CHAPTER 3: DEDUCTIVE REASONING AGENTS

An Introduction to Multiagent Systems

http://www.csc.liv.ac.uk/˜mjw/pubs/imas/

Agent Architectures

- An *agent architecture* is a *software design* for an agent.
- We have already seen a top-level decomposition, into:

  perception – state – decision – action

- An agent architecture defines:
  - key data structures;
  - operations on data structures;
  - control flow between operations

Agent Architectures – Pattie Maes (1991)

'[A] particular methodology for building [agents]. It specifies how . . . the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions . . . and future internal state of the agent. An architecture encompasses techniques and algorithms that support this methodology.'

Agent Architectures – Leslie Kaelbling (1991)

'[A] specific collection of software (or hardware) modules, typically designated by boxes with arrows indicating the data and control flow among the modules. A more abstract view of an architecture is as a general methodology for designing particular modular decompositions for particular tasks.'

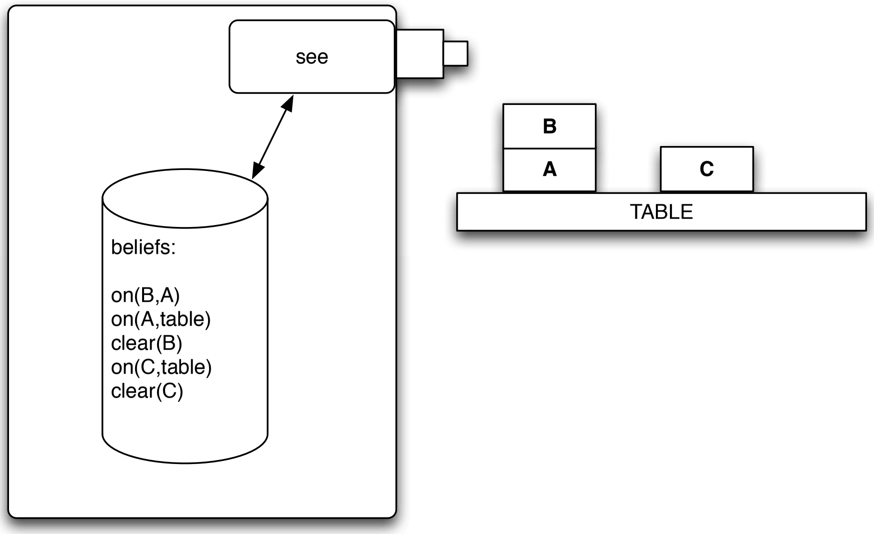
Types of Agents

- 1956–present: *Symbolic Reasoning Agents*
  Its purest expression, proposes that agents use *explicit logical reasoning* in order to decide what to do.
- 1985–present: *Reactive Agents*
  Problems with symbolic reasoning led to a reaction against this — led to the *reactive agents* movement, 1985–present.
- 1990-present: *Hybrid Agents*
  Hybrid architectures attempt to combine the best of symbolic and reactive architectures.

Symbolic Reasoning Agents

- The classical approach to building agents is to view them as a particular type of knowledge-based system, and bring all the associated methodologies of such systems to bear.
- This paradigm is known as *symbolic AI*.
- We define a deliberative agent or agent architecture to be one that:
  - contains an explicitly represented, symbolic model of the world;
  - makes decisions (for example about what actions to perform) via symbolic reasoning.

Representing the Environment Symbolically



The Transduction Problem

The problem of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.

. . . vision, speech understanding, learning.

The representation/reasoning problem

that of how to symbolically represent information about complex real-world entities and processes, and how to get agents to reason with this information in time for the results to be useful.
. . . knowledge representation, automated reasoning, automatic planning.

Problems with Symbolic Approaches

- Most researchers accept that neither problem is anywhere near solved.
- Underlying problem lies with the complexity of symbol manipulation algorithms in general: many (most) search-based symbol manipulation algorithms of interest are *highly intractable*.
- Because of these problems, some researchers have looked to alternative techniques for building agents; we look at these later.

