DIS La Sapienza — PhD Course

Autonomous Agents and Multiagent Systems

Lecture 4: High-Level Programming in the Situation Calculus: Golog and ConGolog

Yves Lespérance

Dept. of Computer Science & Engineering
York University
Toronto, Canada

Lecture Outline

Part 1: Syntax, Informal Semantics, Examples

Part 2: Formal Semantics

Part 3: Implementation

High-level Programming in the Situation Calculus — The Approach

Plan synthesis can be very hard; but often we can sketch what a good plan might look like.

Instead of planning, agent's task is *executing a high-level plan/program.*

But allow *nondeterministic* programs.

Then, can direct interpreter to *search* for a way to execute the program.

The Approach (cont.)

Can still do planning/deliberation.

Can also completely script agent behaviors when appropriate.

Can control nondeterminism/amount of search done.

Related to work on planning with domain specific search control information.

The Approach (cont.)

Programs are *high-level*.

Use primitive actions and test conditions that are *domain dependent.*

Programmer specifies preconditions and effects of primitive actions and what is known about initial situation in a logical theory, a *basic action theory* in the situation calculus.

Interpreter uses this in search/lookahead and in updating world model.

Golog [LRLLS97]

AlGOl in LOGic

Constructs:

α,	primitive action
ϕ?,	test a condition
$(\delta_1; \delta_2)$,	sequence
if ϕ **then** δ_1 **else** δ_2 **endIf**,	conditional
while ϕ **do** δ **endWhile**,	loop
proc $\beta(\vec{x})$ δ **endProc**,	procedure definition
$\beta(\vec{t})$,	procedure call
$(\delta_1 \mid \delta_2)$,	nondeterministic choice of action
$\pi\, \vec{x}\, [\delta]$,	nondeterministic choice of arguments
δ^*,	nondeterministic iteration

Golog Semantics

High-level program execution task is a special case of planning:

Program Execution: Given domain theory $\mathcal{D}$ and program δ, the execution task is to find a sequence of actions $\vec{a}$ such that:

$$\mathcal{D} \models Do(\delta, S_0, do(\vec{a}, S_0))$$

where $Do(\delta, s, s')$ means that program δ when executed starting in situation s has s' as a legal terminating situation.

Since Golog programs can be nondeterministic, may be several terminating situations s'.

Will see how Do can be defined later.

Nondeterminism

A nondeterministic program may have several possible executions. E.g.:

$$ndp_1 = (a \mid b); c$$

Assuming actions are always possible, we have:

$$Do(ndp_1, S_0, s) \equiv s = do([a, c], S_0) \vee s = do([b, c], S_0)$$

Above uses abbreviation $do([a_1, a_2, \ldots, a_{n-1}, a_n], s)$ meaning $do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, s))))$.

Interpreter searches all the way to a final situation of the program, and only then starts executing corresponding sequence of actions.

Nondeterminism (cont.)

When condition of a test action or action precondition is false, backtrack and try different nondeterministic choices. E.g.:

$$ndp_2 = (a \mid b); c; P?$$

If P is true initially, but becomes false iff a is performed, then

$$Do(ndp_2, S_0, s) \;\equiv\; s = do([b, c], S_0)$$

and interpreter will find it by backtracking.

Using Nondeterminism: A Simple Example

A program to clear blocks from table:

$$(\pi\, b\, [OnTable(b)?; putAway(b)])^*; \neg\exists b\, OnTable(b)?$$

Interpreter will find way to unstack all blocks ($putAway(b)$ is only possible if b is clear).

Example: Controlling an Elevator

Primitive actions: $up(n)$, $down(n)$, $turnoff(n)$, $open$, $close$.

Fluents: $floor(s) = n$, $on(n, s)$.

Fluent abbreviation: $next\_floor(n, s)$.

Action Precondition Axioms:

$$
\begin{array}{l}
Poss(up(n), s) \equiv floor(s) < n. \\
Poss(down(n), s) \equiv floor(s) > n. \\
Poss(open, s) \equiv True. \\
Poss(close, s) \equiv True. \\
Poss(turnoff(n), s) \equiv on(n, s). \\
Poss(no\_op, s) \equiv True.
\end{array}
$$

Elevator Example (cont.)

