Golog semantics

Golog/ConGolog programs are syntactic objects.

How do we assign a formal semantics to them?

Let us first consider Golog only.

For simplicity we will not consider procedures, but see [DLL-AIJ00,LRLLS97].

Golog semantics (cont.)

We start by considering a single model of the SitCalc action theory. (That is we start by assuming complete information, just as in normal computer programs)

Any idea of what the semantics should talk about?

Evaluation semantics: intro

Idea: describe the overall result of the evaluation of the Golog program.

Given a Golog program δ *and a situation* s **compute the situation** s' **obtained by executing** δ **in** s.

More formally: Define the **relation**:

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

where δ is a program, s is the situation in which the program is evaluated, and s' is the situation obtained by the evaluation.

Such a relation can be defined inductively in a standard way using the so called **evaluation (structural) rules**

Evaluation semantics: references

The general approach we follows is is the *structural operational semantics* approach[Plotkin81, Nielson&Nielson99].

This whole-computation semantics is often call: *evaluation semantics* or *natural semantics* or *computation semantic*.

Evaluation rules for Golog: deterministic constructs

$$Act: \quad \frac{(a, s) \longrightarrow do(a[s], s)}{true} \quad \text{if } Poss(a[s], s)$$

$$Test: \quad \frac{(\phi?, s) \longrightarrow s}{true} \quad \text{if } \phi[s]$$

$$Seq: \quad \frac{(\delta_1; \delta_2,\ s) \longrightarrow s'}{(\delta_1, s) \longrightarrow s'' \ \wedge\ (\delta_2, s') \longrightarrow s'}$$

$$if: \quad \frac{(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{else } \delta_2, s) \longrightarrow s'}{(\delta_1, s) \longrightarrow s'} \quad \text{if } \phi[s] \qquad \frac{(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{else } \delta_2, s) \longrightarrow s'}{(\delta_2, s) \longrightarrow s'} \quad \text{if } \neg\phi[s]$$

$$while: \quad \frac{(\textbf{while } \phi \textbf{ do } \delta, s) \longrightarrow s}{true} \quad \text{if } \phi[s] \qquad \frac{(\textbf{while } \phi \textbf{ do } \delta, s) \longrightarrow s'}{(\delta, s) \longrightarrow s'' \ \wedge\ (\textbf{while } \phi \textbf{ do } \delta\ s'') \longrightarrow s'} \quad \text{if } \neg$$

Evaluation rules: nondeterministic constructs

$$Nondetbranch: \quad \frac{(\delta_1 \mid \delta_2,\ s) \longrightarrow s'}{(\delta_1, s) \longrightarrow s'} \qquad \frac{(\delta_1 \mid \delta_2,\ s) \longrightarrow s'}{(\delta_2, s) \longrightarrow s'}$$

$$Nondetchoice: \quad \frac{(\pi\, x.\ \delta(x), s) \longrightarrow s'}{(\delta(t), s) \longrightarrow s'}$$ *(for any t)*

$$Nondetiter: \quad \frac{(\delta^*, s) \longrightarrow s}{true} \qquad \frac{(\delta^*, s) \longrightarrow s'}{(\delta, s) \longrightarrow s'' \wedge (\delta^*, s'') \longrightarrow s'}$$

Structural rules

The structural rules have the following schema:

$$\frac{\textsc{consequent}}{\textsc{antecedent}} \quad \text{if } \textsc{side-condition}$$

which is to be interpreted logically as:

$$\forall(\textsc{antecedent} \land \textsc{side-condition} \supset \textsc{consequent})$$

where $\forall Q$ stands for the universal closure of all free variables occurring in Q, and, typically, ANTECEDENT, SIDE-CONDITION and CONSEQUENT share free variables.

Given a model of the SitCalc action theory, the structural rules define inductively a relation, namely: **the smallest relation satisfying the rules**.

Examples

Compute the following assuming actions are always possible:

- $(a; b, S_0) \longrightarrow s_f$
- $((a \mid b); c, S_0) \longrightarrow s_f$
- $((a \mid b); c; P?, S_0) \longrightarrow s_f$ where P true iff a is not performed yet.

