On Procedure Recognition in the Situation Calculus

Jorge A. Baier
Departamento de Ciencia de la Computación,
Pontificia Universidad Católica de Chile
Casilla 306, Santiago 22, Chile
jabaier@ing.puc.cl

Abstract

The aim of our ongoing research is to give a method to construct intelligent tutoring systems for agents who are executing typical procedures *in dynamic environments based on a logical framework. Typical procedures are similar to plans in the sense that they describe the actions an agent should execute to achieve a certain goal. In this paper we address what we consider is the first step toward the construction of this kind of systems: procedure recognition.*

We formalize what does it mean that an agent is performing a procedure in the Situation Calculus [12], a logical first-order language extended with induction. Based upon this formalization, we give two different implementations. The first, which is directly based in our formalization, is proved to be quite inefficient. The second, significantly more efficient, arises from a logical reformulation of the original formalization. Procedures are represented through ConGolog *[6] programs, a logical interpreted language based on the Situation Calculus.*

1 Introduction and Motivation

Plan recognition [8] is a well-known research area in automated reasoning. Applications of this area range from intelligent tutoring systems for natural language processing [11] to military simulation [7].

The aim of our ongoing research is to give a method to construct intelligent tutoring systems for agents who are executing *typical procedures* in dynamic environments based on a logical framework. A typical procedure is a course of action that an agent may execute to accomplish a goal. We distinguish the *course of action* and the *goal* as two components of a procedure. A procedure can be carried out concurrently with other tasks. We assume that procedures are stored in *procedure libraries*, which can be regarded as the users' manual of the system being operated by the agent.

To exemplify these notions consider, for example, an aircraft pilot which commands an aircraft following the standard know-how of piloting. It can be argued that this know-how represents a mental compilation of procedures to achieve several goals, such as landing or taking off. The pilot performs a typical procedure in accordance with his intentions, i.e. it executes the takeoff procedure when he intends to takeoff. Nevertheless, while executing a procedure, the pilot may execute, concurrently, other actions which do not necessarily appear in the procedure description. For example, while landing, the pilot may establish several conversations with the control tower or with the members of the crew without altering the degree of success of the goal.

If one wants to construct an intelligent system to aid and monitor the actions of an autonomous intelligent agent we must first address the problem of recognizing the agent's intentions. We claim that this necessarily implies the recognition of the procedures the agent is executing at a particular moment. Once the intentions of the agent are recognized, the tutoring system can, on request, aid the agent by suggesting what actions she may perform for accomplishing her goals. On the other hand, the system may deliberately suggest corrections to the agent's behavior if it determines that the agent's attitudes contradict her intentions.

In this paper we address the first part of the problem, i.e. procedure recognition. We propose a formalization of procedure recognition in ConGolog [6], an interpreted logical language developed for programming intelligent agents. The ConGolog language is based on its predecessor Golog [10]. The main difference between them is that the former can adequately represent concurrent processes and, therefore, represent procedures in a more general way. ConGolog's semantics is based on the Situation Calculus [12], a family of logical languages developed to logically represent the dynamics of the world. The modeling of procedures in these languages has the following advantages:

1. It is possible to represent a wide variety of programs using ConGolog; in particular, traditional programs

such as those written in the *C* language. However, CONGOLOG was devised as a language for programming intelligent agents and is therefore appropriate for representing actions being carried out by such kind of agents. Furthermore, since a CONGOLOG program is not restricted to be executed by a single agent, it can also represent programs executing in concurrent environments.

2. A CONGOLOG program has a complete account of the environment in which it is executed through a Situation Calculus model of the world. A program can reason about the hypothetical evolution of the world after executing a sub-program without actually executing it. Furthermore, a monitoring system can predict the results of a program being executed by an agent before the agent finishes to execute the program.

3. CONGOLOG semantics and Situation Calculus's theories of action can be straightforwardly implemented in PROLOG. Indeed, we have implemented our formalizations.

There has been a previous attempt to formalize procedures in the GOLOG language [4]. However, since this approach is built upon GOLOG's semantics it cannot represent concurrent procedures. We think that concurrent procedures are commonly carried out by intelligent agents. Consider, for example, the procedure of making *spaghetti bolognesa*. One could argue that this procedure is composed by a series of steps, some of which need not be executed in a particular order. For example, to make that dish one can start by making the sauce and afterwords boil the spaghetti, or vice versa. The subprocess of making sauce can also be decomposed in other concurrent subprocesses. Furthermore, the approach of [4] is rather different with respect to the notion of procedure. We discuss about this in a later section.

This paper is organized as follows. In section 2 we present a brief introduction to the Situation Calculus and how theories of action are built using this language. In section 3 we present a brief introduction to the CONGOLOG language. In section 4 formalize the notion of procedure execution within the Situation Calculus. In section 5 we discuss implementation issues, giving the first step toward efficient implementations. In section 6 we refer to related work. Finally, in section 7, we conclude giving guidelines for future work.

2 The Situation Calculus

In this section we briefly introduce the Situation Calculus, the logical language upon which CONGOLOG semantics is based. We introduce the basic concepts of the language and show how a domain can be modeled by way of an example that will be used throughout the paper. For further details on the Situation Calculus, refer to [17, 14].

