A First-Order Semantics for Golog and ConGolog under a Second-Order Induction Axiom on Situations

Fangzhen Lin
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

Abstract

Golog and ConGolog are languages defined in the situation calculus for cognitive robotics. Given a Golog program $\delta$, its semantics is defined by a macro $Do(\delta, s, s')$ that expands to a logical sentence that captures the conditions under which performing $\delta$ in $s$ can terminate in $s'$. A similar macro is defined for ConGolog programs. In general, the logical sentences that these macros expand to are second-order, and in the case of ConGolog, may involve quantification over programs. In this paper, we show that by making use of the foundational axioms in the situation calculus, in particular, the second-order closure axiom about the space of situations, these macro expressions can actually be defined using first-order sentences.

Introduction

Golog (Levesque et al. 1997) and ConGolog (De Giacomo, Lespérance, and Levesque 2000) are Algol-like programming languages defined in the situation calculus (McCarthy 1968; Reiter 2001). They are intended for high-level control of agents, including both physical robots and virtual agents (Reiter 2001; Burgard et al. 1999; Funge 2000; Lespérance, Levesque, and Ruman 1997; McIlraith and Son 2002). They include non-determinism, and in the case of ConGolog, concurrency.

The semantics of Golog programs is axiomatized in logic by macro expansions (Levesque et al. 1997): given a Golog program $\delta$ and two situation terms $s$ and $s'$, the macro expression $Do(\delta, s, s')$ is used to denote that $s'$ is a terminating situation of performing $\delta$ in $s$. There may be more than one possible terminating situation as the program may be indeterminate. In general, the macro expression $Do(\delta, s, s')$ expands into a second-order sentence because $\delta$ may have loops. In the case of ConGolog programs, which are extensions of Golog programs with some concurrency operators, De Giacomo, Lespérance and Levesque (2000) introduced a one-step transition predicate $Trans(\delta, s, s', s'')$ (performing $\delta$ in $s$ for one step may end up in situation $s''$ with $\delta'$ as the remaining program to be performed) and a final predicate $Final(\delta, s)$ (program $\delta$ terminates in $s$). The Do macros are then defined using these predicates, and in general expand to sentences that are not only second-order, but also involve quantification over programs.

The second-order formulas that are needed in Golog and ConGolog macro expansions are fundamentally the same as the ones for defining transitive closures. However, the foundational axioms of the situation calculus already have a second-order axiom capturing the space of situations as the transitive closure of $Do(\alpha, s)$ starting at the initial situation $S_0$. In this paper, we show that this second-order foundational axiom can be used to capture loops and recursions so that for Golog and ConGolog programs, the macro expression $Do(\delta, s, s')$ can in fact be defined using first-order sentences.

The situation calculus

We first briefly review the situation calculus used for Golog and ConGolog. For more details, see (Reiter 2001; Lin 2007). The language of the situation calculus is a many-sorted one with sorts $\mathit{action}$ for actions, $\mathit{situation}$ for situations, and other domain dependent ones. There is a constant $S_0$ of sort $\mathit{situation}$, and a function $do(\alpha, s) : \mathit{action} \times \mathit{situation} \rightarrow \mathit{situation}$.

The foundational axioms say that the set of situations consists exactly of those that can be constructed from $S_0$ using function $do(\alpha, s)$ (we assume that variables in displayed formulas are universally quantified from outside):

$$S_0 \neq do(\alpha, s),$$

$$do(\alpha_1, s_1) = do(\alpha_2, s_2) \supset a_1 = a_2 \land s_1 = s_2,$$

$$\forall P. [P(S_0) \land \forall a, s(P(s) \supset P(do(\alpha, s)))] \supset \forall s P(s).$$

The first two axioms are unique names axioms, and the third one a second-order induction axiom. Given a sequence $[a_1, ..., a_n]$ of actions, we use $do([a_1, ..., a_n], s)$ to denote the situation resulting from doing the sequence of actions in $s$: $do(a_n, ..., do(a_1, s))$. Using this notation, we can understand the foundational axioms as saying that $s$ is a situation iff there is a sequence $\alpha$ of actions such that $s = do(\alpha, S_0)$.

With these axioms, one can define other relations inductively on situations (Reiter 1993; Lin and Reiter 1994). The following axioms define a partial order $\leq$ on situations:

$$\neg s < S_0,$$

$$s < do(\alpha, s') \equiv s = s' \lor s < s',$$

$$s \leq s' \equiv s < s' \lor s = s'.$$
Again in terms of sequences of actions, we can see that $s \leq s'$ iff there is a sequence $o$ of actions such that $s' = do(o, s)$.

In the following, we denote by $\Sigma$ the set of foundational axioms as well as the above axioms for $\prec$.

### Golog

Golog programs are complex actions and procedures. Complex actions are defined inductively as follows:

\[
\delta ::= A \mid \varphi ? \mid \delta ; \delta \mid (\delta | \delta) \mid (\pi x)\delta \mid \delta^*
\]

where $A$ is a primitive action, $\varphi$ is a pseudo-situation calculus formula with all situation arguments suppressed. In (Levesque et al. 1997), the semantics of these complex actions are defined using a macro $\text{Do}(\delta, s, s')$ that expands to a situation calculus formula, and means that $s'$ is a terminating situation of performing $\delta$ in $s$. As we mentioned, this formula is in general second-order. Using the foundational axioms in the situation calculus, we can define it by first-order sentences. To do this, we introduce a new predicate whenever an iteration is encountered:

\[
\text{Do}(A, s, s') \equiv \text{Poss}(A, s) \land s' = do(A, s),
\]

\[
\text{Do}(\varphi ?, s, s') \equiv s' = s \land \varphi[s],
\]