Deductive Reasoning Agents

- Use logic to encode a theory defining the *best* action to perform in any given situation.
- Let:

  ρ be this theory (typically a set of rules);

  Δ be a logical database that describes the current state of the world;

  Ac be the set of actions the agent can perform;

  $\Delta \vdash_\rho \phi$ mean that ϕ can be proved from Δ using ρ.

Action Selection via Theorem Proving

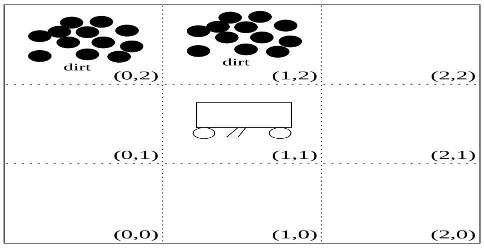
for each $\alpha \in Ac$ do
    if $\Delta \vdash_\rho Do(\alpha)$ then return α
end-for

for each $\alpha \in Ac$ do
    if $\Delta \not\vdash_\rho \neg Do(\alpha)$ then return α
end-for

return `null` /* *no action found* */

An Example: The Vacuum World

- Goal is for the robot to clear up all dirt.



- Use 3 *domain predicates* in this exercise:

$In(x,y)$	agent is at (x,y)
$Dirt(x,y)$	there is dirt at (x,y)
$Facing(d)$	the agent is facing direction d

- Possible actions:

$$Ac = \{turn, forward, suck\}$$

NB: *turn* means "turn right".

- Rules ρ for determining what to do:

$$
\begin{aligned}
\mathit{In}(0,0) \wedge \mathit{Facing}(\mathit{north}) \wedge \neg \mathit{Dirt}(0,0) &\longrightarrow \mathit{Do}(\mathit{forward}) \\
\mathit{In}(0,1) \wedge \mathit{Facing}(\mathit{north}) \wedge \neg \mathit{Dirt}(0,1) &\longrightarrow \mathit{Do}(\mathit{forward}) \\
\mathit{In}(0,2) \wedge \mathit{Facing}(\mathit{north}) \wedge \neg \mathit{Dirt}(0,2) &\longrightarrow \mathit{Do}(\mathit{turn}) \\
\mathit{In}(0,2) \wedge \mathit{Facing}(\mathit{east}) &\longrightarrow \mathit{Do}(\mathit{forward})
\end{aligned}
$$

- . . . and so on!
- Using these rules (+ other obvious ones), starting at $(0,0)$ the robot will clear up dirt.

Problems

- how to convert video camera input to $\mathit{Dirt}(0,1)$?
- decision making assumes a *static* environment: *calculative* rationality.
- decision making via theorem proving is *complex* (maybe event *undecidable*!)

Approaches to Overcoming these Problems

- weaken the logic;
- use symbolic, non-logical representations;
- shift the emphasis of reasoning from *run time* to *design time*.

AGENT0 and PLACA

- Yoav Shoham introduced "agent-oriented programming" in 1990:

  "new programming paradigm, based on a societal view of computation".
- The key idea:
  directly programming agents in terms of intentional notions like belief, commitment, and intention.

Agent0

Each agent in AGENT0 has 4 components:

- a set of capabilities (things the agent can do);
- a set of initial beliefs;
- a set of initial commitments (things the agent will do); and
- a set of *commitment rules*.