2.1 The Situation Calculus

The Situation Calculus is a many-sorted, first-order logical language extended with induction. We use the sorts $\mathcal{S}$, $\mathcal{A}$, $\mathcal{F}$ and $\mathcal{D}$ for situations, actions, fluents, and domain objects, respectively. Situations represent a *snapshot* of the world plus a history of the evolution of the world. Actions are regarded as the only reason by which the world evolves from one situation to another. Fluents are first-order functional terms which denote properties of the world that are static in a situation. For example, the binary fluent *over* can be such that $over(x,y)$ means that x is over y. On the other hand, fluent formulas are like first-order formulas in which every atomic sub-formula is a fluent. For example $(\forall x)\, over(x, Table)$ may be used to represent the fact that all objects x are over $Table$.

The following are distinguished elements of the language:

S_0 Is a constant which denotes the initial situation. This situation represents the world before anything has occurred.

$do : \mathcal{A} \times \mathcal{S} \to \mathcal{S}$ If a denotes an action and s denotes a situation, then $do(a,s)$ denotes the situation which results from executing action a in situation s. For notational convenience we frequently write $do([a_1, a_2, \ldots, a_n], s)$ instead of $do(a_n, do(\ldots, do(a_2, do(a_1, s))\ldots))$.

$holds : \mathcal{F} \times \mathcal{S}$ If f is a fluent and s is a situation, then $holds(f,s)$ is true if and only if f holds in situation s [1]. This predicate can be straightforwardly extended to fluent formulas.

$Poss \subseteq \mathcal{A} \times \mathcal{S}$ states when an input action is possible in a situation, i.e. $Poss(a,s)$ is true if and only if it is possible to execute action a in situation s.

$\prec \subseteq \mathcal{S} \times \mathcal{S}$ Is a binary predicate which represents a partial order between situations. Normally used infix, $s \prec s'$ is true if and only if it is possible to reach situation s' by executing a positive number of actions in s. We use the abbreviation $s \preceq s'$ to denote $s \prec s' \vee s = s'$.

To precisely define the structure of situations it is necessary to give a small set of foundational axioms, which we omit here for the sake of briefness.

[1]This approach differs from that of [14] since fluents are reified, i.e. are not predicates but objects of the language.

2.2 Theories of Action

To model a particular domain, it is necessary to write several axioms. We will show how to do this by modeling the following aeronautic domain.

Example 1: An agent piloting an aircraft can execute the following actions.

Extend/Retract flaps : have the effect of extending/retracting the flaps one position (the aircraft has 3 flaps positions, where 3 indicates fully retracted and 0 indicates fully extended).

Reduce/Increase thrust : have the effect of reducing/increasing the thrust in the engines. Thrust level can be reduced or increased discretely by an amount of 100. The maximum and minimum amounts of thrust are respectively 0 and 500.

Extend/Retract Landing Gear : Have the effect of extending/retracting the landing gear.

Establish ATC communication : Establish a communication with air traffic controllers.

To model this domain we use the actions terms *retFlaps*, *extFlaps*, *incThrust*, *decThrust*, *retGear*, *extGear*, with obvious meanings. The following are the fluents used:

$\mathit{flpLevel}(n)$ True when n is a number describing the state of the flaps.

$\mathit{thrLevel}(n)$ True when n is a number that represents the thrust produced in the engines.

gearIsRet True iff landing gear is retracted.

The set of axioms needed to construct a theory of action are the following.

Action Precondition Axioms A set Σ_{prec} which establishes all the necessary and sufficient conditions for an action to be executed. Their general form is [2]

$$(\forall \mathbf{\bar{x}})\,(\mathit{Poss}(a(\mathbf{\bar{x}}),s) \equiv \Pi(\mathbf{\bar{x}},s)),$$

where $\Pi(\mathbf{\bar{x}},s)$ is a first-order formula simple on s [3].

[2]We use the notation $\mathbf{\bar{x}}$ to denote a tuple of variables of sort domain-object.

[3]A formula simple on s if the only situation term it contains is s, and it does not quantify over s.

Example (cont.): The precondition axioms for the aeronautics domain are[4]:

$$\mathit{Poss}(\mathit{retFlaps},s) \equiv (\exists n)\,\mathit{holds}(\mathit{flpLevel}(n),s) \wedge n < 3 \quad (1)$$

$$\mathit{Poss}(\mathit{extFlaps},s) \equiv (\exists n)\,\mathit{holds}(\mathit{flpLevel}(n),s) \wedge n > 0 \quad (2)$$

$$\mathit{Poss}(\mathit{incThrust},s) \equiv (\exists n)\,\mathit{holds}(\mathit{thrLevel}(n),s) \wedge n < 500 \quad (3)$$

$$\mathit{Poss}(\mathit{decThrust},s) \equiv (\exists n)\,\mathit{holds}(\mathit{thrLevel}(n),s) \wedge n > 0 \quad (4)$$

$$\mathit{Poss}(\mathit{retGear},s) \equiv \neg \mathit{holds}(\mathit{gearIsRet},s) \quad (5)$$

$$\mathit{Poss}(\mathit{extGear},s) \equiv \mathit{holds}(\mathit{gearIsRet},s) \quad (6)$$

$$\mathit{Poss}(\mathit{estblATC},s) \equiv \mathit{True} \quad (7)$$

Effect Axioms A set Σ_{eff} of effect axioms, which describe the direct effect of actions. There are two kinds of effect axioms: positive effect axioms (with the syntactical form of (8)) and negative effect axioms (like (9)). We write one of each for every fluent. For a fluent f, they have the following general form:

$$\mathit{Poss}(a,s) \wedge \gamma_f^+(\mathbf{\bar{x}},s) \supset \mathit{holds}(f(\mathbf{\bar{x}}),do(a,s)) \quad (8)$$

$$\mathit{Poss}(a,s) \wedge \gamma_f^-(\mathbf{\bar{x}},s) \supset \mathit{holds}(f(\mathbf{\bar{x}}),do(a,s)) \quad (9)$$

