Situation Calculus

Reasoning about action and agent programming

Sebastian Sardina
Department of Computer Science and Information Technology
RMIT University
Melbourne, AUSTRALIA

www.cs.toronto.edu/cogrobo

Ray Reiter
Hector Levesque
Outline

1. Introduction

2. Situation Calculus: Reasoning about actions

3. Golog: Offline High-Level Programming

4. ConGolog: Reactivity and Concurrency

5. IndiGolog: Interleaved Execution

6. Conclusions
Outline

1. Introduction
2. Situation Calculus: Reasoning about actions
3. Golog: Offline High-Level Programming
4. ConGolog: Reactivity and Concurrency
5. IndiGolog: Interleaved Execution
6. Conclusions
Cognitive Robotics

The study of the **knowledge representation and reasoning** problems faced by an autonomous agent in a **dynamic and incompletely known** world.

- knowledge of the agent vs. the designer?
- perception vs. reasoning?
- when is initial knowledge adequate to achieve a goal?

Do not want to simply engineer robot controllers that solve a class of problems or that work in a class of application domains.
would like a system that is capable of generating actions in the world that are appropriate, as a function of some current set of beliefs and desires.

**Note:** the descriptions and preferences are inputs (not wired in!)
Classical Planning [SRI Shakey “The Robot” ’70]

- Planing from **first-principle:**
  Goal oriented (usually goals-to-be)

- Dumb executor, little feedback, no reactivity, does not scale!

- Kind of plans $\delta_0$:
  - Linear total-order plans (STRIPS)
  - Partial-order plans (SNLP)
  - Conditional plans (CNLP, S-GP)
  - **Robot programs** [Levesque 96]

- An interesting case: [Sacerdoti 74]
  HTN planning/programming
The Reactive Approach [Brooks 86,91]

- Reactive robots, **headless**!
- Little representation, quick decisions
- But: little or no pro-activity
- Examples:
  - Subsumption architecture [Brooks 86]
  - Universal plans [Schoopers 87]
  - Control Reactive Rules [Baral&Son 98]
  - (PO)MDP and policies?
AGENT = Action Theory + High-Level Program

In *Cognitive Robotics*, we think of an agent equipped with:

- **An action theory**: the agent knows the theory and its consequences (actions’ effects, frame & qualification problems, sensing, etc.)

- **A high-level program**: specifying the agent tasks/behaviors (nondeterministic & domain actions)
AGENT = Action Theory + High-Level Program

In Cognitive Robotics, we think of an agent equipped with:

- **An action theory**: the agent knows the theory and its consequences (actions’ effects, frame & qualification problems, sensing, etc.)

- **A high-level program**: specifying the agent tasks/behaviors (nondeterministic & domain actions)

\[
\text{getOnFlight} \overset{\text{def}}{=} (\pi a.a)^*; \text{At}(airport)\)?
\]

\[
(go(\text{terminal1}) \mid go(\text{terminal2}));
\]

\[
(buy(\text{magazine}) \mid buy(\text{newspaper}));
\]

\[
\text{if } (\text{GateNo} \geq 90) \text{ then } \{go(\text{gate}(\text{GateNo})); buy(\text{coffee})\}
\]

\[
\text{else } \{buy(\text{coffee}); go(\text{gate}(\text{GateNo}))\}
\]

\[
\text{boardPlane}
\]
High-Level Programs

Programs telling the robot what needs to be done:

- primitive statements are actions to be performed by the robot:
  \[ \Rightarrow \text{move to the desk and then pick up the package} \]

- tests within a program pertain to conditions in the world that are affected by the actions of the robot (or other agents):
  \[ \Rightarrow \text{if the door is locked then ... otherwise ...} \]
  \[ \Rightarrow \text{while there is a package on the table do ...} \]

- programs may be non-deterministic: they may contain choice points where the interpreter must make a reasoned (non-random) selection:
  \[ \Rightarrow \text{go through the (appropriate) door and retrieve the package that is waiting} \]
  \[ \Rightarrow \text{either turn left or turn right (as appropriate) ... at which point you must be located in the hall} \]
High-Level Programs

Programs telling the robot what needs to be done:

