Sequential, Temporal GOLOG

Ray Reiter
Department of Computer Science
University of Toronto
Toronto, Canada, M5S 1A4
reiter@cs.toronto.edu

Abstract

We extend the ontology and foundational axioms of the sequential situation calculus to include time. When combined with a view of actions with durations as processes that are initiated and terminated by instantaneous actions, this explicit representation of time yields a very rich account of interleaving concurrency in the situation calculus. Based upon this axiomatization, we extend the semantics and interpreter for the situation calculus-based programming language GOLOG to the temporal domain, and illustrate the resulting increased functionality of the language via a GOLOG program describing the temporal behaviour of a coffee delivery robot. Among other features, this program illustrates how, in the GOLOG framework, one can represent concurrent processes with explicit time.

1 Introduction

The situation calculus [15] has long been the formalism of choice in artificial intelligence for theoretical investigations of properties of actions (e.g. [12, 1, 4]), but until recently, it has not been taken seriously as a specification or implementation language for practical problems in dynamic world modeling. Exceptions to this are the situation calculus-based programming languages GOLOG and CONGOLOG [11, 2], and some of their applications to planning [10, 20], robotics [7, 5, 3] and agent programming [8, 6]. The perspective being pursued by the Cognitive Robotics Group at the University of Toronto is to reduce the "traditional" reliance on *planning* for eliciting interesting robot behaviors, and instead provide the robot with *programs* written in a suitable high-level language [9], in our case, GOLOG or CONGOLOG. Such programming languages must be very expressive, providing a range of primitives for describing agent behaviors in complex worlds, for example, sensing actions, time, inter-agent communication, beliefs, goals, intentions, etc. Moreover, because of their complexity, they must do so in a semantically clear way. As indicated above, our approach to the design of such languages has been through the situation calculus.

The purpose of this paper is to extend the functionality of these languages by endowing them with the ability to represent time explicitly. Specifically, we extend the ontology and foundational axioms of the sequential situation calculus of [19] to include time. By pursuing an idea first proposed for the situation calculus by Pinto [17] and Ternovskaia [24], we show how one can view actions with durations as processes that are initiated and terminated by instantaneous actions. This conceptual shift, when coupled with an explicit representation for time, provides a rich account of interleaving concurrency in the situation calculus. Based upon the axioms for the sequential, temporal situation calculus, we next extend the semantics and interpreter for the situation calculus-based programming language GOLOG to the temporal domain. Finally, we illustrate the resulting increased functionality of the language with a GOLOG program describing the temporal behavior of a coffee delivery robot. Among other features, this program illustrates how, in the GOLOG framework, one can straightforwardly represent concurrent processes with explicit time.

2 The Situation Calculus

The situation calculus is a second order language specifically designed for representing dynamically changing worlds. All changes to the world are the result of named *actions*. A possible world history, which is simply a sequence of actions, is represented by a first order term called a *situation*. The constant S_0 is used to denote the *initial situation*, namely the empty history. Non-empty histories are constructed using a distinguished binary function symbol do; $do(\alpha, s)$ denotes the successor situation to s resulting from performing the action α. Actions may be parameterized. For example, $put(x, y)$ might stand for the action of putting object x on object y, in which case $do(put(A, B), s)$ denotes that situation resulting from placing A on B when the history is s. In the situation calculus, actions are denoted by first order terms, and situations (world histories) are also first order terms. For example, $do(putdown(A), do(walk(L), do(pickup(A), S_0)))$ is the situation denoting the world history consisting of the sequence of actions [pickup(A), walk(L), putdown(A)]. Notice that the sequence of actions in a history, in the order in which they occur, is obtained from a situation term by reading off the actions from right to left.

Relations whose truth values vary from situation to situation are called *relational fluents*. They are denoted by predicate symbols taking a situation term as their last argument. Similarly, functions whose values vary from situation to situation are called *functional fluents*, and are denoted by function symbols taking a situation term as their last argument. For example, $isCarrying(robot, p, s)$, meaning that a $robot$ is carrying package p in situation s, is a relational fluent; $location(robot, s)$, denoting the location of $robot$ in situation s, is a functional fluent.

To characterize the domain of situations, various foundational axioms have been proposed for the situation calculus [22, 13, 18]. The following set of axioms modifies these earlier proposals, and appears to be the simplest appropriate formulation of foundational axioms for the situation calculus [19]:

$$(\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s), \tag{1}$$

$$do(a, s) = do(a', s') \supset a = a' \wedge s = s',^{1} \tag{2}$$

$$\neg s \sqsubset S_0, \tag{3}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'. \tag{4}$$

The first is a second order induction axiom. The next is a unique names axiom for situations. The last two axioms define an ordering relation $\sqsubset$ on situations. (Here, $s \sqsubseteq s'$ is an abbreviation for $s \sqsubset s' \vee s = s'$.) The intuitive reading of a situation is as a finite sequence of actions, in which case $s \sqsubseteq s'$ means that situation s' can be obtained from s by adding a finite number of actions onto s. Here, S_0 is a distinguished constant symbol of the language of the situation calculus, denoting the initial situation.[2]

[1] In what follows, lower case Roman characters will denote variables in formulas. Moreover, free variables will always be implicitly universally prenex quantified.

[2] Notice that in the situation calculus, the constant S_0 is just like NIL in the programming language LISP, and do acts like $cons$. Situations are simply *lists* of primitive actions. For example, the situation term $do(C, do(B, do(A, S_0)))$ is simply an alternative syntax for the LISP list (CBA) $(= cons(C, cons(B, cons(A, NIL))))$. To obtain the action *history* corresponding to this term, namely the performance of action A, followed by B, followed by C, read this list from right to left. Therefore, when situation terms are read from right to left, the relation $s \sqsubset s'$ means that situation s is a proper subhistory of the situation s'. The induction axiom (1) simply provides induction for lists: If the empty list has property P and if, whenever list s has property P so does $cons(a, s)$, then all lists have property P.

