Integrating True Concurrency into the Robot Programming Language GOLOG

Jorge Baier and Javier Pinto
Depto. de Ciencia de la Computación,
Pontificia Universidad Católica de Chile,
Casilla 306, Santiago 22, Chile.
Email: {**jabaier, jpinto**}**@ing.puc.cl**

Abstract

Research in Knowledge Representation and Theories of Action has led to the development of several logical languages to describe the dynamics of the world. One of the most influential languages developed is the Situation Calculus. Stemming from this research, the situation calculus-based programming language GOLOG *has been proposed as a tool for implementing simulators and controllers of dynamical systems using a repertoire of user specified primitive actions. Lately, this language has been extended in order to incorporate the notion of* concurrent action execution*, leading to the dialect* CONGOLOG*, where an interleaving view of concurrent execution is considered. In this paper, we take this work one step further by introducing* true concurrency*, defining the language* TCONGOLOG*. In our view, true concurrency arises when primitive actions can be taken to be executed at the same instant.*

1. Motivation

Research in Knowledge Representation and Theories of Action has led to the development of several logical languages to describe the dynamics of the world. One of the most influential languages developed is the Situation Calculus. This language, was originally proposed by John McCarthy in order to model actions and change in discrete worlds where only one action can be performed at any given time [10]. In the last decade, a great deal of work has been done in order to extend the situation calculus with a much greater degree of expressiveness. In this article we do not discuss the situation calculus in detail. The reader is referred to [15], where an up to date review of the language and its properties is discussed. Also, [18] presents a detailed discussion of the applications of the situation calculus, for example to databases, programming language semantics and robot programming.

One important feature of the extended situation calculus, is the integration of concurrency [9, 11, 17, 1, 13]. However, much of this work considers an approach to concurrency in which an interleaving view of concurrent execution is taken. For example, if A_1 and A_2 are two primitive actions, and the term $A_1||A_2$ denotes the complex action where A_1 and A_2 are concurrent, then an interleaving view of concurrency takes the result of executing $A_1||A_2$ as equivalent to executing either A_1 and then A_2 or vice-versa. In [13, 14], a different view is taken, in which the importance of taking true concurrency is underlined and an approach to integrate it into the situation calculus is presented.

An offspring of theories of action research is the development of the field of *cognitive robotics*. In this area, several *cognitive robotic languages* have been proposed (see [19, 21]). Salient among these languages is GOLOG [7, 6], whose semantics is expressed as situation calculus theories. In fact, GOLOG is an interpreted language whose programs are translated into situation calculus logical formulas, which in turn are interpreted as logic programs executable by a Prolog interpreter. A shortcoming of the original GOLOG language is its inability to express concurrent execution of programs. This shortcoming has been addressed in [4, 5], where an interleaving view of concurrency is taken, leading the the dialect CONGOLOG. This language is limited by the underlying situation calculus semantics for concurrency as *interleaving*.

In this article we extend CONGOLOG and present a dialect TCONGOLOG, in which true concurrency is considered. This is possible given that we take a situation calculus with true concurrency to provide for the underlying semantics.

The article is structured as follows. In section 2, we present an introduction to the GOLOG language. In section 3 a brief introduction to the concurrent situation calculus is presented. In section 4 we show how to extend CONGOLOG with true concurrency, and present examples where true concurrency is necessary. Finally, in section 5 we conclude and discuss further research.

2. Introduction to GOLOG

GOLOG is a logic-based interpreted language designed to model dynamic worlds at a high abstraction level. This language can be used in high-level robot control, intelligent agent programming, etc. The semantics of the language is based on the situation calculus, a language of second order logic, which we describe in section 3. The result of the execution of a GOLOG program is a sequence of primitive actions, i.e. a sequence of actions that a robot or agent may execute.

In a GOLOG program we distinguish the following elements:

Primitive actions and Wait Conditions

- Primitive actions, denoted by α (possible with subscripts). They are single (non-parallel) atomic actions. In a single step of execution of the program, the interpreter executes one of these actions.
- ϕ?: test(wait) conditions. Here ϕ is a first order formula. These formulas play the same role in GOLOG as boolean expressions in traditional imperative languages such as C or Pascal. The main difference is that their expressiveness is higher since they can include quantifications over objects.

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

- 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 *sequences 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*. Here 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.
  - $\pi x.\sigma$, is a *non-deterministic choice of arguments*. The variable x is 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, is a *conditional sentence*. σ_1 is executed if ϕ holds, in other case, σ_2 is executed.
  - **while** ϕ **do** σ: *while loops*. σ is executed while ϕ holds.