- primitive statements are actions to be performed by the robot:
  \[
  \implies \text{move to the desk and then pick up the package}
  \]

- tests within a program pertain to conditions in the world that are affected by the actions of the robot (or other agents):
  \[
  \implies \text{if the door is locked then ... otherwise ...}
  \]
  \[
  \implies \text{while there is a package on the table do ...}
  \]

- programs may be non-deterministic: they may contain choice points where the interpreter must make a reasoned (non-random) selection:
  \[
  \implies \text{go through the (appropriate) door and retrieve the package that is waiting}
  \]
  \[
  \implies \text{either turn left or turn right (as appropriate) ... at which point you must be located in the hall}
  \]
High-Level Programs

Programs telling the robot what needs to be done:

- primitive statements are actions to be performed by the robot:
  \[ \text{move to the desk and then pick up the package} \]

- tests within a program pertain to conditions in the world that are affected by the actions of the robot (or other agents):
  \[ \text{if the door is locked then} \ldots \text{otherwise} \ldots \]
  \[ \text{while there is a package on the table do} \ldots \]

- programs may be non-deterministic: they may contain choice points where the interpreter must make a reasoned (non-random) selection:
  \[ \text{go through the (appropriate) door and retrieve the package that is waiting} \]
  \[ \text{either turn left or turn right (as appropriate) \ldots at which point you must be located in the hall} \]
The Agent Programming Approach I

- Planning with *procedural* clues
  - Goals-to-do
- Less complexity! But still hard!
- Little reactive and feedback.
- Examples:
  - **GoLog**, **ConGoLog**, **DTGoLog**
  - MDPs + HAM [Parr 98]
  - HTN-planning [Eral et. al 94]
The Agent Programming Architecture II [IndiGoLog]

- High-level programs executed online or incrementally
- Planning is just a sub-module
- Control over extended periods
- Reactive: can adapt
- Proactive: plan locally
- Can handle runtime sensing
We will talk about...

1. Action theories in the situation calculus.
2. Offline execution of high-level programs.
3. Online execution of high-level programs and local planning.
Outline

1. Introduction

2. Situation Calculus: Reasoning about actions

3. GOLOG: Offline High-Level Programming

4. CONGOLOG: Reactivity and Concurrency

5. INDIGOLOG: Interleaved Execution

6. Conclusions
The description of the initial state, and the prerequisites and effects of actions can be formulated in the language of the *situation calculus*:

1. A dialect of the predicate calculus with special sorts for:
   - **actions**: \( closeDoor \), \( pickUp(package_{13}) \), \( moveFwd(2) \);
   - **situations**: \( S_0 \), the initial situation,
     \( do(a, s) \), where \( a \) is an action term and \( s \) is a situation term.

2. Predicates or functions whose value changes from situation to situation are called **fluents**:
   
   \[
   \neg Holding(x, S_0) \land Holding(x, do(pickUp(x), S_0))
   \]

3. Use a distinguished predicate \( Poss(a, s) \) to assert that action \( a \) is possible to execute in situation \( s \).
Basic Action Theories [Lin and Reiter 94]

Theory $\mathcal{D} = \Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{Poss} \cup \mathcal{D}_{SSA} \cup \mathcal{D}_{UNA}$, where:

- $\mathcal{D}_{S_0}$: axioms describing the initial state of the fluents in $S_0$; $\implies$ allows for incomplete knowledge!
- $\mathcal{D}_{poss}$: axioms describing the prerequisites of the primitive actions, of the form:

$$Poss(A(\vec{x}), s) \equiv \Theta_A(\vec{x}, s),$$

one for each primitive action $A$.
- $\mathcal{D}_{ssa}$: axioms describing the effects and non-effects of actions, of the form:

$$F(\vec{x}, do(a, s)) \equiv \psi_F(\vec{x}, a, s),$$

one for each fluent $F$.
- $\mathcal{D}_{una}$: unique name axioms for the actions.
- $\Sigma$: some foundational domain-independent axioms.
A solution to the frame problem (sometimes) [Reiter 91]

1. Positive effect axioms:

   \[ \text{Holding}(x, s) \land \text{Fragile}(x, s) \supset \text{Broken}(x, \text{do(drop}(x), s)) \]