\[
\text{Do}(\delta_1; \delta_2, s, s') \equiv \exists s''. \text{Do}(\delta_1, s, s'') \land \text{Do}(\delta_2, s'', s'),
\]

\[
\text{Do}(\delta_1 | \delta_2, s, s') \equiv \text{Do}(\delta_1, s, s') \lor \text{Do}(\delta_2, s, s'),
\]

\[
\text{Do}((\pi x)\delta(x), s, s') \equiv \exists x. \text{Do}(\delta(x), s, s'),
\]

\[
\text{Do}(\delta^*, s, s') \equiv P_\delta(\vec{x}, s, s'),
\]

\[
P_\delta(\vec{x}, s, s') \equiv s = s' \lor \exists s''. s < s'' \leq s' \land \text{Do}(\delta, s, s'') \land P_\delta(\vec{x}, s'', s'),
\]

where $A$ is a primitive action, $\text{Poss}(A, s)$ the action precondition predicate, $\varphi[s]$ the result of restoring $s$ as the situation argument to the fluents in $\varphi$, $P_\delta$ a new predicate, and $\vec{x}$ the tuple of free variables in $\delta$.

For example, consider $\delta = (\pi x)(A(x) \mid (\pi x)B(x)^*)^*$. Let $\delta_1 = (A(x) \mid (\pi x)B(x)^*)^*$, $\delta_2 = A(x) \mid (\pi x)B(x)^*$, $\delta_3 = (\pi x)B(x)^*$, and $\delta_4 = B(x)^*$. Then

\[
\text{Do}(\delta, s, s') \equiv \exists x. \text{Do}(\delta_1, s, s'),
\]

\[
\text{Do}(\delta_1, s, s') \equiv P_\delta_2(x, s', s'),
\]

\[
P_\delta_2(x, s', s') \equiv s = s' \lor \exists s''. s < s'' \leq s' \land \text{Do}(\delta_2, s, s'') \land P_\delta_2(x, s'', s'),
\]

\[
\text{Do}(\delta_2, s, s') \equiv \text{Do}(A(x), s, s') \lor \text{Do}(\delta_3, s, s'),
\]

\[
\text{Do}(A(x), s, s') \equiv \text{Poss}(A(x), s) \land s' = do(A(x), s),
\]

\[
\text{Do}(\delta_3, s, s') \equiv \exists x. \text{Do}(\delta_4, s, s'),
\]

\[
\text{Do}(\delta_4, s, s') \equiv P_{B(x)}(x, s, s'),
\]

\[
P_{B(x)}(x, s, s') \equiv s = s' \lor \exists s''. s < s'' \leq s' \land \text{Do}(B(x), s, s'') \land P_{B(x)}(x, s'', s'),
\]

\[
\text{Do}(B(x), s, s') \equiv \text{Poss}(B(x), s) \land s' = do(B(x), s).
\]

Thus the axioms about the two extra predicates are

\[
P_{\delta_2}(x, s, s') \equiv s = s' \lor \exists s''. s < s'' \leq s' \land P_{\delta_2}(x, s'', s') \land \text{Poss}(A(x), s) \land s' = do(A(x), s) \lor \exists x. P_{B(x)}(x, s, s''),
\]

\[
P_{B(x)}(x, s, s') \equiv s = s' \lor \exists s''. s < s'' \leq s' \land P_{B(x)}(x, s'', s') \land \text{Poss}(B(x), s) \land s' = do(B(x), s).
\]

In contrast, according to (Levesque et al. 1997), the macro $\text{Do}(\delta, s, s')$ is defined as follows:

\[
\text{Do}(\delta, s, s') \equiv \exists x. \text{Do}(\delta_1, s, s'),
\]

\[
\text{Do}(\delta_1, s, s') \equiv (\forall x) \{ (\forall s_1) P(s_1, s_1) \land (\forall s_2, s_3) (P(s_1, s_2) \land \text{Do}(\delta_2, s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s') \},
\]

\[
\text{Do}(\delta_2, s, s') \equiv \text{Do}(A(x), s, s') \lor \text{Do}(\delta_3, s, s'),
\]

\[
\text{Do}(A(x), s, s') \equiv \text{Poss}(A(x), s) \land s' = do(A(x), s),
\]

\[
\text{Do}(\delta_3, s, s') \equiv \exists x. \text{Do}(\delta_4, s, s'),
\]

\[
\text{Do}(\delta_4, s, s') \equiv (\forall x) \{ (\forall s_1) P(s_1, s_1) \land (\forall s_2, s_3) (P(s_1, s_2) \land \text{Do}(B(x), s_2, s_3) \supset P(s_1, s_3)) \supset P(s, s') \},
\]

\[
\text{Do}(B(x), s, s') \equiv \text{Poss}(B(x), s) \land s' = do(B(x), s).
\]

As can be seen, when fully expanded, $\text{Do}(\delta, s, s')$ is a second-order sentence with nested second-order quantification.

To illustrate how to reason with our axiomatization, suppose there are two objects $a$ and $b$, and that $\text{Poss}(A(a), S_0)$ and $\text{Poss}(B(b), do(A(a), S_0))$ hold. We show that

\[
\text{Do}(\delta, S_0, \text{do}(B(b), do(A(a), S_0)))
\]

holds. It is easy to see that this follows from

\[
\exists x. P_{\delta_2}(x, S_0, S_2),
\]

where $S_2 = \text{do}(B(b), S_1)$ and $S_1 = \text{do}(A(a), S_0)$. This follows from

\[
\text{Poss}(A(a), S_0) \land P_{\delta_2}(a, S_1, S_2).
\]

The first conjunct is assumed to be true. The second conjunct follows from

\[
\exists x. P_{B(x)}(x, S_1, S_2).
\]

This is proved by showing $P_{B(x)}(b, S_1, S_2)$, which follows from

\[
\text{Do}(B(b), S_1, S_2),
\]

which is true because $\text{Poss}(B(b), S_1)$ is assumed to be true.

