge \\
& \qquad \neg\phi(s) \supset ext(c_3', c_3, c^*, s)]
\end{aligned}
$$

For example, let $c = a_1 \cdot a_2 \cdot [p, a_3, \epsilon]$ and let p be false at $do(a_2, do(a_1, s))$. Then $ext(c', c, c^*, s)$ holds for $c^* = [q, \epsilon, a_4]$ and $c' = a_1 \cdot a_2 \cdot [p, a_5, [q, \epsilon, a_4]]$. Note that c' can differ arbitrarily in the branches which are not taken, in this case the true-branch for p.

Lastly, we introduce a two-place function *cdo*, which takes a CAT c and a situation s and returns a situation which is obtained from s using the actions along a particular path in c. The idea is that *cdo* follows a certain branch in the tree depending on the truth value of the respective branch-formula at the current situation.

$$
\begin{aligned}
& cdo(\epsilon, s) = s. \\
& cdo(a, s) = do(a, s). \\
& cdo(a \cdot c, s) = cdo(c, do(a, s)). \\
& cdo([\phi, c_1, c_2]), s) = \text{if } \phi(s) \text{ then } cdo(c_1, s) \text{ else } cdo(c_2, s).
\end{aligned}
$$

5 sGOLOG

Programs in sGOLOG are those of GOLOG augmented by sensing actions for both formulas and terms. The main difference compared to the original GOLOG is that the interpreter now produces CAT's instead of situations. When constructing a CAT, a decision must be made as to when new branches, that is, constructs of the form $[\phi, c_1, c_2]$ are introduced. One possibility is to introduce them automatically whenever a sensing action occurs. This seems fine if we are sensing the truth value of a formula (like ϕ), but what should we branch on if we are sensing the value of a term, in particular if there are (infinitely) many potential values for the term? To overcome this problem, we leave the introduction of new branches under the control of the user by introducing a new special action $branch_on(\phi)$, whose "effect" is to introduce a new CAT $[\phi, \epsilon, \epsilon]$. Since we want sGOLOG to produce CAT's which are ready for execution, we need to make sure that the truth value of the formula which decides on which branch to take is known. This is taken care of by attaching an appropriate `Kwhether`-term to the definition of *branch_on*.

Technically, the sGOLOG interpreter is defined in a way very similar to the original GOLOG. We introduce a three place macro $Do(\rho, s, c)$ which expands into a formula of the situation calculus augmented by CAT's. It may be read as "executing the program ρ in situation s results in CAT c." Note the difference compared to the original *Do*, where the last argument was a situation rather than a CAT.

We begin with an auxiliary four-place macro $Do4$. Intuitively, $Do4(\rho, s, c, c')$ may be read as "starting in situation s, executing the CAT c and then the program ρ leads to c', which is an extension of c."

$$Do4(\rho, s, c, c') \doteq \exists c^* Do(\rho, cdo(c, s), c^*) \wedge ext(c', c, c^*, s).$$

$Do(\rho, s, c)$ is then defined as follows:

$Do(a, s, c) \doteq Poss(a, s) \wedge c = a$ for every primitive action a.
$Do(branch_on(\phi), s, c) \doteq \mathtt{Kwhether}(\phi(now), s) \wedge c = [\phi, \epsilon, \epsilon]$.
$Do(\phi?, s, c) \doteq \phi(s) \wedge c = \epsilon$.
$Do((\rho_1|\rho_2), s, c) \doteq Do(\rho_1, s, c) \vee Do(\rho_2, s, c)$.
$Do((\rho_1; \rho_2), s, c) \doteq \exists c' Do(\rho_1, s, c') \wedge Do4(\rho_2, s, c', c)$.
$Do(\pi x \rho, s, c) \doteq \exists x Do(\rho, s, c)$.
$Do(\rho^*, s, c) \doteq \forall P[P(\epsilon, \epsilon) \wedge \forall c_1, c_2, c_3. P(c_1, c_2) \wedge Do4(\rho, s, c_2, c_3) \supset P(c_1, c_3)] \supset P(\epsilon, c)$.
$Do(\textbf{proc}\ P\ \rho\ \textbf{endproc}, s, c) \doteq Do(\rho, s, c)$[8]

Note that the definition of the various program constructs are not all that different from the original ones. In fact, if we confine ourselves to GOLOG programs without sensing and without occurrences of the special action $branch_on$, it is not hard to show that the two interpreters coincide.

