

# Concurrent Processes and ConGolog

## References:

Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus: Language and Implementation.

Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus: Foundations.

Revised version will appear in *Artificial Intelligence*, 2000.

## 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.

## Modeling Concurrency

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

Important notion: process becoming *blocked*. Happens when a process  $\sigma$  reaches a primitive action whose preconditions are false or a test action  $\phi?$  and  $\phi$  is false.

Then execution need not fail as in Golog. In ConGolog, execution may continue provided another process executes next. The  $\sigma$  process is blocked.

## New ConGolog Constructs

$(\sigma_1 \parallel \sigma_2),$	concurrent execution
$(\sigma_1 \gg \sigma_2),$	concurrent execution with different priorities
$\sigma \parallel,$	concurrent iteration
$\langle \phi \rightarrow \sigma \rangle,$	interrupt.

In  $(\sigma_1 \gg \sigma_2)$ ,  $\sigma_1$  has higher priority than  $\sigma_2$ .  $\sigma_2$  executes only when  $\sigma_1$  is done or blocked.

$\sigma \parallel$  is like nondeterministic iteration  $\sigma^*$ , but the instances of  $\sigma$  are executed concurrently rather than in sequence.

An interrupt  $\langle \phi \rightarrow \sigma \rangle$  has a trigger condition  $\phi$  and a body  $\sigma$ . If the interrupt gets control from higher priority processes and the condition  $\phi$  is true, the interrupt triggers and the body is executed. Once the body completes execution, the interrupt may trigger again.



## E.g. Two Robots Lifting a Table

- Objects:  
Two agents:  $\forall r \text{ Robot}(r) \equiv r = \text{Rob}_1 \vee r = \text{Rob}_2$ .  
Two table ends:  $\forall e \text{ TableEnd}(e) \equiv e = \text{End}_1 \vee e = \text{End}_2$ .
- Primitive actions:  
 $\text{grab}(\text{rob}, \text{end})$   
 $\text{release}(\text{rob}, \text{end})$   
 $\text{vmove}(\text{rob}, z)$       move robot arm up or down by  $z$  units.
- Primitive fluents:  
 $\text{Holding}(\text{rob}, \text{end})$   
 $\text{vpos}(\text{end}) = z$       height of the table end
- Initial state:  
 $\forall r \forall e \neg \text{Holding}(r, e, S_0)$   
 $\forall e \text{ vpos}(e, S_0) = 0$
- Preconditions:  
 $\text{Poss}(\text{grab}(r, e), s) \equiv \forall r^* \neg \text{Holding}(r^*, e, s) \wedge \forall e^* \neg \text{Holding}(r, e^*, s)$   
 $\text{Poss}(\text{release}(r, e), s) \equiv \text{Holding}(r, e, s)$   
 $\text{Poss}(\text{vmove}(r, z), s) \equiv \text{TRUE}$

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

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

$$\text{TableUp}(s) \stackrel{\text{def}}{=} \text{vpos}(\text{End}_1, s) \geq H \wedge \text{vpos}(\text{End}_2, s) \geq H$$

(both ends of table are higher than some threshold  $H$ )

$$\text{Level}(s) \stackrel{\text{def}}{=} |\text{vpos}(\text{End}_1, s) - \text{vpos}(\text{End}_2, s)| \leq T$$

(both ends are at same height to within a tolerance  $T$ )

