Multi-agent programming in IndiGolog

Adrian Pearce

13 July 2011

Includes slides by Ryan Kelly

Outline

Outline

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

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
- δ^* - 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:

Operator	Meaning
a	Execute action a in the world
$\phi?$	Proceed if condition ϕ is true
$\delta_1 ; \delta_2$	Execute δ_1 followed by δ_2
$\delta_1 \vert \delta_2$	Execute either δ_1 or δ_2
$\pi(x)\delta(x)$	Nondet. select arguments for δ
$\delta *$	Execute δ zero or more times
if ϕ **then** δ_1 **else** δ_2	Exec. δ_1 if ϕ holds, δ_2 otherwise
while ϕ **do** δ	Execute δ while ϕ holds
proc $P(\overrightarrow{x})\delta(\overrightarrow{x})$ **end**	Procedure definition
$\delta_1 \vert\vert \delta_2$	Concurrent execution (ConGolog)
$\Sigma\delta$	Plan execution offline (IndiGolog)

A Quick Example

Consider a Golog program for getting to university of a morning:

$$\begin{gathered} ringAlarm; (hitSnooze; ringAlarm)^*; turnOffAlarm; \\ \pi(food,\ edible(food)?; eat(food)); (haveShower || brushTeeth); \\ (driveToUni \mid trainToUni); (time < 11:00)? \end{gathered}$$

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 *Legal Execution*:

$$\mathcal{D} \models \exists s, \delta' : Trans^*(\delta, S_0, \delta', s) \wedge Final(\delta', s)$$

Extending the Situation Calculus

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: $Legal(a, s) \rightarrow \neg\exists n : nat(n) \wedge Poss(n, s)$
- Incomplete knowledge (from last lecture): $\mathbf{Knows}(\phi, s)$

Outline

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.

$$\begin{array}{c} \textbf{proc } \mathit{MakeSalad}(\mathit{bowl}) \\ (\mathit{ChopTypeInto}(\mathit{Lettuce}, \mathit{bowl}) \; || \\ \mathit{ChopTypeInto}(\mathit{Carrot}, \mathit{bowl}) \; || \\ \mathit{ChopTypeInto}(\mathit{Tomato}, \mathit{bowl}) \;) \; ; \\ \pi(\mathit{agt}, \mathit{Mix}(\mathit{agt}, \mathit{bowl}, 1)) \\ \textbf{end} \end{array}$$

$$\begin{array}{c} \textbf{proc } \mathit{ChopTypeInto}(\mathit{type}, \mathit{dest}) \\ \pi((\mathit{agt}, \mathit{obj}), \\ \mathit{IsType}(\mathit{obj}, \mathit{type})? \; ; \\ \mathit{Chop}(\mathit{agt}, \mathit{obj}) \; ; \\ \mathit{PlaceIn}(\mathit{agt}, \mathit{obj}, \mathit{dest})) \\ \textbf{end} \end{array}$$

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

Outline

MIndiGolog Semantics

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

$$\delta = \delta_{agt1} || \delta_{agt2} || \ldots || \delta_{agtN}$$

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

$$\delta = \delta_{task1} || \delta_{task2} || \ldots || \delta_{taskN}$$

MIndiGolog takes the second approach

Algorithm for multiple agents

Algorithm: ReadyLog

$\sigma \Leftarrow S_0$
while $\mathcal{D} \cup \mathcal{D}_{golog} \not\models \mathit{Final}(\delta, s)$ **do**
  Find an action a and program δ' such that:

$$\mathcal{D} \cup \mathcal{D}_{golog} \models \mathit{Trans}^*(\delta, \sigma, \delta', do(a, \sigma))$$

  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

Algorithm for multiple agents

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

$$MakeSalad() || MakePasta() || MakeCake()$$

They can even plan to have different dishes ready at different times $[MakeSalad() || MakePasta()]; ?(time < 7:30))$

$$|| (MakeCake(); ?(8:15 < time < 8:30))$$

Time

We modify the original transition rule

$$Trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = Nil \wedge s' = do(a, s)$$

Modifying this to use CONCURRENT#TIMEPOINT pairs and *Legal* gives

$$Trans(a, s, \delta', s') \equiv \exists t : Legal(\{a\}\#t, s) \wedge \delta' = Nil \wedge s' = do(\{a\}, s)$$

This ensures that the temporal component respects the ordering between predecessor and successor situations.