Theorem 1. *Let Do_{Old} stand for the old definition of Do. Let ρ be a GOLOG program without sensing and branch_on-actions and let $c = a_1 \cdot c_2 \ldots \cdot c_n$ for primitive actions a_i. Then $\models Do(\rho, s, c) \equiv Do_{Old}(\rho, s, cdo(c, s))$.*

5.1 The airport example revisited

Let us now see how the airport example could be handled in sGOLOG. To keep the formalization brief, we make various simplifying assumptions. For example, we assume implicitly that the agent is at the airport and that *buy_coffee*, *buy_paper*, and *sense_gate* are always possible. *goto_gate* requires the referent of *gate* to be known and *board_plane* requires being at the (right) gate.

$Poss(buy_coffee, s) \equiv TRUE$
$Poss(buy_paper, s) \equiv TRUE$
$Poss(sense_gate, s) \equiv TRUE$
$Poss(goto_gate, s) \equiv \mathtt{Kref}(gate, s)$
$Poss(board_plane, s) \equiv am_at_gate(s)$

sense_gate is the only sensing action. Hence for the others *SF* always returns the same value:

[8] For simplicity, we only consider simple nonrecursive procedures without parameters. See [12] for how to handle general procedures in GOLOG.

$SF(buy_coffee, s) = 1$
$SF(buy_paper, s) = 1$
$SF(goto_gate, s) = 1$
$SF(sense_gate, s) = gate(s)$

gate and *am_at_gate* are the only fluents, and *gate* never changes its value:

$Poss(a, s) \supset gate(do(a, s)) = y \equiv gate(s) = y.$
$Poss(a, s) \supset am_at_gate(do(a, s)) \equiv a = goto_gate \;\vee\; am_at_gate(s)$
(Assumes that boarding the plane leaves you at the gate.)

Finally, the sGOLOG program to catch the plane is like the one in the introduction except for the explicit *branch_on* action:

proc *catch_plane*
  sense_gate;
  buy_paper;
  $branch_on(gate \geq 90)$;
  if $gate \geq 90$ **then** *goto_gate*;*buy_coffee*
      else *buy_coffee*;*goto_gate*
  endif;
  board_plane
endproc

Assuming that AX consists of the foundational axioms of our extended situation calculus, the above airport axioms, and simple arithmetic to compare numbers, we obtain

$$\begin{array}{l} \mathrm{AX} \models Do(catch_plane, S_0, c) \text{ with} \\ \quad c = sense_gate \cdot buy_paper \cdot [gate \geq 90, \\ \qquad goto_gate \cdot buy_coffee \cdot board_plane, \\ \qquad buy_coffee \cdot goto_gate \cdot board_plane]. \end{array}$$

Note that in the airport example we make use of the fluent K only in a very limited way, namely in the form of $\mathtt{Kref}(gate, s)$ and $\mathtt{Kwhether}(gate(now) \leq 90, s)$. (The latter results from interpreting $branch_on(gate \geq 90)$.) In particular, we do not have to deal with nested occurrences of `Knows`. For this reason, there is no need to stipulate any special properties of the K-relation such as reflexivity or transitivity. In principle, there is no problem adding such restrictions. Indeed Scherl and Levesque [18] have shown that it suffices to stipulate those for initial situations and that the successor state axiom for K (Definition 1) guarantees that these properties hold in all successor situations as well. However, as the work by Scherl and Levesque also shows, reasoning about K is not easy. Even if we restrict the use of K to the `Knows`-macro, we still need some form of modal reasoning to deal with it. When it comes to implementing sGOLOG, this seems like a high price to pay. Fortunately, as we will see in a moment, with reasonable restrictions on the use of sensing, there is a way to avoid this problem by not using the K-fluent at all.

6 A simple implementation

In this section we present a very simple implementation of sGOLOG in Eclipse-Prolog. Besides the usual constraints that come with the use of Prolog, like negation-as-failure and atomic facts only to describe the initial situation, we restrict sensing actions and their use as follows:

1. Only the truth value of atomic facts can be sensed. In particular, sensing actions have the form $sense(P)$, where P is a fluent.
2. The truth value of a sensed fluent is never tested before the corresponding sensing action has been performed.
3. $branch_on(P)$ actions are only allowed in case P is a sensed fluent.
4. Whenever a $branch_on(P)$ action is reached, both truth values are conceivable for P.