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

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
     $SafeToLift(r)?;$ 
     $vmove(r, A)$ 
  end

```

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

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

### Proposition

$$\begin{aligned}
 Ax \models & \forall s. Do(ctrl(Rob_1) \parallel ctrl(Rob_2), S_0, s) \\
 & \supset Goal(s)
 \end{aligned}$$

## 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
- defined fluents:
 

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

- initial state:

$$\begin{aligned} \text{floor}(e, S_0) = 1 \quad \neg \text{FanOn}(e, S_0) \quad \text{temp}(e, S_0) = 0 \\ \text{ButtonOn}(3, S_0) \quad \text{ButtonOn}(6, S_0) \end{aligned}$$

- exogenous actions:

$$\begin{aligned} \forall a. \text{Exo}(a) \equiv a = \text{detectSmoke} \vee a = \text{resetAlarm} \vee \\ \exists e a = \text{changeTemp}(e) \vee \exists n a = \text{reqElevator}(n) \end{aligned}$$

- precondition axioms:

$$\begin{aligned} \text{Poss}(\text{goDown}(e), s) &\equiv \text{floor}(e, s) \neq 1 \\ \text{Poss}(\text{goUp}(e), s) &\equiv \text{floor}(e, s) \neq 6 \\ \text{Poss}(\text{buttonReset}(n), s) &\equiv \text{TRUE} \\ \text{Poss}(\text{toggleFan}(e), s) &\equiv \text{TRUE} \\ \text{Poss}(\text{ringAlarm}) &\equiv \text{TRUE} \\ \text{Poss}(\text{reqElevator}(n), s) &\equiv (1 \leq n \leq 6) \wedge \\ &\quad \neg \text{ButtonOn}(n, s) \\ \text{Poss}(\text{changeTemp}(s)) &\equiv \text{TRUE} \\ \text{Poss}(\text{detectSmoke}, s) &\equiv \neg \text{Smoke}(s) \\ \text{Poss}(\text{resetAlarm}, s) &\equiv \text{Smoke}(s) \end{aligned}$$

- successor state axioms:

$$\begin{aligned} \text{Poss}(a, s) \supset [\text{floor}(e, \text{do}(a, s)) = n &\equiv \\ (a = \text{goDown}(e) \wedge n = \text{floor}(e, s) - 1) \vee & \\ (a = \text{goUp}(e) \wedge n = \text{floor}(e, s) + 1) \vee & \\ (n = \text{floor}(e, s) \wedge a \neq \text{goDown}(e) \wedge & \\ a \neq \text{goUp}(e))] & \\ \text{Poss}(a, s) \supset [\text{temp}(e, \text{do}(a, s)) = t &\equiv \\ (a = \text{changeTemp}(e) \wedge \text{FanOn}(e, s) \wedge & \\ t = \text{temp}(e, s) - 1) \vee & \\ (a = \text{changeTemp}(e) \wedge \neg \text{FanOn}(e, s) \wedge & \\ t = \text{temp}(e, s) + 1) \vee & \\ (t = \text{temp}(e, s) \wedge a \neq \text{changeTemp}(e))] & \\ \text{Poss}(a, s) \supset [\text{FanOn}(e, \text{do}(a, s)) &\equiv \\ (a = \text{toggleFan}(e) \wedge \neg \text{FanOn}(e, s)) \vee & \\ (a \neq \text{toggleFan}(e) \wedge \text{FanOn}(e, s))] & \\ \text{Poss}(a, s) \supset [\text{ButtonOn}(n, \text{do}(a, s)) &\equiv \\ a = \text{reqElevator}(n) \vee & \\ (\text{ButtonOn}(n, s) \wedge a \neq \text{buttonReset}(n))] & \\ \text{Poss}(a, s) \supset [\text{Smoke}(\text{do}(a, s)) &\equiv \\ a = \text{detectSmoke} \vee & \\ (\text{Smoke}(s) \wedge a \neq \text{resetAlarm})] & \end{aligned}$$

In Golog you might write an elevator controller as follows:

```

proc controlG(e)
  while  $\exists n. ButtonOn(n)$  do
     $\pi n. \{ BestButton(n)?; serveFloor(e, n) \};$ 
  while  $floor(e) \neq 1$  do goDown(e)
end
proc serveFloor(e, n)
  while  $floor(e) < n$  do goUp(e);
  while  $floor(e) > n$  do goDown(e);
  buttonReset(n)
end

```

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.

Better solution in ConGolog, use interrupts:

$$\begin{aligned}
 &< \exists n. ButtonOn(n) \rightarrow \\
 &\quad \pi n. \{ BestButton(n)?; serveFloor(e, n) \} > \\
 &\gg \\
 &< floor(e) \neq 1 \rightarrow goDown(e) >
 \end{aligned}$$

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

$$\begin{aligned}
 &< \exists n. EButtonOn(n) \rightarrow \\
 &\quad \pi n. \{ EButtonOn(n)?; serveEFloor(e, n) \} >
 \end{aligned}$$

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) ∧ ¬FanOn(e) → toggleFan(e)> ||
   <TooCold(e) ∧ FanOn(e) → toggleFan(e)>) >>
  <∃n.EButtonOn(n) →
   πn.{EButtonOn(n)?; serveEFloor(e, n)}>>
  <Smoke → ringAlarm> >>
  <∃n.ButtonOn(n) →
   πn.{BestButton(n)?; serveFloor(e, n)}>>
  <floor(e) ≠ 1 → goDown(e)>
end

```

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

To control  $n$  elevators, we can simply write:

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

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  $\sigma\parallel$ .