Procedure definitions A procedure in GOLOG has the following form:

$$\textbf{proc}\ \beta(\vec{x})\,\sigma\ \textbf{end},$$

where β is the name of the procedure, $\vec{x}$ are the arguments and σ is the body.

Definition 1 *A* GOLOG *program is a set of zero or more procedure definitions followed by a complex action, which is the main program. A* GOLOG *program looks like:*

$$\textbf{proc}\ P_1(\vec{v}_1)\ \sigma_1\ \textbf{end};\ldots\textbf{proc}\ P_n(\vec{v}_n)\sigma_n\ \textbf{end};\ \sigma$$

Example 1 We can define a procedure *clean*, which puts all the blocks in a given domain of discourse outside the *Box*.

proc *clean*
  while $\neg(\forall x)[block(x) \supset in(x,Box)]?$ **do** $(\pi x)\ put(x,Box)$
 end;
clean

In this program, the procedure *clean* executes the action $put(x,Box)$, for an arbitrary x. This is repeated while there are blocks inside the *Box* . The main program calls the *clean* procedure. 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* [20]; 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.

This program can be directly translated into a situation calculus theory[1], which, in turn, can be translated into a PROLOG program.

[1] in section 4 we explain how this is done

3. The Situation Calculus and Concurrency

3.1. Formal Foundations

Our work is based on the concurrent Situation Calculus [12, 17], which extends the Situation Calculus with a sort for concurrent actions. Concurrent actions are treated as sets of atomic actions.

The concurrent Situation Calculus is a second order language with sorts $\mathcal{A}$, $\mathcal{C}$, $\mathcal{S}$ and $\mathcal{D}$ for atomic actions, concurrent actions, situations and domain objects. The sort of the variables and constants used in the examples should be obvious from the context. There is a distinguished initial situation denoted with the constant S_0. The function *do* takes a concurrent action and a situation and yields another situation. Fluents are predicates that take one argument of type situation, and represent properties that are static within a situation, but that may change between situations. Also, we have the predicate *Poss*, which takes a concurrent action and a situation, and should hold when the concurrent action is executable in the situation.

The foundational axioms for the concurrent Situation Calculus define the structure of situations. The definition states that there is an initial situation, called S_0, and that any situation that exists is obtained by the application of the *do* function. Thus, if $c_1 \dots c_n$ are concurrent actions, then $do(c_1, S_0)$, $do(c_1, do(c_1, S_0))$, $do(c_1, do(c_3, do(c_4, S_0)))$ are situations. Also, the foundational axioms state that no situation exists, unless it can be reached by the execution of finitely many action starting in S_0 (for this, second order induction is necessary). Finally, the foundational axioms also establish a partial order between situations. Thus, if s_1 and s_2 are situations, then $s_1 < s_2$ would be true if s_1 is a situation along the path to s_2 starting from S_0.

Concurrent actions are treated as sets of primitive actions. Thus, $\in$ is used as a relation between atomic actions and concurrent actions. We write $a \in c$ to mean that action a is part of the concurrent action c.

In the following section we describe how to write theories of action based on the language of the concurrent Situation Calculus. Our approach is based upon Reiter's monotonic solution to the frame problem.

3.2. Domain Axiomatization

A domain axiomatization is divided in several sets of axioms. First, we have a set (T_d) of axioms that only mention terms of sort $\mathcal{D}$. Also, we have a suitable set of unique names axioms (T_{una}, which we omit) for terms of sorts $\mathcal{A}$ and $\mathcal{D}$. The other sets of axioms are described below.

Action Precondition Axioms The set T_{prec} of action precondition axioms for atomic actions, each of the form[2]:

$$Poss(\{a_l(\underline{\mathbf{x}})\}, s) \supset \Phi_{a_l}(\underline{\mathbf{x}}, s),$$

where $a_l(\underline{\mathbf{x}})$ is an $\mathcal{A}$ term, and $\Phi_{a_l}(\underline{\mathbf{x}}, s)$ is a formula simple on s[3]. In general, preconditions for actions are formulated as necessary conditions for *Poss*.

Example: Let us consider the following extension to the soup bowl example from [3]: There is a bowl filled with soup, the bowl can be lifted from either of two sides (left and right). If the bowl is lifted from one side only, then the soup is spilled. If the bowl is lifted from both sides simultaneously, then it is not spilled, unless it is shaken at the same time. When the bowl is shaken, the soup always spills. To model this example, we have the action constants: $Lift_l$, $Lift_r$, and *Shake*. There are three fluents $lifted_l$, $lifted_r$, and *spilled*.