According to these axioms, a situation is a finite sequence of actions. There are no constraints on the actions entering into such a sequence, so that it may not be possible to actually execute these actions one after the other. Frequently, we shall be interested only in *executable* situations, namely, those action sequences in which it is actually possible to perform the actions one after the other. To characterize such situations, we rely on the distinguished predicate symbol $Poss$ of the language of the situation calculus. Intuitively, $Poss(a, s)$ means that it is possible to perform the action a in situation s. In specifying a domain of application, one would include axioms characterizing $Poss$ for each primitive action of the domain. Now we can introduce the abbreviation:

$$s < s' \stackrel{def}{=} s \sqsubset s' \wedge (\forall a, s^*).s \sqsubset do(a, s^*) \sqsubseteq s' \supset Poss(a, s^*). \tag{5}$$

So $s < s'$ means that s is an initial subsequence of s', and all the actions of s' following those of s can be executed one after the other. In particular, $S_0 < s$ means that every action in s is possible, so they can all be executed in sequence. To captures formally this intuitive concept of the executable situations, we introduce the abbreviation:

$$executable(s) \stackrel{def}{=} S_0 < s \vee S_0 = s.$$

In addition to the above domain independent axioms, one must specify other axioms when formalizing an application domain (details in [11]):

- *Action precondition axioms*, one for each primitive action, characterizing the relation $Poss$.
- *Successor state axioms*, one for each fluent. These capture the causal laws of the domain, together with a solution to the frame problem [21]. The solution to the frame problem embodied in these axioms applies only when all the primitive actions of the application domain are deterministic. Moreover, [21] does not treat state constraints, and therefore, does not address the ramification or qualification problems.[3]
- Unique names axioms for the primitive actions.
- Axioms describing the initial situation – what is true initially, before any actions have occurred. This is any finite set of sentences mentioning no situation term, or only the situation term S_0.

3 GOLOG

GOLOG [11] is a situation calculus-based logic programming language for defining complex actions us-

[3] But see [13, 16, 17] for possible ways to do this, while preserving the successor state axiom approach.

ing a repertoire of user specified primitive actions. GOLOG provides the usual kinds of imperative programming language control structures as well as three flavours of nondeterministic choice:

1. *Sequence:* α ; β. Do action α, followed by action β.
2. *Test actions:* p? Test the truth value of expression p in the current situation.
3. *While loops:* **while** p **do** α **endWhile.**
4. *Conditionals:* **if** p **then** α **else** β.
5. *Nondeterministic action choice:* $\alpha \mid \beta$. Do α or β.
6. *Nondeterministic choice of arguments:* $(\pi\ x)\alpha$. Nondeterministically pick a value for x, and for that value of x, do the action α.
7. *Nondeterministic repetition:* α^*. Do α a nondeterministic number of times.
8. *Procedures*, including recursion.

The semantics of GOLOG programs is defined by macro-expansion, using a ternary relation *Do* (see [11] for a full description). *Do* is defined inductively on the structure of its first argument as follows:

Primitive actions:

$$Do(a, s, s') \stackrel{def}{=} Poss(a, s) \wedge s' = do(a, s). \quad (6)$$

Test actions:

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

Here, ϕ is a test expression of our programming language; it stands for a situation calculus formula, but with all situation arguments suppressed. $\phi[s]$ denotes the situation calculus formula obtained by restoring situation variable s to all fluent names mentioned in ϕ.

Sequence:

$$Do(\delta_1;\delta_2, s, s') \stackrel{def}{=} (\exists s^*).Do(\delta_1, s, s^*) \wedge Do(\delta_2, s^*, s').$$

Nondeterministic choice of two actions:

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

Nondeterministic choice of action arguments:

$$Do((\pi x)\, \delta(x), s, s') \stackrel{def}{=} (\exists x)\, Do(\delta(x), s, s').$$

Conditionals:

$$\textbf{if } p \textbf{ then } \alpha \textbf{ else } \beta \stackrel{def}{=} p? ; \alpha \mid \neg p? ; \beta.$$

Similar (but more complicated) definitions are given for iteration and procedures.

$Do(program, s, s')$ is an *abbreviation* for a situation calculus formula whose intuitive reading is that s' is one of the situations reached from s by executing *program*. This means that to execute *program*, one must *prove*, using the situation calculus axiomatization of some background domain, the situation calculus formula $(\exists s)Do(program, S_0, s)$. Any binding for s obtained by a constructive proof of this sentence is an execution trace of *program*.

In [11] a GOLOG interpreter was given, written in Prolog. We present a variant of this here because we shall be suitably modifying it to accommodate time, and because we shall be presenting an example of a corresponding temporal GOLOG program.

