1

Chronolog: It's about Time for Golog

GIUSEPPE DE GIACOMO AND MAURICE PAGNUCCO

ABSTRACT. In this paper we introduce a notion of discrete time into the cognitive robotics languages of the Golog family. Our variant of Golog, named *Chronolog*, is achieved in a rather straightforward manner but we show that it allows one to express a rich variety of temporal concepts. It is based on exogenous `tick` actions generated by an external "clock" and a special functional fluent to keep track of the passing of time. Moreover, we consider on-line and off-line versions of our proposal as well as real-time and reactive variants and the ability to deal with exogenous actions.

1 Introduction

In this paper we introduce a notion of discrete time into the cognitive robotics language Golog [Levesque, Reiter, Lespérance, Lin, and Scherl 1997; Reiter 2001]. The main advantages of our approach are that it is relatively straightforward yet novel and surprisingly expressive. We consider that an external "clock" will generate discrete clock "ticks" and that their arrival will be kept track of by a special functional fluent for time.

The introduction of time into Golog and the situation calculus is not new. It has already been considered by several authors. Grosskreutz and Lakemeyer in describing their Golog variant cc-Golog [Grosskreutz and Lakemeyer 2003] suggest the use of an explicit clock with discrete clock tick actions but object to it due to the problem of determining the correct granularity of clock ticks. They also object to this on the basis that the execution traces would be "glutted with irrelevant 'clock tick' actions". Gans *et al.* [2003] also use the notion of clock ticks in Golog but do not develop the idea formally. Reiter [1996, 1998, 2001] considers instantaneous actions with a temporal argument $A(\bar{x}, t)$ denoting that the action occurred at time t. He is able to deal with continuous actions, concurrent actions and actions with duration by allowing processes (or events) to have a start action that initiates the process and an end action that terminates it. This approach to actions with duration is based on one developed by Pinto [1997]. Davis [1990] also discusses time in the situation calculus and draws his account mainly on the work of McDermott [1982]. Here a fluent *clock_time* associates a time with situations. Situations can also belong to intervals (sets of situations belonging to the interval). This allows for continuous actions and branching time.

In this paper, we give a nice and simple framework to deal with discrete time, under certain assumptions. We consider our agent's behavior to be controlled by (variants of) Golog programs, and we consider the discrete passing of time (i.e., clock ticks) to be outside the control of the agent. We then make the simplifying assumption that program execution is orders of magnitudes quicker than the external ticking. And we equip agents' programs with simple abilities for testing time flow and synchronize with time ticks when needed. In this setting we show that we can actually go surprisingly far in modeling sophisticated time related properties within programs.

Again we stress that the simplicity and the effectiveness of the approach rely strongly on the assumption that the computation of the agent is so fast that it can execute any amount of computation in between time ticks. This is obviously an assumption that we cannot hope to hold in every context. However there are contexts where it can be thought of as a good approximation of reality. Obvious examples are robots that perform physical movements directed by a high level control program.

The remainder of the paper is organized as follows. In the next section we provide some background to the situation calculus and the cognitive robotics language Golog and some of its variants. In Section 3 we introduce our notion of time and the axioms required to capture it while in Section 4 we introduce some additional syntactic constructs to enhance readability. Section 4 also illustrates the expressiveness of our approach with a list of temporal notions that are captured. Section 5 discusses the execution of Chronolog programs. We provide some examples in Section 6. In Section 7 we show the modifications required of the IndiGolog [De Giacomo and Levesque 1999] interpreter to implement Chronolog and in Section 8 we show how to implement a timed search construct.

2 Preliminaries

The basis for Golog and its variants is provided by the situation calculus [McCarthy 1963; McCarthy and Hayes 1969; Reiter 2001]. We will not go over the language in detail here except to note the following components: there is a special constant S_0 used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol do where $do(a, s)$ denotes the situation resulting from performing the action a at situation s; relations whose truth values vary from situation to situation, are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument; and, there is a special predicate $Poss(a, s)$ used to state that action a is executable in situation s.