The action precondition axioms for the actions in this domain are:

$$\begin{aligned}
&Poss(\{Lift_l\}, s) \supset \neg lifted_l(s),\\
&Poss(\{Lift_r\}, s) \supset \neg lifted_r(s),\\
&Poss(\{Shake\}, s) \supset True.
\end{aligned}$$

These axioms embody the assumption that the preconditions for performing an action are completely known. In the presence of no qualification state constraints (see below), one normally assumes that all necessary conditions are also sufficient. Thus, in the previous example, the implications are converted into equivalences.

Direct Effect Axioms The set T_{eff} of effect axioms corresponding to the *causal rules* of the domain. These axioms are restricted to specify only the effects that are a direct consequence of the action, regardless of what other actions, if any, are performed concurrently.

Since *Poss* only takes concurrent actions as first parameter, these axioms have a slightly different syntactic form from Reiter's effect axioms [16]. A positive direct effect axiom is of the syntactic form (1), while a negative direct effect axiom is of the form (2).

$$Poss(c, s) \wedge A \in c \wedge G_f^+(\underline{\mathbf{x}}, A, s) \supset f(\underline{\mathbf{x}}, do(c, s)), \quad (1)$$

$$Poss(c, s) \wedge A \in c \wedge G_f^-(\underline{\mathbf{x}}, A, s) \supset \neg f(\underline{\mathbf{x}}, do(c, s)). \quad (2)$$

Now, all the effect axioms can be compiled together in one positive and one negative general direct effect axiom of the forms:

$$\begin{aligned}
Poss(c, s) \wedge \gamma_f^+(\underline{\mathbf{x}}, c, s) &\supset f(\underline{\mathbf{x}}, do(c, s)),\\
Poss(c, s) \wedge \gamma_f^-(\underline{\mathbf{x}}, c, s) &\supset \neg f(\underline{\mathbf{x}}, do(c, s)).
\end{aligned}$$

[2] We write $\underline{\mathbf{x}}$ to denote a tuple of domain variables (i.e., variables of sort $\mathcal{D}$).

[3] A formula is simple on a situation term s_t, if it does not mention any situation terms, other than s_t, and does not quantify over s_t.

If there are no positive (or negative) effect axioms for a fluent f, then the formula $\gamma_f^+(\underline{\mathbf{x}},c,s)$ (or $\gamma_f^-(\underline{\mathbf{x}},c,s)$ for the negative case) is the atom *False*.

Example (cont.): For the example, we have:

$$\begin{aligned}
&Poss(c,s) \wedge Lift_l \in c \supset lifted_l(do(c,s)),\\
&Poss(c,s) \wedge Lift_r \in c \supset lifted_r(do(c,s)),\\
&Poss(c,s) \wedge Shake \in c \supset spilled(do(c,s)).
\end{aligned}$$

Ramification State Constraints A set T_{rc} of ramification state constraints [8]. In general, ramification constraints are of the form:

$$\Psi(\underline{\mathbf{x}},s),$$

where $\Psi(\underline{\mathbf{x}},s)$ is a simple formula. We assume that constraints are written as clauses. Also, it might be convenient to write these clauses as implications.

Example (cont.): For this domain we have:

$$\begin{aligned}
&(\forall s).\ lifted_l(s) \vee \neg lifted_r(s) \vee spilled(s)\\
&(\forall s).\ \neg lifted_l(s) \vee lifted_r(s) \vee spilled(s).
\end{aligned}$$

One reading of these constraints is that if one side is lifted, and the other is not lifted, then *spilled* holds. Combining both constraints one can derive that if *spilled* is not true, then both sides are lifted, or no side is lifted.

Qualification State Constraints A set T_{qc} of *qualification state constraints* [8]. They have the same syntactic form as the ramification constraints, that is $\Psi(\underline{\mathbf{x}},s)$. Where $\Psi(\underline{\mathbf{x}},s)$ is a simple formula.

Example (cont.): For the example, we might assume that the fluent *inPlate* is true in a situation if the soup is not spilled. A typical example of a qualification constraint is:

$$at(d_1,l,s) \wedge at(d_2,l,s) \supset d_1 = d_2,$$

This constraint states that if an object d_1 is in location l and object d_2 is also in l, then d_1 and d_2 must be the same object. The fact that this axiom is listed as a *qualification state constraint* says that if an action were, by its direct effects, to violate the constraint, then the action would not be possible. If the constraint were of a ramification type, then the action would be considered possible, but we would need new effects (aside from the direct ones) in order to ensure that the constraint is satisfied in the resulting situation.

The Initial Situation A set of axioms, T_{S_0}, characterizing the initial situation S_0.