Although these axioms completely specify when the truth value of fluents change from one situation to another, they do not specify when these properties do not change. The problem of specifying succinctly what does not change is known as the *frame problem*. For example, for the aeronautics domain, one can have a positive effect axiom to state when *gearIsRet* becomes true and a negative effect axiom to state when is becomes false; in fact, this happens exactly when, respectively, *retGear* and *extGear* are executed. Such effect axioms, however, do not specify that when an ATC communication is established the state of the landing gear persists. In [16], Reiter presents a solution to the frame problem. He proposes that *successor state axioms*, one for each fluent, can be mechanically derived from direct effect axioms. These axioms have the following syntactical form:

$$\mathit{Poss}(a,s) \supset [\mathit{holds}(f(\mathbf{\bar{x}}),do(a,s)) \equiv (\gamma_f^+(\mathbf{\bar{x}},a,s) \vee \mathit{holds}(f(\mathbf{\bar{x}}),s) \wedge \neg\gamma_f^-(\mathbf{\bar{x}},a,s))],$$

and we gather them in a set Σ_{ssa}. These axioms establish necessary and sufficient conditions for the change of fluent values between successive situations, provided that $\gamma_f^+(\mathbf{\bar{x}},a,s)$ and $\gamma_f^-(\mathbf{\bar{x}},a,s)$ are never simultaneously true.

[4]All free variables in formulas are supposed implicitly universally prenex quantified.

Example (cont.): The frame axioms for the aeronautics example are:

$$\begin{aligned} Poss(a,s) \supset [&holds(flpLevel(n),do(a,s)) \equiv \\ &(holds(flpLevel(m),s) \wedge \\ &((a = retFlaps \wedge n = m+1) \vee \\ &\ (a = extFlaps \wedge n = m-1))) \vee \\ &(holds(flpLevel(n),s) \wedge \\ &\neg a = extFlaps \wedge \neg a = retFlaps)], \end{aligned} \tag{10}$$

$$\begin{aligned} Poss(a,s) \supset [&holds(thrLevel(n),do(a,s)) \equiv \\ &(holds(thrLevel(m),s) \wedge \\ &((a = incThrust \wedge n = m+100) \vee \\ &\ (a = decThrust \wedge n = m-100))) \vee \\ &(holds(thrLevel(n),s) \wedge \\ &\neg a = incThrust \wedge \neg a = decThrust)], \end{aligned} \tag{11}$$

$$\begin{aligned} Poss(a,s) \supset [&holds(gearIsRet,do(a,s)) \equiv \\ &(holds(thrLevel(m),s) \wedge a = retGear) \vee \\ &(holds(gearIsRet,s) \wedge \neg a = extGear], \end{aligned} \tag{12}$$

Unique Name Axioms for Actions and Domain Objects A set Σ_{una} containing unique name axioms for terms denoting actions, fluents, and constants which denote domain objects.

Example (cont.): For our example, the unique name axioms are the following:

$$\begin{aligned} &\neg incThrust = decThrust, \neg incThrust = retFlaps, \\ &\neg incThrust = extFlaps, \ldots \\ &thrLevel(x) = flpLevel(y), \ldots \end{aligned}$$

Initial Description Axioms A set Σ_{S_0} which describes the initial situation. **Example** (cont.): For our example, initial description axioms could be

$$\begin{aligned} holds(x,S_0) \equiv [&x = gearIsUp \vee \\ &x = flpLevel(n) \wedge n = 500 \vee \\ &x = thrLevel(n) \wedge n = 3]. \end{aligned}$$

Definition 1 (Theory of Action) *A theory of action in the Situation Calculus is a set of first-order sentences containing the foundational axioms together with precondition, successor state, unique name, and initial description axioms.*

3 Introduction to CONGOLOG

The CONGOLOG [6] language is an interpreted, logical language designed for programming intelligent agents at a high abstraction level. CONGOLOG's semantics is based on the Situation Calculus. The semantics of the CONGOLOG language is written as logical axioms; therefore, the execution of programs can be fully analyzed at a logical level. CONGOLOG dialect is based on the original GOLOG language proposition [10]. The main difference between GOLOG and CONGOLOG is that the latter can adequately represent concurrent processes as interleaving[5]. Moreover, their semantics is defined differently: GOLOG programs are macros which expand to formulas in the Situation Calculus; CONGOLOG programs are logical objects whose semantics is defined using logical predicates.

CONGOLOG programs contain constructs present in traditional structured programming languages. Formally, a CONGOLOG program can have the following elements:

Primitive Actions and Test Conditions

- Primitive actions, denoted by α (possibly with subscripts). They are atomic actions. In a single step of execution of the program, the interpreter can execute at most one of these actions.
- ϕ?: test conditions. Here ϕ is a fluent formula. These formulas play the same role in GOLOG as boolean expressions in traditional imperative languages such as C or Pascal.

