

What Does it Mean that an Agent is Performing a Typical Procedure? A Formal Definition in the Situation Calculus

Robert Demolombe
ONERA Toulouse
Toulouse, France

Robert.Demolombe@cert.fr

Erwan Hamon^{*}
ONERA Toulouse
Toulouse, France

erwan.hamon.2001@supaero.fr

ABSTRACT

We briefly recall the basic notions of the Situation Calculus and of the programming language GOLOG. Then, it is shown that typical procedures, like procedures performed by pilots in the field of aeronautics, do not have the same status as programs. In particular they may specify that some actions should not be performed. The GOLOG language is extended to fit these differences, and the semantics of the extended language is defined in the Situation Calculus. In the following we introduce the predicate *Doing*(α, s, s') which characterises the fact that the execution of the procedure α has started and not ended between the observed situations s and s' . That gives a logical answer to the question posed in the title.

We present an implementation in PROLOG that recognises procedures that satisfy the *Doing* predicate. Also, an interface allows to simulate the actions performed by a pilot and shows the procedures that have been recognised.

The presented results can be used to define a method to select among the recognised procedures, the procedure that can be assigned to pilot's intention. The results are not specific to the application domain and can be applied to the interactions among any kinds of agents.

Categories and Subject Descriptors

H.1 [Information Systems]: Models and Principles; I.2.4 [Computing Methodologies]: Artificial Intelligence Knowledge Representation Formalisms and Methods

General Terms

Theory

Keywords

Formalisms and logics, Intention, Situation Calculus

^{*}Also student at Ecole Nationale Supérieure de l'Aéronautique et de l'Espace

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS'02, July 15-19, 2002, Bologna, Italy.
Copyright 2002 ACM 1-58113-480-0/02/0007 ...\$5.00.

1. INTRODUCTION

There are many application domains where it is useful for an agent to recognise the other agents' intentions. In particular in the field of aeronautics, if an aircraft system could analyse the commands performed by the pilot, and infer pilot's intentions from this analysis, then this information could be used for two different purposes.

The first purpose is to help the pilot to perform the other commands he intends to execute. For instance, there are commands that have two different meanings depending on the context that has been previously set. This may cause pilot errors because the pilot may wrongly believe that some given context has been set, and when the command is performed the effect differs from what he expected. In this kind of situation, if the system has the ability to recognise the pilot's intention, it can automatically set the right context before the command is performed.

The second purpose is to control what the pilot does either (1) when he forgot to perform some action that he should have performed (for instance, if he forgot to extend the landing gear before landing) or (2) when he performed an action that he should not have performed (for instance, if he changed his route without asking for authorisation from air traffic control).

Whatever the system goal is (to help or to control the pilot), it is important that it recognises the pilot's intention. To do that, at least two sorts of information have to be known by the system: (1) the commands that have been actually performed (this information can be obtained from sensors at the level of the pilot/aircraft interface), and (2) the set of typical procedures described in the flight manual of the aircraft. For instance, the flight manual includes one, or more, typical procedures for taking off, for landing, or for certain emergency situations.

Roughly speaking, the procedures that partially match the observed performed commands are assumed to be the procedures that the pilot may intend to perform.

This approach raises several non-trivial problems: to find an appropriate formalism to represent the typical procedures (as it will be shown later on these procedures significantly differ to programs in Computer Science), to find a precise definition of the fact that an agent is performing a procedure, and to select the most plausible procedure that can be assigned to the pilot's intention.

Our contribution in this paper focuses on the first and the second problems: formal definition of procedures (sections 4 and 6), and formal definition of performing a procedure (section 5). A preliminary implementation of these results

is presented in section 7.

There are many works on intention recognition. For instance, in [9] Schank and Abelson have introduced the concept of script for natural language understanding. Perreault and Allen in [5] used plan recognition for natural language dialogue understanding. In [8] Sadek has formalised in modal logic the process of user's intention analysis for a similar objective. In [1, 2] Ghallab used the notion of chronicle to control on-going processes. However, none of these works give a formal definition of the fact that an agent is performing a procedure, which is one of the key points in recognising intentions. Finally, in [3] Lespérance et al. have defined in Situation Calculus the fact that a situation occurs during a procedure execution. This is a related but different problem of the problem we consider in this paper.

2. A BRIEF INTRODUCTION TO SITUATION CALCULUS

Situation Calculus is a first order ¹ predicate calculus language designed by a group of researchers led by H. Levesque and R. Reiter at The Cognitive Robotic group of the University of Toronto. Most of the results about Situation Calculus can be found in Reiter's book [7].

To represent the evolution of one or several agents, the situations under consideration are defined as the situations where we are after performance of a sequence of actions, and properties that hold in a given situation are determined by successor state axioms and by the initial situation.

The successor state axioms define how atomic sentences change their truth value when an action has been performed depending on that action and on the properties that hold in the previous situation.