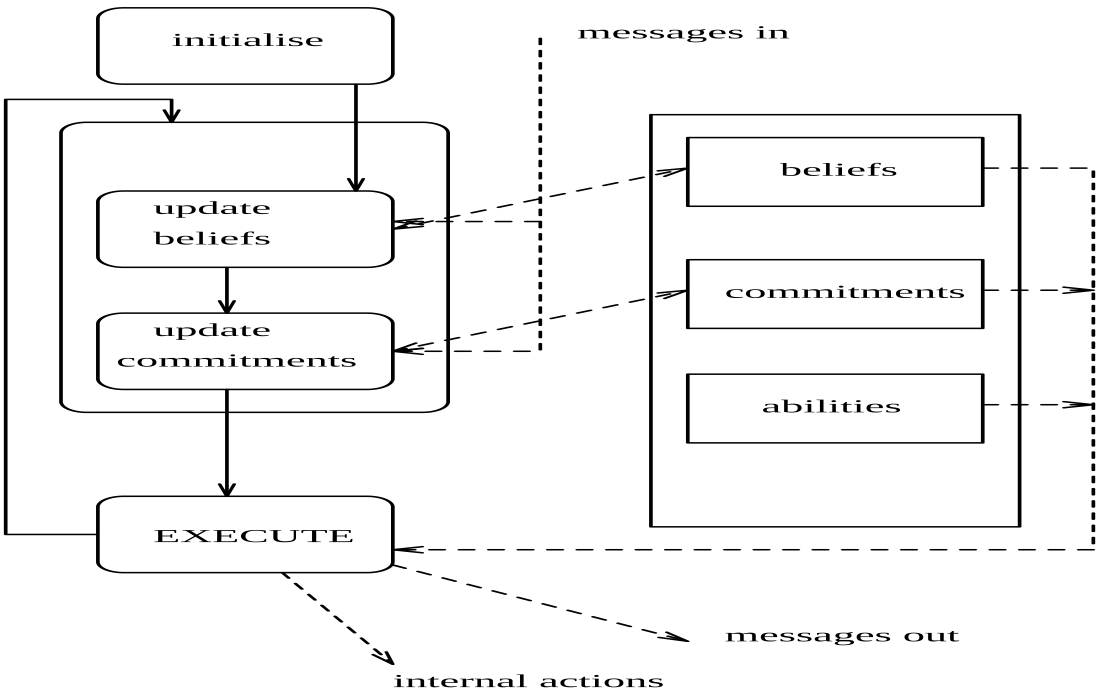
Commitment Rules

- The key component, which determines how the agent acts, is the commitment rule set.
- Each commitment rule contains
  - a *message condition*;
  - a *mental condition*; and
  - an action.

AGENT0 Decision Cycle

- On each *decision cycle* . . .
  The message condition is matched against the messages the agent has received;
  The mental condition is matched against the beliefs of the agent.
  If the rule fires, then the agent becomes committed to the action (the action gets added to the agents commitment set).

- Actions may be
  - *private*:
    an internally executed computation, or
  - *communicative*:
    sending messages.
- Messages are constrained to be one of three types:
  - “requests” to commit to action;
  - “unrequests” to refrain from actions;
  - “informs” which pass on information.



- A commitment rule:

```
COMMIT(
   ( agent, REQUEST, DO(time, action)
   ), ;;; msg condition
   ( B,
     [now, Friend agent] AND
     CAN(self, action) AND
     NOT [time, CMT(self, anyaction)]
   ), ;;; mental condition
   self,
   DO(time, action)
)
```

- This rule may be paraphrased as follows:

  if I receive a message from *agent* which requests me to do *action* at *time*, and I believe that:

  - *agent* is currently a friend;
  - I can do the action;
  - at *time*, I am not committed to doing any other action,

  then commit to doing *action* at *time*.

PLACA

- A more refined implementation was developed by Thomas, for her 1993 doctoral thesis.
- Her Planning Communicating Agents (PLACA) language was intended to address one severe drawback to AGENT0: the inability of agents to plan, and communicate requests for action via high-level goals.
- Agents in PLACA are programmed in much the same way as in AGENT0, in terms of *mental change* rules.

- An example mental change rule:

```
(((self ?agent REQUEST (?t (xeroxed ?x)))
 (AND (CAN-ACHIEVE (?t xeroxed ?x)))
      (NOT (BEL (*now* shelving)))
      (NOT (BEL (*now* (vip ?agent))))
 ((ADOPT (INTEND (5pm (xeroxed ?x)))))
 ((?agent self INFORM
       (*now* (INTEND (5pm (xeroxed ?x)))))))
```

- Paraphrased:

  if someone asks you to xerox something, and you can, and you don't believe that they're a VIP, or that you're supposed to be shelving books, then

  – adopt the intention to xerox it by 5pm, and
  – inform them of your newly adopted intention.