Complex Actions Are denoted by the letters σ and δ and can be defined as:

- $\{\}$, is a complex action which denotes the *empty* program.
- α, a primitive action, is a complex action.
- If σ_1 and σ_2 are complex actions then the following are also complex actions:
  - $(\sigma_1;\sigma_2)$, is a *sequence of actions*. The execution of the sequence corresponds to the execution of σ_1 followed by σ_2.
  - $(\sigma_1|\sigma_2)$, is a *non-deterministic choice between actions*. The interpreter non-deterministically chooses to execute either σ_1 or σ_2. If, for example, one transition of σ_2 were not possible, the only choice of the interpreter is to execute one transition of σ_1.

[5]However, the language has also been extended to represent true concurrency, i.e. a combination of interleaving and parallelism [3].

– $(\sigma_1||\sigma_2)$, is a *concurrent execution*. Here, complex actions σ_1 and σ_2 may execute concurrently, i.e. there is no particular order imposed to the execution actions belonging to σ_1 or σ_2.

– $\pi x.\sigma$, is a *non-deterministic choice of arguments*. The variable x is nondeterministically instantiated with an object of the domain. σ is executed considering this non-deterministic instantiation.

– σ^*, is a *non-deterministic iteration*. The complex action σ may be executed an arbitrary number of times.

– **if** ϕ **then** σ_1 **else** σ_2 **endIf**, is a *conditional sentence*. σ_1 is executed if ϕ holds, otherwise, σ_2 is executed.

– **while** ϕ **do** σ **endWhile**: *while loops*. σ is executed while ϕ holds.

To simplify this introduction to CONGOLOG we have omitted some constructs that we will not need. In particular, we have omitted the definition of CONGOLOG procedure calls. These procedures, though, do not have the same character as the procedures we formalize in this paper; they correspond to the notion of procedure of traditional programming languages.

Definition 2 (CONGOLOG program) *A* CONGOLOG *program is complex action.*

Definition 3 (CONGOLOG linear program) *A* CONGOLOG *linear program is a* CONGOLOG *program that does not contain conditional sentences, while loops, nondeterministic or concurrency constructs. I.e., a linear program is a sequence of primitive actions, e.g.* $\alpha_1;\ldots;\alpha_n$.

We denote linear programs with the letter σ^-, possibly with subscripts.

Example : Let $put(x,Box)$ denote the action of putting block x inside Box. The program

$$\begin{array}{l}\textbf{while}\ \neg(\forall x)[block(x) \supset in(x,Box)]\ \textbf{do}\\ \quad (\pi x)\ put(x,Box)\\ \textbf{endWhile},\end{array}$$

repeatedly executes action $put(x,Box)$, for an arbitrary block x while there are blocks outside Box. This program illustrates the expressive power afforded by the language. The condition is a quantified, first-order statement.

The GOLOG language also allows for constructs that refer to the *mental state* of the agent that executes the program. For this, constructs have been introduced in order to deal with *knowledge producing actions* [18]; i.e., actions whose sole effect is to change the state of mind of the agent, rather than affect the world. For instance, the action of reading a phone number.

3.1 CONGOLOG's Semantics

What does it mean that CONGOLOG's semantics is defined in the Situation Calculus? Roughly, it means that it is possible to define, for any given program σ and any situation s which situations will be visited by an agent who starts the execution of σ in s.

CONGOLOG's semantics is given by transitional axioms which establish the evolution of a program decomposing it in single steps or *transitions*. To this extent, CONGOLOG semantics exhaustively defines the following predicates for all types of complex actions:

- $Final(\sigma,s)$, which is true if and only if program σ may finish in situation s. For example, the definition states that an empty program may finish in any situation.
- $Trans(\sigma,s,\sigma',s')$, which is true if and only if σ is executed in s, in a single step of execution, it may reach situation s', with program σ' remaining to be executed.

Example 2: Suppose program $\sigma = \alpha_1;$ **if** ϕ **then** α_2 **else** α_3 is executed in situation s. From CONGOLOG's axiomatization it follows that

- $Trans(\sigma,s,\sigma',do(\alpha_1,s))$ is true when $\sigma' =$ **if** ϕ **then** α_2 **else** α_3, provided that α_1 is possible in situation s.
- $Trans(\sigma',do(\alpha_1,s),\alpha_2,do(\alpha_1,s))$ is true iff fluent formula ϕ holds in $do(\alpha_1,s)$.
- $Trans(\sigma',do(\alpha_1,s),\alpha_3,do(\alpha_1,s))$ is true iff fluent formula ϕ does not hold in $do(\alpha_1,s)$.
- $Trans(\alpha_2,do(\alpha_1,s),\{\},do(\alpha_2,(do(\alpha_1,s)))$ is true iff α_2 is possible in $do(\alpha_1,s)$.

It is possible the transitive closure of $Trans$, $Trans^*$ in terms of $Trans$ and $Final$. Thus, $Trans^*(\sigma,s,\sigma',s')$ holds iff when program σ is started in situation s it can get to situation s', in 0 or more transitions, with program σ' remaining to be executed. Furthermore, the predicate macro Do is defined by

$$Do(\sigma,s,s') \stackrel{\text{def}}{=} (\exists\sigma')\,Trans^*(\sigma,s,\sigma',s') \wedge Final(\sigma',s').$$

$Do(\sigma,s,s')$ is true if and only if program σ, when started in situation s may halt in situation s'.

4 Defining and Executing Procedures

For us, a procedure is very similar to a plan in the sense that the agent executes a course of action to achieve a goal. In [9] it was argued that an appropriate representation of a plan in the presence of sensing is a GOLOG program. We consider that a suitable representation for procedures are CONGOLOG programs since they provide even more generality.