An action theory of the following form is a common scenario [Reiter 2001]:

- Axioms describing the initial situation, S_0;
- Action precondition axioms, one for each primitive action a, that are used to characterize $Poss(a, s)$—when it is possible to perform action a at situation s;
- Successor state axioms, one for each fluent F, stating the conditions under which $F(\vec{x}, do(a, s))$ holds as a function of what holds in situation s; these take the place of effect axioms, but also provide a solution to the frame problem;
- Unique names axioms for the primitive actions; and,
- Some foundational, domain independent axioms [Levesque and Lakemeyer 2000; Reiter 2001].

Next we turn to programs. The programs we consider here are based on the ConGolog language defined in [De Giacomo, Lespérance, and Levesque 2000], which provides a rich set of programming constructs summarized below:

α	primitive action
ϕ	wait for a condition
$\delta_1; \delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic branch

$\pi\, x.\, \delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	while loop
$\delta_1 \parallel \delta_2$	concurrency with equal priority
$\delta_1 \rangle\!\rangle\, \delta_2$	concurrency with δ_1 at a higher priority
$\delta^{\parallel}$	concurrent iteration
$\langle\ \phi \rightarrow \delta\ \rangle$	interrupt
$p(\vec{\theta})$	procedure call[1]

Among these constructs, we notice the presence of nondeterministic constructs. These include $(\delta_1 \mid \delta_2)$, which nondeterministically chooses between programs δ_1 and δ_2, $\pi\, x.\, \delta$, which nondeterministically picks a binding for the variable x and performs the program δ for this binding of x, and δ^*, which performs δ zero or more times. It should be noted that these forms of nondeterminism represent "reasoned" choices; during execution any choices to be made are carried out on the basis of what will guarantee termination of the program. Also notice that ConGolog includes constructs for dealing with concurrency. In particular $(\delta_1 \parallel \delta_2)$ expresses the concurrent execution (interpreted as interleaving) of the programs δ_1 and δ_2. Observe that a program may become blocked when it reaches a primitive action whose preconditions are false or a wait action $\phi?$ whose condition ϕ is false. Then, execution of $(\delta_1 \parallel \delta_2)$ may continue provided another program executes next. Beside $(\delta_1 \parallel \delta_2)$ ConGolog includes other constructs for dealing with concurrency, such as prioritized concurrency $(\delta_1 \rangle\!\rangle\, \delta_2)$, and interrupts $\langle \phi \rightarrow \delta\ \rangle$. In $(\delta_1 \rangle\!\rangle\, \delta_2)$, δ_1 has higher priority than δ_2, and δ_2 may only execute when δ_1 is done or blocked. $\delta^{\parallel}$ is like nondeterministic iteration δ^*, but the instances of δ are executed concurrently rather than in sequence. Finally, an interrupt $<\ \vec{x} :\ \phi \rightarrow \delta\ >$ has variables $\vec{x}$, a trigger condition ϕ, and a body δ. If the interrupt gets control from higher priority processes and the condition ϕ is true for some binding of the variables, the interrupt triggers and the body is executed with the variables taking these values. Once the body completes execution, the interrupt may trigger again. We refer the reader to [De Giacomo, Lespérance, and Levesque 2000] for a detailed account of ConGolog.

In [De Giacomo, Lespérance, and Levesque 2000], a single step transition semantics in the style of [Plotkin 1981] is defined for ConGolog programs. Two special predicates $Trans$ and $Final$ are introduced. $Trans(p, s, p', s')$ means that by executing program p starting in situation s, one can get to situation s' in one elementary step with the program p' remaining to be executed, that is, there is a possible transition from the configuration (p, s) to the configuration (p', s'). $Final(p, s)$ means that program p may successfully terminate in situation s, i.e., the configuration (p, s) is final.[2]

Offline executions of programs, which are the kind of executions originally proposed for Golog and ConGolog [Levesque, Reiter, Lespérance, Lin, and Scherl 1997; De Giacomo,

[1]For the sake of simplicity, we will not consider procedures in this paper.

[2]For example, the transition requirements for sequence are

$$\begin{aligned} &Trans([p_1; p_2], s, p', s') \equiv \\ &\quad Final(p_1, s) \wedge Trans(p_2, s, p', s') \vee \\ &\quad\ \ \exists q'.\, Trans(p_1, s, q', s') \wedge p' = (q'; p_2) \end{aligned}$$

i.e., to single-step the program $(p_1; p_2)$, either p_1 terminates and we single-step p_2, or we single-step p_1 leaving some q', and $(q'; p_2)$ is what is left of the sequence.