Now suppose that there is another primitive action $C(x)$. Furthermore, assume the unique names axioms for actions, $C(x) \neq A(y) \neq B(z)$. Then we can show that $\text{Do}(\delta, S_0, \text{do}(C(a), S_0))$ does not hold:

\[
\neg \text{Do}(\delta, S_0, S_3),
\]
where $S_3 = do(C(a), S_0)$, follows from
\[ \forall x. \neg P_{\delta_2}(x, S_0, S_3), \]
which is equivalent to
\[ \forall x \forall s. S_0 < s \leq S_3 \land P_{\delta_2}(x, s, S_3) \land \{ \text{Poss}(A(x), S_0) \land s = do(A(x), S_0) \} \lor \exists x. P_{B(x)}(x, S_0, s) \}
which is equivalent to
\[ \forall x \forall s. S_0 < s \leq S_3 \lor \left( \neg \{ \text{Poss}(A(x), S_0) \land s = do(A(x), S_0) \} \land \neg \exists x. P_{B(x)}(x, S_0, s) \right) \].

Because $S_3$ is the only $s$ that can make $S_0 < s \leq S_3$ true, thus the above assertion is equivalent to
\[ \forall x. \neg \{ \text{Poss}(A(x), S_0) \land S_3 = do(A(x), S_0) \} \land \neg \exists x. P_{B(x)}(x, S_0, S_3), \]
But $S_3 \neq do(A(x), S_0)$ for any $x$, thus the above assertion is equivalent to
\[ \forall x. \neg P_{B(x)}(x, S_0, S_3), \]
which is equivalent to
\[ \forall x. \neg \{ \text{Poss}(B(x), S_0) \land S_3 = do(B(x), S_0) \}, \]
which is equivalent to
\[ \forall x. S_3 \neq do(B(x), S_0), \]
which is always true as $S_3 = do(C(a), S_0)$.

The following result is not hard to prove.

**Theorem 1** Let $\text{Do1}(\delta, s, s')$ and $\text{Do2}(\delta, s, s')$ be the macro operators defined in (Levesque et al. 1997) and this paper, respectively. For any Golog complex action $\delta$, we have that
\[ \Sigma \cup T_3 \models \forall \vec{x}, s, s'. \text{Do1}(\delta, s, s') \equiv \text{Do2}(\delta, s, s'), \]
where $\vec{x}$ is the tuple of free variables in $\delta$, and $T_3$ is the set of axioms about $P_{\delta_1}$ for any $\delta_1$ such that $\delta_1$ occurs in $\delta$.

Notice that for a program like
\[ \delta = [A(x)^*]; ([\pi x]A(x))^*; (\pi x)A(x)^*, \]
$T_3$ would contain axioms about $P_{A(a)}(s, s')$, $P_{(\pi x)A(x)}(s, s')$, and $P_{A(x)}(s, s')$.

**Procedures**

Golog programs can have recursively defined procedures in the following form (Levesque et al. 1997):
\[ \text{proc } P_1(\vec{v_1}) \delta_1 \text{ endProc } ; \ldots ; \text{proc } P_n(\vec{v_n}) \delta_n \text{ endProc } ; \delta_0. \]

Here $P_i, 1 \leq i \leq n$, are procedures, and $\delta_0$ is the main program. The procedure bodies and the main program $\delta_0$, $0 \leq i \leq n$, are complex actions, extended by procedure calls. In the main program $\delta_0$, a procedure call $P_i(\vec{v_i})$ is treated like an action:
\[ Do(P_i(\vec{v_i}), s, s') \triangleq P_i(\vec{v_i}[s], s, s'), \]
where $P_i(\vec{v_i}, s, s')$ is a new predicate for the procedure $P_i$, and $\vec{v_i}[s]$, like $\varphi[s]$, is the tuple of terms obtained by restoring $s$ as the situation arguments in the functional fluents occurring in $\vec{v_i}$. As discussed in (Levesque et al. 1997), this amounts to call by value. These new predicates are defined by second-order sentences in (Levesque et al. 1997) to capture least fixed-point semantics of recursive procedures. Here we axiomatize these predicates using first-order axioms, with the help of some additional new predicates: For each $P_i(\vec{v_i}, s, s')$, we introduce a new one with the same name but with one more situation argument $P_i(\vec{v_i}, s, s', s'')$, and relate them with the following axioms:
\[ P_i(\vec{v_i}, s, s') \equiv \exists s'' . P_i(\vec{v_i}, s, s', s''), \]
\[ \neg P_i(\vec{v_i}, s, s', S_0), \]
\[ P_i(\vec{v_i}, s, s', do(a, s'')) \equiv \text{Do}(\delta_i[s''], s, s'), \]
\[ \text{Do}(P_i(\vec{v_i}, s', s')) s, s' \triangleq P_i(\vec{v_i}[s], s, s'), \]
where $\delta_i[s'']$ is the result of replacing every procedure call of the form $P_j(\vec{v_j})$ in $\delta_i$ by the call $P_j(\vec{v_j}, s'')$.

Intuitively, $P_i(\vec{v_i}, s, s', s'')$ if the procedure $P_i$ can return and yields the new situation $s'$ when performed in $s$, with at most $|s''|$ number of activations for all procedures, where $|s''|$ is the number of actions in the sequence $s$ such that $s'' = do(a, S_0)$.