A Golog Interpreter in Prolog

```
:- op(950, xfy, [:]). /* Sequence.*/
:- op(950, xfy, [#]). /* Nondeterministic action
                         choice.*/
:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880,xfy, [<=>]).  /* Equivalence */

do(E1 : E2,S,S1) :- do(E1,S,S2), do(E2,S2,S1).
do(?(P),S,S) :- holds(P,S).
do(E1 # E2,S,S1) :- do(E1,S,S1) ; do(E2,S,S1).
do(if(P,E1,E2),S,S1) :-
             do(?(P) : E1 # ?(-P) : E2,S,S1).
do(star(E),S,S1) :- S1 = S ;  do(E : star(E),S,S1).
do(while(P,E),S,S1):-
                  do(star(?(P) : E) : ?(-P),S,S1).
do(pi(V,E),S,S1) :- sub(V,_,E,E1), do(E1,S,S1).
do(E,S,S1) :- proc(E,E1), do(E1,S,S1).
do(E,S,do(E,S)) :- primitive_action(E), poss(E,S).

/* sub(Name,New,T1,T2): T2 is T1 with Name
   replaced by New. */

sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1],
         sub_list(X1,X2,L1,L2), T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :-
       sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

/* The holds predicate implements the Lloyd-Topor
   transformations on test conditions.  */

holds(P & Q,S) :- holds(P,S), holds(Q,S).
holds(P v Q,S) :- holds(P,S); holds(Q,S).
holds(P => Q,S) :- holds(-P v Q,S).
holds(P <=> Q,S) :- holds((P => Q) & (Q => P),S).
holds(-(-P),S) :- holds(P,S).
holds(-(P & Q),S) :- holds(-P v -Q,S).
holds(-(P v Q),S) :- holds(-P & -Q,S).
holds(-(P => Q),S) :- holds(-(-P v Q),S).
holds(-(P <=> Q),S) :-
        holds(-((P => Q) & (Q => P)),S).
holds(-all(V,P),S) :- holds(some(V,-P),S).
      /* Negation by failure */
holds(-some(V,P),S) :- not holds(some(V,P),S).
holds(-P,S) :- isAtom(P), not holds(P,S).
holds(all(V,P),S) :- holds(-some(V,-P),S).
holds(some(V,P),S) :- sub(V,_,P,P1), holds(P1,S).

/* The following clause treats the holds predicate
   for all atoms, including Prolog system
```

```
    predicates. For this to work properly, the
    GOLOG programmer must provide, for all atoms
    taking a situation argument, a clause giving
    the result of restoring its suppressed
    situation argument, for example:
      restoreSitArg(ontable(X),S,ontable(X,S)). */

holds(A,S) :- restoreSitArg(A,S,F), F ;
       not restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- not (A = -W ; A = (W1 & W2) ;
    A = (W1 => W2) ; A = (W1 <=> W2) ;
    A = (W1 v W2) ; A = some(X,W) ; A = all(X,W)).

restoreSitArg(poss(A),S,poss(A,S)).
```

It is possible to prove the soundness of this interpreter, with respect to the above semantics, when the usual Prolog closed world assumption is made about the initial database, namely, that complete information is available about the initial situation. Therefore, this interpreter is only suitable for applications in which this closed world assumption holds.[4] The `holds` predicate in this interpreter evaluates test conditions of GOLOG programs. Since such test conditions can be arbitrary first order formulas, the `holds` predicate first converts them to Prolog executable form using the Lloyd-Topor transformations [14].

4 Interleaving Concurrency in the Situation Calculus

The possibility of concurrent execution of actions leads to many difficult formal and conceptual problems, quite independently of the underlying knowledge representation language. For example, what can one mean by the concurrent action $\{walk(A, B), chewGum\}$? Intuitively, both actions have durations. By this concurrent action, do we mean that both actions have the same duration? That the time segment occupied by one is entirely contained in that occupied by the other? That their time segments merely overlap? What if there are three actions and the first overlaps the second, the second overlaps the third, but the first and third do not overlap; do they all occur concurrently? A representational device in the situation calculus for overcoming these problems is to conceive of such actions as *processes*, represented by relational fluents, and to introduce durationless (instantaneous) actions that initiate and terminate these processes [17, 24]. For example, instead of the monolithic action representation $walk(x, y)$, we might have instantaneous actions $startWalk(x, y)$ and $endWalk(x, y)$, and the process of walking from x to y, represented by the relational fluent $walking(x, y, s)$. $startWalk(x, y)$ causes the fluent $walking$ to become true, $endWalk(x, y)$ causes it to become false. Similarly, we might represent the $chewGum$ action by the pair of instantaneous actions $startChewGum$ and $endChewGum$, and the relational fluent $chewingGum(s)$. It is straightforward to represent these fluents and instantaneous actions in the situation calculus. For example, here are the action precondition and successor state axioms for the walking action:

$$Poss(startWalk(x, y), s) \equiv \neg(\exists u, v) walking(u, v, s) \wedge location(s) = x,$$

$$Poss(endWalk(x, y), s) \equiv walking(x, y, s),$$

$$walking(x, y, do(a, s)) \equiv a = startWalk(x, y) \vee walking(x, y, s) \wedge a \neq endWalk(x, y),$$

$$location(do(a, s)) = y \equiv (\exists x) a = endWalk(x, y) \vee location(s) = y \wedge \neg(\exists x, y') a = endWalk(x, y').$$

With this device of instantaneous *start* and *end* actions in hand, we can represent arbitrarily complex concurrency. For example,

$$do(endWalk(A, B), do(endChewGum, do(startChewGum, do(startWalk(A, B), S_0)))),$$

in which the gum-chewing process is initiated after the walking process, and terminated before the end of the walking process. Or, we can have the gum-chewing start before the walking, and terminate before the walking ends:

$$do(endWalk(A, B), do(endChewGum, do(startWalk(A, B), do(startChewGum, S_0)))).$$

In other words, we can represent any overlapping occurrences of the walking and chewing gum processes, *except for exact co-occurrences of any of the instantaneous initiating and terminating actions.* For many applications, this is sufficient. The great advantage is that this style of interleaved concurrency can be represented in the "classical" sequential situation calculus, and no new extensions of the theory are necessary. However, as yet, we have no explicit representation for time; the axioms for the situation calculus capture a purely qualitative notion of time. Sequential action occurrence is the only temporal concept represented by the axioms; an action occurs *before* or *after* another. It may occur one millisecond or one year before its successor; the axioms are neutral on this question. So a situation $do(A_n, do(A_{n-1}, \ldots, do(A_1, S_0) \cdots))$ must be understood as a world history in which, after a nondeterministic period of time, A_1 occurs, then, after a nondeterministic period of time, A_2 occurs, etc. If, for example, this action sequence was the result of a GOLOG robot program execution, then the robot's action execution system would make the decision about the exact times at which these actions would be performed sequentially in the physical world, but the axioms, being silent on action occurrence times, con-

[4] Of course, the general theory of GOLOG does not suffer from this restriction, only the specific implementation given above.

tribute nothing to this decision. Our objective now is to show how to incorporate time into the situation calculus, after which one can specify axiomatically and via GOLOG programs the times at which actions are to occur.

