DRONACHARYA
College of Engineering

INTELLIGENT SYSTEMS (CSE-303-F)

Section C

Situational Calculus

Essentials of Situation Calculus

- Situation Calculus was introduced by John McCarthy in 1969.
- It describes dynamic domains in FOL using:
  - situations (denote world states; include world history)
  - actions (named, parameterized functions)
  - axioms (to specify actions and domain knowledge)
- Planning (or: reasoning with actions) in the situation calculus is done through theorem proving:
  - Infer a goal situation from the initial situation using the given axioms.

Situation Calculus - Overview

- Situation Calculus is a specific, enriched FOL language.
- Actions denote changes of the world and are referred to by a name and a parameter-list (like functions).
- Situations refer to worlds and can be used to represent a (possible) world history for a given sequence of actions.
- The special function Result or do expresses that an action is applied in a situation.
- The effect (changes) and frame (remains) of an action are specified through axioms.
- Planning in situation calculus involves theorem-proving, inferring a goal situation from the initial situation.
- The actions involved in a proof and the bindings of their parameters represent the plan.

Situations

- A situation corresponds to a world (state).
- Situations are denoted through FOL terms: e.g. s, s'
- Actions transform situations, i.e. the application of an action in a given situation s yields a situation s'.
- Situations thus also refer to possible world histories.
- For example, the expression

$$do(putdown(A), do(walk(L), do(pickup(A), S_0)))$$

refers to the action sequence:

$$[\mathrm{pickup}(A), \mathrm{walk}(L), \mathrm{putdown}(A)]$$

yielding a new situation s when applied to S_0.

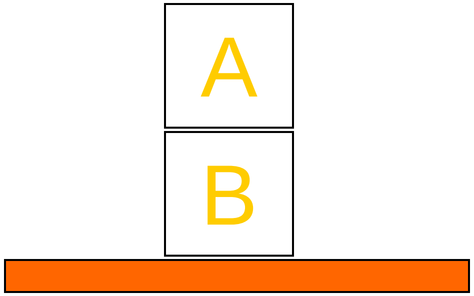
Situations - Example

Situation s_0

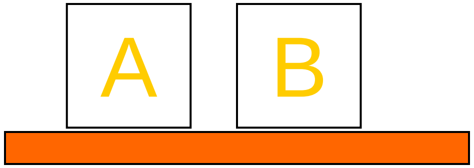
s_0 = {on(A,B),on(B,Fl),clear(A),clear(Fl)}

on(A,B,s_0),on(B,Fl,s_0),clear(A,s_0),

clear(Fl,s_0)



Action: move (A, B, Fl)



Situation s_1

s_1 = {on(A,Fl), on(B,Fl), clear(A),clear(B),clear(Fl)}

on(A,F,s_1),on(B,Fl,s_1),clear(A,s_1),clear(B,s_1),clear(Fl,s_1)

Actions

Actions are written as functions with their name and a parameter list. They can also be referred to by variables ($\rightarrow$ reification).

Actions transform situations.

The performance of an action in a situation is denoted through the Result or do function.

The performance (do) of an action a in a situation s yields a new situation s'.

Result- or do-Function

Result (or: do) is a function from actions and situations into situations.

<u>Example</u>

$$s' = do\ (move\ (x, y, z), s)$$

specifies a new situation s' which is the result of performing a move-action in situation s.

<u>General</u>

s' = do (a, s) for action a and situations s, s'

do-Function - Example

situation s = {on(A,B), on(B,Fl), clear(C)}

action a = move (A,B,C)

apply action a in situation s

do (move (A,B,C) , s) = s'

s' = {on(A,C), on(B,Fl), clear (B)}

Instead of specifying the situation s' this way, we add situations into the basic formulas (certain basic formulas - and terms).

Fluents

- Predicates and functions, whose values change due to actions, are called fluents.
- Predicates, whose truth values can change, are called relational fluents.

  example: is_holding(robot, p, s) or on(x,y,s) .
- Functions, whose denotations can vary, are called functional fluents.

  example: loc(robot, s) or under(x,s)
- Actions in a domain are specified by providing action precondition axioms, effect axioms and frame axioms.