Some remarks regarding each of these assumptions:

1. While restricting ourselves to atomic facts is essential, there is no problem in principle to allow for sensing the referent of a term as well. However, it would add some overhead to the implementation, and we have chosen to ignore this issue here for the sake of simplicity.
2. This is what De Giacomo and Levesque [4] call the *dynamic closed world assumption.* It lets us deal with incomplete information even in Prolog, which makes the closed world assumption. The idea is that whenever a fluent F is tested for the first time, complete information about F has been achieved.
3. Applying *branch_on* only to sensed fluents enables us to avoid having to work explicitly with a K-fluent in a very simple way. Testing whether a sensed fluent P is known can be reduced to testing whether the action $sense(P)$ was performed earlier. This test is easily implementable by introducing a new fluent $sensed(P, s)$ which becomes true when $sense(P)$ is executed and remains true from then on. In essence, under the above assumptions the truth value of $sensed(P, s)$ keeps track of whether P is known. Of course, the use of $sensed(P, s)$ to simulate `Kwhether`(P, s) is not restricted to *branch_on*. For example, it may also be part of a definition of $Poss(a, s)$.
4. If it is possible that P can take on either truth value, we can safely use hypothetical reasoning for both cases when evaluating $branch_on(P)$.[9] The idea is that the true-branch C1 of the CAT $[P, C1, C2]$, which results from $branch_on(P)$, is constructed by assuming that P is true in the current situation. Similarly, the false-branch $C2$ is developed by assuming that P is false in the current situation.
   We implement this by introducing new primitive actions $assm(P, 1)$ and $assm(P, 0)$ whose only effect is to turn P true and false, respectively.

[9] If it would follow that P is, say, true, then considering the case where P is false might lead to failure, even though that case never arises.

(See the definition of *do*4 below.) Note that that actions occurring after *assm* are allowed to change the truth value of P. An example previously discussed in De Giacomo and Levesque [4] is an elevator controller, which first senses the value of a button and, if it is "on," resets it to "off" afterwards.

We now turn to the actual implementation. The reader familiar with the original paper on GOLOG [12] will notice the close similarity between the Prolog implementation of GOLOG and the one below for sGOLOG. Note, in particular, that the definitions of *do* in GOLOG and sGOLOG are practically identical except for sequence (:) and, of course, *branch_on*, which does not exist in GOLOG.

```
:- dynamic(holds/2).
:- op(970,xfy, [:]).  /* Sequence.*/
:- op(950, xfy, [#]). /* Nondeterministic action choice.*/

/* do4(P,S,C,C1) recursively descends the CAT C (first two clauses). */
/* Once a leaf of the CAT is reached (third clause), "do" is called, */
/* which then extends this branch according to P */
do4(E,S,[A|C],C1) :- primitive_action(A),C1=[A|C2],do4(E,do(A,S),C,C2).
do4(E,S,[[P,C1,C2]],C) :- do4(E,do(assm(P,1),S),C1,C3),
                          do4(E,do(assm(P,0),S),C2,C4),
                          C = [[P,C3,C4]].
do4(E,S,[],C) :- do(E,S,C).

do(E1 : E2, S, C) :- do(E1,S,C1), do4(E2,S,C1,C). /* sequence */
do(?(P),S,C) :- C=[],holds(P,S). /* test */
do(E1 # E2, S, C) :- do(E1,S,C) ; do (E2,S,C). /* nond. act. choice */
do(if(P,E1,E2),S,C) :- do((?(P) : E1) # (?(neg(P)) : E2),S,C).
do(star(_),_,[]). /* nondet. iteration */
do(star(E),S,C) :- do(E : star(E),S,C).
do(while(P,E),S,S1):- do(star(?(P) : E) : ?(neg(P)),S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1)./* nond. arg. choice */
do(E,S,C) :- proc(E,E1), do(E1,S,C). /* procedure */

/* the base cases: primitive actions and branch_on(P) */
do(E,S,[E]) :- primitive_action(E), poss(E,S).
do(branch_on(P),S,[[P,[],[]]]) :-  holds(sensed(P),S).

/* sub and sub_list are auxiliary predicates */
/* sub(Name,New,Term1,Term2): */
/*    Term2 is Term1 with Name replaced by New. */
sub(_,_,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
T2 =..[F|L2].
sub_list(_,_,[],[]).
```

```
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2),
                                   sub_list(X1,X2,L1,L2).

/* Definition of holds for arbitrary nonatomic formulas */
holds(and(P1,P2),S) :- holds(P1,S), holds(P2,S).
holds(or(P1,P2),S) :- holds(P1,S); holds(P2,S).
holds(neg(P),S) :- not holds(P,S).   /* Negation by failure */
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* the successor state axiom for sensed */
holds(sensed(P),do(A,S)) :- A = sense(P) ; holds(sensed(P),S).
```