5 The Sequential, Temporal Situation Calculus

Now, we add an explicit representation for time to the sequential situation calculus of Section 2. This will allow us to specify the exact times, or a range of times, at which actions in a world history occur. For the reasons indicated in the previous section, we consider only instantaneous actions. We want to represent the fact that a given such action occurs at a particular time. Recall that in the situation calculus, actions are denoted by first order terms, like $start\_meeting(Susan)$ or $bounce(ball, wall)$. Our proposal for adding a time dimension to the situation calculus is to provide a new temporal argument to all instantaneous actions, denoting the time at which that action occurs. Thus, $bounce(ball, wall, 7.3)$ might be the instantaneous action of $ball$ bouncing against $wall$ at time 7.3.

We now extend the foundational axioms for the sequential situation calculus of Section 2 to accommodate time. We retain these axioms, and add only one new axiom. Also, the definition of executable situation requires modification. We begin by introducing a function symbol $time$: $time(a)$ denotes the time of occurrence of action a. This means that in any application involving a particular action $A(\vec{x}, t)$, we shall need an axiom telling us the time of the action A: $time(A(\vec{x}, t)) = t$, for example, $time(start\_meeting(person, t)) = t$. Next, it will be convenient to have a function $start$: $start(s)$ denotes the start time of situation s. This requires the axiom:

$$start(do(a, s)) = time(a). \tag{7}$$

We do not define the start time of S_0; this is arbitrary, and may (or may not) be specified to be any number, depending on the application. Notice also that we imagine temporal variables to range over the reals, although nothing prevents them from ranging over the integers, rationals, or anything else on which a binary relation $<$ is defined. In this connection, we do not provide axioms for the reals (or integers), but rely instead on the standard interpretation of the reals and their operations (addition, multiplication, etc) and relations ($<$, $\leq$, etc).

Next we reconsider the relation $<$ on situations. Intuitively, $s \leq s'$ means that one can get to s' from s by a sequence of possible actions. Consider the situation $do(bounce(B, W, 4), do(start\_meeting(Susan, 6), S_0))$, in which the time of the second action precedes that of the first. Intuitively, we do not want to consider such an action sequence executable, and we amend our definition (5) for $<$ accordingly:

$$\begin{aligned} s < s' \stackrel{def}{=} {} & s \sqsubset s' \wedge \\ & (\forall a, s^*).s \sqsubset do(a, s^*) \sqsubseteq s' \supset \\ & \quad Poss(a, s^*) \wedge start(s^*) \leq time(a). \end{aligned} \tag{8}$$

Now, $s < s'$ means that one can get to s' from s by a sequence of possible actions, and moreover, the times of those action occurrences must be nondecreasing. We are here overloading the predicate $<$; it is used to order situations as well as numbers in the temporal domain. Its meaning will always be clear from context.

Finally, notice that the constraint $start(s^*) \leq time(a)$ in abbreviation (8) permits executable action sequences in which the time of an action may be the same as the time of a preceding action. For example,

$$\begin{aligned} & do(end\_lunch(Bill, 4), \\ & \qquad do(start\_meeting(Susan, 4), S_0)) \end{aligned}$$

might be a perfectly good executable situation, which is defined by a sequence of two actions, each of which has the same occurrence time, but one of which ($start\_meeting(Susan, 4)$) "occurs before" the other ($end\_lunch(Bill, 4)$). This means that we provide for concurrent execution of instantaneous actions, but unlike the true concurrency treated in [23], we are here giving an *interleaving* account of concurrency. There are many reasons for allowing two or more interleaved actions to have the same occurrence times. One is we can often give an interleaving account of action co-occurrences without introducing the more complex formal machinery of [23]. Another is that often an action occurrence serves as an enabling condition for the simultaneous occurrence of another action. For example, cutting a weighted string at time t enables the action $startFalling(t)$. Both actions occur at the same time, but conceptually, the falling event happens "immediately after" the cutting. Accordingly, we want to treat the situation $do(startFalling(t), do(cutString(t), S_0))$ as an executable situation.

There are many advantages to using interleaving instead of true concurrency, whenever this is possible. For example, the precondition interaction problem [17] cannot arise in this case, neither can the possibility of infinitely many action co-occurrences [23]. Moreover, as we shall see with the example to follow, the combination of instantaneous actions, explicit representation of time, and an interleaving account of concurrency provides for a very rich, yet formally simple representation language for processes that overlap in temporally complex ways.

(1) - (4) and (7) are the foundational axioms for the *sequential, temporal situation calculus*. The development given above of these foundational axioms has many similarities to that given by Reiter in [23] for the *concurrent, temporal situation calculus*. The principal difference is that [23] treats true concurrency,

where concurrent actions are sets of primitive instantaneous actions. It is possible to obtain the foundational axioms for the sequential, temporal situation calculus from those of [23] by requiring that all concurrent actions be singleton sets, and identifying the primitive action A with the "concurrent" action $\{A\}$, but the above approach, where we started from scratch, seemed to us conceptually more attractive.