Situations in Formulas

Integrating situations into the formulas above yields:

situation s   on(A,B,s), on(B,Fl,s), clear(C,s)

action a   move (A,B,C)

apply action a in situation s

do (a, s)

do (move (A,B,C), s) = s'

situation s'   on(A,C,s'), on(B,Fl,s'), clear(B,s')

Note: Persistent predicate expressions like Block(A), Block(B), ... remain without s.

The Calculus of Situation Calculus

Sit Calc Axioms "lite"

Action Description - Axioms

Axioms specify what changes and what remains.
Consider every combination of action and fluent.

effect-axioms – specify effects, i.e. what changes

- positive effects $\rightarrow$ a formula becomes true
- negative effects $\rightarrow$ a formula becomes false

frame-axioms – specify frame, i.e. what remains

- positive effects $\rightarrow$ a formula remains true
- negative effects $\rightarrow$ a formula remains false

In addition, general axioms specify general laws or rules of the domain.

Effect Axiom - move-example

action: move (x, y, z)

effect-axiom:

$$(on\ (x, y, s) \wedge clear\ (z, s) \wedge x \neq z) \Rightarrow$$
$$on\ (x, z, do\ (move\ (x, y, z), s))$$

Explanation:

If the left side (condition) of the axiom holds, then the action can be performed, and the right side (consequence) follows.

The consequence states what is true in the resulting situation, here: on(x,z,s)

Effect Axioms - move-example

positive effect

$on\ (x, y, s) \wedge clear\ (x, s) \wedge clear\ (z, s) \wedge y \neq z \Rightarrow$
$on\ (x, z, do\ (move\ (x, y, z), s))$

If x is on y, both x and z are clear, and z is not the block onto which x is moved, then a result of the move-action is that x is on z.

negative effect

$on\ (x, y, s) \wedge clear\ (x, s) \wedge clear\ (z, s) \wedge y \neq z \Rightarrow$
$\neg on\ (x, y, do\ (move\ (x, y, z), s))$

If x is on y, both x and z are clear, and z is not the block onto which x is moved, then a result of the move-action is that x is not anymore on y.

Frame Axiom - move-example

action: move (x, y, z)

Frame Axiom:

$$on(u, v, s) \wedge x \neq u \quad \Rightarrow$$
$$on(u, v, do(move(x, y, z), s))$$

Explanation:

A Frame Axiom states, what remains true or unaffected, when an action is performed.

In the example here: a block u, which is not the one moved, remains where it is, i.e. on (u, v) is still valid after the action.

Frame Axioms - move-example

positive frame axiom

$$on\ (u, v, s) \wedge x \neq u \Rightarrow$$
$$on\ (u, v, do\ (move\ (x, y, z), s))$$

If a block u is on another block v, and u is not the block being moved, then it stays on v.

negative frame axiom

$$\neg on\ (u, v, s) \wedge (x \neq u \vee y \neq v) \Rightarrow$$
$$\neg on\ (u, v, do\ (move\ (x, y, z), s))$$

If a block u is not on another block v, and u is not moved, or nothing is put on v, then u will still not be on v after the move.

Sit Calc Axioms in GOLOG

Axioms for Actions

Actions are specified by providing a certain set of domain-dependent axioms.

These are:

- action precondition axioms

  describe under what conditions an action can occur

  use additional function Poss

- effect axioms

  describe what is changed due to an action

- frame axioms

  describe what remains unchanged, when an action takes place

GOLOG Axioms - Example

$$Poss(a, s) \wedge (\exists r)a = repair(r, x) \supset \neg broken(x, do(a, s)).$$

If a is possible in s, and there is a robot r, such that a is the action that the robot repairs x, then x is not broken after the "robot repairs x action" was done in s.