Consider the following program in the blocks world from (Levesque et al. 1997) ($\text{maketower}(n)$ makes a tower of $n$ blocks; $\text{tower}(x, m)$ is true if there is a tower of $m$ blocks whose top one is $x$; $\text{stack}(x, n)$ places $n$ blocks on the tower whose top one is $x$; $\text{unstack}(x, n)$ removes $n$ blocks from the tower whose top is $x$):

```
proc maketower(n) 
    (\pi x, m)(tower(x, m))?;
    if m \leq n then stack(x, n - m)
    else unstack(x, m - n)
    endIf
endProc;
```

```
proc stack(x, n) 
    n = 0? \[ (\pi y)(put(y, x); stack(y, n - 1))\]
endProc;
```

```
proc unstack(x, n) 
    n = 0? \[ (\pi y)(on(x, y)?; movetotable(x);
                unstack(y, n - 1))\]
endProc;
```

```
maketower(7); \neg(\exists x)on(x, A)?
```

We have the following axioms for $\text{maketower}(n, s, s')$, $\text{stack}(x, n, s, s')$, and $\text{unstack}(x, n, s, s')$:
\[ \text{maketower}(n, s, s') \equiv \exists s_1 . \text{maketower}(n, s, s', s_1), \]
\[ \text{stack}(x, n, s, s') \equiv \exists s_1 . \text{stack}(x, n, s, s', s_1), \]
\[ \text{unstack}(x, n, s, s') \equiv \exists s_1 . \text{unstack}(x, n, s, s', s_1), \]
\[ \neg \text{maketower}(n, s, s', S_0), \]
maketower(n, s, s', do(a, s₁)) ≡ ∃x, m. [lower(x, m, s) ∧ m ≤ n ⊃ stack(x, n - m, s, s', s₁)] ∧ m > n ⊃ unstack(x, m - n, s, s', s₁),
¬stack(x, n, s, s', S₀),

stack(x, n, s, s', do(a, s₁)) ≡ (n = 0 ∧ s = s') ∨ 
∃y, s′′. s′′ = do(put(y, x), s) ∧ stack(y, n - 1, s′′, s', s₁),
unstack(x, n, s, s', do(a, s₁)) ≡ (n = 0 ∧ s = s') ∨ 
[∃y, s′′. on(x, y, s) ∧ s′′ = do(movetotable(x), s) ∧ unstack(y, n - 1, s′′, s', s₁)].

To see how our approach handles cycles, consider the following cyclic procedure:

\[
\text{proc } P \{ P \} \text{ endProc}
\]

Clearly, the call \( P \) would never terminate. Our axioms for \( P \) are:

\[
P(s, s') \equiv ∃s₁. P(s, s', s₁), \quad ¬P(s, s', S₀), \quad P(s, s', do(a, s₁)) \equiv P(s, s', s₁)
\]

By induction using the foundational axioms, it is easy to see that these axioms entail \( ∀s, s', s₁. ¬P(s, s', s₁) \), thus \( ∀s, s'. ¬P(s, s') \).

Again the following result is not hard to see.

**Theorem 2** Given a Golog program below

\[
\text{proc } P_{i}(\bar{v}_{i})\delta_{1} \text{ endProc} ; \ldots ; \text{proc } P_{n}(\bar{v}_{n})\delta_{n} \text{ endProc} ; \delta_{0}
\]

we have that

\[
Σ \cup T_{δ₀} \models ∀\bar{x}, s, s'. Do1(δ₀, s, s') \equiv Do2(δ₀, s, s'),
\]

where \( \bar{x} \) is the tuple of free variables in \( δ \), \( Do1(δ₀, s, s') \) and \( Do2(δ₀, s, s') \) the macro operators defined in (Levesque et al. 1997) and this paper, respectively, and \( T_{δ₀} \) the set of axioms about \( P₀ \) for any \( δ \) such that \( δ^* \) occurs in some \( δ_i \), \( 0 ≤ i ≤ n \).

### Concurrent processes

ConGolog (De Giacomo, Lespérance, and Levesque 2000) extends Golog by adding some concurrency operators that allows two programs to be executed concurrently. Instead of the \( Do \) macro, the main construct is a transition predicate \( Trans(δ, s, δ', s') \) which says one way to do \( δ \) in \( s \) is to perform a primitive or a testing action that will lead to situation \( s' \) with \( δ' \) as the remaining program to be executed. For example, \( Trans(a; b, s, b, do(a, s)) \) holds because performing the sequence \( a; b \) in \( s \) will result in performing \( a \), which leads to \( do(a, s) \) with \( b \) as the remaining program to do.

In (De Giacomo, Lespérance, and Levesque 2000), the axiomatization of \( Trans \) requires programs be refined and quantified over. For instance, the axiom for sequence is

\[ Trans(δ₁; δ₂, s, δ', s') \equiv \exists γ, δ = (γ; δ₂) \land Trans(δ₁, s, γ, s') \lor Final(δ₁, s) \land Trans(δ₂, s, δ', s'), \]

where \( Final(δ, s) \) is a predicate which says that \( δ \) has finished in \( s \).

Using \( Trans \), and by quantifying over programs, \( Do \) is then defined as

\[ Do(δ, s, s') \equiv ∃δ'. Trans*(δ, s, δ', s') \land Final(δ', s'), \]

where \( Trans^* \) is the transitive closure of \( Trans \).

Here we show that even for concurrent processes, the \( Do \) macro can be defined using first-order axioms only, and without needing to refine and quantify over programs.

Consider concurrent execution by interleaving \( δ₁ \parallel δ₂ \). Executing this program in \( s \) can result in \( s' \) if there is a sequence of primitive actions, some from \( δ₁ \) and the rest from \( δ₂ \), that can be executed in \( s \) to yield \( s' \). For example, there are six possible ways to interleave \( (P?; A) \) and \( (B; A) \), resulting in the following four possible sequences of primitive and test actions:

\[
(P?; A; B; A), (P?; B; A; A), (B; A; P?; A), (B; P?; A; A).
\]

If we ignore the test actions, as they do not result in changes of situations, we then get two possible sequences of primitive actions: \([A, B, A]\) and \([B, A, A]\). Notice that these are possible interleavings. Not all of them may come to fruit. For example, \( P? \) may not be true initially, thus cannot be performed as the first action.