Lespérance, and Levesque 2000], are characterized using the $Do(p, s, s')$ predicate, which means that there is an execution of program p that starts in situation s and terminates in situation s':

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

where $Trans^*$ is the reflexive transitive closure of $Trans$. An offline execution of program p from situation s is a sequence of actions $a_1, \ldots, a_n$ such that:

$$Axioms \models Do(p, s, do(a_n, \ldots, do(a_1, s))).$$

Observe that an offline executor is in fact similar to a planner that given a program, a starting situation, and a theory describing the domain, produces a sequence of actions to execute in the environment. In doing this, it has no access to sensing results, which will only be available at runtime. See [De Giacomo, Lespérance, and Levesque 2000] for more details.

In [De Giacomo and Levesque 1999], IndiGolog, an extension of ConGolog that deals with online executions with sensing is developed. The semantics defines an *online execution* of an IndiGolog program p starting from a history σ, as a sequence of *online configurations* $(p_0 = p, \sigma_0 = \sigma), \ldots, (p_n, \sigma_n)$ such that for $i = 0, \ldots, n-1$:

$$\begin{aligned} &Axioms \cup \{Sensed[\sigma_i]\} \models \\ &\qquad\qquad Trans(p_i, end[\sigma_i], p_{i+1}, end[\sigma_{i+1}]), \end{aligned}$$

$$\sigma_{i+1} = \begin{cases} \sigma_i & \text{if} \quad end[\sigma_{i+1}] = end[\sigma_i], \\ \sigma_i \cdot (a, x) & \text{if} \quad end[\sigma_{i+1}] = do(a, end[\sigma_i]) \\ & \qquad \text{and } a \text{ returns } x. \end{cases}$$

An *online execution successfully terminates* if

$$Axioms \cup \{Sensed[\sigma_n]\} \models Final(p_n, end[\sigma_n]).$$

There is no automatic lookahead in IndiGolog. Instead, a *search* operator $\Sigma(p)$ is introduced to allow the programmer to specify when lookahead should be performed [De Giacomo and Levesque 1999; De Giacomo, Lespérance, Levesque, and Sardina 2002].

3 Time in our Framework

We presume the existence of an external "clock" that generates exogenous `tick` actions. All actions, including `tick`, are instantaneous however and have no duration. While an action may occur after a certain number of `tick`s t have occurred and so can be considered to have occurred at t, it takes no time for the action itself to be performed. We shall discuss how actions with duration may be handled later. No further assumptions are made about the clock, or the regularity of `tick` actions, etc.

Chronolog introduces the special functional fluent *Time* and exogenous action `tick`. In the initial situation, *Time* has the value 0 and is incremented by one each time an exogenous `tick` occurs. In other words, *Time* serves to keep track of the number of exogenous `tick`s that have occurred. Given these assumptions we can proceed as follows.
The initial state axiom for time:

$$Time(S_0) = 0$$

The preconditions for `tick` are:

$$Poss(\texttt{tick},\ s) \equiv true$$

The successor state axiom for *Time* is:

$$\begin{array}{l} Time(do(a,\ s)) = t \equiv \\ \quad a = \texttt{tick} \ \wedge\ Time(s) = t - 1 \vee \\ \quad a \neq \texttt{tick} \ \wedge\ Time(s) = t \end{array}$$

Observe that these axioms conform to the requirements for Reiter's Action Theories and hence we can use regression and, more generally, all results on Basic Action Theories apply to them as well.

These axioms can be added to the standard Golog, ConGolog and IndiGolog axioms in a straightforward way as described in Section 2. The repercussions of doing so, we shall return to shortly. First however, it will be convenient to introduce some additional syntactic constructs that will simplify our presentation. These will be defined in terms of the notions introduced above.

4 Synchronization

Now we turn to *Chronolog* programs. These are simply IndiGolog programs that *access the time ticking* by testing the fluent $Time(s)$. However, here we make the fundamental assumption that no time ticks occur during the execution of a Chronolog program, unless it explicitly *synchronizes* with time. In practice we are requiring that Chronolog programs can be executed order of magnitudes quicker than time ticking.