Let us consider, for instance, the property "the landing gear is extended", and the two actions "to flip the switch to extend the landing gear" and "to flip the switch to retract the landing gear". The fact that the the landing gear is extended in the situation s is formally represented by the atomic formula $gear.extended(s)$. The two actions are represented by the two constant symbols of the sort action: $extend.gear$ and $retract.gear$.

In general, situations are denoted by terms. They may be constant symbols, like S_0 to denote the initial situation, or variable symbols, like s , or by terms of the form $do(a, s)$, where do is a designated function symbol, and a and s respectively are of sorts action and situation; $do(a, s)$ denotes the situation that results of performance of the action a from the situation s . For instance, S_0 , $do(extend.gear, S_0)$, $do(extend.gear, s)$ and $do(retract.gear, do(extend.gear, S_0))$ are terms of the sort situation. Sometimes the notation $do([a_1, \dots, a_n], s)$ will be used to denote $do(a_n, \dots, do(a_1, s) \dots)$.

To define the successor state axioms, **all** the conditions that cause $gear.extended$ to be true or to be false have to be given. For example, if we are in the situation $do(a, s)$ and the action a is $extend.gear$ (respectively $retract.gear$) then the property $gear.extended$ is true (respectively false). In formal terms we have :

$$\begin{aligned} \forall s \forall a (a = extend.gear \rightarrow gear.extended(do(a, s))) \\ \forall s \forall a (a = retract.gear \rightarrow \neg gear.extended(do(a, s))) \end{aligned}$$

¹In fact, some limited extensions to second order logic are needed when we have to define the transitive closure of a relation

To represent the fact that these conditions are "all" the conditions we need additional axioms that express that if $gear.extended$ turns out to be true (respectively false), then the performed action is necessarily $extend.gear$ (respectively $retract.gear$). Then, in formal terms, we have:

$$\begin{aligned} \forall s \forall a (\neg gear.extended(s) \wedge gear.extended(do(a, s)) \rightarrow \\ a = extend.gear) \\ \forall s \forall a (gear.extended(s) \wedge \neg gear.extended(do(a, s)) \rightarrow \\ a = retract.gear) \end{aligned}$$

It can be easily shown that these four axioms are logically equivalent to the following successor state axiom:

$$\forall s \forall a (gear.extended(do(a, s)) \leftrightarrow a = extend.gear \vee gear.extended(s) \wedge \neg(a = retract.gear))$$

Actions may have parameters. In that case they are denoted by function symbols. For instance, the two actions "to set the altitude selector for a climb of y feet" and "to set the altitude selector for a descent of y feet" can be respectively denoted by the terms: $climb(y)$ and $descent(y)$. If $altitude(x, s)$ is a predicate that means that in the situation s the aircraft altitude is x , the corresponding successor state axiom is:

$$\begin{aligned} \forall s \forall a \forall x (altitude(x, do(a, s)) \leftrightarrow \\ \exists y \exists z (a = climb(y) \wedge altitude(z, s) \wedge x = z + y) \vee \\ \exists y \exists z (a = descent(y) \wedge altitude(z, s) \wedge x = z - y) \vee \\ altitude(x, s) \wedge \neg \exists y (a = climb(y) \vee a = descent(y))) \end{aligned}$$

The formal definition of the language L_{SC} is given below. Predicates, except $<$ and $=$, may have one, and only one, argument of sort situation. This argument is the last in the list of arguments. Predicates that have an argument of the type situation are called "fluents". The equality predicate is denoted by $x = y$ as usual. There is a designated function symbol $do(a, s)$ which is of sort situation. Its first argument is of sort action, and the second argument is of sort situation. Formulas in L_{SC} are all defined by the following set of rules:

- Atomic formulas are in L_{SC} .
- If ϕ and ψ are in L_{SC} , then $(\neg\phi)$ and $(\phi \vee \psi)$ are in L_{SC} .
- If ϕ is in L_{SC} , then $(\exists x\phi)$ is in L_{SC} .

The other logical connectives and quantifiers are defined as usual.

The set of situations has the structure of a tree in the sense that a given situation cannot be reached via two distinct sequences of actions. We have formally:

$$\forall s_1 \forall s_2 \forall a_1 \forall a_2 (do(a_1, s_1) = do(a_2, s_2) \rightarrow (a_1 = a_2) \wedge (s_1 = s_2))$$

A successor relation is defined on the set of situations. Intuitively $s \leq s'$ means that the situation s' is reached from the situation s after some sequence of action. In semiformal terms, $s \leq s'$ is the smallest relation that satisfies the following properties:

$$\begin{aligned} s \leq s' \stackrel{\text{def}}{=} (s < s') \vee (s = s') \\ \forall s \forall s' \forall a (s' = do(a, s) \rightarrow s < s') \\ \forall s \forall s' \forall s'' ((s < s') \wedge (s' < s'') \rightarrow (s < s'')) \end{aligned}$$

To define the dynamics of a Multi Agent System successor state axioms have to be given for each fluent p . These axioms have the following general form:

$$\forall s \forall a \forall \vec{x} (p(\vec{x}, do(a, s)) \leftrightarrow \Gamma_p^+(\vec{x}, a, s) \vee p(\vec{x}, s) \wedge \neg \Gamma_p^-(\vec{x}, a, s))$$

where $\Gamma_p^+(\vec{x}, a, s)$ and $\Gamma_p^-(\vec{x}, a, s)$ must be inconsistent formulas.