Example (cont.): The initial situation for the example can be characterized with:

$$\neg lifted_l(S_0) \wedge \neg lifted_r(S_0) \wedge \neg spilled(S_0).$$

3.3. A Theory of Action and Change

A theory of action in Reiter's style, is derived from the domain axiomatization. Such a theory includes all the foundational axioms of the Situation Calculus, a set of complete precondition axioms, derived from the set T_{qc} of qualification constraints and the Action Precondition Axioms. These axioms are equivalences that, for each action (concurrent or primitive), it tells us whether the action is possible. A set of *successor state axioms* derived from the direct effect axioms and the ramification constraints. These axioms have the form:

$$\begin{aligned}
&Poss(c,s) \supset [f(\underline{\mathbf{x}},do(c,s)) \equiv\\
&\qquad (\gamma_f^+(\underline{\mathbf{x}},c,s) \vee f(\underline{\mathbf{x}},s) \wedge \neg\gamma_f^-(\underline{\mathbf{x}},c,s))].
\end{aligned}$$

These axioms establish necessary and sufficient conditions for change of fluent values between successive situations. These are the central axioms in a theory of action and change. From these axioms it is possible to define the *regression operator* $\mathcal{R}$ for formulas in the situation calculus in the following form. Given a formula $\varphi(s)$ simple in situation variable s. The regression of $\varphi(do(c,s))$ can be obtained by *unwinding* the $do(c,s)$ term by means of the successor state axioms. In that way, the formula $\mathcal{R}(\varphi(do(c,s)))$ is a formula that is simple in s and that is true if and only if φ is true in $do(c,s)$. Thus, the truth of formulas can be regressed from situation $do(c,s)$ to situation s. This operator is essential for deriving the successor state axioms for concurrent actions [14].

4. Concurrent GOLOG: TCONGOLOG

We now extend the GOLOG language with true concurrency. Our work relies heavily on the language CONGOLOG proposed in [4]. Concurrency is achieved by adding the construct $(\sigma_1||\sigma_2)$ to the programming language meaning that complex action σ_1 and σ_2 execute concurrently. The notion of concurrent execution is the standard one used in computer science: it means that nothing can be said about the ordering of execution between any two primitive actions from σ_1 and σ_2. Since our aim is to add true concurrency, primitive actions from both σ_1 and σ_2 can occur in parallel in a single step of execution. Primitive non-parallel actions, denoted by α in the original GOLOG are now denoted by the letter c.

As shown in [4] it is convenient for handling concurrency to give GOLOG a *computational semantics*, which is based on single steps of computation or *transitions*. In the original GOLOG a transition corresponds to the execution of a single primitive action or a test condition. In our extension, transitions correspond to the execution of simple concurrent (parallel) actions.

In order to define the language semantics, we to introduce the predicates *Final* and *Trans* as in [4], where $Final(\sigma,s)$ means that the program σ may terminate in situation s. $Trans(\sigma,s,\sigma',s')$ is true when program σ may execute one step in situation s, ending in situation s' with program σ' remaining.

Our definition of *Trans* and *Final* differs from that of [4] in the fact that our transitions may be either legal or illegal. We mean by legal transition one that results from the execution of a possible action. The correct execution of a program will be defined as a series of transitions composed only by legal transitions.

Final and *Trans* are defined by two sets of equivalence axioms, one axiom for each different type of action of the language. The following are the equivalence axioms Γ_F for *Final*.

$$\begin{aligned}
&Final(\{\},s) \equiv True\\
&Final(\alpha,s) \equiv False\\
&Final(\phi?,s) \equiv False
\end{aligned}$$

This first three axioms state that only an empty program can terminate in any situation. For complex actions, *Final* is extended as follows:

$$\begin{aligned}
&Final((\sigma_1;\sigma_2),s) \equiv Final(\sigma_1,s) \wedge Final(\sigma_2,s)\\
&Final((\sigma_1|\sigma_2),s) \equiv Final(\sigma_1,s) \vee Final(\sigma_2,s)\\
&Final((\sigma_1||\sigma_2),s) \equiv Final(\sigma_1,s) \wedge Final(\sigma_2,s)\\
&Final(\pi x.\sigma,s) \equiv \exists x.Final(\sigma,s)\\
&Final(\mathbf{if}\ \phi\ \ \mathbf{then}\ \sigma_1\ \ \mathbf{else}\ \sigma_2,s) \equiv\\
&\qquad \phi(s) \wedge Final(\sigma_1,s) \vee \neg\phi(s) \wedge Final(\sigma_2,s)\\
&Final(\sigma^*,s) \equiv TRUE\\
&Final(\mathbf{while}\ \phi\ \ \mathbf{do}\ \sigma,s) \equiv \phi(s) \wedge Final(\sigma,s) \vee \neg\phi(s)
\end{aligned}$$