Example Output: *MakeSalad*()

```
do [acquire(thomas,lettuce1), acquire(richard,tomato1),
   acquire(harriet,carrot1)] at _U
do [acquire(thomas,board1), acquire(harriet,board2)] at _T
do [place_in(thomas,lettuce1,board1), place_in(harriet,carrot1,board2)]
do [begin_task(thomas,chop(board1)), begin_task(harriet,chop(board2))]
do [end_task(thomas,chop(board1)), end_task(harriet,chop(board2))] at _
do [acquire(thomas,bowl1)] at _P
do [transfer(thomas,board1,bowl1)] at _O
do [release(thomas,board1)] at _N
do [release(thomas,bowl1), acquire(richard,board1)] at _M
do [place_in(richard,tomato1,board1), acquire(harriet,bowl1)] at _L
do [begin_task(richard,chop(board1)), transfer(harriet,board2,bowl1)] a
```

Example Output

```
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: $MakeSalad(Bowl1) \;||MakePasta(Bowl2) \;||\; MakeCake(Bowl3)$

MIndiGolog Semantics

Agents should take advantage of true concurrency. Basic idea:

$$Trans(\delta_1||\delta_2, s, \delta', s') \equiv \exists\gamma : \; Trans(\delta_1, s, \gamma, s') \wedge \delta' = (\gamma||\delta_2)$$
$$\vee \; \exists\gamma : \; Trans(\delta_2, s, \gamma, s') \wedge \delta' = (\delta_1||\gamma)$$
$$\vee \; \exists c_1, c_2, \gamma_1, \gamma_2, t : \; Trans(\delta_1, s, \gamma_1, do(c_1\#t, s))$$
$$\wedge Trans(\delta_2, s, \gamma_2, do(c_2\#t, s)) \quad \wedge Legal((c_1 \cup c_2)\#t, s) \wedge \forall a : [a \in c_1 \wedge a \in$$
$$\wedge \; \delta' = (\gamma_1||\gamma_2) \wedge s' = do((c_1 \cup c_2)\#t, s)$$

Robust Concurrency

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

Robust Concurrency

Consider two programs both wanting to initiate agent actions:

$$\delta_1 = placeIn(Jim, Flour, Bowl); placeIn(Jim, Sugar, Bowl)$$

$$\delta_2 = placeIn(Jim, Flour, Bowl); placeIn(Jim, Egg, Bowl)$$

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

However, an individual transition for both programs is $c_1 = c_2 = \{placeIn(Jim, Foour, Bowl)\}$.
Naively executing $c_1 \cup c_2$ to transition both programs would result in only one unit of flour being added.

Robust Concurrency

Consider two programs waiting for a timer to ring:

$$\delta_1 = \mathit{ringTimer}; \mathit{acquire}(\mathit{Jim}, \mathit{Bowl})$$

$$\delta_2 = \mathit{ringTimer}; \mathit{acquire}(\mathit{Joe}, \mathit{Bowl})$$

Both programs should be allowed to proceed using the same (natural) *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.

MIndiGolog Execution

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 ommitted> ...
  end
end
```

MIndiGolog Execution

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

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)