Getting logical

Till now we have defined the relation $(\delta, s) \longrightarrow s'$ in a single model of the SitCalc action theory of interest.

But what about if the action theory has incomplete information and hence admits several models?

Idea: *Define a logical predicate* $Do(\delta, s, s')$ *starting from the definition of the relation* $(\delta, s) \longrightarrow s'$.

Definition of Do: intro

How: *do we define a logical predicate* $Do(\delta, s, s')$ *starting from the definition of the relation* $(\delta, s) \longrightarrow s'$*?*

- Rules correspond to logical conditions;

- The minimal predicate satisfying the rules is expressible in 2nd-order logic by using the formulas of the following form:

$\forall D.\{$

logical formulas corresponding to the rules

that use the **predicate variable** D in place of the relation

$\} \supset D(\delta, s, s').$

Definition of Do

$$
\begin{aligned}
&Do(\delta, s, s') \;\equiv \\
&\quad \forall D.\{ \\
&\qquad \forall[\; Poss(a[s], s) \;\supset\; D(a, s, do(a[s], s)) \;] \;\wedge \\
&\qquad \forall[\; \phi[s] \;\supset\; D(\phi?, s, s) \;] \;\wedge \\
&\qquad \forall[\; D(\delta_1, s, s'') \wedge D(\delta_2, s'', s') \;\supset\; D(\delta_1;\delta_2,\; s, s')] \;] \;\wedge \\
&\qquad \forall[\; \phi[s] \wedge D(\delta_1, s, s') \;\vee\; \neg\phi[s] \wedge D(\delta_2, s, s')] \;\supset\; D(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{else } \delta_2, s, s') \;] \;\wedge \\
&\qquad \forall[\; \phi[s] \wedge s' = s \;\vee\; \neg\phi[s] \wedge D(\delta_2, s, s') \wedge D(\textbf{while } \phi \textbf{ do } \delta, s, s') \;\supset\; D(\textbf{while } \phi \textbf{ do } \delta, s, s') \;] \;\wedge \\
&\qquad \forall[\; D(\delta_1, s, s') \;\vee\; D(\delta_2, s'', s') \;\supset\; D(\delta_1 \mid \delta_2,\; s, s') \;] \;\wedge \\
&\qquad \forall[\; D(\delta(t), s, s') \;\supset\; D(\pi\, x.\, \delta(x), s, s') \;] \;\wedge \\
&\qquad \forall[\; s' = s \;\vee\; D(\delta, s, s'') \wedge D(\delta^*, s'', s') \;\supset\; D(\delta^*, s, s') \;] \;\wedge \\
&\quad \} \;\supset\; D(\delta, s, s').
\end{aligned}
$$

Examples

Assuming the action theory Γ does not logically implies $Poss(a, S_0)$, but all other actions are possible, find all s_f that constitute (certain) executions of the programs seen before, i.e., such that the following logical implication holds:

- $\Gamma \models Do(a; c,\ S_0,\ s_f)$
- $\Gamma \models Do((a \mid b); c,\ S_0,\ s_f)$
- $\Gamma \models Do((a \mid b); c; P?,\ S_0,\ s_f)$ where P holds iff a is not performed yet.

Original Definition of Do

In [LRLLS97], $Do(\delta, s, s')$ is defined by induction on the structure of the program instead of using structural rules as above.

The main advantage of this definition is that $Do(\delta, s, s')$ can be is simply viewed as an abbreviation for a formula of the SitCalc.

Programs do not even need to be formally introduced!!!

Original Definition of Do (cont.)

$$Act: \quad Do(a, s, s') \stackrel{def}{=} Poss(a[s], s) \land s' = do(a[s], s)$$

$$Test: \quad Do(\phi?, s, s') \stackrel{def}{=} \phi[s] \land s = s'$$

$$Seq: \quad Do(\delta_1; \delta_2,\ s, s') \stackrel{def}{=} \exists s''.\ Do(\delta_1, s, s'') \land Do(\delta_2, s'', s')$$

$$Nondetbranch: \quad Do(\delta_1 \mid \delta_2,\ s, s') \stackrel{def}{=} Do(\delta_1, s, s') \lor Do(\delta_2, s, s')$$

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

$Nondetiter:$ It is not definable in 1st-order logic! ...