The set Γ_T for the *Trans* predicate is the following:

$$\begin{aligned}
&Trans(\{\},s,\delta,s') \equiv False\\
&Trans(c,s,\delta,s') \equiv \delta = \{\} \wedge s' = do(c,s)\\
&Trans(\phi?,s,\delta,s') \equiv \phi(s) \wedge \delta = \{\} \wedge s' = s
\end{aligned}$$

The first axiom says that an empty program cannot have a transition because it has finished. The second says that the transition of a simple action c leads from s to $do(c,s)$. The third says that a transition of a test condition does not change the situation in which the program is executing and that it can be true if and only if the condition holds.

$$\begin{aligned}
&Trans((\sigma_1;\sigma_2),s,\delta,s') \equiv Final(\sigma_1,s) \wedge Trans(\sigma_2,s,\delta,s') \vee\\
&\qquad\qquad \exists\delta'.\delta = (\delta';\sigma_2) \wedge Trans(\sigma_1,s,\delta',s')\\
&Trans((\sigma_1|\sigma_2),s,\delta,s') \equiv\\
&\qquad\qquad Trans(\sigma_1,s,\delta',s') \vee Trans(\sigma_2,s,\delta',s')\\
&Trans((\sigma_1||\sigma_2),s,\delta,s') \equiv\\
&\quad \exists\delta_1,\delta_2,c_1,c_2(Trans(\sigma_1,s,\delta_1,do(c_1,s)) \wedge\\
&\qquad\qquad Trans(\sigma_2,s,\delta_2,do(c_2,s)) \wedge\\
&\qquad\qquad s' = do(c_1 \cup c_2,s) \wedge \delta = (\delta_1||\delta_2)) \vee\\
&\quad \exists\delta'.\delta = (\delta'||\sigma_2) \wedge Trans(\sigma_1,s,\delta',s') \vee\\
&\qquad \delta = (\sigma_1||\delta') \wedge Trans(\sigma_2,s,\delta',s').
\end{aligned}$$

In other words, a transition of $(\sigma_1;\sigma_2)$ corresponds to a transition of σ_2 if σ_1 has terminated, in other case it corresponds to a transition of σ_1. A transition of $(\sigma_1|\sigma_2)$ correspond to either a transition of σ_1 or a transition of σ_2. A transition of $(\sigma_1||\sigma_2)$ correspond to either a transition of σ_1, a transition of σ_2 or the execution of a concurrent action formed by a simple action from σ_1 and a simple action from σ_2. In other words, a transition of $(\sigma_1||\sigma_2)$ is a combination of interleaving and parallelism.

$$\begin{aligned}
&Trans(\pi x.\sigma,s,\delta,s') \equiv \exists x.Trans(\sigma,s,\delta,s')\\
&Trans(\mathbf{if}\ \phi\ \ \mathbf{then}\ \sigma_1\ \ \mathbf{else}\ \sigma_2,s) \equiv\\
&\quad \phi(s) \wedge Trans(\sigma_1,s,\delta,s') \vee \neg\phi(s) \wedge Trans(\sigma_2,s,\delta,s')\\
&Trans(\sigma^*,s,\delta,s') \equiv \exists\delta'.\delta = (\delta';\sigma^*) \wedge Trans(\sigma,s,\delta',s')\\
&Trans(\mathbf{while}\ \phi\ \ \mathbf{do}\ \sigma,s,\delta,s') \equiv\\
&\quad \phi(s) \wedge \exists\delta'.\delta = (\delta';\mathbf{while}\ \phi\ \ \mathbf{do}\ \sigma) \wedge Trans(\sigma,s,\delta',s').
\end{aligned}$$

The first axiom says that one transition of $\pi x.\sigma$ is a transition of σ^v_x, where σ^v_x is the result of substituting variable x with some constant v. The second says that a transition of a conditional is a transition of σ_1 or σ_2 depending on whether φ holds. The third establishes that the transition of a non-deterministic iteration σ^* is a transition of σ an that the remaining program is $(\delta';\sigma^*)$ where δ' is the remaining program of one transition of σ. The final axiom says that the transition of a while loop is a transition of its body provided φ holds.