2. Positive effect axioms:

   \[ \text{HasGlue}(s) \land \text{Broken}(x, s) \supset \neg \text{Broken}(x, \text{do(repair}(x), s)) \]

3. Frame axiom axioms:

   \[ \neg \text{Broken}(x, s) \land (y \neq x \lor \neg \text{Fragile}(x, s)) \supset \neg \text{Broken}(x, \text{do(drop}(y), s)) \]

4. Completeness assumption:

   \[ \neg \text{Broken}(x, s) \land \text{Broken}(x, \text{do}(a, s)) \supset \text{Fragile}(x, s) \land a = \text{drop}(x) \]

Successor state axioms = positive effect axioms + positive effect axioms + frame axioms + completeness assumption
A solution to the frame problem (sometimes) [Reiter 91]

So, suppose that for fluent $F(\vec{x}, s)$:

$\gamma_F^+(\vec{x}, a, s)$ encodes all positive effects; and

$\gamma_F^-(\vec{x}, a, s)$ encodes all negative effects.

The successor state axiom solution has the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \lor F(\vec{x}, s) \land \neg \gamma_F^-(\vec{x}, a, s)$$

In our example:
A solution to the frame problem (sometimes) [Reiter 91]

So, suppose that for fluent $F(\vec{x}, s)$:

$\gamma^+_F(\vec{x}, a, s)$ encodes all *positive* effects; and
$\gamma^-_F(\vec{x}, a, s)$ encodes all *negative* effects.

The *successor state axiom* solution has the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma^+_F(\vec{x}, a, s) \lor F(\vec{x}, s) \land \neg \gamma^-_F(\vec{x}, a, s)$$

In our example:

$$\gamma^+_{\text{Broken}}(\vec{x}, a, s) \overset{\text{def}}{=} \ldots$$
$$\gamma^-_{\text{Broken}}(\vec{x}, a, s) \overset{\text{def}}{=} \ldots$$

$$\text{Broken}(\vec{x}, do(a, s)) \equiv$$

$$\gamma^+_{\text{Broken}}(\vec{x}, a, s) \lor$$
$$\text{Broken}(\vec{x}, s) \land \neg \gamma^-_{\text{Broken}}(\vec{x}, a, s)$$
A solution to the frame problem (sometimes) [Reiter 91]

So, suppose that for fluent $F(\vec{x}, s)$:

- $\gamma_F^+(\vec{x}, a, s)$ encodes all positive effects; and
- $\gamma_F^-(\vec{x}, a, s)$ encodes all negative effects.

The successor state axiom solution has the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \lor F(\vec{x}, s) \land \neg \gamma_F^-(\vec{x}, a, s)$$

In our example:

$$\gamma_{\text{Broken}}^+(\vec{x}, a, s) \overset{\text{def}}{=} a = \text{drop}(x) \land \text{Holding}(x, s) \land \text{Fragile}(x, s)$$

$$\gamma_{\text{Broken}}^-(\vec{x}, a, s) \overset{\text{def}}{=} \ldots$$

$$\text{Broken}(x, do(a, s)) \equiv$$

$$[\text{Holding}(x, s) \land \text{Fragile}(x, s) \land a = \text{drop}(x)] \lor$$

$$\text{Broken}(x, s) \land \neg \gamma_{\text{Broken}}^-(\vec{x}, a, s)$$
A solution to the frame problem (sometimes) [Reiter 91]

So, suppose that for fluent $F(\vec{x}, s)$:

$\gamma_F^+(\vec{x}, a, s)$ encodes all *positive* effects; and

$\gamma_F^-(\vec{x}, a, s)$ encodes all *negative* effects.