3. A BRIEF INTRODUCTION TO GOLOG

The language of Situation Calculus has been extended in order to represent complex actions. This extension is called GOLOG (see [4]), for *al*GOL in LOGic, and it has the same expressive power as ALGOL. The originality of GOLOG lies in the definition of its semantics. In fact, complex actions, or programs, in GOLOG are abbreviation for sentences of the Situation Calculus (sometimes with extensions to second order logic). That is, what a GOLOG program does is defined by a sentence (possibly complex) that characterises the situations s' that may be reached from a situation s after execution of a program δ . This is formally denoted by $Do(\delta, s, s')$.

The semantic of $Do(\delta, s, s')$ is inductively defined from atomic actions and for each program composition operator.

If δ is the atomic action a , the situation s' is $s' = do(a, s)$, provided it is possible to perform the action a in the situation s . This precondition is formally represented by the designated predicate $Poss(a, s)$, and for each atomic action the precondition is given by a sentence of the Situation Calculus.

For instance, if a precondition to extend the landing gear is that it is not already extended, the precondition is defined by:

$$\forall s (Poss(extend.gear, s) \leftrightarrow \neg extended.gear(s))$$

In general:

$$Do(a, s, s') \stackrel{\text{def}}{=} Poss(a, s) \wedge s' = do(a, s)$$

The test action is denoted by $\phi?$, where ϕ is a Situation Calculus sentence where the situation arguments in the fluents have been removed. For instance, the test action to check if the altitude is lower than 300 feet and the landing gear has not been extended is represented by: $\exists x (altitude(x) \wedge x < 300) \wedge \neg gear.extended$. To determine in a given situation s , during a program execution, if this condition holds, the actual situation s has to be restored in ϕ . That is denoted by $\phi[s]$. In our example, $\phi[s]$ is the sentence $\exists x (altitude(x, s) \wedge x < 300) \wedge \neg gear.extended(s)$.

In this notation, the test action $\phi?$ can be executed only if $\phi[s]$ holds in s , and, if so, the resulting situation s' is s itself. Then:

$$Do(\phi?, s, s') \stackrel{\text{def}}{=} \phi[s] \wedge s' = s$$

The sequence operator is denoted by $;$ and its semantics is defined as expected by:

$$Do(\delta_1; \delta_2, s, s') \stackrel{\text{def}}{=} \exists s_1 (Do(\delta_1, s, s_1) \wedge Do(\delta_2, s_1, s'))$$

The nondeterministic choice of two actions is denoted by $|$ and its semantics is defined by:

$$Do(\delta_1 | \delta_2, s, s') \stackrel{\text{def}}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')$$

The nondeterministic iteration of δ is denoted by δ^* . Its semantics is defined by a complex second order formula which is not given here for simplification.

Then, *if...then...else*, *while* and *until* instructions are just abbreviations for complex sentences in the Situation Calculus. We have:

$$if \phi then \delta_1 else \delta_2 endif \stackrel{\text{def}}{=} (\phi?; \delta_1) | (\neg \phi?; \delta_2)$$

$$while \phi do \delta endwhile \stackrel{\text{def}}{=} ((\phi?; \delta)^*; \neg \phi?)$$

$$until \phi do \delta until \stackrel{\text{def}}{=} ((\neg \phi?; \delta)^*; \phi?)$$

The definition of procedures is beyond the scope of this paper, but it is worth knowing that procedures, even recursive procedures, have been defined in a similar way in GOLOG.

4. FORMALISATION OF TYPICAL PROCEDURES

Now we would like to show with an example that GOLOG programs are not entirely appropriate to represent the sort of typical procedures we have mentioned in the introduction, and that the GOLOG language has to be extended with new features for that purpose.

Let us, for instance, consider the typical procedure called “fire on board”, which is described for a small private aircraft. The procedure says that in case of engine fire the pilot 1) turns off fuel feed, 2) set full throttle, and 3) set mixture off. These three primitive actions, or commands, are respectively denoted by *fuel.off*, *full.throttle* and *mixture.off*, and the procedure is denoted by *fire.on.board*. In a first attempt, this procedure could be formally represented by the GOLOG program:

$$fire.on.board \stackrel{\text{def}}{=} fuel.off; full.throttle; mixture.off$$

Let us consider now a situation s_2 where the pilot has turned off the fuel, and he has called an air traffic controller (action *atc.call*), and then he has set full throttle. That is the situation s_2 is of the form $s_2 = do([fuel.off, atc.call, full.throttle], s_1)$.