Precondition Axiom - Example

Action precondition axiom for pickup:

$$Poss\ (pickup\ (x),\ s) \equiv \forall x.\ \neg Holding\ (x,\ s) \land NextTo\ (x,\ s) \land \neg Heavy\ (x)$$

Effect Axiom - Examples

Effect axioms for drop, explode, repair:

$$Poss(drop(r,x),s) \wedge fragile(x,s) \supset broken(x, do(drop(r,x),s))$$

$$Poss(explode(b),s) \wedge nexto(b,x,s) \supset broken(x, do(explode(b),s))$$

$$Poss(repair(r,x),s) \supset \neg broken(x, do(repair(r,x),s))$$

Frame Axiom - Example

Frame axioms for drop:

$$Poss(drop(r,x),s) \wedge colour(y,s) = c \supset colour(y, do(drop(r,x),s)) = c.$$

The Frame-Problem

- There can be a large number of frame axioms necessary to describe a domain.
- This complicates the task of axiomatising a domain and makes planning or reasoning in situation calculus (theorem proving) extremely inefficient.
- This is the famous Frame Problem.

Successor-State Axioms

Collect all the effect axioms which affect a given fluent. Assume that they specify all of the ways that the value of the fluent can change. Then apply a syntactic transformation to the effect axioms to obtain a successor state axiom for the fluent.

successor-state-axioms:
combine frame and effect axioms;
specified for each fluent - action pair

Successor-State Axioms

general structure

predicate expression is true in follow state ⇔
the action made it true
<u>or</u>
it was true and the action did not make it false.

$$
\begin{aligned}
Poss(a,s) \supset [broken(x, do(a,s)) \equiv \\
&(\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \vee \\
&(\exists b)\{a = explode(b) \wedge nexto(b,x,s)\} \vee \\
&broken(x,s) \wedge \neg(\exists r)a = repair(r,x)].
\end{aligned}
$$

How to Derive Successor-State Axioms?

$$Poss(a, s) \wedge (\exists r) a = repair(r, x) \supset \neg broken(x, do(a, s)).$$

<u>Effect Axioms Schema</u>:

a action; s situation; F fluent; γ condition for F to become true (false) for a in s.

$$Poss(a, s) \ \wedge \ \gamma_F^+(\vec{x}, a, s) \ \supset \ F(\vec{x}, do(a, s))$$

$$Poss(a, s) \ \wedge \ \gamma_F^-(\vec{x}, a, s) \ \supset \ \neg F(\vec{x}, do(a, s))$$

<u>General Successor State Axiom</u>:

$$Poss\,(a, s) \ \supset$$

$$[F(\vec{x}, do(a, s)) \ \equiv \ \gamma_F^+(\vec{x}, a, s) \ \vee \ (F(\vec{x}, s) \ \wedge \ \neg\gamma_F^-(\vec{x}, a, s))]$$

General and Specific Successor State Axiom

$$Poss(a,s) \supset$$
$$\boxed{[F(\vec{x}, do(a,s))} \equiv \boxed{\gamma_F^+(\vec{x}, a, s)} \vee \boxed{(F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s))]}$$

$$\begin{aligned} Poss(a,s) \supset \boxed{[broken(x, do(a,s))} \equiv \\ \boxed{\begin{aligned}&(\exists r)\{a = drop(r,x) \wedge fragile(x,s)\} \vee \\ &(\exists b)\{a = explode(b) \wedge nexto(b,x,s)\}\end{aligned}} \vee \\ \boxed{broken(x,s) \wedge \neg(\exists r)a = repair(r,x)]}\end{aligned}$$

Situation Calculus Axioms - so far

Effect axioms describe how an action changes a situation, when the action is performed.

Frame axioms describe, what remains unchanged between situations.

Successor-state axioms combine effect and frame axioms.

Add domain knowledge!

General Axioms

General axioms

Describe formulas, which are true in all situations.

Example:

$$\forall x, y, s: on(x, y, s) \wedge \neg(y = Table) \Rightarrow \neg clear(y, s)$$

For all situations s and all objects x and y: if something is on object y in s, and y is not the table, then y is not clear in s.

$$\forall s: clear(Table, s)$$

The table (or floor) is always clear.