The **successor state axiom** solution has the following form:

$$F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \lor F(\vec{x}, s) \land \lnot \gamma_F^-(\vec{x}, a, s)$$

In our example:

$$\gamma_{\text{Broken}}^+(\vec{x}, a, s) \overset{\text{def}}{=} a = \text{drop}(x) \land \text{Holding}(x, s) \land \text{Fragile}(x, s)$$

$$\gamma_{\text{Broken}}^-(\vec{x}, a, s) \overset{\text{def}}{=} a = \text{repair}(x) \land \text{HasGlue}(x, s)$$

$$\text{Broken}(x, do(a, s)) \equiv$$

$$[\text{Holding}(x, s) \land \text{Fragile}(x, s) \land a = \text{drop}(x)] \lor$$

$$\text{Broken}(x, s) \land \lnot[a = \text{repair}(x) \land \text{HasGlue}(s)]$$
Reasoning about actions

- **Projection task:** given a sequence of actions $a_1, \ldots, a_n$, does condition $\phi(s)$ hold?:
  \[ \mathcal{D} \models \phi(do(do(a_n, do(a_{n-1}, \ldots, do(a_1, S_0)) \ldots)) \]

- **Deductive planning:** find a sequence of actions to make $\phi(s)$ true:
  \[ \mathcal{D} \vdash \exists s. \phi(s) \land \text{executable}(s) \]

- **Database transaction formalization:**
  \[ \mathcal{D} \models \exists c. \text{Enrolled}(john, c, do(reg(mary, c10), do(drop(john, c21), S_0))) \]

- **Inductive proofs:**
  \[ \mathcal{D} \models \forall p, n, n'. s \sqsubseteq s' \land \text{sal}(p, n, s) \land \text{sal}(p, n', s') \supset n \leq n' \]
Reasoning about actions (cont.)

- Knowledge and beliefs: what does the agent knows?
  \[ \phi(s) \text{ vs } \text{Know}(\phi, s). \]
- Explicit time and continuous change:
  \[ \text{start}(s) = t \text{ and } \text{Velocity}(x, s) = 87. \]
- Natural actions: actions that must occur (bouncing of the ball).
- Concurrent actions:
  \[ D \models \neg \text{Spilled(coffee, do\{liftRight(table), liftLeft(table)\}, S_0)}, \]
  \[ D \models \text{Spilled(coffee, do\{liftRight(table)\}, S_0)}. \]
- Ramifications: indirect effects
  \[ D \models \forall s. \text{Inside}(x, y) \supset \text{At}(x, l, \text{do(move(y, l), S_0))}. \]
Reasoning about actions (cont.)

- **Knowledge and beliefs**: what does *the agent* knows?
  \[ \phi(s) \text{ vs } \textbf{Know}(\phi, s). \]

- **Explicit time and continuous change**:
  \[ \text{start}(s) = t \text{ and } \text{Velocity}(x, s) = 87. \]

- **Natural actions**: actions that *must* occur (bouncing of the ball).

- **Concurrent actions**:
  \[ \mathcal{D} \models \neg \textbf{Spilled}(\text{coffee}, \text{do}\{\text{liftRight}(\text{table}), \text{liftLeft}(\text{table})\}, S_0), \]
  \[ \mathcal{D} \models \textbf{Spilled}(\text{coffee}, \text{do}\{\text{liftRight}(\text{table})\}, S_0). \]

- **Ramifications**: indirect effects
  \[ \mathcal{D} \models \forall s. \textbf{Inside}(x, y) \supset \textbf{At}(x, l, \text{do}(\text{move}(y, l), S_0)). \]

**All in plain classical (i.e., predicate) logic!**
Outline

1 Introduction

2 Situation Calculus: Reasoning about actions

3 **Golog**: Offline High-Level Programming

4 **ConGolog**: Reactivity and Concurrency

5 **IndiGolog**: Interleaved Execution

6 Conclusions
**Golog** is a *logic-based* programming language whose primitive actions are those of a background action theory.

It includes the following constructs:

- $\alpha$, *primitive action*
- $\phi?$, *test/wait for a condition*
- $(\delta_1; \delta_2)$, *sequence*
- **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ **endIf**, *conditional*
- **while** $\phi$ **do** $\delta$ **endWhile**, *loop*
- **proc** $\beta(\vec{x})$ $\delta$ **endProc**, *procedure definition*
- $\beta(\vec{t})$, *procedure call*
- $(\delta_1 | \delta_2)$, *nondeterministic choice of action*
- $\pi \vec{x} [\delta]$, *nondet. choice of arguments*
- $\delta^*$, *nondeterministic iteration*
An offline execution of a Golog program $\delta$ relative to an action theory $\mathcal{D}$, is a situation term $S$ such that:

$$\mathcal{D} \models Do(\delta, S_0, S).$$

$Do(\delta, s, s')$: it is possible to reach situation $s'$ from situation $s$ by executing a sequence of actions specified by $\delta$. 