Outline

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

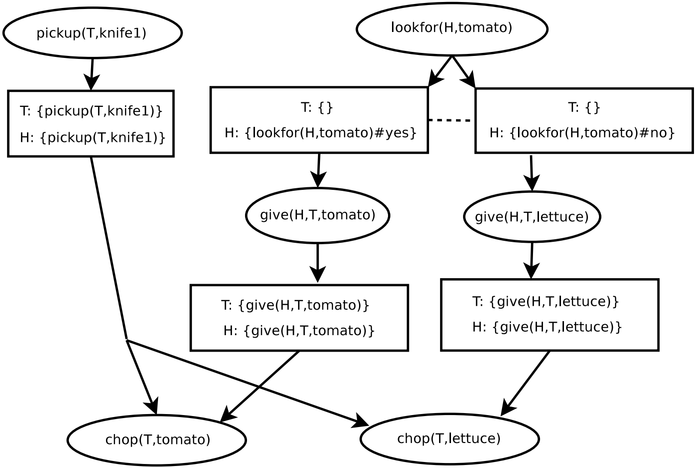
Prime event structures are a canonical representation for partially-ordered branching sequences of events:

- 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:

- $j \in ens(i) \equiv j \prec i \wedge \forall k \in ens(i) : \neg(j \prec k)$
- $j \in alts(i) \equiv j \# i \wedge \forall k \in ens(i) : \neg(j \# k)$

Joint Executions



Joint Executions

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

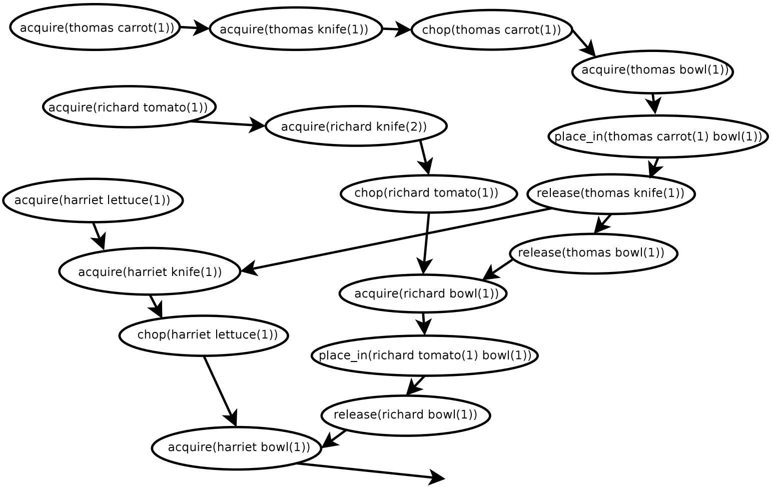
- Independent events have independent actions
- All possible outcomes are considered
- Actions are enabled by observable events:
- Overlapping views enable identical actions:

Planning with Joint Executions

Our implementation maintains these restrictions while building a JE one action at a time - just like an ordinary situation term.

```
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

Outline

Publications

- Sebastian Sardina, Giuseppe De Giacomo, Yves Lesperance, and Hector Levesque. On the Semantics of Deliberation in IndiGolog - From Theory to Implementation. Annals of Mathematics and Artificial Intelligence, 41(2-4):259-299, August 2004
- Ryan F. Kelly and Adrian R. Pearce. Towards High-Level Programming for Distributed Problem Solving. In Proceedings of the IEEE/WIC/ACM Inter- national Conference on Intelligent Agent Technology (IAT'06), pages 490-497, 2006
- Ryan Kelly. Asynchronous Multi-Agent Reasoning in the Situation Calculus, PhD Thesis, The University of Melbourne, 2008

Publications

- A. Ferrein, Ch. Fritz, and G. Lakemeyer. Using Golog for Deliberation and Team Coordination in Robotic Soccer. Kunstliche Intelligenz, I:24-43, 2005.
- Alessandro Farinelli, Alberto Finzi, Thomas Lukasiewicz: Team Programming in Golog under Partial Observability. IJCAI: 2097-2102, 2007
- Fiora Pirri and Ray Reiter. Planning with natural actions in the situation calculus. In Logic-Based Artificial Intelligence. Kluwer Press, 2000.

Download

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

Summary