The following definition states what is, for us, a procedure.

Definition 4 (Procedure) *A procedure τ is a pair $\langle\sigma,\varphi\rangle$, where φ is a fluent formula which represents the goal being pursued by the procedure, and σ is a* GOLOG *program which represents the steps the agent should execute to achieve the goal.*

To simplify the exposition we have defined the operators $proc(\langle\sigma,\varphi\rangle) \stackrel{\text{def}}{=} \sigma$ and $goal(\langle\sigma,\varphi\rangle) \stackrel{\text{def}}{=} \varphi$ as syntactic sugar.

Example 3: We can define a simplified landing procedure for the aeronautics domain as $\tau_{land} = \langle\delta_{land}, \phi_{ready}\rangle$, were

$$
\begin{aligned}
\delta_{land} \stackrel{\text{def}}{=} & \, extFlaps; extFlaps; extFlaps; \\
& \textbf{while } thrLevel(n) \wedge n \geq 100 \textbf{ do} \\
& \quad decThrust \\
& \textbf{endWhile}; \\
& extGear.
\end{aligned}
$$

and $\phi_{ready} \stackrel{\text{def}}{=} thrLevel(100) \wedge flpLevel(0) \wedge \neg gearIsRet$

In order to be able to recognize the execution of procedures on-line, it is necessary to precisely define what does it mean that a procedure is being performed. Although we consider that procedures can be represented as a program, the problem of determining if a procedure is being performed and the problem of determining if a program is being executed are quite different.

When performing a procedure, agents may concurrently execute other actions. We say that the procedure is being performed as long as these actions do not occlude the achievement of the goal; otherwise, we say that the procedure is not being performed. For instance, suppose a system that can observe the actions carried out by an agent in a kitchen. Furthermore, suppose that the system has observed the following actions: (1) agent chops meat (2) agent fries a chopped onion (3) agent answers the phone. If both frying the onion and chopping the meat are part of a procedure which describes how to cook a spaghetti bolognesa, then we could infer from the observations that the agent is making spaghetti bolognesa. Observe that we can infer this even when answering the phone has nothing to do with the process of making spaghetti. Nevertheless, if observation (3) is "agent dumps the meat and the onion" we can no longer infer that the agent is cooking that dish.

From a more abstract point of view, we say a procedure $\tau = \langle\sigma,\phi\rangle$ is being performed in situation s if

1. concurrently with σ, any actions can occur, provided that:
2. there exists a situation in the past of s, where the execution of program σ started;
3. if σ' is a portion of σ that remains to be executed, then there exists a situation in the future of s where the goal can be achieved by executing σ' in s, and
4. program σ has not finished in the past of s.

To formalize these conditions, we define the abbreviation *Perf* such that $Perf(\tau,\delta',s)$ is true if and only if procedure τ is being performed in situation s and δ' (a segment of $proc(\tau)$) remains to be executed in s. The predicate can be defined by the following axiom:

$$
\begin{aligned}
& Perf(\tau,\delta',s) \stackrel{\text{def}}{=} (\exists s_p, s_f, \sigma_c^-, \delta). \\
& \quad \delta = proc(\tau) \wedge s_p \prec s \wedge \\
& \quad Trans^*(\delta || \sigma_c^-, s_p, \delta', s) \wedge \\
& \quad (\exists a, s'_p, \delta'')\,(s_p \preceq s'_p \preceq do(a,s'_p) \preceq s \wedge \\
& \qquad\qquad Trans^*(\delta, s_p, \delta'', do(a,s'_p))) \wedge \\
& \quad \neg(\exists \sigma^-)\, Trans^*(\sigma_c^-, s_p, \sigma^-, s) \wedge \\
& \quad (\forall \delta_1, \sigma_2^-)\,(\delta' = \delta_1 || \sigma_2^- \supset \neg(\exists \sigma_1^-)\sigma_c = \sigma_1^-;\sigma_2^-) \wedge \\
& \quad (\exists s_f)\,(s \prec s_f \wedge Do(\delta', s, s_f) \wedge holds(goal(\tau), s_f)).
\end{aligned}
\tag{13}
$$

Intuitively, the first two lines of the definition (without considering the initial existential quantification) are necessary and sufficient to establish condition 1. Notice that, without loss of generality, we restrict the concurrent sequence of actions σ_c^- to be a linear program. The third and fourth lines say, roughly, that an action a, executed in the past of s, has been originated from δ, and therefore it establishes condition 2. The fifth line of the definition may seem redundant but it is useful for implementation purposes, since it facilitates the computation of the exact segment of the program that has been executed in the past of s. Finally, conditions 3 and 4 are set by the last remaining lines of the definition.

Now we can define the macro[6] $Performed(\tau,s)$ intended to hold exactly when procedure τ is being performed in situation s.

$$Performed(\tau,s) \stackrel{\text{def}}{=} (\exists\delta)\, Perf(\tau,\delta,s) \tag{14}$$

Remark 1 *Let τ_{land} be defined as in example 3. Furthermore, let*

$$
\begin{aligned}
& S_1 = do([extFlaps, estblATC, extFlaps], S_0), \\
& S_2 = do([extFlaps, retFlaps, estblATC], S_0), \\
& S_3 = do([extFlaps, estblATC, retFlaps, extFlaps], S_0).
\end{aligned}
$$

[6]From now on we may refer to *Perf* or *Performed* as predicates. The reader, though, must be aware that these are actually macro definitions.