We now define the predicate $Trans^*$ which is the transitive closure of *Trans* plus the condition of legality of transitions, given by $S_0 < s'$.

$$\begin{aligned}
&Trans^*(\sigma,s,\sigma',s') \stackrel{def}{=} S_0 < s' \wedge \\
&\forall T[\{\forall s.T(\sigma,s,\sigma,s) \wedge \\
&\forall s,\delta',s',\delta'',s''.T(\sigma,s,\delta',s') \wedge \\
&\quad Trans(\delta',s',\delta'',s'') \supset T(\sigma,s,\delta'',s'')\} \supset T(\sigma,s,\sigma',s')]
\end{aligned} \tag{3}$$

In other words $Trans^*(\sigma,s,\sigma',s')$ holds when the execution of any number of *legal* transitions of program σ in situation s leads to situation s' with program σ' remaining.

Example 2 In this example we have two agents whose goal is to get as many balls as they can out of a box and keep them in private storage. To get a ball out of the box, the agent must first grab it and then take it away.

Actions:

- $grab(ag,x)$: Agent ag grabs the ball x.
- $release(ag,x)$: Agent ag releases ball x.
- $store(ag,x)$ means the agent ag stores the ball x and keeps it.

Fluents:

- $inbox(x,s)$: ball x is inside the box in situation s.
- $grabbed(ag,x,s)$: agent ag has the ball x in one of its hands in situation s.
- $has(ag,x,s)$: agent ag is holding the ball x in situation s.

Direct Effect Axioms The set of **direct effect axioms** contains:

$$Poss(c,s) \wedge grab(ag,x) \in c \supset grabbed(ag,x,do(c,s)). \tag{4}$$

$$\begin{aligned}
&Poss(c,s) \wedge [release(ag,x) \in c \vee store(ag,x) \in c] \supset \\
&\quad \neg grabbed(ag,x,do(c,s))
\end{aligned} \tag{5}$$

$$Poss(c,s) \wedge store(ag,x) \in c \supset \neg inbox(x,do(a,s)). \tag{6}$$

$$Poss(c,s) \wedge store(ag,x) \in c \supset has(ag,x,do(a,s)). \tag{7}$$

Precondition Axioms The set of **precondition axioms**, T_{prec}, is defined by the following axioms:

To execute $store(ag,x)$ the ball x must be grabbed only by agent ag:

$$\begin{aligned}
&Poss(c,s) \supset [store(ag,x) \in c \supset \\
&\quad (grabbed(ag,x,s) \wedge \\
&\quad ((\forall ag').\, grabbed(ag,x,s) \supset ag = ag'))].
\end{aligned} \tag{8}$$

Agent ag can grab ball x if it has its hand free and the ball is inside the box.

$$\begin{aligned}
&Poss(c,s) \supset [grab(ag,x) \in c \supset \\
&\quad (inbox(x,s) \wedge \neg grabbed(ag,x,s))].
\end{aligned} \tag{9}$$

Agent ag can release the ball x if it has it grabbed:

$$Poss(c,s) \supset [release(ag,x) \in c \supset grabbed(ag,x,s)]. \tag{10}$$

Qualification Constraints In any legal situation it must be true that a ball is in a unique position, i.e., a ball is either inside the box or one agent has it. Thus, we add the following **qualification constraints**:

$$(\forall s,x)\, inbox(x,s) \equiv \neg \exists ag.\, has(ag,x,s)$$
$$(\forall ag_1,ag_2,s,x)\, has(ag_1,x,s) \wedge has(ag_2,x,s) \supset ag_1 = ag_2.$$

Successor State Axioms Using [16] we generate the following successor state axioms for each action. The following is the set T_{succ} of successor state axioms for *inbox*, *has* and *grabbed*.

$$\begin{aligned}
Poss(c,s) \supset &[inbox(x,do(c,s)) \equiv \\
&inbox(x,s) \wedge \neg \exists ag.\, store(ag,x) \in c],
\end{aligned} \tag{11}$$

$$\begin{aligned}
Poss(c,s) \supset &[has(ag,x,do(c,s)) \equiv \\
&has(ag,x,s) \vee store(ag,x) \in c],
\end{aligned} \tag{12}$$

$$\begin{aligned}
Poss(c,s) \supset &[grabbed(ag,x,do(c,s)) \equiv \\
&grabbed(ag,x,s) \wedge \neg(store(ag,x,s) \in c \vee \\
&release(ag,x,s) \in c) \vee \\
&grab(ag,x,s) \in c].
\end{aligned} \tag{13}$$

From the qualification constraint and the successor state axioms we generate the following precondition axioms, using [13, 14]. These are also added to T_{prec}. The idea is to leave out of the extension of *Poss* all the concurrent actions which would invalidate the qualification constraints.