If we are told that \([A, B, A]\) is the actual sequence of actions performed, then we know that the first action \( A \) must come from the process \( (P?; A) \), and the last from \( (B; A) \). However, if \([B, A, A]\) is the actual sequence, then for the two \( A \)'s in the sequence, we do not know from which process each of them come.

To keep track of which action comes from which process in a sequence of primitive actions, we introduce two special primitive actions \( Y \) (for yes), and \( N \) (for no). Situations consisting of only these two actions are called ‘schedules’:

\[ sch(S₀), sch(do(a, s)) \equiv sch(s) \land (a = Y \lor a = N). \]

For example, the schedule \( do(Y, do(N, do(Y, do(Y, S₀)))) \)

\( (do(Y, Y, N, Y), S₀) \) tells a process to first execute two actions, then wait for the other process to execute one action, then execute one more action. Referring back to our earlier example, if we know that \([B, A, A]\) is the actual sequence, then for process \( (P?; A) \), there can be only two possible schedules: \( do([N, Y, N], S₀) \) or \( do([N, Y, Y], S₀) \). The first implies that the first \( A \) in the sequence \([B; A; A]\) comes from the process, and the second that the second \( A \) in the sequence comes from it. Notice that our “schedule situations” only tell when to execute primitive actions while ignore test actions. For process \( (P?; A) \) to be successfully executed, we know that \( P \) must be tested and succeeded sometime before \( A \) is executed, but we do not care exactly when it was actually tested.

These special schedule situations will be used to model possible interleavings. Before we proceed, we first introduce some useful operators about situations:

- We say that two situations are of equal length, written \( el(s, s') \), if the number of actions in \( s \) is the same as that
in $s'$:
\[
el(S_0, S_0),
\]
\[
-\el(do(a, s), S_0),
\]
\[
el(do(a, s), do(b, s')) \equiv el(s, s').
\]

• We say one schedule $s_1$ is earlier than $s_2$, written \textit{earlier}($s_1, s_2$), if at least one action is scheduled earlier in $s_1$:

\[
\textit{earlier}(s_1, s_2) \equiv \exists s_3. do(Y, s_3) \leq s_1 \land do(N, s_3) \leq s_2.
\]

• A schedule can be split into two: \textit{split}($s_1, s_2$) holds if for any action, it is scheduled to be performed in $s$ iff it is scheduled to be performed either in $s_1$ or $s_2$, but not both:

\[
\text{split}(S_0, S_0),
\]
\[
\text{split}(do(N, s), s_1, s_2) \equiv \exists s_4. \text{split}(s, s_3, s_4) \land
\]
\[
s_1 = do(N, s_3) \land s_2 = do(N, s_4),
\]
\[
\text{split}(do(Y, s), s_1, s_2) \equiv \exists s_4. \text{split}(s, s_3, s_4) \land
\]
\[
[(s_1 = do(Y, s_3) \land s_2 = do(N, s_4)) \lor
\]
\[
(s_1 = do(N, s_3) \land s_2 = do(Y, s_4))].
\]

It is easy to see that splitting preserves the length of the schedule:

\[
sch(s) \land \text{split}(s, s_1, s_2) \supset el(s, s_1) \land el(s, s_2).
\]

• We have two special schedules: \textit{Full}($s, s'$) if the schedule from $s$ to $s'$ is full (all “$Y$”), and \textit{Empty}($s, s'$) if the schedule from $s$ to $s'$ is empty (all “$N$”):

\[
\text{Full}(s, s') \equiv \forall a, s_1 (s < do(a, s_1) \leq s' \supset a = Y),
\]
\[
\text{Empty}(s, s') \equiv \forall a, s_1 (s < do(a, s_1) \leq s' \supset a = N).
\]

Notice that \textit{Full}($s, s$) and \textit{Empty}($s, s$) are always true.

ConGolog has three main concurrency operators: $\delta_1 \parallel \delta_2$ (execute $\delta_1$ and $\delta_2$ concurrently by interleaving), $\delta_1 \triangleright \delta_2$ (interleaving $\delta_1$ and $\delta_2$ with priority to $\delta_1$), and $\delta\parallel$ (interleave $\delta$ any number of times).

The prioritized concurrency $\delta_1 \triangleright \delta_2$ needs special attention. Consider

\[(A; \text{fail?}) \mid B) \triangleright C.
\]

Suppose that initially both $A$ and $C$ are executable, but $B$ is not. Suppose further that if $C$ is executed, then $B$ becomes executable. According to the semantics in (De Giacomo, Lespérance, and Levesque 2000), the above program cannot be executed as (($A; \text{fail?}) \mid B) \triangleright C$, so its first action, which is $A$, is executed. But once $A$ is executed, it is at a dead end. In this view, if a process has priority, then its first executable action is performed, regardless whether executing that action can lead to a successful execution of this process. However, if we insist on only executing actions that can lead to termination, then the process with lower priority, $C$, would be executed first as the process with higher priority cannot be executed. After $C$ is performed, $B$ from the other process can then be executed.

Which interpretation is more appropriate depends on the applications. We’ll axiomatize both. In our framework, it is easier to axiomatize the latter, so we do it first.