Domain Modelling in Sit Calc

A particular domain of application will be specified by a theory in the following form:

Axioms describing the initial situation, S_0.

Action precondition axioms, one for each primitive action a, characterizing $Poss(a, s)$.

Successor state axioms, one for each fluent F, stating under what conditions $F(\vec{x}, do(a, s))$ holds as function of what holds in situation s.

Unique names axioms for the primitive actions.

Some foundational, domain independent axioms.

Frame-Problem

Frame-Problem

specify everything which remains stable

Leads to too many conditions which would have to be explicitly stated for any state transformation. Computationally very expensive.

Approach: successor-state axioms; STRIPS

Qualification-Problem

Qualification-Problem

specify everything which is precondition to an action

Difficult to include every precondition, which could prevent (if not fulfilled) the action to be performed.

Approach: non-monotonic reasoning with standard preconditions and effects as defaults.

Ramification-Problem

Ramification-Problem

conflict between change and frame for derived formulas

Some axioms state conclusions about fluents indirectly affected by actions. This can contradict frame-axioms.

Example: An agent grabs an object and holds it. When the agent moves, the object moves too (domain model), though this is not explicitly stated (not an effect axiom). Normally, objects are supposed to stay, where they are (frame-axiom).

Frame: every object stays where it is unless it is moved.

Domain: if an object is attached to another object and one of the objects moves, the other object moves too.

Approach: Integrate TMS for derived formulae.

Planning

Situation Calculus and Planning

Planning starts with a specified start situation and the specification of a goal situation.

Planning comprises of finding a proof which infers the goal situation from the start situation using successor-state and other axioms.

For example, prove S' = at (A, L) from S0 = at (A, S0)

A Plan can be read from the proof: it is the sequence of actions causing the sequence of transformations of situations from the initial situation to the final situation.

$$do(putdown(A), do(walk(L), do(pickup(A), S_0)))$$

$$[\mathrm{pickup(A),\ walk(L),\ putdown(A)}]$$

GOLOG

Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin and Richard Scherl, Golog: A logic programming language for dynamic domains, *Journal of Logic Programming,* **31**, 59-84, (1997).

M. Shanmugasundaram, Presentation in 74.757, 2004.

Golog

- Golog is a kind of logic programming language for reasoning with actions, based on situation calculus.
  - Golog → "alGOL in LOGic"
- It allows in addition to express and reason with more complex action structures, like:

- **if** $car\_in\_driveway$ **then** $drive$ **else** $walk$ **endIf**
- **while** $(\exists block)\ ontable(block)$ **do** $remove\_a\_block$ **endWhile**
- **proc** $remove\_a\_block\ (\pi x)[pickup(x); putaway(x)]$ **endProc**

Golog - Basics

- Complex action expressions are defined using additional extralogical symbols (e.g., *while*, *if*, etc.), which act as abbreviations for logical expressions in the language of the situation calculus.
- These extralogical expressions are like macros, which expand into genuine formulas of the situation calculus.
- The abbreviation $Do(\delta, s, s')$ is the most basic abbreviation used in the Golog language, where δ is a complex action expression.
- $Do(\delta, s, s')$ means that executing δ in situation s has s' as a legal terminating situation.
- Complex actions may be nondeterministic, i.e. they may have several different executions terminating in different situations.

Golog - Definitions 1

Do is defined inductively for the structure of its first argument:

1. Primitive actions:

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

2. Test actions:

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

3. Sequence:

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

Golog - Definitions 2

4. Nondeterministic choice of two actions:

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

5. Nondeterministic choice of action arguments:

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

6. Nondeterministic iteration:

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

Golog - Conditionals

- Conditionals and while loops are defined in terms of the above constructs as follows:

$$\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endIf} \stackrel{def}{=} [\phi? \,;\, \delta_1] \mid [\neg\phi? \,;\, \delta_2],$$

$$\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile} \stackrel{def}{=} [[\phi? \,;\, \delta]^* \,;\, \neg\phi?].$$