Original Definition of Do (cont. 2)

Nondeterministic iteration:

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

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, s) is in the set for all situations s.

2. Whenever (s, s'') is in the set, and doing δ in situation s'' takes you to situation s', then (s, s'') is in the set.

Must use 2nd-order logic because transitive closure is not 1st-order definable.

And concurrency?

Unfortunately evaluation semantics does not extend to construct for concurrency.

We need a finer form of semantics, namely **Transition Semantics**, where we specify what executing a **single step** of the program amounts to.

Transition semantics: intro

Idea: describe the result of executing a **single step** of the Golog program.

- *Given a Golog program* δ *and a situation* s **compute the situation** s' **and the program** δ' **that remains to be executed obtained by executing a single step of** δ **in** s.

- *Assert when a Golog program* δ *can be considered* **successfully terminated** *in a situation* s.

Transition semantics: intro

More formally:

- Define the **relation**, named $Trans$ and denoted by "$\longrightarrow$"):

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

  where δ is a program, s is the situation in which the program is executed, and s' is the situation obtained by executing a single step of δ and δ' is what remains to be executed of δ after such a single step.

- Define a **predicate**. named $Final$ and denoted by "$\surd$":

$$(\delta, s)^{\surd}$$

  where δ is a program that can be considered (successfully) terminated in the situation s.

Such a relation and predicate can be defined inductively in a standard way, using the so called **transition (structural) rules**

Transition semantics: references

The general approach we follows is is the *structural operational semantics* approach[Plotkin81, Nielson&Nielson99].

This single-step semantics is often call: *transition semantics* or *computation semantics*.

Transition rules for Golog: deterministic constructs

$$Act: \quad \frac{(a,s) \longrightarrow (nil, do(a[s],s))}{true} \quad \text{if } Poss(a[s],s)$$

$$Test: \quad \frac{(\phi?,s) \longrightarrow (nil,s)}{true} \quad \text{if } \phi[s]$$

$$Seq: \quad \frac{(\delta_1;\delta_2,\ s) \longrightarrow (\delta_1';\delta_2,s')}{(\delta_1,s) \longrightarrow (\delta_1';s')} \qquad \frac{(\delta_1;\delta_2,\ s) \longrightarrow (\delta_2',s')}{(\delta_2,s) \longrightarrow (\delta_2';s')} \quad \text{if } (\delta_1,s)\surd$$

$$if: \quad \frac{(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{else } \delta_2,s) \longrightarrow (\delta_1',s')}{(\delta_1,s) \longrightarrow (\delta_1',s')} \quad \text{if } \phi[s] \qquad \frac{(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{else } \delta_2,s) \longrightarrow (\delta_2',s')}{(\delta_2,s) \longrightarrow (\delta_2',s')} \quad \text{if } \neg\phi[s]$$

$$while: \quad \frac{(\textbf{while } \phi \textbf{ do } \delta,s) \longrightarrow (\delta';\textbf{while } \phi \textbf{ do } \delta,s)}{(\delta,s) \longrightarrow (\delta',s')} \quad \text{if } \phi[s]$$

Termination rules for Golog: deterministic constructs

$$Nil: \quad \frac{(nil, s)^{\surd}}{true}$$

$$Seq: \quad \frac{(\delta_1; \delta_2,\ s)^{\surd}}{(\delta_1, s)^{\surd}\ \wedge\ (\delta_2; s)^{\surd}}$$

$$if: \quad \frac{(\textbf{if}\ \phi\ \textbf{then}\ \delta_1 \textbf{else}\ \delta_2, s)^{\surd}}{(\delta_1, s)^{\surd}}\ \text{if}\ \phi[s] \qquad \frac{(\textbf{if}\ \phi\ \textbf{then}\ \delta_1 \textbf{else}\ \delta_2, s)^{\surd}}{(\delta_2, s)^{\surd}}\ \text{if}\ \neg\phi[s]$$

$$while: \quad \frac{(\textbf{while}\ \phi\ \textbf{do}\ \delta, s)^{\surd}}{true}\ \text{if}\ \neg\phi[s] \qquad \frac{(\textbf{while}\ \phi\ \textbf{do}\ \delta, s)^{\surd}}{(\delta, s)^{\surd}}\ \text{if}\ \phi[s]$$