Using schedule situations, we extend the $\Delta$ macro to concurrency as follows: $\Delta\delta, s, s'$ if with a full schedule, executing $\delta$ in $s$ can lead to $s'$:

\[
\Delta(\delta, s, s') \overset{df}{=} \exists s_1. \text{Full}(S_0, s_1) \land \text{el}(s', s_1) \land \Delta(\delta, s, s_1),
\]

where $\Delta(\delta, s, s', s_1)$ is a new macro meaning that executing the sequence of actions from $s$ to $s'$ according to schedule $s_1$ is a valid way to execute $\delta$ in $s$:

\[
\Delta(A, s, s', s_1) \overset{df}{=} \exists s_2, s_3, s'', s < do(A, s'') \leq s \land
\]
\[
s_2 < do(Y, s_3) \leq s_1 \land \text{el}(s, s_2) \land \text{el}(s'', s_3) \land
\]
\[
\text{Empty}(s_2, s_3) \land \text{Empty}(do(Y, s_3), s_1) \land \text{Poss}(A, s''),
\]
\[
\Delta(\delta, s, s', s_1) \overset{df}{=} \exists s_2, s_3, s'' \leq s'' \leq s' \land
\]
\[
s_2 \leq s_1 \land \text{el}(s, s_2) \land \text{Empty}(s_2, s_1) \land \text{Poss}(s''),
\]
\[
\Delta(\delta_1; s, s', s_1) \overset{df}{=} \exists s_2, s_3, s' \leq s_1 \land \text{el}(s_2, s'') \land
\]
\[
\Delta(\delta_1, s, s'', s_1) \land \Delta(\delta_2, s, s', s_1),
\]
\[
\Delta(\delta_1 \parallel \delta, s, s', s_1) \overset{df}{=} \exists s_2, s_3, \text{split}(s_1, s_2, s_3) \land
\]
\[
\Delta(\delta_1, s, s', s_2) \land \Delta(\delta_2, s, s', s_3),
\]
\[
\Delta(\delta_1 \triangleright \delta_2, s, s', s_1) \overset{df}{=} \exists s_2, s_3, \text{split}(s_1, s_2, s_3) \land
\]
\[
\Delta(\delta_2, s, s', s_2) \land \Delta(\delta_2, s, s', s_3),
\]
\[
\Delta(\delta \parallel, s, s', s_1) \overset{df}{=} s = s' \lor P_\delta^\parallel(\bar{x}, s', s_1),
\]
\[
\exists s_2 (s_2 \leq s_1 \land \text{el}(s_2, s_1) \land \text{Empty}(s_2, s_1) \supset P_\delta^\parallel(\bar{x}, s', s_1),
\]
\[
\neg \exists s_2 (s_2 \leq s_1 \land \text{el}(s_2, s_1) \land \text{Empty}(s_2, s_1) \supset P_\delta^\parallel(\bar{x}, s', s_1),
\]
\[
\exists s_2, s_3, \text{split}(s_1, s_2, s_3) \land s_1 \neq s_2 \land P_\delta^\parallel(\bar{x}, s', s_2) \land
\]
\[
\text{Poss}(s_2, s_3).
\]

Notice that for $\Delta(\delta, s, s', s_1)$ to make sense, the length of the terminating situation $s'$ should be the same as the length of the schedule $s_1$, and only the segment of the schedule from $s$ to $s'$ is relevant. Thus, for example,

\[
\Delta(\delta, do(a, S_0), do(b, do(a, S_0)), do(Y, do(N, S_0)))
\]

holds iff

\[
\Delta(\delta, do(a, S_0), do(b, do(a, S_0)), do(Y, do(Y, S_0)))
\]

holds as the two schedules are the same as far as the last action is concerned. We explain the first two definitions in more details below:
• For a primitive action $A$, $Do(A, s, s', s_1)$ holds if the schedule $s_1$ has exactly one occurrence of “$Y$”, and at that point, the corresponding action in $s'$ is $A$. Reformatted in terms of sequences of actions, the definition says that $Do(A, s, s', s_1)$ holds if for some sequences of actions $\alpha, \gamma, \beta$, and $\gamma_i$, $i = 1, 2$, we have:

$$s = do(\alpha, S_0),$$
$$s' = do(\beta_1, do(A, do(\beta_2, s))),$$
$$s_1 = do(\gamma_1, do(Y, do(\gamma_2, do(\gamma, S_0)))),$$
$$|\alpha| = |\gamma|,$$
$$|\beta_2| = |\gamma_i|, i = 1, 2,$$
$$a \in \gamma_i \supset a = N, i = 1, 2.$$  

• For a test action, $Do(\varphi?, s, s', s_1)$ holds if the schedule $s_1$ is empty, and $\varphi$ is true somewhere between $s$ and $s'$. Again, it is easier to understand in terms of sequences of actions. The definition says that $Do(\varphi, s, s', s_1)$ holds if for some sequences of actions $\alpha, \beta$, and $\gamma$, we have:

$$\exists s'' s \leq s'' \leq s' \land \varphi(s''),$$
$$s = do(\alpha, S_0),$$
$$s_1 = do(\gamma, do(\beta, S_0),$$
$$|\alpha| = |\beta|,$$
$$a \in \gamma \supset a = N.$$  

As an example, consider again the following prioritized concurrent program:

$$\delta = ((A; fail? \mid B) \gg C),$$
where $A$, $B$, and $C$ are primitive actions, and $fail$ is a fluent that is never true. Suppose that $Poss(A, S_0)$, $\neg Poss(B, S_0)$, $Poss(C, S_0)$, and $Poss(B, do(C, S_0))$. We show that

$$Do(\delta, S_0, do(B, do(C, S_0))).$$  

We need to prove

$$Do(\delta, S_0, do(B, do(C, S_0)), do(Y, do(Y, Y, S_0))).$$  

Let $\delta_1 = (A; fail?) \mid B$. The above assertion follows from the following three assertions:

$$Do(\delta_1, S_0, do(B, do(C, S_0)), do(Y, do(N, Y, S_0))),$$
$$Do(C, S_0, do(B, do(C, S_0)), do(N, do(Y, Y, S_0))),$$
$$\neg \exists s_1, s_2, s, split(do(Y, do(Y, Y, S_0)), s_1, s_2) \land$$
$$\text{earlier}(s_1, do(Y, do(N, Y, S_0))) \land Do(\delta_1, S_0, s, s_1).$$