Golog - Conditionals

- Procedures are hard to define in situation calculus semantics using macro expansion, because there is no straightforward way to expand a procedure body, when that body includes a recursive call to itself.
- Use an auxiliary macro definition for any predicate symbol P of arity n+2, taking a pair of situation arguments:

$$Do(P(t_1, \ldots, t_n), s, s') \stackrel{def}{=} P(t_1[s], \ldots, t_n[s], s, s').$$

Golog - Procedures

- Semantics of procedures: A Golog program follows the block-structured programming style. A program of the form

$$\textbf{proc } P_1\ (\vec{v}_1)\ \delta_1 \textbf{ endProc } ;\ \cdots\ ;\ \textbf{proc } P_n\ (\vec{v}_n)\ \delta_n \textbf{ endProc } ;\ \delta_0$$

will then be evaluated as:

$$\begin{aligned} &Do(\{\textbf{proc } P_1\ (\vec{v}_1)\ \delta_1 \textbf{ endProc } ;\ \cdots\ ;\ \textbf{proc } P_n\ (\vec{v}_n)\ \delta_n \textbf{ endProc } ;\ \delta_0\}, s, s') \\ &\quad \stackrel{def}{=} (\forall P_1, \ldots, P_n).[\bigwedge_{i=1}^{n} (\forall s_1, s_2, \vec{v}_i).Do(\delta_i, s_1, s_2) \supset Do(P_i(\vec{v}_i), s_1, s_2)] \\ &\qquad \supset Do(\delta_0, s, s'). \end{aligned}$$

Golog - Blocks World Example

A blocks world program to make a seven block tower with block A clear in the final situation.

proc *maketower* (n) % Make a tower of n blocks.

$(\pi\, x, m)[tower(x,m)?\ ;$ % $tower(x,m)$ means that there is a tower % of m blocks, whose top block is x.

if $m \leq n$ **then** $stack(x, n-m)$

else $unstack(x, m-n)$

endIf]

endProc ;

proc *stack* (x, n) % Place n blocks on the tower whose top block is x.

$n = 0? \mid (\pi\, y)[put(y,x)\ ;\ stack(y, n-1)]$

endProc ;

proc *unstack* (x, n) % Remove n blocks from the tower % whose top block is x.

$n = 0? \mid (\pi\, y)[on(x,y)?\ ;\ movetotable(x)\ ;\ unstack(y, n-1)]$

endProc ;

% main: create a seven block tower, with A clear at the end.

$maketower(7)\ ; \neg(\exists x) on(x, A)?$

Programming in / Planning with Golog

- Golog programs are "executed" using theorem proving.
- Program execution means, given a program δ and an initial situation s_0, find a terminating situation s for δ, if one exists.
- To do so, we prove the termination of δ as:

$$Axioms \models (\forall s_0)(\exists s) Do(\delta, s_0, s).$$

and then extract from the proof a binding for the terminating situation.

Elevator Controller in GOLOG

Primitive actions:

- $up(n)$ – Move the elevator up to floor n.
- $down(n)$ – Move the elevator down to floor n.
- $turnoff(n)$ – Turn off call button n.
- $open$ – Open the elevator door.
- $close$ – Close the elevator door.

Fluents:

- $current\_floor(s) = n$ – In situation s, the elevator is at floor n.
- $on(n, s)$ – In situation s, call button n is on.
- $next\_floor(n, s)$ – In situation s, the next floor to be served is n.

GOLOG - Elevator Controller

Primitive action preconditions:

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

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

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

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

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

Successor state axioms:

$$\begin{aligned} Poss(a, s) \supset [current\_floor(do(a, s)) = m \equiv \{a = up(m) \vee a = down(m) \vee \\ current\_floor(s) = m \wedge \neg(\exists n) a = up(n) \wedge \neg(\exists n) a = down(n)\}]. \end{aligned}$$

$$Poss(a, s) \supset [on(m, do(a, s)) \equiv on(m, s) \wedge a \neq turnoff(m)].$$

GOLOG - Elevator Controller

The next floor (to be served) is the nearest floor to the floor, where the elevators is now, in s.