Transition rules: nondeterministic constructs

$$Nondetbranch: \quad \frac{(\delta_1 \mid \delta_2,\, s) \longrightarrow (\delta_1', s')}{(\delta_1, s) \longrightarrow (\delta_1', s')} \qquad \frac{(\delta_1 \mid \delta_2,\, s) \longrightarrow (\delta_2', s')}{(\delta_2, s) \longrightarrow (\delta_2', s')}$$

$$Nondetchoice: \quad \frac{(\pi\, x.\ \delta(x), s) \longrightarrow (\delta'(t), s')}{(\delta(t), s) \longrightarrow (\delta'(t), s')}$$ *(for any t)*

$$Nondetiter: \quad \frac{(\delta^*, s) \longrightarrow (\delta'; \delta^*, s')}{(\delta, s) \longrightarrow (\delta', s')}$$

Termination rules: nondeterministic constructs

$$Nondetbranch: \qquad \frac{(\delta_1 \mid \delta_2,\ s)^{\surd}}{(\delta_1, s)^{\surd} \ \vee \ (\delta_2, s)^{\surd}}$$

$$Nondetchoice: \qquad \frac{(\pi\, x.\ \delta(x), s)^{\surd}}{(\delta(t), s)^{\surd}} \quad \textit{(for some } t\textit{)}$$

$$Nondetiter: \qquad \frac{(\delta^*, s)^{\surd}}{true}$$

Structural rules

The structural rules have the following schema:

$$\frac{\textsc{consequent}}{\textsc{antecedent}} \quad \text{if } \textsc{side-condition}$$

which is to be interpreted logically as:

$$\forall(\textsc{antecedent} \wedge \textsc{side-condition} \supset \textsc{consequent})$$

where $\forall Q$ stands for the universal closure of all free variables occurring in Q, and, typically, ANTECEDENT, SIDE-CONDITION and CONSEQUENT share free variables.

Given a model of the SitCalc action theory, the structural rules define inductively a relation, namely: **the smallest relation satisfying the rules**.

Examples

Compute the following assuming actions are always possible:

- $(a; b, S_0) \longrightarrow (nil; b, do(a, S_0)) \longrightarrow (nil, do(b(do(a, S_0)))$
- $((a \mid b); c, S_0) \longrightarrow ???$
- $((a \mid b); c; P?, S_0) \longrightarrow ???$
- $(a; (b \mid c), S_0) \longrightarrow ???$
- $((a; b \mid a; c), S_0) \longrightarrow ???$

  where P true iff a is not performed yet.

Evaluation vs. transition semantics

How do we characterize a whole computation using single steps?

First we define the relation, named $Trans^*$, denoted by $\longrightarrow^*$ by the following rules:

$$0steps: \quad \frac{(\delta, s)\longrightarrow^*(\delta, s)}{true}$$

$$nsteps: \quad \frac{(\delta, s)\longrightarrow^*(\delta'', s'')}{(\delta, s)\longrightarrow(\delta', s') \ \wedge \ (\delta', s')\longrightarrow^*(\delta'', s'')} \quad \textit{(for some } \delta', s'\textit{)}$$

Then it can be shown that:

$$\begin{aligned}(\delta, s_0) \longrightarrow s_f \equiv \\ (\delta, s_0)\longrightarrow^*(\delta_f, s_0) \ \wedge \ (\delta_f, s_f)^{\surd} \quad \text{for some } \delta_f\end{aligned}$$

Getting logical

Till now we have defined the relation $(\delta, s) \longrightarrow (\delta', s')$ and the predicate $(\delta, s)^{\surd}$ in a single model of the SitCalc action theory of interest.

But what about if the action theory has incomplete information and hence admits several models?