## ConGolog Semantics

We had originally developed a Golog-style semantics for ConGolog where  $Do(\delta, s, s')$  was macro. But handling prioritized concurrency is difficult in that approach, and the resulting semantics was unintuitive.

So the definitive semantics is based on transition systems, a fairly standard approach in their theory of programming languages. 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  $(\delta, s)$ , the computation may be considered completed.

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

In this semantics,  $Trans$  and  $Final$  are predicates that take programs as arguments. So need to introduce terms that denote programs (reify programs). In the third axiom,  $\phi$  is a term that denotes a formula, and  $\phi[s]$  stands for  $Holds(\phi, s)$ , which is true iff the formula denoted by  $\phi$  is true in  $s$ . The details are in the “Foundations” paper.

$$\begin{aligned}
Trans(\delta^*, s, \delta, s') &\equiv \\
&\exists \delta'. \delta = (\delta'; \delta^*) \wedge Trans(\delta, s, \delta', s') \\
Trans(\mathbf{if} \ \phi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2, s, \delta, s') &\equiv \\
&\phi(s) \wedge Trans(\delta_1, s, \delta, s') \quad \vee \\
&\neg \phi(s) \wedge Trans(\delta_2, s, \delta, s') \\
Trans(\mathbf{while} \ \phi \ \mathbf{do} \ \delta, s, \delta', s') &\equiv \\
&\phi(s) \wedge \exists \delta''. \delta' = (\delta''; \mathbf{while} \ \phi \ \mathbf{do} \ \delta) \wedge \\
&\quad Trans(\delta, s, \delta'', s') \\
Trans([\delta_1 \parallel \delta_2], s, \delta, s') &\equiv \\
&\exists \delta'. \delta = (\delta' \parallel \delta_2) \wedge Trans(\delta_1, s, \delta', s') \quad \vee \\
&\delta = (\delta_1 \parallel \delta') \wedge Trans(\delta_2, s, \delta', s') \\
Trans([\delta_1 \gg \delta_2], s, \delta, s') &\equiv \\
&\exists \delta'. \delta = (\delta' \gg \delta_2) \wedge Trans(\delta_1, s, \delta', s') \quad \vee \\
&\delta = (\delta_1 \gg \delta') \wedge Trans(\delta_2, s, \delta', s') \wedge \\
&\quad \neg \exists \delta'', s''. Trans(\delta_1, s, \delta'', s'') \\
Trans(\delta \parallel, s, \delta', s') &\equiv \\
&\exists \delta''. \delta' = (\delta'' \parallel \delta) \wedge Trans(\delta, s, \delta'', s')
\end{aligned}$$

$$\begin{aligned}
Final(nil, s) &\equiv TRUE \\
Final(\alpha, s) &\equiv FALSE \\
Final(\phi?, s) &\equiv FALSE \\
Final([\delta_1; \delta_2], s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
Final([\delta_1 \mid \delta_2], s) &\equiv Final(\delta_1, s) \vee Final(\delta_2, s) \\
Final(\pi x. \delta, s) &\equiv \exists x. Final(\delta, s) \\
Final(\delta^*, s) &\equiv TRUE \\
Final(\mathbf{if} \ \phi \ \mathbf{then} \ \delta_1 \ \mathbf{else} \ \delta_2, s) &\equiv \\
&\phi(s) \wedge Final(\delta_1, s) \vee \neg \phi(s) \wedge Final(\delta_2, s) \\
Final(\mathbf{while} \ \phi \ \mathbf{do} \ \delta, s) &\equiv \\
&\phi(s) \wedge Final(\delta, s) \vee \neg \phi(s) \\
Final([\delta_1 \parallel \delta_2], s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
Final([\delta_1 \gg \delta_2], s) &\equiv Final(\delta_1, s) \wedge Final(\delta_2, s) \\
Final(\delta \parallel, s) &\equiv TRUE
\end{aligned}$$