6 Sequential, Temporal GOLOG

With the above axioms for the sequential, temporal situation calculus in hand, it is easy to modify the GOLOG semantics and interpreter to accommodate time. Semantically, we need only change the definition of the Do macro for primitive actions (6) to:

$$Do(a,s,s') \stackrel{def}{=} \\ Poss(a,s) \wedge start(s) \leq time(a) \wedge s' = do(a,s).$$

Everything else about the definition of Do remains the same. One can prove that with this modification, whenever a sequential temporal GOLOG program terminates, it does so in an executable situation according to the revised definition (8) for executable situations. To suitably modify the GOLOG interpreter of Section 3, replace the clause

```
do(E,S,do(E,s)):- primitive_action(E),
                  poss(E,S).
```

by

```
do(E,S,do(E,S)) :- primitive_action(E),
                   poss(E,S), start(S,T1),
                   time(E,T2), T1 <= T2.
```

Finally, because we have introduced a new predicate, $start$, taking a situation argument, we must also augment the earlier GOLOG interpreter with the clause:

```
start(do(A,S),T) :- time(A,T).
```

We can now write sequential, temporal GOLOG programs. However, to execute such programs, the GOLOG interpreter must have a temporal reasoning component. It must, for example, be able to infer that $T_1 = T_2$ when given that $T_1 \leq T_2 \wedge T_2 \leq T_1$. While such a special purpose temporal theorem prover could be written and included in the GOLOG interpreter, we prefer to rely on a logic programming language with a built-in constraint solving capability. Specifically, we appeal to the ECRC Common Logic Programming System ECLIPSE 3.5.2; this provides a built-in Simplex algorithm for solving linear equations and inequalities over the reals. So we shall assume that our GOLOG program makes use of linear temporal relations like $2 * T_1 + T_2 = 5$ and $3 * T_2 - 5 \leq 2 * T_3$, and rely on ECLIPSE to perform the temporal reasoning for us. ECLIPSE provides a special syntax for those relations over the reals for which it provides a built-in theorem prover. These are: $ =, $ <>, $ >=, $ >, $ <=, $ <, with the obvious meanings. So, in ECLIPSE, the above modification of the GOLOG interpreter to include time is:

```
do(E,S,do(E,S)) :- primitive_action(E),
                   poss(E,S), start(S,T1),
                   time(E,T2), T1 $<= T2.
```

All other clauses of the earlier interpreter as given in Section 3 will work correctly under ECLIPSE.

6.1 Example: A Coffee Delivery Robot

Here, we describe a robot whose task is to deliver coffee in an office environment. The robot is given a schedule of every employee's preferred coffee periods, as well as information about the times it takes to travel between various locations in the office. The robot can carry just one cup of coffee at a time, and there is a central coffee machine from which it gets the coffee. Its task is to schedule coffee deliveries such that, if possible, everyone gets coffee during his/her preferred time periods. For simplicity, we assume that both the actions of picking up a cup of coffee, and giving it to someone are instantaneous. We represent the action of going from one location to another, which intuitively does have a duration, by a process and a pair of instantaneous $start$ and end actions, as described in Section 4.

Primitive actions:

- $pickupCoffee(t)$. The robot picks up a cup of coffee from the coffee machine at time t.
- $giveCoffee(p,t)$. The robot gives a cup of coffee to p at time t.
- $startGo(loc_1, loc_2, t)$. The robot starts to go from location loc_1 to loc_2 at time t.
- $endGo(loc_1, loc_2, t)$. The robot ends its process of going from location loc_1 to loc_2 at time t.

Fluents:

- $robotLocation(s)$. A functional fluent denoting the robot's location in situation s.
- $hasCoffee(person, s)$. $person$ has coffee in s.
- $going(loc_1, loc_2, s)$. In situation s, the robot is going from loc_1 to loc_2.
- $holdingCoffee(s)$. In situation s, the robot is holding a cup of coffee.

Situation Independent Predicates and Functions

- $wantsCoffee(person, t_1, t_2)$. $person$ wants to receive coffee at some point in the time period $[t_1, t_2]$.
- $office(person)$. Denotes the office of $person$.
- $travelTime(loc_1, loc_2)$. Denotes the amount of time that the robot takes to travel between loc_1 and loc_2.
- CM. Constant denoting coffee machine's location.

- *Sue*, *Mary*, *Bill*, *Joe*. Constants denoting people.

Action Precondition Axioms:

$$Poss(pickupCoffee(t), s) \equiv \neg holdingCoffee(s) \wedge robotLocation(s) = CM,$$

$$Poss(giveCoffee(person, t), s) \equiv holdingCoffee(s) \wedge robotLocation(s) = office(person),$$

$$Poss(startGo(loc_1, loc_2, t), s) \equiv \neg(\exists l, l')going(l, l', s) \wedge loc_1 \neq loc_2 \wedge robotLocation(s) = loc_1,$$

$$Poss(endGo(loc_1, loc_2, t), s) \equiv going(loc_1, loc_2, s).$$

Successor State Axioms

$$hasCoffee(person, do(a, s)) \equiv (\exists t)a = giveCoffee(person, t) \vee hasCoffee(person, s),$$

$$robotLocation(do(a, s)) = loc \equiv (\exists t, loc')a = endGo(loc', loc, t) \vee robotLocation(s) = loc \wedge \neg(\exists t, loc', loc'')a = endGo(loc', loc'', t),$$

$$going(l, l', do(a, s)) \equiv (\exists t)a = startGo(l, l', t) \vee going(l, l', s) \wedge \neg(\exists t)a = endGo(l, l', t),$$

$$holdingCoffee(do(a, s)) \equiv (\exists t)a = pickupCoffee(t) \vee holdingCoffee(s) \wedge \neg(\exists person, t)a = giveCoffee(person, t).$$