Let us now consider an implementation of the airport example. Since we restrict ourselves to sensing the truth values of fluents, we need to adapt the example accordingly. For simplicity, let us assume that there are only two gates, *gate_A* and *gate_B*. We introduce a fluent *it_is_gate_A*, whose truth value can be sensed and which then tells us which gate to take. In contrast to the original example, we also use the slightly more general action $goto(x)$ and fluent $am_at(x)$ instead of *goto_gate* and *am_at_gate*, respectively. The general definition of sGOLOG requires the use of $SF(a,s)$ to define the result of a sensing action. However, SF is only needed for the definition of the epistemic K-fluent. Since the implementation does not use K, there is no need to use SF either.

```
/* declare the primitive actions */
primitive_action(goto(_)).
primitive_action(buy_paper).
primitive_action(buy_coffee).
primitive_action(board_plane).
primitive_action(sense(_)).

/* all actions are always possible except board_plane, */
/* which requires being at the right gate */
poss(goto(_),_).
poss(buy_paper,_).
poss(buy_coffee,_).
poss(sense(_),_).
/* boarding requires being at the right gate */
poss(board_plane,S) :- holds(it_is_gate_A,S) ->
                       holds(am_at(gate_A),S) ; holds(am_at(gate_B),S).

/* successor state axioms */

/* I am at X if I just went there or */
/* if I was there and did not go anywhere else */
/* (assumes that boarding the plane leaves you at the gate) */
holds(am_at(X),do(A,S)) :- A = goto(X) ;
```

```
                          (holds(am_at(X),S), not (A = goto(_))).

holds(it_is_gate_A,do(A,S)) :- holds(it_is_gate_A,S) ;
                               A = assm(it_is_gate_A,1).

/* facts about the initial situation s0 */
/* none necessary here */

proc(catch_plane,
           (sense(it_is_gate_A):
           buy_paper:
           branch_on(it_is_gate_A):
           if(it_is_gate_A,
                 goto(gate_A):buy_coffee,
                 buy_coffee:goto(gate_B)):
           board_plane)
    ).

/* sample run, slightly reformatted for better readability */

[eclipse 3]: do(catch_plane,s0,C).

C = [sense(it_is_gate_A), buy_paper, [it_is_gate_A,
               [goto(gate_A), buy_coffee, board_plane],
               [buy_coffee, goto(gate_B), board_plane]]]  More? (;)

no (more) solution.
```

One might object that the implementation requires that the user is aware of the internals of the interpreter because the pseudo-action *assm* occurs in the user's code. In the example, it is part of the successor state axiom of `holds(it_is_gate_A,do(A,S))`. The objection can be dealt with by allowing the user to write successor state axioms without *assm* and then modifying the axioms automatically in the following way. Let F be a fluent whose truth value can be sensed. According to Reiter [17], the general form of the successor state axiom of F is (provided $Poss(a,s)$ holds):

$$Holds(F, do(a,s)) \equiv \Phi_F^+ \vee (Holds(F,s) \wedge \neg\Phi_F^-),$$

where Φ_F^+ and Φ_F^- describe the conditions which lead to F being true and false, respectively. We can compile the "effects" of the pseudo-action *assm* into the successor state axiom by replacing the above axiom by

$$\begin{aligned} Holds(F, do(a,s)) \equiv\ & [\Phi_F^+ \vee a = assm(F,1)] \vee \\ & (Holds(F,s) \wedge \neg[\Phi_F^- \vee a = assm(F,0)]). \end{aligned}$$

Another issue that needs to be addressed is that of correctness. In other words, is the interpreter a faithful implementation of the specification of sGOLOG? Here the answer may not be that easy to come by. A reasonable

intermediate step would be to first give a purely logical specification of our way to avoid the K-fluent using *sensed* and *assm* and then prove that the two formalizations coincide under the restrictions laid out at the beginning of this section. We leave this to future work.

7 Summary and discussion

In this paper we proposed sGOLOG, which extends GOLOG by adding sensing actions for sensing the truth values of formulas as well as the referents of terms. We provided an off-line interpreter for sGOLOG, whose definition is simple and remarkably similar to the original one proposed for GOLOG. Instead of producing a linear sequence of primitive actions, the sGOLOG interpreter generates a tree of actions, if one exists, with the idea that branching is conditioned on the outcome of sensing actions.