**Golog semantics**
**GOLOG semantics**

An offline execution of a **GOLOG** program $\delta$ relative to an action theory $\mathcal{D}$, is a situation term $S$ such that:

$$\mathcal{D} \models \text{Do}(\delta, S_0, S).$$

$\text{Do}(\delta, s, s')$: *it is possible to reach situation $s'$ from situation $s$ by executing a sequence of actions specified by $\delta$.*

This execution formula can be defined inductively:

- $\text{Do}(a, s, s') \overset{\text{def}}{=} \text{Poss}(a, s) \land s' = \text{do}(a, s)$
- $\text{Do}(\phi, s, s') \overset{\text{def}}{=} \phi(s) \land s' = s$
- $\text{Do}(\delta_1; \delta_2, s, s') \overset{\text{def}}{=} \exists s''. \text{Do}(\delta_1, s, s'') \land \text{Do}(\delta_2, s'', s')$
- $\text{Do}(\delta_1 | \delta_2, s, s') \overset{\text{def}}{=} \text{Do}(\delta_1, s, s') \lor \text{Do}(\delta_2, s, s')$
- $\text{Do}(\pi x.\delta(x), s, s') \overset{\text{def}}{=} \exists x. \text{Do}(\delta(x), s, s')$
GoLog implementation

Do can be lifted directly from its semantic specification:

\[\text{do}([E_1,E_2],S,S_1) \leftarrow \text{do}(E_1,S,S_2), \text{do}(E_2,S_2,S_1).\]
\[\text{do}(\neg(P),S,S) \leftarrow \text{holds}(P,S).\]
\[\text{do}(\text{ndet}(E_1,E_2),S,S_1) \leftarrow \text{do}(E_1,S,S_1) ; \text{do}(E_2,S,S_1).\]
\[\text{do}(\text{if}(P,E_1,E_2),S,S_1) \leftarrow \text{do}(\text{ndet}([\neg(P),E_1],[\neg(-P),E_2]),S,S_1).\]
\[\text{do}(\text{star}(E),S,S_1) \leftarrow S_1 = S ; \text{do}([E,\text{star}(E)],S,S_1).\]
\[\text{do}(\text{while}(P,E),S,S_1) \leftarrow \text{do}([\text{star}([\neg(P),E]),\neg(-P)],S,S_1).\]
\[\text{do}(\text{pi}(V,E),S,S_1) \leftarrow \text{sub}(V,_,E,E_1), \text{do}(E_1,S,S_1).\]
\[\text{do}(E,S,S_1) \leftarrow \text{proc}(E,E_1), \text{do}(E_1,S,S_1).\]
\[\text{do}(E,S,\text{do}(E,S)) \leftarrow \text{prim_action}(E), \text{poss}(E,P), \text{holds}(P,S)\]
\[\text{holds}([P,Q],S) \leftarrow \text{holds}(P,S), \text{holds}(Q,S).\]
\[\ldots\]
Some **Golog** applications

A number of applications have been written in **Golog** and variants:

- a simulated elevator controller;
- mail delivery robots at Toronto and York Universities
  \[\Rightarrow\] demonstrates hardware independence
- a robot museum guide [Burgard et al AAAI 98]
  \[\Rightarrow\] controllable online via the Web
- business process simulation [Yu & Mylopoulos 97]
  \[\Rightarrow\] for analysis of the processes, not execution
- characters in computer animation [Funge SIGGRAPH 99]
  \[\Rightarrow\] for high-level behaviour specification
- a large “softbot” application [Ruman MSc 96]
  personal banking: ATM controller, personal agents, bank
  agents, routers agents run as processes under Unix, communicate
  over TCP/IP
Why is Golog popular?

[ Copied from Ryan Kelly’s presentation, 2005 ]
Why is Golog popular?