In that situation s_2 we would like to say that the pilot is **performing** the procedure *fire.on.board*. However, the sequence of performed actions does not partially match the program *fire.on.board*, if partial matching is taken in the sense that a part of this program has been executed in s_2 . Indeed, this program excludes the performance of any action in between *fuel.off* and *full.throttle*.

In general, a program specifies the actions, and **all** the actions that must be executed. A typical procedure does not preclude execution of other actions. Roughly speaking, to perform the typical procedure one has to perform at least the indicated actions, but may also be performed other actions.

It is possible to extend GOLOG to adapt it to the representation of typical procedures, though these procedures do not have exactly the same meaning as programs. To do that, the language has been extended with a designated constant σ that is intended to represent any sequence of actions. Then, the procedure *fire.on.board* can be represented by:

$$fire.on.board \stackrel{\text{def}}{=} fuel.off; \sigma; full.throttle; \sigma; mixture.off$$

Now we can say that in s_2 the procedure has been partially executed.

However, it is not exactly true that between the commands *fuel.off* and *full.throttle* the pilot is allowed to execute any command. In particular it must be excluded that he turns on fuel feed (action *fuel.on*) because to stop the fire there must be no more fuel in the engine. That is another kind of things that cannot be specified by a program, because programs specify what the pilot should do, and they do not allow specifying what the pilot **should not** do.

This is our motivation to extend GOLOG with a new program composition operator denoted by $/$. Intuitively, $\sigma/fuel.on$ represents any sequence of actions that does not contain the command *fuel.on*. So, *fire.on.board* can be reformulated as:

$$fire.on.board \stackrel{\text{def}}{=} fuel.off; (\sigma/fuel.on); full.throttle; (\sigma/fuel.on); mixture.off$$

In general, typical procedures (procedures for short) are defined by terms, like complex actions in GOLOG, whose syntax is formally defined by the following rules:

- atomic actions, test actions and σ are procedures,
- if α_1 and α_2 are procedures, then $(\alpha_1; \alpha_2)$, $(\alpha_1|\alpha_2)$ and $(\alpha_1 - \alpha_2)$ are procedures.

We adopt the notation: $\alpha_1/\alpha_2 \stackrel{\text{def}}{=} \alpha_1 - (\sigma; \alpha_2; \sigma)$. The intuition is that an execution of α_1/α_2 is an execution of α_1 that does not contain an execution of α_2 .

The semantics of procedures is defined in the same way as the semantics of GOLOG programs by $Do_p(\alpha, s, s')$, whose intuitive meaning is that the sequence of actions performed between s and s' can be interpreted as an execution of α . We have:

$$Do_p(a, s, s') \stackrel{\text{def}}{=} s' = do(a, s)$$

The difference from $Do(a, s, s')$ is that here the condition $Poss(a, s)$ is not imposed. The reason is that $Do_p(\alpha, s, s')$ is intended to match the procedure α with an observed sequence of actions, and, since actions have been performed, there is no need to check if it was possible to perform them.

$$Do_p(\sigma, s, s') \stackrel{\text{def}}{=} s \leq s'$$

$$Do_p(\phi?, s, s') \stackrel{\text{def}}{=} Do(\phi?, s, s')$$

$$Do_p(\alpha_1; \alpha_2, s, s') \stackrel{\text{def}}{=} \exists s_1 (Do_p(\delta_1, s, s_1) \wedge Do_p(\delta_2, s_1, s'))$$

$$Do_p(\alpha_1|\alpha_2, s, s') \stackrel{\text{def}}{=} Do_p(\delta_1, s, s') \vee Do_p(\delta_2, s, s')$$

$$Do_p(\alpha_1 - \alpha_2, s, s') \stackrel{\text{def}}{=} Do_p(\alpha_1, s, s') \wedge \neg Do_p(\alpha_2, s, s')$$

Notice that according to these definitions we have:

$$Do_p(\alpha_1/\alpha_2, s, s') \leftrightarrow (Do_p(\alpha_1, s, s') \wedge \neg \exists s_1 \exists s_2 (s \leq s_1 \wedge Do_p(\alpha_2, s_1, s_2) \wedge s_2 \leq s'))$$

The procedures *if...then...else*, *while* and *until* are defined exactly like in GOLOG programs.

5. FORMAL DEFINITION OF “PERFORMING A PROCEDURE”

In our approach, to characterise the fact that an agent is performing a procedure α when a given sequence of performed actions has been observed between two situations s and s' , we require that three conditions hold:

1. The agent has begun executing a part α' of α between s and s' .
2. The agent has not completely executed α between s and s' .
3. The actions performed between s and s' do not prevent the continuation of the execution of α .