Let us look at how synchronization can be performed in a Chronolog program. The most basic synchronization facility is to wait for the passing of a certain number of clock ticks. We introduce the construct $wait(t)$ where t is an integer number of `tick`s and define it as follows.

$$wait(t) \equiv \pi t'.[(Time = t')?;\ (Time = t' + t)?]$$

or, equivalently,

$$wait(t) \equiv \pi t'.[(t' = Time + t)?;\ (Time = t')?]$$

In other words, $wait(t)$ in a Chronolog program causes the program to wait for t exogenous `tick` actions to occur.

The definitions above make use of Chronolog's blocking wait construct $?$. The first instance is used to determine the current time (the number of clock ticks that have occurred since the system was started in the initial situation S_0). The second instance causes the Golog program to block until an additional t exogenous `tick` actions have occurred. It can be easily seen that $wait(1)$ can be used to synchronize with the next clock tick. $wait(0)$ essentially does nothing.

Notice that here the assumption that Chronolog execution is much quicker than time ticking plays an important role. If this was not the case, and an exogenous tick occurs after the execution of the first blocking wait construct $?$ of a $wait(0)$ but before the second, the program will block forever and never terminate.

Other basic synchronization constructs that can be immediately expressed in Chronolog as well include the following.

wait until time t:
$(Time = t)?$

wait until after time t:
$\pi u.[(u \geq t)?;\ (Time = u)]$

wait no more than time t:
$\pi u.[(u \leq t)?;\ (Time = u)]$

Using these notions it turns out that we can express a surprisingly rich set of constructs. To illustrate the range of possibilities we list a number of useful temporal concepts that our approach can provide. To simplify notation we make the assumptions that unless otherwise stated $t \geq Time$ and also, $t \leq t'$.

perform action a at t:
$\pi u.[(u = t - Time)?; wait(u); a]$

perform action a before t:
$\pi u.[(u \leq t - Time)?; wait(u); a]$

perform action a after t:
$\pi u.[(u \geq t - Time)?; wait(u); a]$

In the following constructs we consider actions with duration. $start\_a$ indicates the initiation of the action and end_a indicates its termination. We consider actions with duration further in the discussion.

perform action a between t and t':
$\pi u.[(u = t - Time)?; wait(u); start\_a; wait(t' - t); end\_a]$

perform action a for time t:
$start\_a;\ wait(t);\ end\_a$

perform action a for at least time t:
$\pi u.(u \geq t)?;\ start\_a;\ wait(u);\ end\_a$

perform action a for no more than time t:
$\pi u.(u \leq t)?;\ start\_a;\ wait(u);\ end\_a$

perform action a at t for time t':
$(Time = t)?; start\_a;\ wait(t');\ end\_a$

perform action a at t for at least time t':
$(Time = t)?; \pi u.(u \geq t'); start\_a;\ wait(u);\ end\_a$

perform action a at t for no more than time t':
$(Time = t)?; \pi u.(u \leq t'); start\_a;\ wait(u);\ end\_a$

perform action a start before t for time t':
$\pi u.(u \leq t - Time)?;\ wait(u);\ start\_a;\ wait(t);\ end\_a$

perform action a start before t for at least time t':
$\pi u.(u \leq t - Time)?;\ wait(u);\ \pi u'.(u' \geq t')?;\ start\_a;$
$wait(u');\ end\_a$

perform action a start before t for no more than t':
$\pi u.(u \leq t - Time)?;\ wait(u);\ \pi u'.(u' \leq t')?;\ start\_a;$
$wait(u');\ end\_a$

perform action a start after t for time t':
$\pi u.(u \geq t - Time)?;\ wait(u);\ start\_a;\ wait(t);\ end\_a$

perform action a start after t for at least time t':
$\pi u.(u \geq t - Time)?;\ wait(u);\ \pi u'.(u' \geq t')?;\ start\_a;$
$wait(u');\ end\_a$

perform action a start after t for no more than time t':
$\pi u.(u \geq t - Time)?;\ wait(u);\ \pi u'.(u' \leq t')?;\ start\_a;$
$wait(u');\ end\_a$