The first assertion (5) follows from

$$Do(B, S_0, do(B, do(C, S_0)), do(Y, do(N, Y, S_0))),$$
which follows from the following facts:

$$el(S_0, S_0),$$
$$el(do(C, S_0), do(N, S_0)),$$
$$Empty(S_0, do(N, S_0)),$$
$$Empty(do(Y, do(N, S_0)), do(Y, do(N, S_0))),$$
$$Poss(\beta, do(C, S_0)).$$

The second assertion (6) follows easily from the following facts:

$$el(S_0, S_0),$$
$$Empty(S_0, S_0),$$
$$Empty(do(Y, S_0), do(N, do(Y, S_0))),$$
$$Poss(C, S_0).$$

For the last assertion (7), notice first that for

$$split(do(Y, do(Y, Y, S_0)), s_1, s_2) \land \text{earlier}(s_1, do(Y, do(N, S_0)))$$

to be true, $s_1$ must be either $do(Y, do(Y, S_0))$ or $do(N, do(Y, S_0))$. But in either case, there is no $s$ such that

$$Do(\delta_1, S_0, s, s_1)$$

can be true.

Notice that under ConGolog’s semantics (De Giacomo, Lespérance, and Levesque 2000), there is no $s$ such that $Do(\delta, S_0, s)$ can be true. This shows that our treatment of the prioritized concurrency operator $\gg$ is different from the one in ConGolog (De Giacomo, Lespérance, and Levesque 2000). If we ignore operator $\gg$, then our semantics is equivalent to the one in (De Giacomo, Lespérance, and Levesque 2000). Before we prove this, we first prove some formal properties about our axiomatization.

We mentioned that in $Do(\delta, s, s', s_1)$, only the part of the schedule between $s$ and $s'$ matters. We now make this precise.

We say that two situations $s_1$ and $s_2$ are the same between $s$ and $s'$, written $eq(s_1, s_2, s, s')$, if there are sequences $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ of actions such that:

- $s' = do(\alpha_1, s)$,
- $s_1 = do(\alpha_3, do(\alpha_2, s_3))$ and $s_2 = do(\alpha_4, do(\alpha_2, s_4))$,
- $|\alpha_1| = |\alpha_2|$,
- $s_3, s_4$, and $s$ are of equal length.

In terms of situation calculus formulas, we have

$$eq(s_1, s_2, s, s') \overset{def}{=}$$

$$\exists s_3, s_4, s_5, s_6, s_7 \leq s_1 \land s_4 \leq s_2 \land el(s_3, s) \land el(s_4, s) \land s_5 \leq s_1 \land s_6 \leq s_2 \land el(s_5, s') \land el(s_6, s') \land$$

$$\forall a, s_7, s_8, [s_7 \leq s_7 < s_8 \leq s_8 < s_6 \land el(s_7, s_8) \supset$$
$$do(a, s_7) \leq s_5 \equiv do(a, s_8) \leq s_6]$$

**Lemma 1** Under our axiomatization, we have that for any $\delta, s, s', s_1, s_2$, if $s'$, $s_1$, and $s_2$ are all of equal length, and the schedules $s_1$ and $s_2$ are the same between $s$ and $s'$, then

$$Do(\delta, s, s', s_1) \iff Do(\delta, s, s', s_2).$$

We mentioned that our formalization of prioritized concurrency $\gg$ is different from that in (De Giacomo, Lespérance, and Levesque 2000). We now show that when a program does not have this operator, and does not have the special $nil$ action used in (De Giacomo, Lespérance, and Levesque 2000), then our axiomatization agrees with that in (De Giacomo, Lespérance, and Levesque 2000).
Theorem 3 Let $\delta$ be a ConGolog complex action without the prioritized concurrency operator $\Rightarrow$, and without the special action nil. Let $\text{Do}1$ and $\text{Do}2$ be the macro operators defined in (De Giacomo, Lespérance, and Levesque 2000) and here in this paper, respectively. We have

$$\Sigma \cup T_1 \cup T_2 = \forall \bar{x}, s, s'. \text{Do1} (\delta, s, s') \equiv \text{Do2} (\delta, s, s'),$$

where $\bar{x}$ is the tuple of the free variables in $\delta$, $T_1$ the set of axioms about Trans, Trans’, and Final, and $T_2$ the set of axioms in this paper about sch, el, split, Full, Empty, $P_{\delta}$, for each $\delta_1$ such that $\delta_1^i$ is in $\delta$, and $P_{\delta}^i$ for each $\delta_1$ such that $\delta_1^i$ is in $\delta$.

This theorem is proved using the following lemma

Lemma 2 Let $\delta$ be a ConGolog complex action without $\Rightarrow$ and nil. Let $s \leq s'$ be two situations, and $s_1 \leq s_2 \leq s_3$ two schedules such that $el(s_2, s)$ and $el(s_1, s')$. Then $\text{Do} (\delta, s, s', s_1)$ iff for some sequence $s \leq s_1 \leq s_2 \leq s_3 \leq \cdots \leq s_n \leq s_n' \leq s'$ of situation terms, and a sequence $\delta = \delta_1, \cdots , \delta_{n+1}$ of ConGolog complex actions, we have that

- $\text{Trans} (\delta_i, S_i, \delta_{i+1}, S'_i), 1 \leq i \leq n$;
- $\text{Final} (\delta_{n+1}, S_{n+1})$;
- the sequence $S_1 \leq S'_1 \leq S_2 \leq S'_2 \leq \cdots \leq S_n \leq S_n'$ conforms with the schedule $s_1$ starting from $s_2$: for any $s_3, s_2 < do(Y, s_3) \leq s_1$ iff there is an $i$ such that $el(s_3, S_i)$ and $s_i < S_i'$.