Idea: *Define a logical predicates* $Trans(\delta, s, \delta', s')$ *and* $Final(\delta, s)$ *starting from the definitions of the relation* $(\delta, s) \longrightarrow (\delta', s')$*, and* $(\delta, s)^{\surd}$*.*

Definition of Do: intro

How: *do we define a logical predicate* $Trans(\delta, s, \delta', s')$ *starting from the definition of the relation* $(\delta, s) \longrightarrow (\delta', s')$*?* and the predicate $(\delta, s)\surd$.

- Rules correspond to logical conditions;

- The minimal predicate satisfying the rules is expressible in 2nd-order logic by using the formulas of the following form (for $Trans$, similarly for $Final$):

$$\forall T.\{$$

logical formulas corresponding to the rules

that use the **predicate variable** T in place of the relation

$$\} \supset T(\delta, s, \delta', s').$$

Definition of Trans

$Trans(\delta, s, \delta', s') \;\equiv\; \forall T.[\;\ldots\; \supset T(\delta, s, \delta', s')]$, where ... stands for the conjunction of the universal closure of the following implications:

$$
\begin{array}{rcl}
Poss(a[s], s) & \supset & T(a, s, nil, do(a[s], s)) \\
\phi[s] & \supset & T(\phi?, s, nil, s) \\
T(\delta, s, \delta', s') & \supset & T(\delta;\gamma, s, \delta';\gamma, s') \\
Final(\gamma, s) \wedge T(\delta, s, \delta', s') & \supset & T(\gamma;\delta, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\delta \mid \gamma, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\gamma \mid \delta, s, \delta', s') \\
T(\delta^v_x, s, \delta', s') & \supset & T(\pi v.\delta, s, \delta', s') \\
T(\delta, s, \delta', s') & \supset & T(\delta^*, s, \delta';\delta^*, s') \\
T(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s, \delta', s') & \supset & T(\{Env;\delta\}, s, \delta', s') \\
T(\{Env;\delta_P{}^{\vec{v}_P}_{\vec{t}[s]}\}, s, \delta', s') & \supset & T([Env : P(\vec{t})], s, \delta', s')
\end{array}
$$

Definition of Final

$Final(\delta, s) \;\equiv\; \forall F.[\;\ldots\;\supset\; F(\delta, s)]$, where ... stands for the conjunction of the universal closure of the following implications:

$$
\begin{array}{rcl}
True & \supset & F(nil, s) \\
F(\delta, s) \wedge F(\gamma, s) & \supset & F(\delta; \gamma, s) \\
F(\delta, s) & \supset & F(\delta \mid \gamma, s) \\
F(\delta, s) & \supset & F(\gamma \mid \delta, s) \\
F(\delta^v_x, s) & \supset & F(\pi v.\delta, s) \\
True & \supset & F(\delta^*, s) \\
F(\delta^{P_i(\vec{t})}_{[Env:P_i(\vec{t})]}, s) & \supset & F(\{Env; \delta\}, s) \\
F(\{Env; \delta_{P}{}^{\vec{v}_P}_{\vec{t}[s]}, s) & \supset & F([Env : P(\vec{t})], s)
\end{array}
$$

Concurrency

ConGolog is an extension of Golog that incorporates a rich account of concurrency:

- concurrent processes,
- priorities,
- high-level interrupts.

We model concurrent processes by **interleaving**: *A concurrent execution of two processes is one where the primitive actions in both processes occur, interleaved in some fashion.*

It is OK for a process to remain **blocked** for a while, the other processes will continue and eventually unblock it.

Congolog

The ConGolog language is exactly like Golog except with the following additional constructs:

if ϕ **then** δ_1 **else** δ_2,	synchronized conditional
while ϕ **do** δ,	synchronized loop
$(\delta_1 \parallel \delta_2)$,	concurrent execution
$(\delta_1 \rangle\!\rangle \delta_2)$,	concurrency with different priorities
$\delta^{\parallel}$,	concurrent iteration
$< \phi \rightarrow \delta >$,	interrupt.

The constructs **if** ϕ **then** δ_1 **else** δ_2 and **while** ϕ **do** δ are the synchronized: *testing the condition* ϕ *does not involve a transition per se, the evaluation of the condition and the first action of the branch chosen are executed as an atomic unit.*

Similar to test-and-set atomic instructions used to build semaphores in concurrent programming.