start action a before t and complete it at t'
$\pi u.(u \leq t - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' = t' - Time)?;\ wait(u');\ end\_a$

start action a before t and complete it before t'
$\pi u.(u \leq t - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' \leq t' - Time \wedge u' \geq u)?;\ wait(u');\ end\_a$

start action a before t and complete it after t'
$\pi u.(u \leq t - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' \geq t' - Time)?;\ wait(u');\ end\_a$

start action a after t and complete it before t'
$\pi u.(u \geq t - Time \wedge u \leq t' - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' \leq t' - Time \wedge u' \geq u)?;\ wait(u');\ end\_a$

start action a after t and complete it after t'
$\pi u.(u \geq t - Time \wedge u \leq t' - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' \geq t' - Time \wedge u' \geq u)?;\ wait(u');\ end\_a$

start action a at t and complete it after t'
$\pi u.(u = t - Time)?;\ wait(u);\ start\_a;$
$\pi u'.(u' \geq t' - Time)?;\ wait(u');\ end\_a$

4.1 Triggers and Exceptions

By using ConGolog's interrupt construct, we can arrange for certain actions to occur at a particular time or on a regular basis while executing a program (denoted by δ here).

trigger action a at time t:
$\langle (Time = t) \rightarrow a \rangle\ \rangle\delta$

trigger action a at every t ticks:
$\langle ((Time \% t) = 0) \rightarrow a \rangle\ \rangle\delta$

Where $\%$ is the modulus operator.

5 Chronolog execution

To understand the semantics of a Chronolog program, one has to keep in mind that a Chronolog program is not executable in isolation in general, but needs to be executed concurrently with a process emitting ticks in a continuous way. Also the ticking should be slow enough to allow for complete execution of the parts of a Chronolog program that do not include synchronization with time.

Observe that even if our program does not contain concurrency explicitly, in order to understand the semantics we need to resort to concurrency. Hence the semantics, even for the offline version, must be single-step. We use notions from ConGolog [De Giacomo, Lespérance, and Levesque 2000] to discuss the execution of Chronolog programs.

The next question to settle is how we model the ticking process. The ticking must go on forever in principle, but in practice it is sufficient to have enough ticks to complete the execution of the ConGolog program: i.e., a finite but unbounded number of ticks. Hence we can render in ConGolog the ticking process simply as:

$$\texttt{tick}^*$$

Finally we must formalize the execution of a ConGolog program δ concurrently with the ticking process, and since we said that the Chronolog program executes freely unless a synchronization with the ticking is required, we can formalize the execution of the program δ together with the ticking process $\texttt{tick}^*$ as:

$$\delta \rangle\rangle \texttt{tick}^*$$

In this way if the Chronolog program can execute (make a transition) it does so, and only when it synchronizes with a tick (i.e., waits for a given tick) does it stop and allow the ticking to go on.

Observe that when we turn to the online semantics, we must pay special attention to the search operator. Indeed, we can retain IndiGolog's normal search $search(\delta)$ that will not take time synchronization into account. But can also introduce a "timed" search for parts of Chronolog programs that require time synchronization. This last variant of search can be defined on the basis of the original one as follows:

$$timed\_search(\delta) \equiv search(\delta \rangle\rangle \texttt{tick}^*)$$

That is, we search as normal but simulate the occurrence of $\texttt{tick}$s as required by the program.

6 Examples

In the following examples we shall omit the specification of action precondition axioms and successor state axioms for fluents as they do not add anything to what we're trying to demonstrate here. Furthermore, action and fluent names will give an indication of their purpose.

One common use for temporal notions is to arrange the scheduling of regular actions. For instance, consider the cron daemon under Unix. It's purpose is to schedule the regular execution of certain programs for effecting system maintenance tasks. As we have seen above when discussing triggers, this can be easily implemented in Chronolog through the use of ConGolog's interrupt mechanism. Here is an example of how we might schedule regular maintenance tasks in Chronolog.