For instance, if s' is $s' = do([fuel.off, atc.call, full.throttle], s)$, the procedure *fire.on.board* can be expressed as:

fire.on.board = $\alpha'; \alpha''$, where $\alpha' = fuel.off; (\sigma/fuel.on); full.throttle$ and $\alpha'' = (\sigma/fuel.on); mixture.off$.

The first condition holds because we have $Do_p(\alpha', s, s')$. The second condition also holds because we have $\neg Do_p(\alpha, s, s')$. Finally, the third condition holds because there exists a situation s'' that follows s' where we have $Do_p(fire.on.board, s, s'')$; for example, s'' may be: $s'' = do([fuel.off, atc.call, full.throttle, atc.call, mixture.off], s)$.

We note that if after having set full throttle the pilot had turned on fuel feed (i.e. if we had $s' = do([fuel.off, atc.call, full.throttle, fuel.on], s)$, then it would not have been possible to satisfy condition 3.

We have refined this definition to remove some possible matches that are counter-intuitive, at least in our application domain. Let us assume, for example, that the procedure to fire a rocket is represented by $fire \stackrel{\text{def}}{=} load; \sigma; shoot$. If the sequence of observed actions is of the form: $load, \dots, load, \dots, shoot, \dots, shoot$, there is no doubt that the first execution of the procedure has to be matched with the first occurrence of *load* and the first occurrence of *shoot*, and not with the second occurrence of *shoot*.

In general, we shall say that a procedure has been executed between s and s' if there is no sub-sequence of action $\langle s, s_1 \rangle$ in $\langle s, s' \rangle$ such that the procedure has been executed between s and s_1 . This notion of minimal sequence of actions that matches a procedure is represented by $Do_m(\alpha, s, s')$, and it is formally defined by:

$$Do_m(\alpha, s, s') \stackrel{\text{def}}{=} Do_p(\alpha, s, s') \wedge \neg \exists s_1 (Do_p(\alpha, s, s_1) \wedge s_1 \leq s')$$

From this definition, if $s' = do([load, \dots, load, \dots, shoot, \dots, shoot], s)$ we do not have $Do_m(fire, s, s')$, but if $s'' = do([load, \dots, load, \dots, shoot], s)$ we do have $Do_m(fire, s, s'')$.

The condition 2, if it is strictly interpreted would lead to say for $s' = do([load, \dots, shoot, \dots, load], s)$ that the agent is not performing the procedure *fire*, but the genuine meaning of condition 2 is that there is between s and s' at least one execution of α which has started and not ended. That is, we should not say that the condition 2 is not satisfied just because α has been completely executed one, or several, times between s and s' .

For that reason we have first characterised with $Do_s(\alpha, s, s')$ the fact that the conditions 1, 2 and 3 are satisfied in a strict sense (i.e. there is no complete execution of α between s and s'), and $Doing(\alpha, s, s')$ will be used to characterise sequences of actions where α may have been executed, but there is an execution which started and did not end.

So, we formally have:

$$Do_s(\alpha, s, s') \stackrel{\text{def}}{=} \exists \alpha' (start(\alpha', \alpha) \wedge \exists s_1 (s_1 \leq s' \wedge Do_m(\alpha', s, s_1)) \wedge \neg \exists s_2 (s_2 \leq s' \wedge Do_m(\alpha, s, s_2)) \wedge \exists s_3 (s' < s_3 \wedge Do_m(\alpha, s, s_3)))$$

where $start(\alpha', \alpha)$ means that α can be reformulated into a procedure of the form: $(\alpha'; \alpha'')|\beta$ which has the same semantics as α , i.e. $\forall s \forall s' (Do_p(\alpha, s, s') \leftrightarrow Do_p((\alpha'; \alpha'')|\beta, s, s'))$.

The condition 1 is expressed by $\exists \alpha' (start(\alpha', \alpha) \wedge \exists s_1 (s_1 \leq s' \wedge Do_m(\alpha', s, s_1))$, the strict interpretation of condition 2 is expressed by $\neg \exists s_2 (s_2 \leq s' \wedge Do_m(\alpha, s, s_2))$, and the condition 3 is expressed by $\exists s_3 (s' < s_3 \wedge Do_m(\alpha, s, s_3))$.