Transition rules: concurrency

$$Conc: \quad \frac{(\delta_1 \parallel \delta_2,\ s) \longrightarrow (\delta_1' \parallel \delta_2, s')}{(\delta_1, s) \longrightarrow (\delta_1', s')} \qquad \frac{(\delta_1 \parallel \delta_2,\ s) \longrightarrow (\delta_1 \parallel \delta_2', s')}{(\delta_2, s) \longrightarrow (\delta_2', s')}$$

$$PriorConc: \quad \frac{(\delta_1 \rangle\!\rangle\ \delta_2,\ s) \longrightarrow (\delta_1' \rangle\!\rangle\ \delta_2, s')}{(\delta_1, s) \longrightarrow (\delta_1', s')} \qquad \frac{(\delta_1 \rangle\!\rangle\ \delta_2,\ s) \longrightarrow (\delta_1 \rangle\!\rangle\ \delta_2', s')}{(\delta_2, s) \longrightarrow (\delta_2', s')\ \wedge\ (\delta_1, s) \not\longrightarrow}$$

$$IterConc: \quad \frac{(\delta^{\parallel}, s) \longrightarrow (\delta' \parallel \delta^{\parallel}, s')}{(\delta, s) \longrightarrow (\delta', s')}$$

$$Interrupts: \quad \frac{(<\phi \rightarrow \delta>, s) \longrightarrow (\delta'; <\phi \rightarrow \delta>, s')}{(\delta, s) \longrightarrow (\delta', s')} \quad \text{if } \phi[s]\ \wedge\ Interrups\_running[s]$$

Termination rules: concurrency

$$Conc: \quad \frac{(\delta_1 \parallel \delta_2,\ s)^{\surd}}{(\delta_1, s)^{\surd} \ \wedge \ (\delta_2, s)^{\surd}}$$

$$PrioConc: \quad \frac{(\delta_1 \rangle\!\rangle\ \delta_2,\ s)^{\surd}}{(\delta_1, s)^{\surd} \ \wedge \ (\delta_2, s)^{\surd}}$$

$$IterConc: \quad \frac{(\delta^{\parallel}, s)^{\surd}}{true}$$

$$Interrupts: \quad \frac{(<\phi \rightarrow \delta>, s)^{\surd}}{true} \quad \text{if } \neg Interrups\_running[s]$$

ConGolog Transition Semantics (cont.)

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

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, ϕ is a term that denotes a formula, and $\phi[s]$ stands for $Holds(\phi, s)$, which is true iff the formula denoted by ϕ is true in s. Details are in [DLL00].

ConGolog Transition Semantics (cont.)

$$Trans(\delta^*, s, \delta, s') \equiv \exists \delta'.\delta = (\delta'; \delta^*) \wedge Trans(\delta, s, \delta', s')$$

$$Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta, s') \equiv$$
$$\quad \phi(s) \wedge Trans(\delta_1, s, \delta, s') \vee \neg\phi(s) \wedge Trans(\delta_2, s, \delta, s')$$

$$Trans(\textbf{while } \phi \textbf{ do } \delta, s, \delta', s') \equiv \phi(s) \wedge$$
$$\quad \exists \delta''.\ \delta' = (\delta''; \textbf{while } \phi \textbf{ do } \delta) \wedge Trans(\delta, s, \delta'', s')$$

$$Trans([\delta_1 \parallel \delta_2], s, \delta, s') \equiv \exists \delta'.$$
$$\quad \delta = (\delta' \parallel \delta_2) \wedge Trans(\delta_1, s, \delta', s') \vee$$
$$\quad \delta = (\delta_1 \parallel \delta') \wedge Trans(\delta_2, s, \delta', s')$$

$$Trans([\delta_1 \rangle\!\rangle \delta_2], s, \delta, s') \equiv \exists \delta'.$$
$$\quad \delta = (\delta' \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \delta', s') \vee$$
$$\quad \delta = (\delta_1 \rangle\!\rangle \delta') \wedge Trans(\delta_2, s, \delta', s') \wedge$$
$$\qquad \neg\exists \delta'', s''.Trans(\delta_1, s, \delta'', s'')$$

$$Trans(\delta^{\parallel}, s, \delta', s') \equiv$$
$$\quad \exists \delta''.\delta' = (\delta'' \parallel \delta^{\parallel}) \wedge Trans(\delta, s, \delta'', s')$$

ConGolog Transition Semantics (cont.)

