Multi-agent programming in IndiGolog

Adrian Pearce

13 July 2011

Includes slides by Ryan Kelly
Agent = action theory + high-level program

- 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/behaviours (nondeterministic & domain actions)
High-Level Programming is a promising approach from single-agent systems:

- Primitive actions from the agents world
- Connected by standard programming constructs
- Containing controlled amounts of nondeterminism
- Agent plans a ”Legal Execution”
- e.g. GOLOG

Vision: the cooperative execution of a shared high-level program by a team of autonomous agents.
Golog (Revisited)

- $a$ - Perform a primitive action
- $\delta_1; \delta_2$ - Perform two programs in sequence
- $\phi? - Assert that a condition holds
- $\delta_1|\delta_2$ - Choose between programs to execute
- $\pi(x, \delta(x))$ - Choose suitable bindings for variables
- $\delta^*$ - Execute a program zero or more times
- $\delta_1||\delta_2$ - Execute programs concurrently

Key Point: programs can include nondeterminism
Why High-Level Programming?

- Natural, flexible task specification
- Powerful nondeterminism control
  - order of actions, who does what, ...
- Sophisticated logic of action
  - Concurrent actions, continuous actions, explicit time, ...

Ferrein, Lakemeyer et.al. have successfully controlled a RoboCup team using a Golog variant called ”ReadyLog” (Ferrein, Fritz and Lakemeyer 2005).
Why is Golog popular?

- 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
Golog for Multiple Agents?

The "Golog Family" includes:

- Original GOLOG
- ConGolog: interleaved concurrency
- IndiGolog: online execution

MIndiGolog facilitates this approach in multi-agent domains:

- Robust integration of *true concurrency*
- Explicit temporal component
- Seamless integration of *natural actions*
IndiGolog operators (Revisited)

IndiGolog introduces a larger range of operators such as:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Execute action $a$ in the world</td>
</tr>
<tr>
<td>$\phi?$</td>
<td>Proceed if condition $\phi$ is true</td>
</tr>
<tr>
<td>$\delta_1; \delta_2$</td>
<td>Execute $\delta_1$ followed by $\delta_2$</td>
</tr>
<tr>
<td>$\delta_1</td>
<td>\delta_2$</td>
</tr>
<tr>
<td>$\pi(x)\delta(x)$</td>
<td>Nondet. select arguments for $\delta$</td>
</tr>
<tr>
<td>$\delta^*$</td>
<td>Execute $\delta$ zero or more times</td>
</tr>
<tr>
<td>if $\phi$ then $\delta_1$ else $\delta_2$</td>
<td>Exec. $\delta_1$ if $\phi$ holds, $\delta_2$ otherwise</td>
</tr>
<tr>
<td>while $\phi$ do $\delta$</td>
<td>Execute $\delta$ while $\phi$ holds</td>
</tr>
<tr>
<td>proc $P(\overrightarrow{x})\delta(\overrightarrow{x})end$</td>
<td>Procedure definition</td>
</tr>
<tr>
<td>$\delta_1</td>
<td></td>
</tr>
<tr>
<td>$\Sigma \delta$</td>
<td>Plan execution offline (IndiGolog)</td>
</tr>
</tbody>
</table>
Consider a Golog program for getting to university of a morning:

\[
\text{ringAlarm}; (\text{hitSnooze}; \text{ringAlarm})^*; \text{turnOffAlarm;} \\
\pi(\text{food}, \text{edible(food)}?; \text{eat(food)}); (\text{haveShower} \parallel \text{brushTeeth}) \\
(\text{driveToUni} \mid \text{trainToUni}); (\text{time} < 11:00)\]

There are potentially many ways to execute this program, depending on which actions are possible in the world.

Use theory of action to plan a \textit{Legal Execution}:

\[
\mathcal{D} \models \exists s, \delta': \text{Trans}^*(\delta, S_0, \delta', s) \land \text{Final}(\delta', s)
\]
For asynchronous multi-agent domains, we must handle:

- Concurrent Actions:  \( do(\{a_1, a_2\}, s) \)
- Continuous time:  \( do(c, t, s) \)
- Long-running tasks:  \( begin(t), \) \( doing(t, s), \) \( end(t) \)
- Natural processes:  \( Legal(a, s) \rightarrow \neg \exists n : nat(n) \land Poss(n, s) \)
- Incomplete knowledge (from last lecture):  \( \text{Knows}(\phi, s) \)
Motivating Example: The Cooking Agents

Several robotic chefs inhabit a kitchen, along with various ingredients, appliances and utensils. They must cooperate to produce a meal consisting of several dishes.

```plaintext
proc MakeSalad(bowl) 
    ChopTypeInto(Lettuce, bowl) || 
    ChopTypeInto(Carrot, bowl) || 
    ChopTypeInto(Tomato, bowl) ; 
    π(agt, Mix(agt, bowl, 1)) 
end

proc ChopTypeInto(type, dest) 
    π((agt, obj), 
    IsType(obj, type)? ; 
    Chop(agt, obj) ; 
    PlaceIn(agt, obj, dest)) 
end
```
MIndiGolog (Multi-agent IndiGolog)