Initial Situation

Unique names axioms stating that the following terms are pairwise unequal:

$$Sue,\ Mary,\ Bill,\ Joe,\ CM,\ office(Sue),\ office(Mary),\ office(Bill),\ office(Joe).$$

Initial Fluent values:

$$robotLocation(S_0) = CM, \quad start(S_0) = 0, \quad \neg(\exists p)hasCoffee(p, S_0), \neg holdingCoffee(S_0), \quad \neg(\exists l, l')going(l, l', S_0).$$

Coffee delivery preferences. The following expresses that $(Sue, 140, 160), \ldots, (Joe, 90, 100)$ are all, and only, the tuples in the *wantsCoffee* relation.

$$\begin{aligned} wantsCoffee(p, t_1, t_2) \equiv\ & p = Sue \wedge t_1 = 140 \wedge t_2 = 160 \vee \\ & p = Mary \wedge t_1 = 130 \wedge t_2 = 170 \vee \\ & p = Bill \wedge t_1 = 100 \wedge t_2 = 110 \vee \\ & p = Joe \wedge t_1 = 90 \wedge t_2 = 100. \end{aligned}$$

Robot travel times:

$$\begin{aligned} & travelTime(CM, office(Sue)) = 15, \\ & travelTime(CM, office(Mary)) = 10, \\ & travelTime(CM, office(Bill)) = 8, \\ & travelTime(CM, office(Joe)) = 10. \\ & travelTime(l, l') = travelTime(l', l), \\ & travelTime(l, l) = 0. \end{aligned}$$

Action Occurrence Times:

$$\begin{aligned} & time(pickupCoffee(t)) = t, \\ & time(giveCoffee(person, t)) = t, \\ & time(startGo(loc_1, loc_2, t)) = t, \\ & time(endGo(loc_1, loc_2, t)) = t. \end{aligned}$$

GOLOG Procedures

proc $deliverCoffee(t)$ % Beginning at time t
% the robot serves coffee to everyone,
% if possible. Else the program fails.
$now \leq t?$;
$\{[(\forall p, t', t'').wantsCoffee(p, t', t'') \supset hasCoffee(p)]?$
$\mid$
if $robotLocation = CM$ **then** $deliverOneCoffee(t)$
else $goto(CM, t)$; $deliverOneCoffee(now)$
endIf$\}$
endProc

The above procedure introduces a functional fluent $now(s)$, which is identical to the fluent $start(s)$. We use it instead of *start* because it has a certain mnemonic value, but, like *start*, it denotes the *current time*.

proc $deliverOneCoffee(t)$ % Assuming the robot
% is at the coffee machine,
% it delivers one cup of coffee.
$(\pi p, t_1, t_2, wait)[\{wantsCoffee(p, t_1, t_2) \wedge \neg hasCoffee(p) \wedge wait \geq 0 \wedge t_1 \leq t + wait + travelTime(CM, office(p)) \leq t_2\}?$;
$pickupCoffee(t + wait)$;
$goto(office(p), now)$;
$giveCoffee(p, now)$;
$deliverCoffee(now)]$
endProc

proc $goto(loc, t)$ % Beginning at time t the
% robot goes to loc.
$goBetween(robotLocation, loc, travelTime(robotLocation, loc), t)$
endProc

proc $goBetween(loc1, loc2, \Delta, t)$ % Beginning at
% time t the robot goes from loc1 to loc2,
% taking Δ time units for the transition.
$startGo(loc1, loc2, t)$; $endGo(loc1, loc2, t + \Delta)$
endProc

The following sequential temporal GOLOG program implements the above specification.