$$\begin{aligned}
Poss(c,s) \supset &\, inbox(x,s) \wedge \\
&\neg \exists ag.\, store(ag,x) \in c \equiv \\
&\quad \neg \exists ag.\, has(ag,x,s) \vee store(ag,x) \in c,
\end{aligned} \tag{14}$$

$$\begin{aligned}
Poss(c,s) \supset &\forall ag_1,ag_2.\, (has(ag_1,x,s) \vee \\
&\quad store(ag_1,x) \in c) \wedge \\
&(has(ag_2,x,s) \vee \\
&\quad store(ag_2,x) \in c) \supset ag_1 = ag_2.
\end{aligned} \tag{15}$$

The set T_{prec} is reconstructed by applying Clark's completion [2] to all the precondition axioms previously defined. The completion replaces all the axioms already defined by a unique equivalence axiom for $Poss(c,s)$.

In the initial situation, there are four balls in the box. The set of axioms T_{S_0} is composed by only one axiom:

$$inbox(x,S_0) \equiv x = B_1 \vee x = B_2 \vee x = B_3 \vee x = B_4 \quad (16)$$

Finally, we add unique name axioms for the objects in the language into the set T_{una}:

$$\begin{aligned} Ag_1 \neq Ag_2 \wedge B_1 \neq B_2 \wedge B_1 \neq B_3 \wedge B_1 \neq B_4 \wedge \\ B_2 \neq B_3 \wedge B_2 \neq B_4 \wedge B_3 \neq B_4 \end{aligned} \quad (17)$$

The agents are programmed to take all the objects away from the box. The following is the GOLOG procedure for the agents.

$$\begin{aligned} & remove(ag) \stackrel{def}{=} \textbf{while } \exists x\, inbox(x) \textbf{ do} \\ & \quad \pi x.(grab(ag,x); \\ & \quad\quad \textbf{if } \neg\exists ag'\, grabbed(ag',x) \wedge ag \neq ag' \textbf{ then} \\ & \quad\quad\quad store(ag,x) \textbf{ else } release(ag,x)) \end{aligned} \quad (18)$$

Proposition 1 *Let* $\sigma \stackrel{def}{=} (remove(Ag_1)||remove(Ag_2))$. *Let* Σ *be a theory of action including the foundational axioms of the situation calculus along with the sets* T_{prec}, T_{ssa}, T_{una}, T_{S_0}, *the* TCONGOLOG *semantics definition axioms* Γ_F *and* Γ_T. *Let* S_1 *and* S_2 *be defined as follows:*

$$\begin{aligned} S_1 \stackrel{def}{=} do([&\{grab(ag_1,B_1),grab(ag_2,B_2)\}, \\ &\{store(ag_1,B_1),store(ag_2,B_2)\}, \\ &\{grab(ag_1,B_3),grab(ag_2,B_4)\}, \\ &\{store(ag_1,B_3),store(ag_2,B_4)\}],S_0) \\ S_2 \stackrel{def}{=} do([&\{grab(ag_1,B_1),grab(ag_2,B_1)\}, \\ &\{release(ag_2,B_1)\},\{grab(ag_2,B_2)\}, \\ &\{store(ag_1,B_1),store(ag_2,B_2)\}, \\ &\{grab(ag_1,B_3),grab(ag_2,B_4)\}, \\ &\{store(ag_1,B_3),store(ag_2,B_4)\}],S_0). \end{aligned}$$

Then S_1 *and* S_2 *are two possible executions of the program, i.e.,*

$$\begin{aligned} \Sigma \models \exists\sigma' Trans^*(\sigma,S_0,\sigma',S_1) \wedge Final(\sigma',S_1) \\ \Sigma \models \exists\sigma' Trans^*(\sigma,S_0,\sigma',S_2) \wedge Final(\sigma',S_2) \end{aligned}$$

PROOF: *Straightforward by applying the axioms of Trans and Final.*

It is interesting to notice that the first execution is a series of parallel actions from both concurrent programs, whereas the second is a mix of interleaving and parallelism.