A defined fluent.

$$\begin{aligned} next\_floor(n,s) \equiv\ & on(n,s) \wedge \\ & (\forall m).on(m,s) \supset |m - current\_floor(s)| \geq |n - current\_floor(s)|. \end{aligned}$$

GOLOG-Procedures for Elevator

$$\textbf{proc } serve(n) \quad go\_floor(n)\ ;\ turnoff(n)\ ;\ open\ ;\ close\ \textbf{endProc.}$$

$$\textbf{proc } go\_floor(n) \quad (current\_floor = n)? \ \mid\ up(n) \ \mid\ down(n)\ \textbf{endProc.}$$

$$\textbf{proc } serve\_a\_floor\ (\pi\ n)[next\_floor(n)?\ ;\ serve(n)]\ \textbf{endProc.}$$

$$\textbf{proc } control\ [\textbf{while } (\exists n) on(n)\ \textbf{do } serve\_a\_floor\ \textbf{endWhile}]\ ;\ park\ \textbf{endProc.}$$

$$\textbf{proc } park\ \textbf{if } current\_floor = 0\ \textbf{then } open\ \textbf{else } down(0)\ ;\ open\ \textbf{endIf}\ \textbf{endProc.}$$

GOLOG - Running the Elevator

Intial State

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

"Running the Elevator Program"

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

Find situation s

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

and collect matching action sequence:

$$[down(3), turnoff(3), open, close, up(5), turnoff(5), open, close, down(0), open],$$

Elevator Controller - Initial and Final Situation

- The initial situation axiom specifies that, initially buttons 3 and 5 are on, and moreover no other buttons are on. Thus, we have complete information initially about which call buttons are on.

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

- A successful proof for the elevator program, for example, may return the following binding for *s*:

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

Elevator Controller - The Plan

- This example shows that Golog is a logic programming language in the following sense:
  - Its interpreter is a general purpose theorem prover.
  - Like Prolog, Golog programs are executed to obtain bindings for the existentially quantified variables of the theorem.

Golog - Planning as Theorem Proving

- Running a program is a theorem proving task, which establishes the following entailment:

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

The meaning of this entailment:

- Do is a macro and not a predicate, and the expression stands for a much longer situation calculus sentence.
- We seek a proof of this macro-expanded sentence from axioms, which characterise the fluents and actions of the domain.
- The execution trace represented by this binding is passed as solution to the elevator's execution module, which uses it for controlling the elevator in the physical world.

References

- Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin and Richard Scherl, Golog: A logic programming language for dynamic domains, *Journal of Logic Programming,* **31**, 59-84, (1997).

Extensions to Golog

Golog - Extensions

- Golog is a sophisticated logic programming language for implementing applications in dynamic domains.
- But Golog lacks or neglects some important features.
  - Sensing and knowledge
  - Exogenous actions
  - Concurrency and reactivity
  - Continuous processes
- The following slides show some extensions of Golog.

ConGolog

- ConGolog is a concurrent programming language based on the situation calculus
- The language includes facilities for prioritizing the execution of concurrent processes, interrupting the execution when certain conditions become true, and dealing with exogenous actions.
- ConGolog differs from other formal models of concurrency in at least two ways. First, it allows incomplete information about the environment. Second, it allows the primitive actions to affect the environment in a complex way and such changes to the environment can affect the execution of the remainder of the program.

ConGolog - Semantics

- By using *Do*, programs are assigned a semantics in terms of a relation, denoted by the formula *Do(δ, s, s')*, which means that a given program *δ* and a situation *s* returns a situation *s'* resulting from executing *δ* starting in the situation *s*.
- Semantics of this form are called *evaluation semantics*, since they are based on the complete evaluation of the program.
- To allow concurrency, it is more convenient to adopt a different form of semantics, so-called *transition semantics* or *computation semantics*.
- Transition semantics are based on defining single steps of computation in contrast to directly defining complete computations.