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

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

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

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

where the ellipsis stands for:

$$\begin{aligned} & \forall s. T(\delta, s, \delta, s) \quad \wedge \\ & \forall s, \delta', s', \delta'', s''. T(\delta, s, \delta', s') \wedge \\ & \quad Trans(\delta', s', \delta'', s'') \supset T(\delta, s, \delta'', s''). \end{aligned}$$

## Trans-based ConGolog Interpreter

```

/* trans(Prog,Sit,Prog_r,Sit_r) */
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(pick(V,P),S,Pr,Sr) :- sub(V,_,P,PP), trans(PP,S,Pr,Sr).
trans(iter(P),S,seq(PP,iter(P)),Sr) :- trans(P,S,PP,Sr).
trans(if(C,P1,P2),S,Pr,Sr) :- holds(C,S),trans(P1,S,Pr,Sr) ;
    holds(neg(C),S),trans(P2,S,Pr,Sr).
trans(while(C,P),S,seq(PP,while(C,P)),Sr) :-
    holds(C,S),trans(P,S,PP,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).
trans(prconc(P1,P2),S,prconc(P1r,P2),Sr) :- trans(P1,S,P1r,Sr).
trans(prconc(P1,P2),S,prconc(P1,P2r),Sr) :-
    not trans(P1,S,_,_),trans(P2,S,P2r,Sr).
trans(iterconc(P),S,conc(PP,iterconc(P)),Sr) :- trans(P,S,PP,Sr).
trans(pcall(P_Args),S,Pr,Sr) :- sub(now,S,P_Args,P_ArgsS),
    proc(P_ArgsS,P), trans(P,S,Pr,Sr).

/* final(Prog,Sit) */
final(nil,S).
final(seq(P1,P2),S) :- final(P1,S),final(P2,S).
final(choice(P1,P2),S) :- final(P1,S) ; final(P2,S).
final(pick(V,P),S) :- sub(V,_,P,PP), final(PP,S).
final(iter(P),S).
final(if(C,P1,P2),S) :- holds(C,S),final(P1,S) ;
    holds(neg(C),S),final(P2,S).
final(while(C,P),S) :- holds(neg(C),S) ; final(P,S).
final(conc(P1,P2),S) :- final(P1,S),final(P2,S).
final(prconc(P1,P2),S) :- final(P1,S),final(P2,S).
final(iterconc(P),S).
final(pcall(P_Args)) :- sub(now,S,P_Args,P_ArgsS),
    proc(P_ArgsS,P),final(P,S).

```

```

/* trans*(Prog,Sit,Prog_r,Sit_r) */
trans*(P,S,P,S).
trans*(P,S,Pr,Sr) :- trans(P,S,PP,SS), trans*(PP,SS,Pr,Sr).

/* do(Prog,Sit,Sit_r) */
do(P,S,Sr) :- trans*(P,S,Pr,Sr),final(Pr,Sr).

/* holds(Cond,Sit) */
holds(and(F1,F2),S) :- holds(F1,S), holds(F2,S).
holds(or(F1,F2),S) :- holds(F1,S); holds(F2,S).
holds(all(V,F),S) :- holds(neg(some(V,neg(F))),S).
holds(some(V,F),S) :- sub(V,_,F,Fr), holds(Fr,S).
holds(neg(neg(F)),S) :- holds(F,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(neg(all(V,F)),S) :- holds(some(V,neg(F)),S).
holds(neg(some(V,F)),S) :- not holds(some(V,F),S)./* Neg by fail
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 by failure */

/* sub(Const,Var,Term1,Term2) */
sub(X,Y,T,Tr) :- var(T), Tr = T.
sub(X,Y,T,Tr) :- not var(T), T = X, Tr = Y.
sub(X,Y,T,Tr) :- T \= X, T =..[F|Ts], sub_list(X,Y,Ts,Trs),
Tr =..[F|Trs].
sub_list(X,Y,[],[]).
sub_list(X,Y,[T|Ts],[Tr|Trs]) :- sub(X,Y,T,Tr),
sub_list(X,Y,Ts,Trs).

```

## E.g. Program – 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.

/* Succ state axioms */
val(vpos(E,do(A,S)),V) :-
(A=vmove(Rob,Amount), holding(Rob,E,S), val(vpos(E,S),V1),
V is V1+Amount);
(A=release(Rob,E), V=0);
(val(vpos(E,S),V), not((A=vmove(Rob,Amount),
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)).

/* 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) */

/* 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)))).

```

```
?- 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