Successor State Axioms:

$$
\begin{array}{l}
floor(do(a, s)) = m \equiv \\
\quad a = up(m) \lor a = down(m) \lor \\
\quad floor(s) = m \land \neg \exists n\, a = up(n) \land \neg \exists n\, a = down(n).
\end{array}
$$

$$
\begin{array}{l}
on(m, do(a, s)) \equiv \\
\quad a = push(m) \lor on(m, s) \land a \neq turnoff(m).
\end{array}
$$

Fluent abbreviation:

$$
\begin{array}{l}
next\_floor(n, s) \stackrel{\text{def}}{=} on(n, s) \land \\
\quad \forall m. on(m, s) \supset |m - floor(s)| \geq |n - floor(s)|.
\end{array}
$$

Elevator Example (cont.)

Golog Procedures:

proc $serve(n)$
  $go\_floor(n); turnoff(n); open; close$
endProc

proc $go\_floor(n)$
  $[current\_floor = n? \mid up(n) \mid down(n)]$
endProc

proc $serve\_a\_floor$
  $\pi\, n\, [next\_floor(n)?; serve(n)]$
endProc

Elevator Example (cont.)

Golog Procedures (cont.):

proc $control$
  while $\exists n\, on(n)$ **do** $serve\_a\_floor$ **endWhile**;
  $park$
endProc

proc $park$
  if $current\_floor = 0$ **then** $open$
  else $down(0); open$
  endIf
endProc

Elevator Example (cont.)

Initial situation:

$$current\_floor(S_0) = 4, \;\; on(5, S_0), \;\; on(3, S_0).$$

Querying the theory:

$$Axioms \models \exists s\, Do(control, S_0, s).$$

Successful proof might return

$$s = do(open, do(down(0), do(close, do(open, do(turnoff(5), do(up(5), do(close, do(open, do(turnoff(3), do(down(3), S_0)))))))))).$$

Using Nondeterminism to Do Planning: A Mail Delivery Example

This control program searches to find a schedule/route that serves all clients and minimizes distance traveled:

proc $control$
  search($minimize\_distance$(0))
endProc

proc $minimize\_distance(distance)$
  $serve\_all\_clients\_within(distance)$
  | % or
  $minimize\_distance(distance + Increment)$
endProc

$mimimize\_distance$ does iterative deepening search.

A Control Program that Plans (cont.)

proc $serve\_all\_clients\_within(distance)$
  $\neg\exists c\ Client\_to\_serve(c)?$ % if no clients to serve, we're done
  $|$ % or
  $\pi c, d\ [(Client\_to\_serve(c) \wedge$ % choose a client
    $d = distance\_to(c) \wedge d \leq distance?)$;
   $go\_to(c)$; % and serve him
   $serve\_client(c)$;
   $serve\_all\_clients\_within(distance - d)]$
endProc

Concurrent Processes and ConGolog: Motivation

A key limitation of Golog is its lack of support for *concurrent processes*.

Can't program several agents within a single Golog program.

Can't specify an agent's behavior using concurrent processes. Inconvenient when you want to program *reactive* or *event-driven* behaviors.

ConGolog Motivation (cont.)

Address this by developing ConGolog (Concurrent Golog) which handles:

- concurrent processes with possibly different priorities,
- high-level interrupts,
- arbitrary exogenous actions.

Concurrency

We model concurrent processes as *interleavings* of the primitive actions in the component processes. E.g.:

$$cp_1 = (a; b) \parallel c$$

Assuming actions are always possible, we have:

$$Do(cp_1, S_0, s) \equiv \\ s = do([a, b, c], S_0) \vee s = do([a, c, b], S_0) \vee s = do([c, a, b], S_0)$$

Concurrency (cont.)

Important notion: process becoming *blocked*. Happens when a process δ reaches a primitive action whose preconditions are false or a test action ϕ? and ϕ is false.

Then execution need not fail as in Golog. May continue provided another process executes next. The process is blocked. E.g.:

$$cp_2 = (a; P?; b) \parallel c$$

If a makes P false, b does not affect it, and c makes it true, then we have

$$Do(cp_2, S_0, s) \equiv s = do([a, c, b], S_0).$$

Concurrency (cont.)