Concurrent METATEM

- Concurrent METATEM is a multi-agent language in which each agent is programmed by giving it a *temporal logic* specification of the behaviour it should exhibit.
- These specifications are executed directly in order to generate the behaviour of the agent.
- Temporal logic is classical logic augmented by *modal operators* for describing how the truth of propositions changes over time.

- For example...

$$\Box \text{important(agents)}$$

means "it is now, and will always be true that agents are important"

$$\Diamond \text{important(ConcurrentMetateM)}$$

means "sometime in the future, ConcurrentMetateM will be important"

$$(\neg \text{friends(us)}) \, \mathcal{U} \, \text{apologise(you)}$$

means "we are not friends until you apologise"

$$\bigcirc \text{apologise(you)}$$

means "tomorrow (in the next state), you apologise".

- MetateM program is a collection of

$$\textit{past} \Rightarrow \textit{future}$$

rules.

- Execution proceeds by a process of continually matching rules against a "history", and *firing* those rules whose antecedents are satisfied.
- The instantiated future-time consequents become *commitments* which must subsequently be satisfied.

- An example MetateM program: the resource controller. . .

$$\begin{aligned} \circledcirc\ \mathrm{ask}(x) &\Rightarrow \Diamond\, \mathrm{give}(x) \\ \mathrm{give}(x) \wedge \mathrm{give}(y) &\Rightarrow (x{=}y) \end{aligned}$$

  – First rule ensure that an ‘ask’ is eventually followed by a ‘give’.
  – Second rule ensures that only one ‘give’ is ever performed at any one time.

- A Concurrent MetateM system contains a number of agents (objects), each object has 3 attributes:
  – a name;
  – an interface;
  – a MetateM program.

- An agent's interface contains two sets:
  - messages the agent will *accept*;
  - messages the agent may *send*.
- For example, a 'stack' object's interface:

stack(pop, push)[popped, stackfull]

{pop, push} = messages received
{popped, stackfull} = messages sent

Snow White & The Dwarves

- To illustrate the language Concurrent MetateM in more detail, here are some example programs...
- Snow White has some sweets (resources), which she will give to the Dwarves (resource consumers).
- She will only give to one dwarf at a time.
- She will always eventually give to a dwarf that asks.
- Here is Snow White, written in Concurrent MetateM:

Snow-White(ask)[give]:

$$\circledcirc\ \text{ask}(x) \Rightarrow \Diamond\, \text{give}(x)$$
$$\text{give}(x) \wedge \text{give}(y) \Rightarrow (x = y)$$

- The dwarf ‘eager’ asks for a sweet initially, and then whenever he has just received one, asks again.

eager(give)[ask]:

$$\begin{array}{rcl} \text{start} & \Rightarrow & \text{ask(eager)} \\ \circledcirc\ \text{give(eager)} & \Rightarrow & \text{ask(eager)} \end{array}$$

- Some dwarves are even less polite: ‘greedy’ just asks every time.

greedy(give)[ask]:

$$\text{start} \Rightarrow \Box\ \text{ask(greedy)}$$

- Fortunately, some have better manners; 'courteous' only asks when 'eager' and 'greedy' have eaten.

courteous(give)[ask]:

$$((\neg \text{ask(courteous)}\ \mathcal{S}\ \text{give(eager)}) \wedge (\neg \text{ask(courteous)}\ \mathcal{S}\ \text{give(greedy)})) \Rightarrow \text{ask(courteous)}$$

- And finally, 'shy' will only ask for a sweet when no-one else has just asked.

shy(give)[ask]:

$$\begin{aligned} \mathit{start} &\Rightarrow \Diamond\ \text{ask(shy)} \\ \circledcirc\ \text{ask(x)} &\Rightarrow \neg\ \text{ask(shy)} \\ \circledcirc\ \text{give(shy)} &\Rightarrow \Diamond\ \text{ask(shy)} \end{aligned}$$