Application:

- Agents cooperate to plan and perform the execution of a shared Golog program

Modifications to Golog

- Merge concurrent actions with concurrent program execution
- Integrate time and natural actions for coordination
- Share planning workload using distributed logic programming
MIndiGolog Semantics

One approach (used in TeamGolog, Farrinelli et al. 2006) defines concurrent execution of the individual agent's programs:

\[ \delta = \delta_{agt1} \parallel \delta_{agt2} \parallel \cdots \parallel \delta_{agtN} \]

In another approach (used in ReadyLog) has all agents cooperate to plan and perform the joint execution of a single, shared program:

\[ \delta = \delta_{task1} \parallel \delta_{task2} \parallel \cdots \parallel \delta_{taskN} \]

MIndiGolog takes the second approach
Algorithm for multiple agents

**Algorithm:** ReadyLog

\[ \sigma \leftarrow S_0 \]

**while** \( D \cup D_{\text{golog}} \not\equiv \text{Final}(\delta, s) \) **do**

Find an action \( a \) and program \( \delta' \) such that:

\[ D \cup D_{\text{golog}} \models Trans^*(\delta, \sigma, \delta', do(a, \sigma)) \]

**if** the action is to be performed by me then

Execute the action \( a \)

**else**

Wait for the action to be executed

**end if**

\[ \sigma \leftarrow do(a, \sigma) \]

\[ \delta \leftarrow \delta' \]

**end while**
Using such an algorithm, the agents can prepare several dishes concurrently

\[\text{MakeSalad()} \parallel \text{MakePasta()} \parallel \text{MakeCake()}\]

They can even plan to have different dishes ready at different times:
\[\text{[MakeSalad()} \parallel \text{MakePasta()}]; \ ?(\text{time} < 7:30)\]
\[\parallel (\text{MakeCake()}; \ ?(8:15 < \text{time} < 8:30))\]
We modify the original transition rule

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

Modifying this to use CONCURRENT\#TIMEPOINT pairs and \textit{Legal} gives

\[ \text{Trans}(a, s, \delta', s') \equiv \exists t : \text{Legal}({a}\#t, s) \land \delta' = \text{Nil} \land s' = \text{do}({a}, s) \]

This ensures that the temporal component respects the ordering between predecessor and successor situations.
Example Output: MakeSalad()
do [release(harriet,board2), end_task(richard,chop(board1))] at _J
do [release(harriet,bowl1)] at _I
do [acquire(richard,bowl1)] at _H
do [transfer(richard,board1,bowl1)] at _G
do [release(richard,board1)] at _F
do [release(richard,bowl1)] at _E
do [acquire(thomas,bowl1)] at _D
do [begin_task(thomas,mix(bowl1,1))] at _C
do [end_task(thomas,mix(bowl1,1))] at _B
do [release(thomas,bowl1)] at _A
.>=.(_U,0),
.<=.(_U,_T),
.<=.(_L,-5+_J),
.<=.(_D,-1+_B),
.=.(_Q,3+_R)
...

Can get concurrency

using: MakeSalad(Bowl1) || MakePasta(Bowl2) || MakeCake(Bowl3)
Agents should take advantage of true concurrency. Basic idea:

\[
\text{Trans}(\delta_1 \| \delta_2, s, \delta', s') \equiv \exists \gamma : \text{Trans}(\delta_1, s, \gamma, s') \land \delta' = (\gamma \| \delta_2) \\
\lor \exists \gamma : \text{Trans}(\delta_2, s, \gamma, s') \land \delta' = (\delta_1 \| \gamma) \\
\lor \exists c_1, c_2, \gamma_1, \gamma_2, t : \text{Trans}(\delta_1, s, \gamma, \text{do}(c_1 \# t, s)) \\
\land \text{Trans}(\delta_2, s, \gamma_2, \text{do}(c_2 \# t, s)) \land \text{Legal}((c_1 \cup c_2) \# t, s) \land \forall a : [a \in c_1 \land a \in c_2] \\
\land \delta' = (\gamma_1 \| \gamma_2) \land s' = \text{do}((c_1 \cup c_2) \# t, s)
\]
The combination of actions \((c_1 \cup c_2)\) may not be possible.
- Must check this explicitly

The same *agent-initiated* action mustn’t *Trans* both programs.
- otherwise dangerous ‘skipping’ of actions can occur
- if two concurrent programs both call for \(pay(Ryan, \$100)\) to be performed, it had better be performed twice!
- Natural actions can transition both programs
Consider two programs both wanting to initiate agent actions:

\[ \delta_1 = \text{placeIn}(Jim, Flour, Bowl); \text{placeIn}(Jim, Sugar, Bowl) \]
\[ \delta_2 = \text{placeIn}(Jim, Flour, Bowl); \text{placeIn}(Jim, Egg, Bowl) \]

Executing \( \delta_1 || \delta_2 \) should result in the bowl containing two units of flour, one unit of sugar and an egg.

