Lecture Outline

Part 1: Syntax, Informal Semantics, Examples

Part 2: Formal Semantics

After Holidays: Implementation

High-level programming in the Situation Calculus: Golog and ConGolog

Yves Lespérance

Department of Computer Science
York University
Toronto, Canada

Giuseppe De Giacomo

Dipartimento di Informatica e Sistemistica
Universitá di Roma “La Sapienza”
Roma, Italy

High-level programming in the Situation Calculus — The Approach

Plan synthesis is often too hard; need to script some behaviors in advance.

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.

So can still do planning/deliberation.

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.

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

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.

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.

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.

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).

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.

Elevator Example (cont.)

- Action Precondition Axioms (cont.):

$$Poss(close, s) \equiv True.$$

$$Poss(turnoff(n), s) \equiv on(n, s).$$

$$Poss(no\_op, s) \equiv True.$$

- Successor State Axioms:

$$\begin{aligned}
&floor(do(a, s)) = m \equiv \\
&\quad a = up(m) \vee a = down(m) \vee \\
&\quad floor(s) = m \wedge \neg \exists n\, a = up(n) \wedge \neg \exists n\, a = down(n).
\end{aligned}$$

$$\begin{aligned}
&on(m, do(a, s)) \equiv \\
&\quad a = push(m) \vee on(m, s) \wedge a \neq turnoff(m).
\end{aligned}$$

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:

$$Poss(up(n), s) \equiv floor(s) < n.$$

$$Poss(down(n), s) \equiv floor(s) > n.$$

$$Poss(open, s) \equiv True.$$

Elevator Example (cont.)

- Golog Procedures (cont.):

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

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

Elevator Example (cont.)

- Fluent abbreviation:

$$next\_floor(n,s) \stackrel{\text{def}}{=} on(n,s) \wedge \\ \forall m.on(m,s) \supset |m - floor(s)| \geq |n - floor(s)|.$$

- 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

Elevator Example (cont.)

- Querying the theory:

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

- Successful proof might return

$$\begin{aligned} s = {} & do(open, do(down(0), do(close, do(open, \\ & do(turnoff(5), do(up(5), do(close, do(open, \\ & do(turnoff(3), do(down(3), S_0)))))))))). \end{aligned}$$

Elevator Example (cont.)

- Golog Procedures (cont.):

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

- Initial situation:

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

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

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.

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:

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

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.

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

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

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.

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

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

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.

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.

ConGolog Constructs (cont.)

In Golog:

if ϕ **then** δ_1 **else** δ_2 **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**.

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

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

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. 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. 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. 2 Robots Lifting Table (cont.)

Claim that goal can be achieved by having Rob_1 and Rob_2 each independently execute the same procedure $ctrl(r)$ defined as:

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. 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.)

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

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.)

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.)

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 $control(e)$

$$
\begin{array}{l}
\quad (< TooHot(e) \wedge \neg FanOn(e) \rightarrow toggleFan(e) > \parallel \\
\quad\; < TooCold(e) \wedge FanOn(e) \rightarrow toggleFan(e) >) \rangle\!\rangle \\
\quad\; < \exists n\, EButtonOn(n) \rightarrow \\
\quad\quad \pi\, n\, [EButtonOn(n)?; serveEFloor(e,n)] > \rangle\!\rangle \\
\quad\; < Smoke \rightarrow ringAlarm > \rangle\!\rangle \\
\quad\; < \exists n\, ButtonOn(n) \rightarrow \\
\quad\quad \pi\, n\, [BestButton(n)?; serveFloor(e,n)] > \rangle\!\rangle \\
\quad\; < floor(e) \neq 1 \rightarrow goDown(e) >
\end{array}
$$

endProc

E.g. Reactive Elevator (cont.)

Better solution in ConGolog, use interrupts:

$$
\begin{array}{l}
< \exists n\, ButtonOn(n) \rightarrow \\
\quad \pi, n\, [BestButton(n)?; serveFloor(e,n)] > \\
\rangle\!\rangle \\
< floor(e) \neq 1 \rightarrow goDown(e) >
\end{array}
$$

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

$$
\begin{array}{l}
< \exists n\, EButtonOn(n) \rightarrow \\
\quad \pi\, n\, [EButtonOn(n)?; serveEFloor(e,n)] >
\end{array}
$$

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 $\delta^{\parallel}$.