$\langle(((Time\%3600) = 0) \rightarrow \ hourly\_task\rangle \ \rangle\rangle$
$\langle(((Time\%86400) = 0) \rightarrow daily\_task\rangle \ \rangle\rangle$
$\langle(((Time\%604800) = 0) \rightarrow weekly\_task\rangle \ \rangle\rangle$
$\langle(((Time\%2592000) = 0) \rightarrow monthly\_task\rangle \ \rangle\rangle$
$\langle true \rightarrow wait(1)\rangle$

Note that in this example we would need to use action preconditions to ensure that an action is only executed once each time the interrupt's condition is true. However, an advantage of our execution model is that more than one rule may fire in between clock ticks. For example, when the time is a multiple of 86400, we can be sure that the $hourly\_task$ will also be performed since 86400 is divisible by 3600. If we wanted to ensure that only one interrupt was triggered between any two clock ticks (and assuming no other exogenous actions), we could add a $wait(1)$ after all other actions that are to be executed in the body of the interrupt. Another thing to be noted in this example is that we have not attempted to model Unix' multitasking abilities.

Of course, it is also possible to use preconditions that involve time. For example, suppose you wish to serve coffee to three people given when they will be available (in this case, between when they arrive and when they leave). A robot r can serve a person x provided that the person is around and that they are thirsty. Once x is served they are not thirsty anymore. The precondition can be specified in Chronolog as follows.

$$\begin{aligned}&Poss(serve(x), s) \equiv \\ &\quad Time \leq arrive(x, s) \wedge Time \geq leave(x, s) \wedge thirsty(x, s))\end{aligned}$$

We omit the successor axiom, and just say that they model the fact that once a person has been served they are not thirsty anymore, while arrival and leaving time remain fixed (situation independent).

A possible initial situation could be described as follows:

$arrive(giuseppe, S_0) = 10, leave(giuseppe, S_0) = 20$
$arrive(maurice, S_0) = 15, leave(maurice, S_0) = 30$
$arrive(bob, S_0) = 1, leave(bob, S_0) = 50$
$thirsty(giuseppe, S_0), thirsty(maurice, S_0),$
$thirsty(bob, S_0)$

A Chronolog program to decide the ordering of the serving of those that want coffee is

$timed_search($
$\quad(\pi x.serve(x) \ \mid \ wait(1))^*; (\forall x.\neg thirsty(x))?$
$)$

This program will attempt to find a sequence of actions (a plan) to serve everyone while they are around. More complex variants where we take into account the time the robots need to serve someone can also be easily modeled.

7 On-line Real-Time Interpreter

Implementing Chronolog requires a fairly straightforward modification to the IndiGolog interpreter [De Giacomo and Levesque 1999] by modifying the `indigo(E, H)` predicate and adding the `exec_inst(E, H, E1, H1)` predicate. The `wait_for_a_tick(H1, H2)` predicate blocks until an exogenous `tick` action occurs. The `exec_inst` predicate executes as many primitive (instantaneous) actions as possible. In other words, the idea is to execute as much of the program as possible (since primitive actions in the situation calculus are considered to be instantaneous) and then check for the occurrence of exogenous `tick`s.

```
indigo(E, H) :-
    exec_inst(E, H, E1, H1),
    (final(E1, H1) ->
        true;
        (wait_for_a_tick(H1, H2), indigo(E1, H2)).

exec_inst(E, H, E, H) :-
    final(E, H).

exec_inst(E, H, E2, H2) :-
    trans(E, H, E1, H1), !,
    exec_inst(E1, H1, E2, H2).

exec_inst(E, H, E, H) :-
    not trans(E, H, _ , _),
    not final(E, H).
```

Modifying IndiGolog in this way means that the program essentially behaves in the following manner (where δ is a Chronolog program) $\delta\rangle\rangle\texttt{tick}^*$ as desired. Notice that the underlying transition semantics, in terms of $trans$ and $final$, is on-line.

8 Conclusions

We have introduced a notion of discrete time into the situation calculus and the cognitive robotics language Golog. Our Golog variant, called Chronolog, achieves this through the introduction of a special functional fluent $Time$ to keep track of the passage of time and exogenous `tick` actions assumed to be generated by an external clock. All we require is a small number of additional axioms, yet the resulting framework allows us to express a surprising number of temporal notions. We have also described the execution of a Chronolog program and introduced the notion of a timed search.