Finally, the definition of $Doing(\alpha, s, s')$ is:

$$Doing(\alpha, s, s') \stackrel{\text{def}}{=} \exists s_1 (s \leq s_1 \wedge Do_s(\alpha, s_1, s')) \wedge \neg \exists s_2 (s \leq s_2 \wedge s_2 < s_1 \wedge Do_s(\alpha, s_2, s_1))$$

The condition $\exists s_1 (s \leq s_1 \wedge Do_s(\alpha, s_1, s'))$ expresses that there is an execution of α that has begun in s_1 and has not ended, and the condition $\neg \exists s_2 (s \leq s_2 \wedge s_2 < s_1 \wedge Do_s(\alpha, s_2, s_1))$ expresses that there is no previous α exe-

²In this paper we do not investigate how to reformulate a procedure in that form. However, we can imagine that a similar transformation as the transformation into disjunctive normal form of formulas of Propositional Calculus should work here. Look, for instance, how the procedure $\alpha \stackrel{\text{def}}{=} (((a; a')|(b; b'))|c)$ can be reformulated into $(a; a'; c)|(b; b'; c)$.

cution which has started and not ended before s_1 .

We have to pay a special attention to the kind of actions whose repetition, in some contexts, produces no new effect. Indeed, if such action appear in the definition of a procedure, the repeated occurrences of this action in an observed sequence should not be interpreted as belonging to several distinct procedure executions.

For instance, let us assume now that the command to extend or to retract the landing gear is not implemented by a switch which can be set in two distinct positions, but by two push buttons, one which has the effect to extend the gear, and the other which has the effect to retract the gear.

In that case, if the pilot pushes the button to extend the gear (action *push.ext*) and if he repeats this action some time later, without having pushed the button to retract the gear (action *push.ret*), the repetition of *push.ext* has no effect on the landing gear.

For example, the sequence of actions: *push.ext*, *call.atc*, *push.ext* may be considered to be equivalent to *push.ext* with respect to some procedure. In particular, if we have to formally represent a landing procedure³ which says that landing gear has to be extended and then flaps have to be extended (action *extend.flaps*) the two above sequence of actions are “equivalent”.

This procedure could be expressed by :

$landing \stackrel{\text{def}}{=} \text{until } \neg done(extend.flaps)? do (push.ext; \sigma) / push.ret \text{ enduntil}$

where $done(a, s)$ is defined by $done(a, s) \stackrel{\text{def}}{=} \exists s' s = do(a, s')$.

In general, a procedure where α may be repeated several times before to execute an action b can be represented by: $\text{until } \neg done(b) do \alpha \text{ enduntil}$.

6. TEMPORAL CONSTRAINTS

There are many procedure descriptions that refer to temporal constraints. For instance, before to take off a pilot has to receive the authorisation from air traffic control, but this authorisation is valid for a limited period of time. If the pilot, for any reason, does not take off in this limited period of time, the authorisation is no more valid. There are other examples where the pilot has to wait a given amount of time before to perform the next action in a procedure.

In the Situation Calculus actions are supposed to be instantaneous, that is, they have no duration. This assumption may seem to be counterintuitive depending on the meaning assigned to an action. If an action performance is understood as what happens when the truth value of a fluent comes to change, then it is sensible to say that there is an instant of time, which has no duration, where a fluent comes to be true, or false, and there is no time where its truth value is undefined.

However, there is another possible interpretation of an action performance where the action starts at a given time and it ends at a further time. If one prefer this interpretation, it is possible, as R. Reiter do to represent concurrent actions in [6], to define for a given action a two corresponding instantaneous actions called *start.a* and *end.a*. For example, if we consider that the action to extend the flaps is not instantaneous, because a button that activates an electric engine that extends the flap has to be pushed until the flaps are completely extended, we can define the instantaneous

³Of course, this is an oversimplified description of the landing procedure, the real procedures are more complex.

actions: *start.extendflaps* and *end.extendflaps*.

Therefore, we can in both approaches consider that actions are instantaneous, and we only have to adapt the definition of actions (either a , or *start.a* and *end.a*). Nevertheless, we need some syntactical means in the language to represent the fact that “time flies”. In our approach, we have defined for that purpose the designated action t , and we have assumed that after performance of t one unit of time has passed. The fact that, for example, the performance of the action a has taken four time units can be represented by: $s' = do([start.a, t, t, t, end.a], s)$. Another action b may have started during a execution, in that case we have: $s' = do([start.a, t, start.b, t, t, end.a], s)$, and the duration is the same.

In the following a technique to represent temporal constraints is presented which can be used for both approaches, and an action symbol a may denote as well an action which has no duration, or the beginning or the end of an action which has a duration.

We shall use t^1 to denote t and t^n to denote $t^{n-1}; \sigma / t; t$.

The fact that at most n time units have passed during the execution of the procedure α is represented by:

$$atmost(\alpha, t^n) \stackrel{\text{def}}{=} \alpha / t^{n+1}$$

To represent the fact that at least n time units have passed during α execution we have introduced a composition operator denoted by $||$. Intuitively $\alpha_1 || \alpha_2$ means that α_1 execution and α_2 execution are interleaved. The formal definition of this operator is:

$$Do_p(\alpha_1 || \alpha_2, s, s') \stackrel{\text{def}}{=} Do_p(\alpha_1, s, s') \wedge Do_p(\alpha_2, s, s')$$

For example, if we have $\alpha_1 \stackrel{\text{def}}{=} \sigma; a; \sigma; b; \sigma$ and $\alpha_2 \stackrel{\text{def}}{=} \sigma; c; \sigma; d; \sigma$, $Do_p(\alpha_1 || \alpha_2, s, s')$ holds for $s' = do([a, c, d, b], s)$, and for $s' = do([c, e, d, a, b], s)$.