From the theory of action of example 1, CONGOLOG *semantics axioms and* (14)*, it follows that*

$$Performed(\tau_{land}, S_1) \wedge \neg Performed(\tau_{land}, S_2) \wedge \\ Performed(\tau_{land}, S_3).$$

PROOF: Follows directly from (14) and the axioms of the semantics of CONGOLOG. □

The following propositions can be straightforwardly proved, and give an account of the change and persistence of the truth value of the predicate *Performed* due to new observations:

Proposition 1 *Let* τ *be an arbitrary procedure and s be a situation. Furthermore, let* Σ *be a theory containing the Situation Calculus's foundational axioms and* CONGOLOG*'s semantics axioms. From definition* (13) *it follows that*

$$\Sigma \models (\forall\delta)\,(Perf(\tau,\delta,s) \wedge \neg(\exists\delta')Trans^*(\delta,s,\delta',do(a,s)) \wedge \\ GoalSat(\tau,\delta,s) \supset Perf(\tau,\delta,do(a,s))),$$

where $GoalSat(\tau,\delta',s) \overset{def}{=} (\exists s_f)\,(s \prec s_f \wedge Do(\delta',s,s_f) \wedge holds(goal(\tau),s_f))$

In simple words, if τ is being performed in s and a new observation a is made, then if a does not come τ and does not affect the achievement of the goal, then τ is still being performed in situation $do(a,s)$.

Proposition 2 *Let* τ *be an arbitrary procedure and s be a situation. Furthermore, let* Σ *be defined as in proposition 1, then*

$$\Sigma \models (\forall\delta,\delta')\,(Perf(\tau,\delta,s) \wedge Trans^*(\delta,s,\delta',do(a,s)) \supset \\ Perf(\tau,\delta',do(a,s)))$$

In short, if τ is being performed in s and a new observation a is made, then if a comes from τ then τ is still being performed in situation $do(a,s)$.

5 Implementation

A Situation Calculus theory of action, such as that presented in section 2.2 can be straightforwardly implemented in PROLOG. As an example, the PROLOG rules corresponding to axioms (1) and (12) can be implemented as follows:

```
poss([retFlaps],S) :-
        holdsf(flaps_level(N),S),
        N<3.

holdsf(gearIsRet,do(A,S)) :-
        A=[extGear];
        (holdsf(gearIsRet,S),
        \+A=[retGear]).
```

We have also implemented the *Performed* predicate following strictly the definition in (14). It is important to notice that our implementation must find a sequence of action σ_c^- which executes concurrently with $proc(\tau)$. The problem of finding such sequence can be viewed as a search problem. Our implementation does this search trying all possible sequences, starting by shorter ones. For example if the program has to determine whether $Performed(\tau,do([a_1,\ldots,a_n],S_0))$, n generic candidate sequences are tried for σ_c^-, starting by a 0-length sequence and finishing with an n-length sequence.

5.1 Complexity of Direct Implementation

The efficiency of recognition is a central issue in the construction of tutoring systems and therefore a careful consideration must be put in its implementation. Although the *Perf* predicate captures the automation of procedure recognition both at a logical and at an implementation level, its direct translation to a PROLOG program yields an inefficient implementation. In fact, the problem of determining whether a procedure τ is being performed is exponential. The following proposition supports our claim:

Proposition 3 *Let procedure* $\tau = \langle\delta,\phi\rangle$ *be such that* $\delta = \alpha_1;\alpha_2;\ldots;\alpha_k$ *and* ϕ *is an arbitrary fluent formula. Let* $S \overset{def}{=} do([a_1,\ldots,a_n],S_0)$*. The complexity of directly determining whether* $Performed(\tau,S)$*, using a direct implementation of* (14)*, is exponential in n or k.*

PROOF: We distinguish two cases $k > n$ and $k \leq n$ and we analyze the worst case, i.e. that $Performed(\tau,S)$ does not hold. Then the algorithm must check whether n subsequences of δ concurrently executed with some σ_c^- can generate S. This implies that $\sum_{i=1}^{n}\binom{n}{i} = 2^n - 1$ possible combinations must be considered.

In the second case $\sum_{i=1}^{k}\binom{n}{i}$ combinations must be considered, which is $\Omega(n^k)$. □

Unfortunately, the worst case is not uncommon. The agent could be performing a few procedures from a set of M procedures from a procedure library. Since the system does not know what procedure is actually being executed by the agent, every procedure must be checked with every new observation that is made.

5.2 Toward an Efficient Implementation

The main drawback present in the direct translation of the *Perf* definition to a PROLOG rule is that to determine whether a procedure is performed it is necessary to completely check the history of observations. In this sense, the problem of determining the truth value of $Perf(\tau,\delta,s)$ is *non-markovian*; furthermore, it is necessary to determine whether δ will eventually make the goal succeed in the future of s.

To give an efficient implementation, we construct a markovian definition for *Perf*. Thus, we do not need to completely check the history of observations. This is possible since the definition of *Perf* can be rewritten in the following way:

$$Perf(\tau,\delta',s) \overset{\text{def}}{=} (\exists\delta').\, hasStarted(\tau,\delta',s) \wedge GoalSat(\tau,\delta',do(a,s)), \tag{15}$$

where *GoalSat* is defined as in proposition 1 and *hasStarted* is defined in such a way to complete definition given by (13). $hasStarted(\tau,\delta',s)$ essentially says that procedure τ has started in the past of s, and that δ' is the piece of the procedure left for execution and $GoalSat(\tau,\delta',s)$ says that δ' achieves the procedures goal in a situation in the future of s.