If no other process can execute, then backtrack. Interpreter still searches all the way to a final situation of the program before executing any actions.

New ConGolog Constructs

$(\delta_1 \parallel \delta_2)$,	concurrent execution
$(\delta_1 \rangle\!\rangle \delta_2)$,	concurrent execution with different priorities
$\delta^{\parallel}$,	concurrent iteration
$<\phi \rightarrow \delta>$,	interrupt.

In $(\delta_1 \rangle\!\rangle \delta_2)$, δ_1 has higher priority than δ_2. δ_2 executes only 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.

ConGolog Constructs (cont.)

An interrupt $<\phi \rightarrow \delta>$ has trigger condition ϕ and body δ. If interrupt gets control from higher priority processes and condition ϕ is true, it triggers and body is executed. Once body completes execution, may trigger again.

ConGolog Constructs (cont.)

In Golog:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{\text{def}}{=} (\phi?;\delta_1)|(\neg\phi?;\delta_2)$$

In ConGolog:

if ϕ **then** δ_1 **else** δ_2 **endIf**, synchronized conditional

while ϕ **do** δ **endWhile**, synchronized loop.

if ϕ **then** δ_1 **else** δ_2 **endIf** differs from $(\phi?;\delta_1)|(\neg\phi?;\delta_2)$ in that no action (or test) from an other process can occur between the test and the first action (or test) in the if branch selected (δ_1 or δ_2).

Similarly for **while**.

Exogenous Actions

One may also specify *exogenous actions* that can occur at random. This is useful for simulation. It is done by defining the Exo predicate:

$$Exo(a) \equiv a = a_1 \vee \ldots \vee a = a_n$$

Executing a program δ with the above amounts to executing

$$\delta \parallel a_1^* \parallel \ldots \parallel a_n^*$$

The current implementation also allows the programmer to specify probability distributions.

E.g. Two Robots Lifting a Table

- Objects:
  Two agents: $\forall r\, Robot(r) \equiv r = Rob_1 \vee r = Rob_2$.
  Two table ends: $\forall e\, TableEnd(e) \equiv e = End_1 \vee e = End_2$.

- Primitive actions:
  $grab(rob, end)$
  $release(rob, end)$
  $vmove(rob, z)$ move robot arm up or down by z units.

- Primitive fluents:
  $Holding(rob, end)$
  $vpos(end) = z$ height of the table end

- Initial state:
  $\forall r \forall e\, \neg Holding(r, e, S_0)$
  $\forall e\; vpos(e, S_0) = 0$

- Preconditions:
  $Poss(grab(r, e), s) \equiv \forall r^* \neg Holding(r^*, e, s) \wedge \forall e^* \neg Holding(r, e^*, s)$
  $Poss(release(r, e), s) \equiv Holding(r, e, s)$
  $Poss(vmove(r, z), s) \equiv True$

E.g. 2 Robots Lifting Table (cont.)

- Successor state axioms:
  $Holding(r, e, do(a, s)) \equiv a = grab(r, e) \vee$
  $\quad Holding(r, e, s) \wedge a \neq release(r, e)$
  $vpos(e, do(a, s)) = p \equiv$
  $\quad \exists r, z(a = vmove(r, z) \wedge Holding(r, e, s) \wedge p = vpos(e, s) + z) \vee$
  $\quad \exists r\, a = release(r, e) \wedge p = 0 \vee$
  $\quad p = vpos(e, s) \wedge \forall r\, a \neq release(r, e) \wedge$
  $\quad\quad \neg(\exists r, z\, a = vmove(r, z) \wedge Holding(r, e, s))$

E.g. 2 Robots Lifting Table (cont.)

Goal is to get the table up, but keep it sufficiently level so that nothing falls off.

$TableUp(s) \stackrel{def}{=} \; vpos(End_1, s) \geq H \; \wedge vpos(End_2, s) \geq H$
(both ends of table are higher than some threshold H)

$Level(s) \stackrel{def}{=} \; |vpos(End_1, s) - vpos(End_2, s)| \leq T$
(both ends are at same height to within a tolerance T)

$Goal(s) \stackrel{def}{=} \; TableUp(s) \wedge \; \forall s^* \leq s \; Level(s^*)$

E.g. 2 Robots Lifting Table (cont.)