The fact that at least n time units have passed during α execution is represented by:

$$atleast(\alpha, t^n) \stackrel{\text{def}}{=} \alpha || (\sigma; t^n; \sigma)$$

Finally, the fact that exactly n time units have passed during α execution is represented by:

$$exactly(\alpha, t^n) \stackrel{\text{def}}{=} (\alpha / t^{n+1}) || (\sigma; t^n; \sigma)$$

If we have to express more complex temporal constraints we can introduce in the language a designated functional symbol $time(s)$ which defines what time it is in the situation s . The successor state axiom for this functional fluent is:

$$\forall s \forall a \forall x (time(do(a, s)) = x \leftrightarrow (a = t \wedge time(s) = x - 1) \vee time(s) = x \wedge \neg(a = t))$$

If we assume, for example, that $time(S_0) = 0$, then temporal constraints can be imposed in procedure definitions like in: $a; ((6 \leq time) \wedge (time \leq 10)); b$. This constraint imposes that action a must be executed before some instant in the interval $[6, 10]$, and b has to be executed after some instant in this interval.

7. PROTOTYPE IMPLEMENTATION

A subset of the procedure definition language has been implemented in PROLOG. This subset does not contain temporal constraints nor the nondeterministic iteration operator. The only reason for these limitations was the lack of time, but they do not raise difficult problems.

Another limitation is that procedures must be in a particular normal form that makes quite easy the decomposition of a procedure in the form $(\alpha'; \alpha'') || \beta$. This normal form is called “linear normal form”.

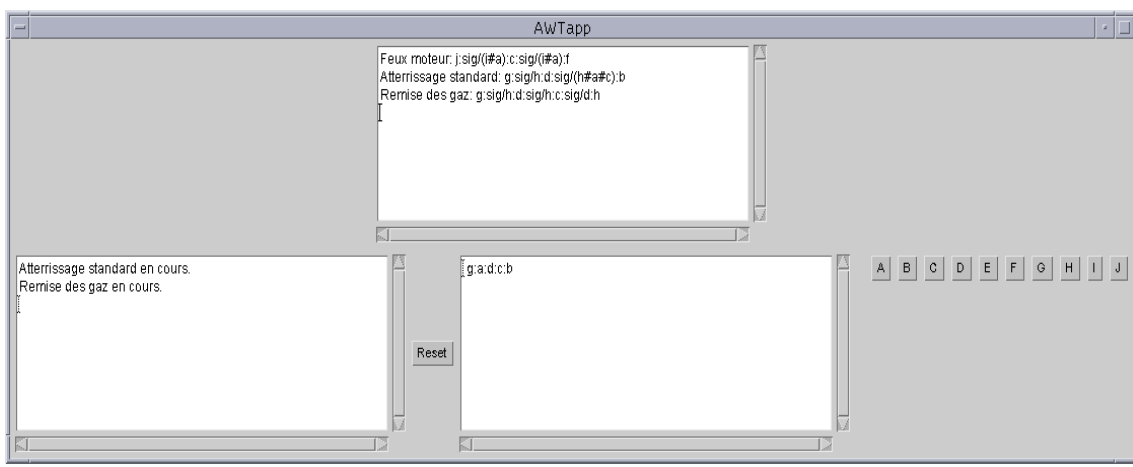


Figure 1: An interface to simulate procedure recognition.

A procedure in linear normal form has the form: $p_1|p_2|\dots|p_n$, where each p_i has the form $\alpha_1;\alpha_2;\dots;\alpha_m$. Each α_j has the form: *atom* or *atom*/ q_l , where *atom* is an atomic action, or a test action or σ , and q_l is a procedure in linear normal form without / operator.

For a given sequence of observed actions $\langle s, s' \rangle$, the PROLOG program determines whether a given procedure α in linear normal form satisfies $Doing(\alpha, s, s')$.

An interactive interface between a user and the PROLOG program has been implemented in JAVA. The user has to give to the interface a set of procedures which have to be matched with observed actions. This set of procedures is presented in the window in the top of the screen (see Figure 1). In the screen the set of buttons A, B,...,J represents pilot commands. The user can click on the buttons to define the sequence of actions performed by the pilot. This sequence of actions is presented in the window on the bottom on the right. After each command “performed” by the pilot (in fact, the interface user) the set of procedures that satisfy the *Doing* predicate are dynamically presented in the window on the bottom on the left.