Several extensions that follow in the same spirit are possible. It would be possible to introduce a construct $wait(action)$ that waits until action a occurs and implement it in a similar way to $wait(t)$. Similarly we could introduce $wait$; wait for an exogenous action (any exogenous action) to occur.

Our introduction of time into Golog would allow one to easily deal with actions with duration. These would need to be "clipped" along the lines that Pinto [Pinto 1997] deals with such actions, so that if we have an action a, it will be clipped into $start\_a$ and end_a. See also the discussion regarding ConGolog [De Giacomo, Lespérance, and Levesque 2000]. Note that it may not be easy to split all actions in this way.

We may also want to introduce constructs into Golog to explicitly allow for such actions. For example:

$$Clipped(a,\ start\_a,\ end\_a)$$

means that action a can be clipped and has $start\_a$ as the initiation of action a and end_a as the termination of action a. Note that end_a can be specified as exogenous or not. In some cases we do not want an exogenous termination but want to explicitly terminate the action at a particular time. It might be interesting to investigate the possibility of allowing end_a to be both exogenous and primitive so that you can do things like: "perform action a and if it has not terminated before a particular condition is true, (explicitly) terminate the action."

Precondition axioms and effect axioms (equivalently, successor state axioms for fluents) would need to be supplied for these actions. What is the relationship between the preconditions and effect/successor state axioms of the three actions? Here is one possibility:

$$\begin{array}{l} Poss(a,\ s) \equiv Poss(start\_a,\ s) \\ Poss(end\_a,\ s) \equiv true \\ effects(do(a,\ s)) \equiv effects(do(end\_a,\ do(start\_a,\ s)) \end{array}$$

Moreover, supposing that we wish to perform a instantaneously, that would be the same as the action sequence $start\_a;\ end\_a$.

Acknowledgments

This paper is dedicated to Hector J. Levesque; a colleague, a mentor, an inspiration and, above all, a friend.

This work was completed while the second author was on Special Studies Program leave at the Università di Roma "La Sapienza".

References

Davis, E. [1990]. *Representations of Commonsense Knowledge*. San Francisco: Morgan Kaufmann.

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

De Giacomo, G., Y. Lespérance, H. J. Levesque, and S. Sardina [2002]. On the semantics of deliberation in indigolg – from theory to implementation. In *Proceedings of the 8th International Conference on the Principles of Knowledge Representation and Reasoning (KR'02)*, pp. 603–614. Morgan Kaufmann Publishers.

De Giacomo, G. and H. J. Levesque [1999]. An incremental interpreter for high-level programs with sensing. In H. J. Levesque and F. Pirri (Eds.), *Logical Foundations for Cognitive Agents*, pp. 86–102. Springer-Verlag.

Gans, G., G. Lakemeyer, M. Jarke, and T. Vits [2003]. Snet: A modeling and simulation environment for agent networks based on i* and congolog. In *Proceedings of the Conference on Advanced Information Systems Engineering (CAiSE'03)*.

Grosskreutz, H. and G. Lakemeyer [2003, March). ccgolog – a logical language dealing with continuous change. *Logic Journal of the IGPL 11*(2), 179–221.

Levesque, H. J. and G. Lakemeyer [2000]. *The Logic of Knowledge Bases*. Cambridge, Massachusetts: MIT Press.

Levesque, H. J., R. Reiter, Y. Lespérance, F. Lin, and R. Scherl [1997]. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31*, 59–84.

McCarthy, J. [1963]. Situations, actions, and causal laws. Technical report, Stanford University Artificial Intelligence Project.

McCarthy, J. and P. Hayes [1969]. Some philosophical problems from the standpoint of artificial intelligence. In D. Michie and B. Meltzer (Eds.), *Machine Intelligence 4*, pp. 463–502. University of Edinburgh Press.

McDermott, D. [1982]. A temporal logic for reasoning about processes and plans. *Cognitive Science 6*, 101–155.

Pinto, J. [1997]. Integrating discrete and continuous change in a logical framework. *Computational Intelligence 14*(1).

Plotkin, G. [1981]. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Dept., Aarhus University, Denmark.

Reiter, R. [1996]. Natural actions, concurrency and continuous change in the situation calculus. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 2–13.

Reiter, R. [1998]. Sequential, temporal golog. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*.

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