Goal can be achieved by having Rob_1 and Rob_2 execute the same procedure $ctrl(r)$:

proc $ctrl(r)$
  $\pi \, e \, [TableEnd(e)?; \; grab(r, e)]$;
  while $\neg TableUp$ **do**
    $SafeToLift(r)?; vmove(r, A)$
  endWhile
endProc

where A is some constant such that $0 < A < T$ and

$$SafeToLift(r, s) \stackrel{def}{=} \; \exists e, e' \, e \neq e' \wedge TableEnd(e) \wedge TableEnd(e') \; \wedge \\ Holding(r, e, s) \; \wedge vpos(e) \leq vpos(e') + T - A$$

Proposition
$Ax \models \forall s. Do(ctrl(Rob_1) \parallel ctrl(Rob_2), S_0, s) \supset \; Goal(s)$

E.g. A Reactive Elevator Controller

- ordinary primitive actions:
  $goDown(e)$ — move elevator down one floor
  $goUp(e)$ — move elevator up one floor
  $buttonReset(n)$ — turn off call button of floor n
  $toggleFan(e)$ — change the state of elevator fan
  $ringAlarm$ — ring the smoke alarm

- exogenous primitive actions:
  $reqElevator(n)$ — call button on floor n is pushed
  $changeTemp(e)$ — the elevator temperature changes
  $detectSmoke$ — the smoke detector first senses smoke
  $resetAlarm$ — the smoke alarm is reset

- primitive fluents:
  $floor(e, s) = n$ — the elevator is on floor n, $1 \leq n \leq 6$
  $temp(e, s) = t$ — the elevator temperature is t
  $FanOn(e, s)$ — the elevator fan is on
  $ButtonOn(n, s)$ — call button on floor n is on
  $Smoke(s)$ — smoke has been detected

E.g. Reactive Elevator (cont.)

- defined fluents:
  $TooHot(e, s) \stackrel{def}{=} temp(e, s) > 3$
  $TooCold(e, s) \stackrel{def}{=} temp(e, s) < -3$

- initial state:
  $floor(e, S_0) = 1 \quad \neg FanOn(e, S_0) \quad temp(e, S_0) = 0$
  $ButtonOn(3, S_0) \quad ButtonOn(6, S_0)$

- exogenous actions:
  $\forall a.Exo(a) \equiv a = detectSmoke \vee a = resetAlarm \vee$
  $\exists e\, a = changeTemp(e) \vee \exists n\, a = reqElevator(n)$

- precondition axioms:
  $Poss(goDown(e), s) \equiv floor(e, s) \neq 1$
  $Poss(goUp(e), s) \equiv floor(e, s) \neq 6$
  $Poss(buttonReset(n), s) \equiv True$, $Poss(toggleFan(e), s) \equiv True$
  $Poss(reqElevator(n), s) \equiv (1 \leq n \leq 6) \wedge \neg ButtonOn(n, s)$
  $Poss(ringAlarm) \equiv True$, $Poss(changeTemp, s) \equiv True$
  $Poss(detectSmoke, s) \equiv \neg Smoke(s)$, $Poss(resetAlarm, s) \equiv Smoke(s)$

E.g. Reactive Elevator (cont.)

- successor state axioms:
  $floor(e, do(a, s)) = n \equiv$
  $(a = goDown(e) \wedge n = floor(e, s) - 1) \vee$
  $(a = goUp(e) \wedge n = floor(e, s) + 1) \vee$
  $(n = floor(e, s) \wedge a \neq goDown(e) \wedge a \neq goUp(e))$
  $temp(e, do(a, s)) = t \equiv$
  $(a = changeTemp(e) \wedge FanOn(e, s) \wedge t = temp(e, s) - 1) \vee$
  $(a = changeTemp(e) \wedge \neg FanOn(e, s) \wedge t = temp(e, s) + 1) \vee$
  $(t = temp(e, s) \wedge a \neq changeTemp(e))$
  $FanOn(e, do(a, s)) \equiv$
  $(a = toggleFan(e) \wedge \neg FanOn(e, s)) \vee$
  $(a \neq toggleFan(e) \wedge FanOn(e, s))$
  $ButtonOn(n, do(a, s)) \equiv$
  $a = reqElevator(n) \vee ButtonOn(n, s) \wedge a \neq buttonReset(n)$
  $Smoke(do(a, s)) \equiv$
  $a = detectSmoke \vee Smoke(s) \wedge a \neq resetAlarm$

E.g. Reactive Elevator (cont.)