Proposition 2 *Let* $Deadlock(\sigma)$ *be true when some execution of the program* σ *leads to a deadlock, i.e.*

$$\begin{aligned} & Deadlock(\sigma) \stackrel{def}{=} \\ & \quad \exists s',\delta.\, Trans^*(\sigma,S_0,\delta,s') \wedge \neg Final(\delta,s') \wedge \\ & \quad \neg\exists s'',\delta'.\,(Trans(\delta,s',\delta',s'') \wedge S_0 < s''). \end{aligned} \quad (19)$$

Then the program σ *defined in proposition 1 does never get into a deadlock situation.*

$$\neg Deadlock(\sigma)$$

PROOF: *The program considers three atomic actions grab, store and release. The idea of the proof is to prove that both actions are allways possible in the steps of the program where they are to be executed.*

5. Conclusions

In this paper we have integrated true concurrency in the robot programming language GOLOG. The incorporation of true concurrency has a special relevance when modeling multi-agent domains. True concurrency is essential in agent cooperation: usually the only way two agents can cooperate is when they execute actions in parallel. For example, think about two robots trying to lift a table with an object over it. If both robots do not lift each side at the same time the object would fall down. Concurrency also appears in other kinds of interactions such as competition. In all these situations it is essential for an agent to have the ability to reason about the interaction between actions execute in parallel; otherwise, flawed reasoning would ensue.

The language can be used in the programming and simulation of multi-agent domains. Part of our current and future research is to see how the language can be used to model more complex domains where agent interaction is important. For instance, in programming robots that interact with other agents. Currently, we are working on the programming aspects of robots in traffic environments.

TCONGOLOG has been implemented in a fairly straightforward manner. This is done by directly translating situation calculus theories into Prolog programs.

The current version of TCONGOLOG is atemporal, that is to say, it does not explicitly incorporate time in the actions. Since time is clearly very important in modeling multi-agent domains, part of our future work is the integration of time in the language. We also plan to see if it is possible to mechanically generate programs for agents given the description of their abilities in order to perform a common goal.

References

[1] J. Baier and J. Pinto. Non-instantaneous Actions and Concurrency in the Situation Calculus (Extended Abstract). In G. de Giacomo and D. Nardi, editors, *10th European Summer School in Logic, Language and Information*, 1998.

[2] K. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, 1978.

[3] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the Limitations of the Situation Calculus? In S. Boyer, editor, *Automated Reasoning, Essays in Honor of Woody Bledsoe*, pages 167–181. Kluwer Academic Publishers, 1991.

[4] G. D. Giacomo, Y. Lespérance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus: foundations. Submitted for publication. Available from http://www.cs.toronto.edu/cogrobo/ConGologFound.ps.gz, February 1999.

[5] G. D. Giacomo, Y. Lespérance, and H. Levesque. Congolog, a concurrent programming language based on the situation calculus: language and implementation. Submitted for publication. Available from http://www.cs.toronto.edu/cogrobo/ConGologLang.ps.gz, February 1999.

[6] H. Levesque, F. Pirri, and R. Reiter. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(18), 1998. http://www.ep.liu.se/ea/cis/1998/018/.

[7] 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.

[8] F. Lin and R. Reiter. State Constraints Revisited. *Journal of Logic and Computation*, 4(5):655–678, 1994. Special Issue on Actions and Processes.

[9] F. Lin and Y. Shoham. Concurrent Actions in the Situation Calculus. In *Working Notes of the 4th International Workshop on Nonmonotonic Reasoning*, pages 133–138, 1992.

[10] J. McCarthy. Situation, actions, and causal laws. Technical Report Memo 2, Stanford Artificial Intelligence Project, 1963.

[11] J. Pinto. Concurrent Events: Synergy and Cancellation of Effects. In *European Conference on Artificial Intelligence, Workshop on Logic and Change*, pages 105–110, 1994. URL = ftp://lyrcc.ing.puc.cl/pub/jpinto/ecai.ps.gz.

[12] 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.

[13] J. Pinto. Concurrent actions and interacting effects. In A. Cohn, L. Schubert, and S. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*, pages 292–303, San Francisco, CA, 1998. Morgan Kaufmann Publishers.

[14] J. Pinto. Concurrency and Action Interaction II: The Precondition Interaction Problem. Forthcoming, 1999.

[15] F. Pirri and R. Reiter. Some Contributions to the Metatheory of the Situation Calculus. *Journal of the ACM*, 1999. To appear.

[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. Natural Actions, Concurrency and Continuous Time in the Situation Calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference(KR'96)*, Cambridge, Massachussetts, U.S.A., Nov. 1996. Morgan Kaufmann.

[18] R. Reiter. KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems. Book Draft, available from `http://www.cs.utoronto.ca/~cogrobo`, 1999.

[19] E. Sandewall. Cognitive robotics logic and its metatheory: Features and fluents revisited. *Electronic Transactions on Artificial Intelligence*, (17), 1998. http://www.ep.liu.se/ea/cis/1998/017/.

[20] 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.

[21] M. Thielscher. Introduction to the fluent calculus. *Linköping Electronic Articles in Computer and Information Science*, 3(14), 1998. http://www.ep.liu.se/ea/cis/1998/014/.