If we can establish all the necessary and sufficient conditions by which the truth value of *hasStarted* may change from an arbitrary situation s to a situation $do(a,s)$ we can transform the nature of the problem to markovian. The following propositions show how a new observation may change the truth value of *hasStarted*.

Proposition 4 *Let τ be an arbitrary procedure and s be a situation. Furthermore, let Σ be a theory containing the Situation Calculus's foundational axioms and* CONGOLOG*'s semantics axioms. From hasStarted's definition it follows that,*

$$\Sigma \models hasStarted(\tau,\delta',s) \supset hasStarted(\tau,\delta',do(a,s))$$

In simple words, if τ has started in the past of situation s, then no observation can alter the fact that τ has already started.

Proposition 5 *Let τ be an arbitrary procedure and s be a situation. Furthermore, let Σ be as in proposition 4. Then*

$$\Sigma \models hasStarted(\tau,\delta',s) \wedge Trans^*(\delta',s,\delta'',do(a,s)) \supset hasStarted(\tau,\delta'',do(a,s))$$

This proposition says that if τ has started in the past of s and δ' remains to be executed, then if a new observation a comes from a transition of program δ' which leaves δ'' to be executed, then in situation $do(a,s)$, τ has started and δ'' remains to be executed.

Proposition 6 *Let τ be an arbitrary procedure and s be a situation. Furthermore, let Σ be as in proposition 4. Then*

$$\Sigma \models \neg hasStarted(\tau,\delta',s) \wedge Trans^*(proc(\tau),s,\delta'',do(a,s)) \supset hasStarted(\tau,\delta'',do(a,s))$$

This proposition establishes when a procedure τ begins to be executed; i.e., when a transition of τ correspond to the observation.

Notice that the syntactic structure of propositions 4–6 is very similar to that of effect axioms. The main difference between them is that *hasStarted* is not a fluent but a definition that stands for a more complex formula. Nevertheless, these results suggest that *hasStarted* could be considered as a fluent, i.e. a property which truth value can be associated to a situation.

Moreover, it is not difficult to see that propositions 4–6 state *all* the conditions under which the truth value of *hasStarted* changes from one situation to another, then we define a fluent $hasStarted_F$ such that $hasStarted_F(\tau,\delta')$ is true in situation s if procedure τ could have started in the past of situation s leaving δ' to be executed. Under this assumption, and following [16], we generate the following pseudo successor state axiom[7] for $hasStarted_F$:

$$\begin{aligned} holds(hasStarted_F(\tau,\delta),do(a,s)) \equiv \\ [holds(hasStarted_F(\tau,\delta'),s) \wedge \\ (\delta=\delta' \vee Trans^*(\delta,s,\delta,do(a,s))) \vee \\ Trans^*(proc(\tau),s,\delta,do(a,s))]. \end{aligned} \tag{16}$$

Furthermore, we use the following axiom to establish that in the initial situation, no procedure has started.

$$(\forall\tau,\delta)\,\neg holds(hasStarted_F(\tau,\delta,S_0)) \tag{17}$$

Through the following theorem, we show an exact correspondence between $hasStarted_F$ and *hasStarted*.

Theorem 1 *Let Σ be a theory containing the Situation Calculus's foundational axioms,* CONGOLOG*'s semantics axioms and axioms* (16) *and* (17)*. Then,*

$$\Sigma \models (\forall\tau,\delta,s).\, holds(hasStarted_F(\tau,\delta),s) \equiv hasStarted(\tau,\delta,s)$$

PROOF: Is done by structural induction on s. The $\Rightarrow$ part is straightforward by using propositions 4 and 5. For the $\Leftarrow$ part we use the definition of *hasStarted*. $\Box$

[7]This axiom is not a real successor state axiom because it mentions the term $do(a,s)$ in the right side of the equivalence, which is not allowed in their syntactic form.

The result expressed in this theorem is of extreme importance since we have shown that it is possible to give an implementation that, to determine whether a procedure is performed in a situation $do(a,s)$ has to check for the truth values of properties that hold only in situation s. In particular such a system can keep in memory, for every situation s, the tuples (τ,δ) such that $hasStarted_F(\tau,\delta)$ holds in s. The amount of tuples needed to be stored is combinatorial in the number and length of plans. However, this number is small compared to time complexity of direct implementation, which is exponential on the number of observations.

6 Related Work

There are several works on plan recognition in the literature. We have knowledge about three works that use logic as its theoretical framework. The pioneering work by Henry Kautz [8], more that a decade ago, treats the problem as the inverse of the planning problem. He uses a logical formalism built upon a temporal logic based on Allen's temporal logic [1]. The main difference between this approach and ours is that it assumes that the set of observations is incomplete and therefore circumscription —a predicate minimization technique— is used. Furthermore, the language for describing plans is quite limited; indeed, no iterations or concurrency is allowed. The latter limitation is also present in work by Rao [15], where the plan recognition problem is treated in a bottom-up manner. Given a set of plans, an algorithm generates an *observational plan* composed by a series of sensing actions whose final goal is to recognize the execution of a plan from a plan library. Plans are described in a modal logic which does not support concurrency. The main drawback is that it is unable to recognize two procedures executing concurrently.