In Golog, might write elevator controller as follows:

proc $controlG(e)$
  while $\exists n. ButtonOn(n)$ **do**
    $\pi\, n\, [BestButton(n)?; serveFloor(e, n)]$;
  endWhile
  while $floor(e) \neq 1$ **do** $goDown(e)$ **endWhile**
endProc

proc $serveFloor(e, n)$
  while $floor(e) < n$ **do** $goUp(e)$ **endWhile**;
  while $floor(e) > n$ **do** $goDown(e)$ **endWhile**;
  $buttonReset(n)$
endProc

E.g. Reactive Elevator (cont.)

Using this controller, get execution traces like:

$$Ax \models Do(controlG(e), S_0,$$
$$do([u, u, r_3, u, u, u, r_6, d, d, d, d, d], S_0))$$

where $u = goUp(e)$, $d = goDown(e)$, $r_n = buttonReset(n)$ (no exogenous actions in this run).

Problem with this: at end, elevator goes to ground floor and stops even if buttons are pushed.

E.g. Reactive Elevator (cont.)

Better solution in ConGolog, use interrupts:

$$< \exists n\, ButtonOn(n) \rightarrow$$
$$\pi, n\, [BestButton(n)?; serveFloor(e, n)] >$$
$$\rangle\!\rangle$$
$$< floor(e) \neq 1 \rightarrow goDown(e) >$$

Easy to extend to handle emergency requests. Add following at higher priority:

$$< \exists n\, EButtonOn(n) \rightarrow$$
$$\pi\, n\, [EButtonOn(n)?; serveEFloor(e, n)] >$$

E.g. Reactive Elevator (cont.)

If we also want to control the fan, as well as ring the alarm and only serve emergency requests when there is smoke, we write:

proc $control(e)$

$(<TooHot(e) \wedge \neg FanOn(e) \rightarrow toggleFan(e)> \parallel$

$<TooCold(e) \wedge FanOn(e) \rightarrow toggleFan(e)>) \rangle\!\rangle$

$<\exists n\, EButtonOn(n) \rightarrow$

$\pi\, n\, [EButtonOn(n)?; serveEFloor(e, n)]> \rangle\!\rangle$

$<Smoke \rightarrow ringAlarm> \rangle\!\rangle$

$<\exists n\, ButtonOn(n) \rightarrow$

$\pi\, n\, [BestButton(n)?; serveFloor(e, n)]> \rangle\!\rangle$

$<floor(e) \neq 1 \rightarrow goDown(e)>$

endProc

E.g. Reactive Elevator (cont.)

To control a single elevator E_1, we write $control(E_1)$.

To control n elevators, we can simply write:

$$control(E_1) \parallel \ldots \parallel control(E_n)$$

Note that priority ordering over processes is only a partial order.

In some cases, want unbounded number of instances of a process running in parallel. E.g. FTP server with a manager process for each active FTP session. Can be programmed using concurrent iteration $\delta^{\parallel}$.

An Evaluation Semantics for Golog

In [LRLLS97], $Do(\delta, s, s')$ is simply viewed as an abbreviation for a formula of the sit. calc.; defined inductively as follows:

$$
\begin{aligned}
&Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \wedge s' = do(a[s], s)\\
&Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \wedge s = s'\\
&Do(\delta_1; \delta_2,\ s, s') \stackrel{def}{=} \exists s''.\ Do(\delta_1, s, s'') \wedge Do(\delta_2, s'', s')\\
&Do(\delta_1 \mid \delta_2,\ s, s') \stackrel{def}{=} Do(\delta_1, s, s') \vee Do(\delta_2, s, s')\\
&Do(\pi\, x, \delta(x), s, s') \stackrel{def}{=} \exists x.\, Do(\delta(x), s, s')
\end{aligned}
$$

Golog Evaluation Semantics (cont.)

$$
\begin{aligned}
&Do(\delta^*, s, s') \stackrel{def}{=} \forall P.\{\, \forall s_1.\, P(s_1, s_1) \wedge\\
&\quad \forall s_1, s_2, s_3.[P(s_1, s_2) \wedge Do(\delta, s_2, s_3) \supset P(s_1, s_3)]\,\}\\
&\supset\ P(s, s').
\end{aligned}
$$

i.e., doing action δ zero or more times takes you from s to s' iff (s, s') is in every set (and thus, the smallest set) s.t.:

1. (s_1, s_1) is in the set for all situations s_1.