This kind of simulation of the recognition process of on doing procedures has been useful to validate the formal definition of the *Doing* predicate.

In our application domain the buttons A, B,...,J have been respectively assigned the meaning of the actions a, b, \dots, j which are intuitively defined below:

- a : to push the stick, b to pull the stick,
- c : to set full throttle, d to set stoped throttle,
- e : to set rich air/fuel mixture, f : to set poor air/fuel mixture,
- g to extend the flaps, h : to retract the flaps,
- i : to turn on fuel feed, j : to turn off fuel feed.

We also have considered the three following procedures:

- Standard landing (“Atterrissage standard” in french) where after to extend the flaps, the pilot has to set stoped throttle and to pull the stick:

$$\alpha = g; (\sigma/h); d; (\sigma/(h|c|a)); b$$

- Avorted landing (“Atterrissage avorté” in french) where after to extend the flaps and to set stoped throttle, like in standard landing, the pilot has to set full throttle and to pull the stick in order to climb:

$$\beta = g; (\sigma/h); d; (\sigma/h); c; (\sigma/d); h$$

- Fire on board (“Feux moteur” in french) has been previously presented in the section 4:

$$\gamma = j; (\sigma/i); c; (\sigma/(i|d)); f$$

The Figure 1 shows that in the situation $s' = do([g, a, d, c, b], s)$ the program has recognised that the pilot may be “Doing” the procedure α or β .

8. CONCLUSION

An extension of the GOLOG language has been presented in the framework of Situation Calculus that allows to represent typical procedures. It has been shown that these extensions are required to fit the difference between programs and typical procedures.

The formal semantics of the *Doing* predicate gives a precise answer to the question: “What does it mean that an agent is performing a typical procedure?”. We do not pretend that this definition is THE definition of *Doing*, but it has the great benefit, with respect to some other works where intention recognition (or chronicle recognition) is defined in the framework of Petri nets or of some ad hoc procedural language, that the proposed answer has a clear meaning.

If, for some other application domain, the definition has to be changed, we can precisely identify the part of the definition which has to be modified with limited risk of side effects.

In addition, this logical definition is relatively close to the PROLOG language and that makes easier its implementation. The presented implementation is not complete but its user interface allows, in some sense, to simulate the pilot’s behaviour and the procedure recognition process.

Further work will be to define among the procedures that satisfy the *Doing* predicate the procedure to which is assigned the pilot’s intention.

If the final goal is to control the pilot, the choice may be postponed until the pilot performs some action which is

not compatible with some previously recognised procedure α . That is, an action a such that we have $Doing(\alpha, s, s')$ and $\neg Doing(\alpha, s, do(a, s'))$. Indeed, this kind of situation can be interpreted in two different ways:

1. the system has considered that the procedure α was compatible with the pilot's intention, and the system was wrong, i.e. the pilot's intention was not to perform α ,
2. the pilot's intention is to perform α but he made an error in performing the action α , and the system has to inform the pilot about his error.

It is definitely not easy to determine whether we are in the case 1 or 2, and we certainly need additional information, that may depend on the application domain, to make the choice.

A possible way to determine in which case we are, that can be investigated in the Situation Calculus, is to "simulate" for the pilot what will happen if he continues to perform α . Indeed, the successor state axioms allow to know properties that will hold after performance of the part α'' of the procedure α that remains to be executed. . By "showing" to the pilot what he is doing, he might tell the system whether that correponds, or not, to his intention.

9. ACKNOWLEDGEMENTS

We would like to thank Yves Lespérance for fruitfull comments on this paper. Thanks also to David Novick for his help in removing english mistakes.

10. REFERENCES

- [1] M. Ghallab. On chronicles: representation, on-line recognition and learning. In *Proceedings of 5th International Conference on Principles of Knowledge Representation and Reasoning (KR'96)*, 1996.
- [2] M. Ghallab. Chronicles as a practical representation for dealing with time, events and actions. In *Proceedings of 6th italian Conference on Artificial Intelligence (AIIA '98)*, 1998.
- [3] Y. Lespérance, H.J. Levesque, F. Lin, and R.B. Scherl. Ability and Knowing How in the Situation Calculus. *Studia Logica*, 60(1), 2000.
- [4] H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *Journal of Logic Programming*, 31:59–84, 1997.
- [5] C. R. Perrault and J. F. Allen. A plan-based analysis of indirect speech acts. *Journal of Computational Linguistics*, 6:167–182, 1980.
- [6] R. Reiter. Sequential, temporal GOLOG. In *Proceeding 6th International Conference on Principles of Knowledge Representation and Reasoning*, 1998.
- [7] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
- [8] D. Sadek. A study in the logic of intention. In *Proc. of the 3rd Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, 1992.
- [9] R.C. Schank and R.P. Abelson. *Scripts, Plans, Goals, and Understanding*. Erlbaum, Hillsdale, NJ, 1977.