The most close approach to ours is the one by Demolombe and Hamon [4], where procedures are regarded as GOLOG programs with restrictions. One of the advantages of our approach with respect to Demolombe and Hamon's is that procedures that involve concurrent programs can also been recognized on-line. For example, consider the procedure $\tau'_{land} = (\delta_{dec}||\delta_{flap}); extGear$ where

$$\delta_{flap} = extFlaps; extFlaps; extFlaps$$
$$\delta_{dec} = \textbf{while } thrLevel(n) \land n \geq 100 \textbf{ do } decThrust \textbf{ endWhile}$$

Using the definitions of the remark 1, it can also be shown that $Perf(\tau'_{land}, S_1)$ and $\neg Perf(\tau'_{land}, S_2)$. Another important difference between our approach an that of [4] is that they regard procedures as programs which specify what can be done and what cannot be done; goals are not part of the procedure definition. In our approach, as shown in remark 1, actions that cannot be done amid the execution of a procedure can be deduced from *Perf*'s definition. This can be done because our notion of procedure considers the goal as part of its definition, therefore actions that cannot be executed are precisely those that will eventually make the procedure fail with respect to the goal.

7 Conclusions and Future Work

We have presented a logical foundation of the problem of mechanically determining what procedure is an agent executing given a complete set of observations. We see this problem as the first step toward the construction of intelligent tutoring systems for autonomous intelligent agents.

Procedures are regarded as composed both by a program and a goal. The program is described in the CONGOLOG programming language, a very expressive language which considers constructs for representing iterations and concurrency. Therefore, we use a language powerful enough to represent almost any describable procedure.

We have given an implementation for plan recognition based upon our logical formalization. Inspired in the idea of history encoding [2] we have given an equivalent formalization which yields a significantly more efficient solution with respect to time complexity.

As part of our future work we would like to incorporate, as in [5], temporal constraints into the definition of procedures. We think that in this case, the semantics of CONGOLOG should be extended. We think this is not difficult to do since the Situation Calculus can be augmented to represent time [13].

We also think it is interesting to incorporate, as in [4] the idea that a procedure could include a set of actions which should *not* be executed while being performed. We think this kind of restrictions may lead to more efficient implementations.

Acknowledgments

I am very grateful to Robert Demolombe for his valuable comments in earlier drafts of this paper.

References

[1] J. F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23:123–154, 1984.

[2] M. Arenas and L. Bertossi. Hypothetical temporal queries in databases. In *Proceedings of the 5th International Workshop on Knowledge Representation meets Databases (KRDB'98)*, 1998.

[3] J. Baier and J. Pinto. Integrating True Concurrency into the Robot Programming Language Golog. In *Proceedings of the 19th Chilean Computer Science Conference*, Talca, Chile, 1999. SCCC.

[4] R. Demolombe and E. Hamon. What does it mean that an agent is performing a typical procedure? a formal definition in situation calculus. In *Joint Conference on Autonomous Agents and Multiagent Systems*, Bologna, Italy, 2002. To appear.

[5] M. Ghallab. On chronicles: Representation, on-line recognition and learning. In Aiello, J. Doyle, and Shapiro, editors, *Proceeding of Principles of Knowledge Representation and Reasoning*, pages 597–606. Morgan-Kauffman, 1996.

[6] G. D. Giacomo, Y. Lespérance, and H. Levesque. ConGolog, a concurrent programming language based on the situation calculus: foundations. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[7] C. Heinze, S. Goss, I. Lloyd, and A. Pearce. Plan recognition in military simulation: Incorporating machine learning with intelligent agents. In *Proceedings of IJCAI-99 Workshop on Team Behaviour and Plan Recognition*, pages 53–64, 1999.

[8] H. A. Kautz. A formal theory of plan recognition and its implementation. In J. F. Allen, H. A. Kautz, R. N. Pelavin, and J. D. Tenenberg, editors, *Reasoning about Plans*, chapter 2, pages 69–126. Morgan Kaufmann Publishers, San Mateo, California, 1991.

[9] H. Levesque. What is planning in the presence of sensing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-96)*, pages 1139–1146, 1996.

[10] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl. GOLOG: A Logic Programming Language for Dynamic Domains. *The Journal of Logic Programming*, 31:59–84, 1997.

[11] D. Litman. *Plan Recognition and Discourse Analysis: An Integrated Approach for Understanding Dialogues*. PhD thesis, University of Rochester, Mar. 1986.

[12] J. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence* **4**, pages 463–502. Edinburgh University Press, Edinburgh, Scotland, 1969.

[13] J. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, Feb. 1994. URL = ftp://ftp.cs.toronto.edu/˜cogrobo/jpThesis.ps.Z.

[14] F. Pirri and R. Reiter. Some Contributions to the Metatheory of the Situation Calculus. *Journal of the ACM*, 46(2):261–325, 1999.

[15] A. S. Rao. Means-end plan recognition : Towards a theory of reactive recognition. In P. Torasso, J. Doyle, and E. Sandewall, editors, *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning*, pages 497–508, Bonn, FRG, 1994. Morgan Kaufmann.

[16] R. Reiter. *The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a completeness result for goal regression*, pages 359–380. Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy. Academic Press, San Diego, CA, 1991.

[17] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.

[18] R. Scherl and H. Levesque. The Frame Problem and Knowledge Producing Actions. In *Proceedings AAAI-93*, pages 689–695, Washington, D.C., July 1993. AAAI.