2. Whenever (s_1, s_2) is in the set, and doing δ in situation s_2 takes you to situation s_3, then (s_1, s_3) is in the set.

Golog Evaluation Semantics (cont.)

The above is the standard 2nd-order way of expressing this set. Must use 2nd-order logic because transitive closure is not 1st-order definable.

For procedures (more complex) see [LRLLS97].

A Transition Semantics for ConGolog

Can develop Golog-style semantics for ConGolog with $Do(\delta, s, s')$ as a macro, but makes handling prioritized concurrency difficult.

So define a *computational semantics* based on *transition systems*, a fairly standard approach in the theory of programming languages [NN92]. First define relations $Trans$ and $Final$.

$Trans(\delta, s, \delta', s')$ means that

$$(\delta, s) \rightarrow (\delta', s')$$

by executing a single primitive action or wait action.

$Final(\delta, s)$ means that in configuration (δ, s), the computation may be considered completed.

ConGolog Transition Semantics (cont.)

$$
\begin{array}{l}
Trans(nil, s, \delta, s') \equiv False \\
Trans(\alpha, s, \delta, s') \equiv \\
\quad Poss(\alpha[s], s) \wedge \delta = nil \wedge s' = do(\alpha[s], s) \\
Trans(\phi?, s, \delta, s') \equiv \phi[s] \wedge \delta = nil \wedge s' = s \\
Trans([\delta_1; \delta_2], s, \delta, s') \equiv \\
\quad Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s') \quad \vee \\
\quad \exists \delta'. \delta = (\delta'; \delta_2) \wedge Trans(\delta_1, s, \delta', s') \\
Trans([\delta_1 \mid \delta_2], s, \delta, s') \equiv \\
\quad Trans(\delta_1, s, \delta, s') \vee Trans(\delta_2, s, \delta, s') \\
Trans(\pi\, x\, \delta, s, \delta, s') \equiv \exists x. Trans(\delta, s, \delta, s')
\end{array}
$$

ConGolog Transition Semantics (cont.)

Here, $Trans$ and $Final$ are predicates that take programs as arguments. So need to introduce terms that denote programs (reify programs). In 3rd axiom, ϕ is term that denotes formula; $\phi[s]$ stands for $Holds(\phi, s)$, which is true iff formula denoted by ϕ is true in s. Details in [DLL00].

ConGolog Transition Semantics (cont.)

$Trans(\delta^*, s, \delta, s') \equiv \exists\delta'.\delta = (\delta'; \delta^*) \wedge Trans(\delta, s, \delta', s')$
$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s, \delta, s') \equiv$
$\quad \phi(s) \wedge Trans(\delta_1, s, \delta, s') \vee \neg\phi(s) \wedge Trans(\delta_2, s, \delta, s')$
$Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv \phi(s) \wedge$
$\quad \exists\delta''.\ \delta' = (\delta''; \textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}) \wedge Trans(\delta, s, \delta'', s')$
$Trans([\delta_1 \parallel \delta_2], s, \delta, s') \equiv \exists\delta'.$
$\quad \delta = (\delta' \parallel \delta_2) \wedge Trans(\delta_1, s, \delta', s') \vee$
$\quad \delta = (\delta_1 \parallel \delta') \wedge Trans(\delta_2, s, \delta', s')$
$Trans([\delta_1 \rangle\!\rangle \delta_2], s, \delta, s') \equiv \exists\delta'.$
$\quad \delta = (\delta' \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \delta', s') \vee$
$\quad \delta = (\delta_1 \rangle\!\rangle \delta') \wedge Trans(\delta_2, s, \delta', s') \wedge$
$\quad\quad \neg\exists\delta'', s''.Trans(\delta_1, s, \delta'', s'')$
$Trans(\delta^{\parallel}, s, \delta', s') \equiv$
$\quad \exists\delta''.\delta' = (\delta'' \parallel \delta^{\parallel}) \wedge Trans(\delta, s, \delta'', s')$

ConGolog Transition Semantics (cont.)