[Copied from Ryan Kelly’s presentation, 2005]

- **Good level of abstraction:**
  - Programs based directly on actions from the domain.
  - Nondeterminism makes programs simpler and more powerful.
  - Symbolic reasoning effortlessly available.
Why is Golog popular?

[Copied from Ryan Kelly’s presentation, 2005]

- **Good level of abstraction:**
  - Programs based directly on actions from the domain.
  - Nondeterminism makes programs simpler and more powerful.
  - Symbolic reasoning effortlessly available.

- **Tradeoff between programming and planning:**
  - Amount of nondeterminism controlled by the programmer.
  - Procedural knowledge easy to encode.
  - Full planning still available.
Why is Golog popular?

[Copied from Ryan Kelly’s presentation, 2005]

- **Good level of abstraction:**
  - Programs based directly on actions from the domain.
  - Nondeterminism makes programs simpler and more powerful.
  - Symbolic reasoning effortlessly available.

- **Tradeoff between programming and planning:**
  - Amount of nondeterminism controlled by the programmer.
  - Procedural knowledge easy to encode.
  - Full planning still available.

- **Logic-based:**
  - Compact implementation in Prolog.
  - Possible to prove safety/liveness properties.
Outline

1 Introduction

2 Situation Calculus: Reasoning about actions

3 GoLog: Offline High-Level Programming

4 ConGoLog: Reactivity and Concurrency

5 IndiGoLog: Interleaved Execution

6 Conclusions
What is missing?

A number of concerns are not well addressed by these efforts:

1. sensing and exogenous actions
2. reactivity: suspending some preprogrammed behaviour
   \[\Rightarrow\] e.g., monitoring power consumption
3. reasoned failure recovery
   \[\Rightarrow\] e.g., encountering an unexpected closed door
4. reliable and non-inmediate primitive actions
   \[\Rightarrow\] e.g., \texttt{go\_to}(loc) \textbf{vs.} \texttt{start\_going\_to}(loc) + exog. reports
5. local decision making
   \[\Rightarrow\] e.g., searching through an entire execution \textbf{vs.}
   committing quickly to the first action

These issues are addressed in more recent dialects of \texttt{GoLog}.
Reactivity in ConGolog [de Giacomo et al AIJ 00]

An extension to Golog, which includes concurrency, prioritized interrupts, and exogenous actions.

\[
\langle \text{TooHot} \land \neg \text{FanOn} \rightarrow \text{toggle\_fan} \rangle
\]
\[
\langle \text{TooCold} \land \text{FanOn} \rightarrow \text{toggle\_fan} \rangle
\]
\[
\langle \text{SmokeDetectorOn} \rightarrow \text{ring\_alarm} \rangle
\]
\[
\langle n : \text{ButtonOn}(n) \rightarrow \pi f.?(\text{BestButton}(f)); \text{serve}(f) \rangle
\]
\[
\langle \neg \text{OnFloor}(1) \rightarrow \text{down} \rangle
\]

Allows reactive controllers, but still via reasoning about actions!

- $\delta_1 \parallel \delta_2$: concurrent execution (equal priority)
- $\delta_1 \rangle \delta_2$: concurrent execution ($\delta_1$ with higher priority)
- $\delta \parallel$: concurrent iteration
- $\langle \vec{x} : \phi(\vec{x}) \rightarrow \delta(\vec{x}) \rangle$: interrupt
Semantics of ConGolog: Trans and Final

To characterize ConGolog behaviour, we need a theory of complex actions based on single steps:

1. **Trans**$(\delta, s, \delta', s')$: program $\delta$ can legally execute a single step to $s'$, with $\delta'$ remaining to execute.

2. **Final**$(\delta, s)$: program $\delta$ can legally terminate.

   
   \[
   \text{Trans}(a, s, \delta', s') \equiv \text{Poss}(a, s) \land s' = s \land \delta' = \text{nil}
   \]

   \[
   \text{Trans}(\delta_1 || \delta_2, s, \delta', s') \equiv \exists \delta''. \left( \text{Trans}(\delta_1, s, \delta'', s') \land \delta' = (\delta''; \delta_2) \lor \text{Trans}(\delta_2, s, \delta'', s') \land \delta' = (\delta_1; \delta'') \right)
   \]

   \[
   \text{Final(while } \phi \text{ do } \delta, s) \equiv \neg \phi(s)
   \]

   \[
   \text{Final}(\delta^*, s) \equiv \text{TRUE}
   \]
Semantics of **ConGolog**: *Trans* and *Final*

To characterize **ConGolog** behaviour, we need a theory of complex actions based on single steps:

1. **Trans**($\delta, s, \delta', s'$): program $\delta$ can legally execute a single step to $s'$, with $\delta'$ remaining to execute.