ConGolog (contd…)

- For this two predicates are defined: *Trans(δ, s, δ', s')* and *Final(δ, s).*
- *Trans(δ, s, δ', s')* holds, if there is a transition from *configuration (δ, s)* to the *configuration (δ', s')*, i.e. if by running program *δ* starting in situation *s,* one can get to situation *s'* in one elemantary step with the program *δ'* remaining to be executed.
- Every elementary step will either be the execution of an atomic action (which changes the situation) or the execution of a test (which does not change the situation).
- Also, if the program is nondeterministic, there are several transitions that are possible in a configuration.

ConGolog (contd…)

- *Final(δ, s)* means that the configuration (*δ, s)* is final; the computation is completed, i.e. no part of the program remains to be executed.
- The final situations reached after a finite number of transitions from a starting situation coincide with those satisfying the *Do* relation.
- Complete computations are thus defined by repeatedly composing single transitions until a final configuration is reached.
- With *Trans* and *Final*, a new definition of *Do* can be given as follows:

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

ConGolog (contd…)

ConGolog is an extended version of Golog that incorporates a rich account of concurrency.

It is rich because it handles:

- Concurrent processes with possibly different priorities
- High-level interrupts
- Arbitrary exogenous actions (something happening outside of the GOLOG-agent)

Concurrent processes are modelled as interleavings of the primitive actions in the component processes.

An important concept is that of a process being blocked.

ConGolog (contd…)

- The ConGolog language has the following constructs:

$$
\begin{array}{lr}
\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, & \text{synchronized conditional} \\
\textbf{while } \phi \textbf{ do } \delta, & \text{synchronized loop} \\
(\delta_1 \parallel \delta_2), & \text{concurrent execution} \\
(\delta_1 \rangle\!\rangle \delta_2), & \text{concurrency with different priorities} \\
\delta^{\parallel}, & \text{concurrent iteration} \\
< \phi \rightarrow \delta >, & \text{interrupt.}
\end{array}
$$

- Exogenous actions:

$$\delta_{EXO} \stackrel{def}{=} (\pi\, a.\, Exo(a)?; a)^*$$

$$\delta \parallel \delta_{EXO}$$

cc-Golog

- cc-Golog is an action language which incorporates continuous change and event-driven behaviour.
- It is used in high-level robot controllers, which often need to specify event-driven behaviour and operate low-level processes that change the world in a continuous fashion.
- Main characteristics of cc-Golog program:
  - Timing of actions is largely *event-driven* thereby providing a reactive behaviour.
  - Actions are executed *as soon as possible*.
  - Conditions change *continuously* over time.
  - Good *blocking policies*.

cc-Golog (contd..)

- Event-driven behaviour is achieved by including a special action *waitFor(τ).*
- Continuous change is incorporated through *continuous fluents*, which are functional fluents whose values range over functions of time.
- Blocking policies are specified by means of a special instruction *withCtrl(φ,σ).*
- Note: cc-Golog only provides deterministic instructions.

IndiGolog

- IndiGolog is an action language, which provides nondeterminism and integrates sensing actions.
- While the Golog interpreter works off-line, Indigolog programs are executed on-line by means of an incremental interpreter.
- The initial state of the world is incompletely specified and the agent or robot must use sensors to determine values of certain fluents.
- Nondeterminism is taken care of by means of an *off-line lookahead search operator Σ.*

Golex

- The field of autonomous mobile robots lacks methods that bridge the gap between high-level symbolic techniques and low-level robot control and navigation systems.
- Golex is an execution and monitoring system with the purpose of bridging the gap between Golog and the complex, distributed RHINO control software.
- Golex provides the following features:
  - High level of abstraction
  - Execution monitoring
  - Sensing and Interaction

pGolog

- Actions of a robot are often best thought of as low level processes with uncertain outcomes.
- A high level robot plan is then a task, that combines the low level processes in an appropriate way and may involve nondeterminism.
- The robot needs to turn a given plan into an executable program through some form of projection such that it satisfies a given goal with a sufficiently high probability.
- This is achieved through pGolog, a probabilistic variant of Golog, whose programs model the low-level processes.
- High-level plans are ordinary Golog programs, except that during projection the names of low-level processes are replaced by their pGolog definitions.