$Final(nil, s) \equiv True$
$Final(\alpha, s) \equiv False$
$Final(\phi?, s) \equiv False$
$Final([\delta_1; \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$
$Final([\delta_1 \mid \delta_2], s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$
$Final(\pi\, x\, \delta, s) \equiv \exists x.Final(\delta, s)$
$Final(\delta^*, s) \equiv True$
$Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf}, s) \equiv$
$\quad \phi(s) \wedge Final(\delta_1, s) \vee \neg\phi(s) \wedge Final(\delta_2, s)$
$Final(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s) \equiv$
$\quad \phi(s) \wedge Final(\delta, s) \vee \neg\phi(s)$
$Final([\delta_1 \parallel \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$
$Final([\delta_1 \rangle\!\rangle \delta_2], s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$
$Final(\delta^{\parallel}, s) \equiv True$

ConGolog Transition Semantics (cont.)

Then, define relation $Do(\delta, s, s')$ meaning that process δ, when executed starting in situation s, has s' as a legal terminating situation:

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

where $Trans^*$ is the transitive closure of $Trans$.

That is, $Do(\delta, s, s')$ holds iff the starting configuration (δ, s) can evolve into a configuration (δ, s') by doing a finite number of transitions and $Final(\delta, s')$.

ConGolog Transition Semantics (cont.)

$$Trans^*(\delta, s, \delta', s') \stackrel{def}{=} \forall T[\ldots \supset T(\delta, s, \delta', s')]$$

where the ellipsis stands for:

$$\begin{array}{l} \forall s.\ T(\delta, s, \delta, s) \quad \wedge \\ \forall s, \delta', s', \delta'', s''.\ T(\delta, s, \delta', s') \wedge \\ \qquad Trans(\delta', s', \delta'', s'') \supset T(\delta, s, \delta'', s''). \end{array}$$

Interrupts

Interrupts can be defined in terms of other constructs:

$$<\phi \rightarrow \delta> \stackrel{def}{=} \quad \textbf{while } Interrupts\_running \textbf{ do} \quad \textbf{if } \phi \textbf{ then } \delta \textbf{ else } False? \textbf{ endIf} \quad \textbf{endWhile}$$

Uses special fluent $Interrupts\_running$.

To execute a program δ containing interrupts, actually execute:

$$start\_interrupts\, ;\, (\delta \rangle\!\rangle\; stop\_interrupts)$$

This stops blocked interrupt loops in δ at lowest priority, i.e., when there are no more actions in δ that can be executed.

Implementation in Prolog

```
trans(act(A),S,nil,do(AS,S)) :- sub(now,S,A,AS), poss(AS,S).

trans(test(C),S,nil,S) :- holds(C,S).

trans(seq(P1,P2),S,P2r,Sr) :- final(P1,S),trans(P2,S,P2r,Sr).
trans(seq(P1,P2),S,seq(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).

trans(choice(P1,P2),S,Pr,Sr) :- trans(P1,S,Pr,Sr) ; trans(P2,S,Pr,Sr).

trans(conc(P1,P2),S,conc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(conc(P1,P2),S,conc(P1,P2r),Sr) :- trans(P2,S,P2r,Sr).
...
final(seq(P1,P2),S) :- final(P1,S),final(P2,S).
...

trans*(P,S,P,S).
trans*(P,S,Pr,Sr) :- trans(P,S,PP,SS), trans*(PP,SS,Pr,Sr).

do(P,S,Sr) :- trans*(P,S,Pr,Sr),final(Pr,Sr).
```