2. **Final**($\delta, s$): program $\delta$ can legally terminate.

   Trans($a, s, \delta', s'$) $\equiv$ Poss($a, s$) $\land$ $s' = s$ $\land$ $\delta' = \text{nil}$

   Trans($\delta_1 || \delta_2, s, \delta', s'$) $\equiv$ $\exists \delta''$. Trans($\delta_1, s, \delta'', s'$) $\land$ $\delta' = (\delta''; \delta_2) \lor$

   Trans($\delta_2, s, \delta'', s'$) $\land$ $\delta' = (\delta_1; \delta'')$

   Final(while $\phi$ do $\delta$, $s$) $\equiv$ $\neg \phi(s)$

   Final($\delta^*, s$) $\equiv$ TRUE

\[ \mathcal{D} \cup \mathcal{C} \models \exists s'. \text{Do}(\delta, s, s'), \]

where $\text{Do}(\delta, s, s') \overset{\text{def}}{=} \exists \delta'. \text{Trans}^*(\delta, s, \delta', s') \land \text{Final}(\delta', s')$
Semantics of \textbf{ConGolog}: \textit{Trans} and \textit{Final}

To characterize \textbf{ConGolog} behaviour, we need a theory of complex actions based on single steps:

1. \textit{Trans}(\delta, s, \delta', s')$: program $\delta$ can legally execute a single step to $s'$, with $\delta'$ remaining to execute.

2. \textit{Final}(\delta, s)$: program $\delta$ can legally terminate.

\begin{align*}
\text{Trans}(a, s, \delta', s') & \equiv \text{Poss}(a, s) \land s' = s \land \delta' = \text{nil} \\
\text{Trans}(\delta_1 \parallel \delta_2, s, \delta', s') & \equiv \exists \delta''. \, \text{Trans}(\delta_1, s, \delta'', s') \land \delta' = (\delta''; \delta_2) \lor \\
& \quad \text{Trans}(\delta_2, s, \delta'', s') \land \delta' = (\delta_1; \delta'')
\end{align*}

\begin{align*}
\text{Final(while } \phi \text{ do } \delta, s) & \equiv \neg \phi(s) \\
\text{Final}(\delta^*, s) & \equiv \text{TRUE}
\end{align*}

$$
\mathcal{D} \cup \mathcal{C} \models \exists s'. \text{Do(} \delta, s, s' \text{)}, \quad \text{*** STILL OFFLINE! ***}
$$

where \text{Do(} \delta, s, s' \text{)} \overset{\text{def}}{=} \exists \delta'. \text{Trans}^*(\delta, s, \delta', s') \land \text{Final}(\delta', s')
Outline

1. Introduction
2. Situation Calculus: Reasoning about actions
3. Golog: Offline High-Level Programming
4. ConGolog: Reactivity and Concurrency
5. IndiGolog: Interleaved Execution
6. Conclusions
To allow sensing or exogenous inputs to help determine what action to perform next, we must execute programs online/incrementally!

We cannot compute an entire execution of a program offline before starting. (This was impractical for large programs anyway!)

Account based on Trans and Final is well-suited to online exec.: 

[Sardina 2000, 2004]
To allow sensing or exogenous inputs to help determine what action to perform next, we must execute programs online/incrementally!

We cannot compute an entire execution of a program offline before starting. (This was impractical for large programs anyway!)

Account based on *Trans* and *Final* is well-suited to online exec.:

find an action $A$ such that $\text{Trans}(\delta, s, \delta', \text{do}(A, s))$ holds, and then commit to it (physically execute it). Repeat.
To allow sensing or exogenous inputs to help determine what action to perform next, we must execute programs online/incrementally!