Sequential Temporal GOLOG Program for a Coffee Delivery Robot

```
/* GOLOG Procedures */

proc(deliverCoffee(T),
  ?(some(t, now(t) & t $<= T)) :
  (?(all(p,all(t1,all(t2,wantsCoffee(p,t1,t2) =>
                          hasCoffee(p)))))
   #
   pi(rloc, ?(robotLocation(rloc)) :
          if(rloc = cm,
```

```
                 /* THEN */
                    deliverOneCoffee(T),
                 /* ELSE */
                    goto(cm,T) : pi(t, ?(now(t)) :
                      deliverOneCoffee(t)))))).

proc(deliverOneCoffee(T),
  pi(p, pi(t1, pi(t2, pi(wait, pi(travTime,
     ?(wantsCoffee(p,t1,t2) & -hasCoffee(p) &
       wait $>= 0 &
       travelTime(cm,office(p),travTime) &
       t1 $<= T + wait + travTime & T + wait +
              travTime $<= t2) :
     pi(t, ?(t $= T + wait) : pickupCoffee(t)) :
     pi(t, ?(now(t)) : goto(office(p),t)) :
     pi(t, ?(now(t)) : giveCoffee(p,t)) :
     pi(t, ?(now(t)) : deliverCoffee(t)))))))).


proc(goto(L,T),
  pi(rloc,?(robotLocation(rloc)) :
    pi(deltat,?(travelTime(rloc,L,deltat)) :
      goBetween(rloc,L,deltat,T)))).

proc(goBetween(Loc1,Loc2,Delta,T),
   startGo(Loc1,Loc2,T) :
   pi(t, ?(t $= T + Delta) : endGo(Loc1,Loc2,t))).

/* Preconditions for Primitive Actions */

poss(pickupCoffee(T),S) :- not holdingCoffee(S),
                           robotLocation(cm,S).

poss(giveCoffee(Person,T),S) :- holdingCoffee(S),
                 robotLocation(office(Person),S).

poss(startGo(Loc1,Loc2,T),S) :- not going(L,LL,S),
           not Loc1 = Loc2, robotLocation(Loc1,S).

poss(endGo(Loc1,Loc2,T),S) :- going(Loc1,Loc2,S).

/* Successor State Axioms */

hasCoffee(Person,do(A,S)) :-
  A = giveCoffee(Person,T) ; hasCoffee(Person,S).

robotLocation(Loc,do(A,S)) :- A=endGo(Loc1,Loc,T) ;
  robotLocation(Loc,S), not A=endGo(Loc2,Loc3,T).

going(Loc1,Loc2,do(A,S)) :- A=startGo(Loc1,Loc2,T) ;
   going(Loc1,Loc2,S), not A = endGo(Loc1,Loc2,T).

holdingCoffee(do(A,S)) :- A = pickupCoffee(T) ;
   holdingCoffee(S), not A = giveCoffee(Person,T).

/* Initial Situation */

robotLocation(cm,s0).   start(s0,0).
wantsCoffee(sue,140,160).
wantsCoffee(bill,100,110).
wantsCoffee(joe,90,100).
wantsCoffee(mary,130,170).
travelTime0(cm,office(sue),15).
travelTime0(cm,office(mary),10).
travelTime0(cm,office(bill),8).
travelTime0(cm,office(joe),10).
travelTime(L,L,0).
travelTime(L1,L2,T) :- travelTime0(L1,L2,T) ;
                       travelTime0(L2,L1,T).

/* Action occurrence time is its last argument. */

time(pickupCoffee(T),T).
time(endGo(Loc1,Loc2,T),T).
time(giveCoffee(Person,T),T).
time(startGo(Loc1,Loc2,T),T).

/* Restore situation arguments to fluents. */

restoreSitArg(robotLocation(Rloc),S,
              robotLocation(Rloc,S)).
restoreSitArg(hasCoffee(Person),S,
              hasCoffee(Person,S)).
restoreSitArg(going(Loc1,Loc2),S,
              going(Loc1,Loc2,S)).
restoreSitArg(holdingCoffee,S,
              holdingCoffee(S)).

/* Primitive Action Declarations */

primitive_action(pickupCoffee(T)).
primitive_action(giveCoffee(Person,T)).
primitive_action(startGo(Loc1,Loc2,T)).
primitive_action(endGo(Loc1,Loc2,T)).

/* Fix on a solution to the temporal
   constraints. */

chooseTimes(s0).
chooseTimes(do(A,S)) :- chooseTimes(S),
                        time(A,T), rmin(T).

/* "now" is a synonym for "start".  */

now(S,T) :- start(S,T).
restoreSitArg(now(T),S,now(S,T)).
```

A problem with our constraint logic programming approach to coffee delivery is that the execution of the GOLOG call `do(deliverCoffee(1),s0,S)` will not, in general, result in a fully instantiated sequence of actions `S`. The actions in that sequence will not have their occurrence times uniquely determined; rather, these occurrence times will consist of all feasible solutions to the system of constraints generated by the program execution. So, to get a fixed schedule of coffee delivery, we must determine one or more of these feasible solutions. The relation `chooseTimes(S)` in the above program does just that. Beginning with the first action in the situation history, `S`, `chooseTimes` determines the time of that action (which, in general, will be a Prolog variable since the ECLIPSE constraint solver will not have determined a unique value for that action's occurrence time). It then minimizes (via `rmin(T)`) that time, relative to the current set of temporal constraints generated by executing the coffee delivery program. Then, having fixed the occurrence time of the first action, it repeats with the second action, etc. In this way, `chooseTimes` selects a particular solution to the linear temporal constraints generated by the program, thereby producing one of many possible schedules for the robot. Of course, there is nothing special about `chooseTimes`; any method for obtaining one or more solutions to the constraints would do just as well.

The following is the output obtained from this program under the temporal GOLOG interpreter of Section 6. We use the relation `chooseTimes(S)` to select a solution to the temporal constraints.

```
[eclipse 2]: do(deliverCoffee(1),s0,S),
             chooseTimes(S).

S = do(giveCoffee(mary,165),do(endGo(cm,
office(mary),165),do(startGo(cm,office(mary),155),
do(pickupCoffee(155),do(endGo(office(sue),cm,155),
do(startGo(office(sue), cm, 140), do(giveCoffee(
sue,140),do(endGo(cm,office(sue),140),do(startGo(
cm,office(sue),125),do(pickupCoffee(125),do(
endGo(office(bill),cm,116),do(startGo(office(bill),
cm,108),do(giveCoffee(bill,108),do(endGo(cm,
office(bill),108),do(startGo(cm,office(bill),100),
do(pickupCoffee(100),do(endGo(office(joe),cm,100),
do(startGo(office(joe),cm,90),do(giveCoffee(joe,
90),do(endGo(cm,office(joe),90),do(startGo(cm,
office(joe),80),do(pickupCoffee(80), s0))))))))))
))))))))))     More? (;)
```

As is usual with GOLOG applications, the above computation would be done off line; as yet, the robot has not performed any physical actions in the world. To have it physically deliver coffee, the action sequence in the above execution trace would be passed to the robot's primitive execution module.

6.2 A Singing Robot

To simplify the exposition, we did not endow the above program with any interleaving execution of processes, as described in Section 4. This would, however, be easy to do. Suppose we wanted our robot to sing a song, but only while it is in transit between locations. Introduce two instantaneous actions $startSing(t)$ and $endSing(t)$, and a process fluent $singing(s)$, with action precondition and successor state axioms:

$$Poss(startSing(t), s) \equiv \neg singing(s).$$

$$Poss(endSing(t), s) \equiv singing(s),$$

$$singing(do(a, s)) \equiv (\exists t)a = startSing(t) \vee \\ singing(s) \wedge \neg(\exists t)a = endSing(t).$$

Then the following version of the GOLOG procedure $goBetween$ turns the robot into a singing waiter:

proc $goBetween(loc1, loc2, \Delta, t)$
  $startGo(loc1, loc2, t)$;
  $startSing(t)$; $endSing(t + \Delta)$;
  $endGo(loc1, loc2, t + \Delta)$
endProc

This provides a temporal, interleaving account of the concurrent execution of two processes: singing and moving between locations.