Prolog Implementation (cont.)

```
holds(and(F1,F2),S) :- holds(F1,S), holds(F2,S).
holds(or(F1,F2),S) :- holds(F1,S); holds(F2,S).
holds(neg(and(F1,F2)),S) :- holds(or(neg(F1),neg(F2)),S).
holds(neg(or(F1,F2)),S) :- holds(and(neg(F1),neg(F2)),S).
holds(some(V,F),S) :- sub(V,_,F,Fr), holds(Fr,S).
holds(neg(some(V,F)),S) :- not holds(some(V,F),S). /* Negation as failure */
...
holds(P_Xs,S) :-
   P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=some(_._),
   sub(now,S,P_Xs,P_XsS), P_XsS.
holds(neg(P_Xs),S) :-
   P_Xs\=and(_,_),P_Xs\=or(_,_),P_Xs\=neg(_),P_Xs\=all(_,_),P_Xs\=some(_._),
   sub(now,S,P_Xs,P_XsS), not P_XsS.              /* Negation as failure */
```

Note: makes closed-world assumption; must have complete knowledge!

Implemented E.g. 2 Robots Lifting Table

```
/* Precondition axioms */

poss(grab(Rob,E),S) :- not holding(_,E,S), not holding(Rob,_,S).
poss(release(Rob,E),S) :- holding(Rob,E,S).
poss(vmove(Rob,Amount),S) :- true.

/* Successor state axioms */

val(vpos(E,do(A,S)),V) :-
  (A=vmove(Rob,Amt), holding(Rob,E,S), val(vpos(E,S),V1), V is V1+Amt);
  (A=release(Rob,E), V=0) ;
  (val(vpos(E,S),V), not((A=vmove(Rob,Amt), holding(Rob,E,S))),
                     A\=release(Rob,E)).

holding(Rob,E,do(A,S)) :-
   A=grab(Rob,E) ; (holding(Rob,E,S), A\=release(Rob,E)).
```

Implemented E.g. 2 Robots (cont.)

```
/* Defined Fluents */

tableUp(S) :- val(vpos(end1,S),V1), V1 >= 3, val(vpos(end2,S),V2), V2 >= 3.

safeToLift(Rob,Amount,Tol,S) :-
   tableEnd(E1), tableEnd(E2), E2\=E1, holding(Rob,E1,S),
   val(vpos(E1,S),V1), val(vpos(E2,S),V2), V1 =< V2+Tol-Amount.

/* Initial state */

val(vpos(end1,s0),0).       /* plus by CWA:           */
val(vpos(end2,s0),0).       /*                        */
tableEnd(end1).             /* not holding(rob1,_,s0) */
tableEnd(end2).             /* not holding(rob2,_,s0) */
```


Implemented E.g. 2 Robots (cont.)

```
/* Control procedures  */

proc(ctrl(Rob,Amount,Tol),
        seq(pick(e,seq(test(tableEnd(e)),act(grab(Rob,e)))),
            while(neg(tableUp(now)),
               seq(test(safeToLift(Rob,Amount,Tol,now)),
                   act(vmove(Rob,Amount)))))).

proc(jointLiftTable,
        conc(pcall(ctrl(rob1,1,2)), pcall(ctrl(rob2,1,2)))).
```

Running 2 Robots E.g.

```
?- do(pcall(jointLiftTable),s0,S).

S = do(vmove(rob2,1), do(vmove(rob1,1), do(vmove(rob2,1), do(vmove(rob1,1),
 do(vmove(rob2,1), do(grab(rob2,end2), do(vmove(rob1,1), do(vmove(rob1,1),
do(grab(rob1,end1), s0))))))))) ;

S = do(vmove(rob2,1), do(vmove(rob1,1), do(vmove(rob2,1), do(vmove(rob1,1),
 do(vmove(rob2,1), do(grab(rob2,end2), do(vmove(rob1,1), do(vmove(rob1,1),
do(grab(rob1,end1), s0))))))))) ;

S = do(vmove(rob1,1), do(vmove(rob2,1), do(vmove(rob2,1), do(vmove(rob1,1),
 do(vmove(rob2,1), do(grab(rob2,end2), do(vmove(rob1,1), do(vmove(rob1,1),
do(grab(rob1,end1), s0)))))))))

Yes
```


References

G. De Giacomo, Y. Lespérance, and H.J. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence*, **121**, 109–169, 2000.

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

Chapter 6 of R. Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

H.R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction*. Wiley Professional Computing, Wiley, 1992.