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

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 $Final(\delta, s')$.

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

where the ellipsis stands for:

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

Induction principles

From such definitions, natural “induction principles” emerge:

These are principles saying that to prove that a property P holds for instances of $Trans$ and $Final$, it suffices to prove that the property P is closed under the assertions in the definition of $Trans$ and $Final$, i.e.:

$$\Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \;\equiv\; P(\delta_1, s_1, \delta_2, s_2)$$
$$\Phi_{Final}(P, \delta_1, s_1) \;\equiv\; P(\delta_1, s_1)$$

Theorem: The following sentences are consequences of the second-order definitions of $Trans$ and $Final$ respectively:

$$\begin{array}{l} \forall P.[\forall \delta_1, s_1, \delta_2, s_2.\, \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \;\supset \\ \qquad \forall \delta, s, \delta', s'.\, Trans(\delta, s, \delta', s') \;\supset\; P(\delta, s, \delta', s') \end{array}$$

$$\begin{array}{l} \forall P.[\forall \delta_1, s_1.\, \Phi_{Final}(P, \delta_1, s_1) \equiv P(\delta_1, s_1)] \;\supset \\ \qquad \forall \delta, s.\, Final(\delta, s, \delta', s') \;\supset\; P(\delta, s) \end{array}$$

Proof

We prove only the first sentence. The proof of the second sentence is analogous.

By definition we have:

$$\begin{aligned}
&\forall \delta, s, \delta', s'. Trans(\delta, s, \delta', s') \;\; \equiv \\
&\qquad \forall P.[\forall \delta_1, s_1, \delta_2, s_2.\, \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \\
&\qquad\quad \supset \; P(\delta, s, \delta', s')
\end{aligned}$$

By considering the only-if part of the above equivalence, we get:

$$\begin{aligned}
&\forall \delta, s, \delta', s'. Trans(\delta, s, \delta', s') \;\wedge \\
&\qquad \forall P.[\forall \delta_1, s_1, \delta_2, s_2.\, \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \\
&\qquad \supset \; P(\delta, s, \delta', s')
\end{aligned}$$

So moving the quantifiers around we get:

$$\begin{aligned}
&\forall P.[\forall \delta_1, s_1, \delta_2, s_2.\, \Phi_{Trans}(P, \delta_1, s_1, \delta_2, s_2) \equiv P(\delta_1, s_1, \delta_2, s_2)] \;\wedge \\
&\qquad \forall \delta, s, \delta', s'. Trans(\delta, s, \delta', s') \\
&\qquad \supset \; P(\delta, s, \delta', s')
\end{aligned}$$

and hence the thesis.

Bisimulation

Bisimulation is a relation $\sim$ satisfing the condition:

$$
\begin{array}{l}
(\delta_1, s_1) \sim (\delta_2, s_2) \;\supset \\
\qquad (\delta_1, s_1)^{\surd} \equiv (\delta_2, s_2)^{\surd} \;\; \wedge \\
\qquad \forall (\delta_1', s_1').(\delta_1, s_1) \longrightarrow (\delta_1', s_1') \supset \\
\qquad\qquad \exists (\delta_2', s_2').(\delta_2, s_2) \longrightarrow (\delta_2', s_2') \;\wedge\; (\delta_1', s_1') \sim (\delta_2', s_2') \;\; \wedge \\
\qquad \forall (\delta_2', s_2').(\delta_2, s_2) \longrightarrow (\delta_2', s_2') \supset \\
\qquad\qquad \exists (\delta_1', s_1').(\delta_1, s_1) \longrightarrow (\delta_1', s_1') \;\wedge\; (\delta_2', s_2') \sim (\delta_1', s_1')
\end{array}
$$

(δ_1, s_1) and (δ_2, s_2) are **bisimilar** if there **exists a bisimulation** between the two.

Note: it can be shown that bisimilarity is an equivalence relation.