7 Conclusions and Future Work

We have extended the ontology and foundational axioms of the sequential situation calculus to include time, and showed how one can view actions with durations as processes that are initiated and terminated by instantaneous actions. This conceptual shift, when coupled with an explicit representation for time, provides a rich account of interleaving concurrency in the situation calculus. Based upon the axioms for the sequential, temporal situation calculus, we extended the semantics and interpreter for the situation calculus-based programming language GOLOG to the temporal domain. Finally, we illustrated the resulting increased functionality of the language with a GOLOG program describing the temporal behavior of a coffee delivery robot, including the concurrent processes of delivering coffee while singing a song.

CONGOLOG [2] is a much richer language than GOLOG, that includes facilities for interleaving concurrent execution, prioritized interrupts and exogenous actions. Augmenting CONGOLOG with a temporal component is a straightforward exercise, and can be done by changing its semantics and implementation in a manner exactly parallel to the approach of Section 6 for GOLOG.

Ongoing work along the lines of this paper includes controlling an RWI B21 autonomous robot to perform temporal scheduling tasks in an office environment. In this setting, it would be unrealistic to expect the robot to execute a schedule like that returned by our coffee delivery program. Frequently, it will be impossible to meet the exact times in such a schedule, for example, if the robot is unexpectedly delayed in traveling to the coffee machine. One approach we are exploring is to have the robot monitor its own execution, using the situation calculus-based execution monitor of [3], recomputing what remains of the schedule after it has determined (by sensing its internal clock) the actual occurrence times of its actions. We do not instantiate a schedule's action occurrence times (as we did using `chooseTimes(S)`), but leave these free, subject to the constraints generated by the GOLOG program. Whenever the robot physically performs an action, it senses the action's actual occurrence time, adds this to the constraints, then computes a remaining schedule, or fails if no continuing schedule can be found.

Acknowledgements:

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada, the Institute for Robotics and Intelligent Systems of the Government of Canada, and the Information Technology

Research Centre of the Government of Ontario. Mikhail Soutchanski provided valuable feedback and corrections for an earlier version of this paper.

References

[1] M. Gelfond, V. Lifschitz, and A. Rabinov. What are the limitations of the situation calculus? In *Working Notes, AAAI Spring Symposium Series on the Logical Formalization of Commonsense Reasoning*, pages 59–69, 1991.

[2] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, Nagoya, Japan, 1997.

[3] G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixth International Conference (KR'98)*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.

[4] S. Hanks and D. McDermott. Default reasoning, nonmonotonic logics, and the frame problem. In *Proceedings of the National Conference on Artificial Intelligence (AAAI'86)*, pages 328–333, 1986.

[5] M. Jenkin, Y. Lespérance, H.J. Levesque, F. Lin, J. Lloyd, D. Marcu, R. Reiter, R.B. Scherl, and K. Tam. A logical approach to portable high-level robot programming. In *Proceedings of the Tenth Australian Joint Conference on Artificial Intelligence (AI'97)*, Perth, Australia, 1997. Invited paper.

[6] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. Foundations of a logical approach to agent programming. In M. Wooldridge, J.P. Muller, and M. Tambe, editors, *Intelligent Agents Vol. II – Proc. 1995 Workshop on Agent Theories, Architectures, and Languages (ATAL-95)*, pages 331–346. Springer-Verlag, Lecture Notes in Art. Intell., 1996.

[7] Y. Lespérance, H.J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high-level robot programming – a progress report. In *Control of the Physical World by Intelligent Systems, Working Notes of the 1994 AAAI Fall Symp.*, 1994.

[8] Y. Lespérance, H.J. Levesque, and R. Reiter. A situation calculus approach to modeling and programming agents. In A. Rao and M. Wooldridge, editors, *Foundations and Theories of Rational Agency*, 1997. In press.

[9] H. L. Levesque and R. Reiter. High-level robotic control: beyond planning. Position paper. AAAI 1998 Spring Symposium: Integrating Robotics Research: Taking the Next Big Leap. Stanford University, March 23-25, 1998. http://www.cs.toronto.edu/˜cogrobo/.

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

[11] H.J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: a logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[12] V. Lifschitz. Toward a metatheory of action. In J. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 376–386, Los Altos, CA, 1991. Morgan Kaufmann Publishers, San Francisco, CA.

[13] F. Lin and R. Reiter. State constraints revisited. *J. of Logic and Computation, special issue on actions and processes*, 4:655–678, 1994.

[14] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

[15] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[16] Sheila A. McIlraith. *Towards a Formal Account of Diagnostic Problem Solving.* PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1997.

[17] J.A. Pinto. *Temporal Reasoning in the Situation Calculus.* PhD thesis, University of Toronto, Department of Computer Science, 1994.

[18] Javier Pinto. Occurrences and Narratives as Constraints in the Branching Structure of the Situation Calculus. Submitted to the Journal of Logic and Computation
URL = ftp://lyrcc.ing.puc.cl/pub/jpinto/jlc.ps.gz.

[19] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. 1998. Submitted for publication.
http://www.cs.toronto.edu/˜cogrobo/.

[20] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems.* In preparation. Draft available at http://www.cs.toronto.edu/˜cogrobo/.

[21] R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, San Diego, CA, 1991.

[22] R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.

[23] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2–13. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[24] E. Ternovskaia. Interval situation calculus. In *Proc. of ECAI'94 Workshop W5 on Logic and Change*, pages 153–164, Amsterdam, August 8-12, 1994.