However, an individual transition for both programs is
\[ c_1 = c_2 = \{ \text{placeIn}(Jim, Four, Bowl) \} \].

Naively executing \( c_1 \cup c_2 \) to transition both programs would result in only one unit of flour being added.
Consider two programs waiting for a timer to ring:

\[ \delta_1 = \text{ringTimer; acquire(Jim, Bowl)} \]

\[ \delta_2 = \text{ringTimer; acquire(Joe, Bowl)} \]

Both programs should be allowed to proceed using the same (natural) \textit{ringTimer} occurrence.
Least natural time point (LNTP)

- Natural actions have been previously utilised in Golog (Pirri and Reiter 2000)
- However, the programmer was typically required to explicitly required to check for them and ensure that they appear in the execution
- We lower the burden on the programmer by guaranteeing that all legal program executions result in legal situations - inserting natural actions into the execution when they are predicted to occur (see page 51 of Kelly 2009)
Distributed Execution

- Agents can each plan a legal execution individually
- Identical search strategy produces identical results
- Coordination without communication!
- Requires a fully observable, completely known world

But, we can also take advantage of communication to share the planning workload between agents.
The semantics of Golog can be neatly encoded as a logic program. Prolog is traditionally used. We have also used Oz for its strong distributed programming support.

```
proc {Trans D S Dp Sp}
    case D of nil then fail
    [] test(C) then {Holds.yes C S} Sp=S Dp=nil
    [] pick(D1 D2) then choice
        {Trans D1 S Dp Sp}
        [] {Trans D2 S Dp Sp}
    end
    [] ... <additional cases omitted> ...
end
end```
Using the built in ParallelSearch object, the agents can transparently share the planning workload:

```proc {ParallelMIndiGolog D S}
    PSearch={New Search.parallel
        init(richard:1#ssh thomas:1#ssh harriet:1#ssh)}
    in
        S={PSearch one(MIndiGolog D $)}
end```

MIndiGolog: a Golog semantics and implementation for shared program execution by a team of cooperating agents:

- Safely taking advantage of true concurrency
- Automatically accounting for predictable environment behaviour
- Using distributed logic programming to share the workload (page 60, Kelly 2009)
Joint Executions

The Golog execution planning process produces a *situation* representing a legal execution of the program.

This is a *linear* and *fully-ordered* sequence of actions, demanding total synchronicity during execution.

Multiple agents should be able to execute independent actions independently.

- need a *partially-ordered* representation
Prime event structures are a canonical representation for partially-ordered branching sequences of events:

- A set of events, $\mathcal{V}$
- A partial order on events, $e_1 \prec e_2$
- A conflict relation, $e_1 \# e_2$
- A labelling function, $\gamma(e) = lbl$

Define *enablers* and *alternatives* as follows:

- $j \in \text{ens}(i) \equiv j \prec i \land \forall k \in \text{ens}(i) : \neg(j \prec k)$
- $j \in \text{alts}(i) \equiv j \# i \land \forall k \in \text{ens}(i) : \neg(j \# k)$
Joint Executions

- pickup(T, knife1)
  - T: \{pickup(T, knife1)\}
  - H: \{pickup(T, knife1)\}

- lookfor(H, tomato)
  - T: \{\}
  - H: \{lookfor(H, tomato)\#yes\}

- give(H, T, tomato)
  - T: \{\}
  - H: \{lookfor(H, tomato)\#no\}

- give(H, T, lettuce)

- chop(T, tomato)
  - T: \{give(H, T, tomato)\}
  - H: \{give(H, T, tomato)\}

- chop(T, lettuce)
  - T: \{give(H, T, lettuce)\}
  - H: \{give(H, T, lettuce)\}
Joint Executions

We enforce several restrictions to ensure a JE can always be executed.

- Independent events have independent actions
- All possible outcomes are considered
- Actions are enabled by observable events:
- Overlapping views enable identical actions:
Our implementation maintains these restrictions while building a JE one action at a time - just like an ordinary situation term.

```plaintext
proc {MakePlan JIn Branches JOut}
  BClosed BRest
in
  {FindOpenBranch JIn Branches BClosed BRest}
  case BRest of (D#R#N)|Bs then Dp Rp S J2 OutNs OutBs in
    {FindTrans1 D R Dp Rp S}
    OutNs = {JointExec.insert JIn N S {MkPrecFunc S Rp} J2}
    OutBs = for collect:C N2 in OutNs do
      {C Dp#ex({JointExec.getobs J2 N2 S} Rp)#N2}
    end
    {MakePlan J2 {Append3 BClosed OutBs Bs} JOut}
  else JOut = JIn end
end
```
Planning with Joint Executions
Joint Executions

**Joint Execution**: a partially-ordered data structure representing actions to be performed by a group of agents

- That ensures synchronisation is always possible
- That can be reasoned about using standard sitcalc techniques
- That can replace situation terms in the Golog planning process
- Implemented in a MIndiGolog execution planner
Publications


MIndiGolog is downloadable from www.agentlab.unimelb.edu.au