We cannot compute an entire execution of a program offline before starting. (This was impractical for large programs anyway!)

Account based on $\text{Trans}$ and $\text{Final}$ is well-suited to online exec.:

\[\text{find an action } A \text{ such that } \text{Trans}(\delta, s, \delta', \text{do}(A, s)) \text{ holds, and then commit to it (physically execute it). Repeat.}\]

Top level interpreter for $\text{IndiGoLog}$:

\[
\begin{align*}
\text{indigo}(P,S) & :\text{- exog\_occurs}(A), !, \text{indigo}(P, [A|S]). \\
\text{indigo}(P,S) & :\text{- final}(P,S), !. \\
\text{indigo}(P,S) & :\text{- trans}(P,S,P1,S), !, \text{indigo}(P1,S). \\
\text{indigo}(P,S) & :\text{- trans}(P,S,P1,[A|S]), \text{exec}(A,R), \text{indigo}(P1,[A|S]).
\end{align*}
\]
Local search $\Sigma(\delta)$

Unlike ordinary Golog, IndiGolog handles non-determinism by making a random selection:

*The non-deterministic program $(\delta_1 \mid \delta_2); \ldots; p?$ may fail if the wrong action is selected and executed!*

The solution: extend the language with a new operator $\Sigma$ where $\Sigma(\delta)$ means “take transitions in $\delta$ that will ultimately succeed”:

$$\text{Trans}(\Sigma(\delta), s, \delta', s') \equiv \text{Trans}(\delta, s, \delta', s') \land (\exists s'') \text{Do}(\delta', s', s'').$$

Can control the amount of lookahead search:

$$\Sigma(\delta_1); \delta_2 \ vs \ \Sigma(\delta_1; \delta_2)$$
An example of local planning

\[ \text{getOnFlight}'_{\text{Sketchy}} \stackrel{\text{def}}{=} \]

\[ \Sigma[(\pi a.a)^*; \text{At}(\text{airport})?]; \]
\[ \Sigma[(\text{go}(\text{term1}) \mid \text{go}(\text{term2})); \]
\[ \text{checkDepartures}^*; \quad /* \text{Sensing Action!} */ \]
\[ (\text{buy}(\text{magazine}) \mid \text{buy}(\text{newspaper})); \]
\[ \text{if GateNo} \geq 90 \text{ then } \{ \text{go}(\text{gate}(\text{GateNo})); \text{buy}(\text{coffee}) \} \]
\[ \text{else } \{ \text{buy}(\text{coffee}); \text{go}(\text{gate}(\text{GateNo})) \} \}; \]

\[ \text{boardPlane} \]
Other work in the area

1. **Execution monitoring**: want to compare the results of executing Golog program to what was expected [de Giacomo et al 98]

2. **Limited planning**: would like to be able to introduce limited forms of planning as part of much larger Golog programs.
   use ideas of [Bacchus & Kabanza 96] for filtering

3. **Knowledge/belief and knowledge-base programming**:
   [Moore 85, Scherl & Levesque 92, Shapiro et al 00, Reiter 01]


5. **Plan correctness**: physical + epistemic feasibility [Sardina et al 04,06]
Golem running **GOLOG**

![Map of the Main Office with characters and Golem's location]

**Golem**
Outline

1. Introduction
2. Situation Calculus: Reasoning about actions
3. Golog: Offline High-Level Programming
4. ConGolog: Reactivity and Concurrency
5. IndiGolog: Interleaved Execution
6. Conclusions
Conclusions

Up until very recently, “robot programming” was tackled mainly by engineers and hobbyists:

- required familiarity with workings of sensors and effectors, and
- programming languages very close to hardware.

(not unlike situation with first computers.)

Increasingly, because of:

- more stable robotic platforms, and
- better low-level interfaces and simulations,

it has become possible to consider robot programs that are much more portable and hardware independent.
Conclusions (cont.)

Situation calculus provides:

1. an expressive framework to reason about action and change;
2. several high-level agent programming languages.

Plus:

✓ everything with a clean semantics in classical logic!
✓ elaboration tolerant (many extensions in a parsimonious way).

Robotics by computer scientists!:

- abstraction, encapsulation, modularity;
- semantics / correctness, computability.