We have already seen an example program that mentions $\Rightarrow$ for which our semantics differs from the one in (De Giacomo, Lespérance, and Levesque 2000). We now proceed to show that our approach can also handle the $\Rightarrow$ operator as defined in (De Giacomo, Lespérance, and Levesque 2000).

To do this, we introduce a new predicate $\text{PD} (\delta, s, s', s_1)$ (partially do the program), meaning that $\delta$ has been carried out from $s$ to $s'$ according schedule $s_1$. The new definition of $\text{Do} (\delta_1 \Rightarrow \delta_2, s, s', s_1)$ is as follows:

$$\text{PD} (\delta_1, s, s', s_1) \equiv \exists s_2, s_3. \{ \text{split} (s_1, s_2, s_3) \land \text{PD} (\delta_1, s, s', s_2) \land \text{PD} (\delta_2, s, s', s_3) \land \neg \exists s_4, s_5, s_6, s_7. \{ \text{split} (s_1, s_4, s_5) \land \text{earlier} (s_4, s_2) \land do(Y, s_6) \leq s_4 \land do(N, s_6) \leq s_2 \land s < s'' \leq s' \land el (do(Y, s_6), s'') \land \text{PD} (\delta_1, s, s'', do(Y, s_6)) \} \}.$$

The new predicate $\text{PD}$ is defined as follows.

$$\text{PD} (A, s, s', s_1) \equiv \exists s_2, s_3, s''. \{ \text{el} (s, s_2) \land \{ \text{Empty} (s_2, s_1) \lor \{ s < do(A, s'') \leq s' \land s_2 < do(Y, s_3) \leq s_1 \land \text{el} (s'', s_3) \land \text{Empty} (s_2, s_3) \land \text{Empty} (do(Y, s_3), s_1) \land \text{Poss} (A, s'') \} \},$$

$$\text{PD} (\varphi, s, s', s_1) \equiv \exists s_2, s_2 \leq s_1 \land el (s, s_2) \land \text{Empty} (s_2, s_1),$$

$$\text{PD} (\delta_1 \Rightarrow \delta_2, s, s', s_1) \equiv \text{PD} (\delta_1, s, s', s_1) \lor \exists s_2, s_2 \leq s_1 \land do(\delta_1, s, s', s_2) \land \text{PD} (\delta_2, s', s', s_1),$$

$$\text{PD} (\delta_1 \Rightarrow \delta_2, s, s', s_1) \lor \text{PD} (\delta_1, s, s', s_1) \lor \text{PD} (\delta_2, s, s', s_1),$$

$$\text{PD} (\delta_1 \Rightarrow \delta_2, s, s', s_1) \lor \text{PD} (\delta_1, s, s', s_1) \lor \text{PD} (\delta_2, s, s', s_1),$$

Notice that the last axiom will not lead to a cyclic definition as splitting $s_1$ into $s_2$ and $s_3$, with $s_3 \neq s_1$, will eventually lead to $s_3$ being an empty schedule.

Related work

If we consider generally the problem of formalizing semantics of programs, there is a huge literature about this in computer science. In particular, Golog and its original semantics was strongly influenced by dynamic logic (Harel 1979). The unique feature of Golog is that actions are specified axiomatically by successor state axioms, and the $\text{Do}$ macro axiomatizes legal execution paths in the situation calculus.

Baier, Fritz and McIlraith (2007) proposed to translate a Golog program to a basic action theory. Later, Fritz, Baier and McIlraith (2008) extended it to ConGolog. Their motivation was to use Golog and ConGolog to encode domain specific control information in planning. Given a basic action theory $D$, and a goal $G$, instead of asking if there is an executable sequence of actions that can achieve $G$: whether $D \models \exists s. \text{Executable} (s) \land G [s]$ holds, they proposed to encode the domain specific control information about $D$ as a ConGolog program $\delta$, and ask if there is a sequence of actions that is a legal execution of $\delta$ and can achieve $G$: whether $D \models \exists s. \text{Do} (\delta, S_0, s) \land G [s]$.

To implement this, they translated $D$ and $\delta$ to a new basic action theory $D'$ in a new language, and showed that there is a mapping $\tau$ from sequences of actions (situations) in the language of $D'$ to sequences of actions (situations) in the language of $D$ such that $D' \models \exists s. \text{Executable} (s) \land G [s]$ iff $D \models \exists s. \text{Do} (\delta, \tau (S), \tau (S)) \land G [\tau (S)]$. Thus, the semantics of $\delta$ is captured by $D'$. While the axioms in $D'$ are also first-order, they use natural numbers. In comparison, our axiomatization uses the same language of the original situation calculus. An interesting question is whether our first-order reformulation of Golog and ConGolog semantics could make the translation from $D$ and $\delta$ to $D'$ simpler.
While we consider Golog and ConGolog programs, Gabaldon (2003; 2004) considered “programs” written in temporal logic formulas, and their translation into first-order situation calculus.

Conclusions
We have shown that by using the foundational axioms in the situation calculus, it is possible to give a first-order semantics to Golog and ConGolog programs. In the case of Golog, this is achieved by introducing a new predicate when an iteration is encountered. In the case of ConGolog, this is done by using so-called scheduling situations. The overall message is that once we have second-order induction on the situations, other inductive structures and fixed-points can be defined in first-order terms. We hope the simpler first-order semantics will lead to more effective ways of doing program verification and synthesis in Golog and ConGolog.

Acknowledgments
My thanks to Hector Levesque for his comments on an earlier version of this paper, and for a pointer to a related work. I also thank the anonymous reviewers of this paper for this symposium and for KR2014 for their comments on an earlier version of this paper. This work was supported in part by HK RGC under GRF 616013, and NSFC of China under grant 6137061

References