In [4], De Giacomo and Levesque propose a different version of GOLOG with sensing. They advocate a combination of off-line and on-line interpretation. On-line interpretation means, roughly, that instead of verifying that the whole program is executable, the interpreter finds the next executable primitive action and commits to it by immediately executing it. The advantage is that, whenever a sensing action occurs, the outcome is immediately known and no branching is necessary. The authors argue that it is infeasible to verify very large programs off-line, in particular those that contain many nondeterministic actions and sensing actions. The authors certainly have a point here. In particular, programs with loops such as **while** ϕ **do** ... $sense(\phi)$ ... **endwhile** generally lead to infinite CAT's in our approach. Despite these shortcomings, we think there is a place for off-line interpretation of programs with sensing. For one, many programs with a moderate number of sensing actions can very well be handled by our approach. Also, as we have seen, sensing does not necessarily lead to branching. Furthermore, we believe that off-line interpreting is a valuable tool during program development, since we want to have some confidence that a program works before running it on an expensive robot. De Giacomo and Levesque seem to believe in off-line interpretation as well, at least partly. They allow a user to specify which parts of a program are to be handled off-line. Their version of off-line interpretation, however, is somewhat limited compared to ours. For one, they only verify that for all outcomes of sensing the program is executable without actually constructing a plan (like our CAT's), which could then be executed without further processing. Moreover, they do not handle sensing terms. It seems interesting to try and combine our ideas with theirs to have the best of both worlds.

References

1. Baral, C. and Son, T. C., Approximate reasoning about actions in presence of sensing and incomplete information., *Proc. of the International Logic Pro-*

gramming Symposium (ILPS'97), 1997.

2. Burgard, W., Cremers, A. B., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W., Thrun, S., The Interactive Museum Tour-Guide Robot, *AAAI-98*.
3. De Giacomo, G. (ed.) *Proceedings of the AAAI Fall Symposium on Cognitive Robotics*, AAAI Technical Report FS-98-02, AAAI Press, 1998.
4. De Giacomo, G. and Levesque, H.J., An incremental interpreter for high-level programs with sensing. in: *Proceedings of the AAAI Fall Symposium on Cognitive Robotics*, AAAI Technical Report FS-98-02, AAAI Press, 1998, pp. 28–34.
5. Etzioni, O., Hanks, S., Weld, D., Draper, D., Lesh, N., and Williamsen, M., An approach to planning with incomplete information. *Proc. KR'92*, Morgan Kaufmann, 1992, pp. 115–125.
6. Golden, K. and Weld, D., Representing sensing actions: the middle ground revisited. *Proc. KR'96*, Morgan Kaufmann, 1996, pp. 174–185.
7. Hintikka, J., *Knowledge and Belief: An Introduction to the Logic of the Two Notions*. Cornell University Press, 1962.
8. Kripke, S. A., Semantical considerations on modal logic. *Acta Philosophica Fennica* **16**, 1963, pp. 83–94.
9. Lakemeyer, G. and Levesque, H. J., AOL: a logic of acting, sensing, knowing, and only knowing, *Proc. of the 6th International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, Morgan Kaufmann, 1998, pp. 316–327.
10. Levesque, H. J., What is Planning in the Presence of Sensing. AAAI-96, AAAI Press, 1996.
11. Levesque, H. J. and Reiter, R., High-level robotic control: beyond planning. Position Statement. *Working Notes of the AAAI Spring Symposium on Integrating Robotic Research: Taking the Next Leap*, AAAI Press, 1998.
12. H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming,* **31**, 59-84, 1997.
13. Lin, F. and Reiter, R., State constraints revisited. *J. of Logic and Computation, special issue on actions and processes*, **4**, 1994, pp. 665–678.
14. McCarthy, J., *Situations, Actions and Causal Laws.* Technical Report, Stanford University, 1963. Also in M. Minsky (ed.), *Semantic Information Processing*, MIT Press, Cambridge, MA, 1968, pp. 410–417.
15. Moore, R. C., A Formal Theory of Knowledge and Action. In J. R. Hobbs and R. C. Moore (eds.), *Formal Theories of the Commonsense World*, Ablex, Norwood, NJ, 1985, pp. 319–358.
16. Morgenstern, L., Knowledge preconditions for actions and plans. *Proc. IJCAI-87*, pp. 867–874.
17. Reiter, R., The Frame Problem in the Situation Calculus: A simple Solution (sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz (ed.), *Artificial Intelligence and Mathematical Theory of Computation*, Academic Press, 1991, pp. 359–380.
18. Scherl, R. and Levesque, H. J., The Frame Problem and Knowledge Producing Actions. in *Proc. of the National Conference on Artificial Intelligence (AAAI-93)*, AAAI Press, 1993, 689–695.