LECTURE 3: DEDUCTIVE REASONING AGENTS

An Introduction to Multiagent Systems

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

1 Agent Architectures

- Introduce the idea of an *agent* as a computer system capable of *flexible autonomous action*.
- Briefly discuss the issues one needs to address in order to build agent-based systems.
- Three types of agent *architecture*:
  - symbolic/logical;
  - reactive;
  - hybrid.

- We want to build agents, that enjoy the properties of autonomy, reactiveness, pro-activeness, and social ability that we talked about earlier.
- This is the area of *agent architectures*.
- Maes defines an agent architecture as:

  > '[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.'

- Kaelbling considers an agent architecture to be:

  > '[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.'

- Originally (1956-1985), pretty much all agents designed within AI were *symbolic reasoning* agents.

  Its purest expression proposes that agents use *explicit logical reasoning* in order to decide what to do.

- Problems with symbolic reasoning led to a reaction against this — the so-called *reactive agents* movement, 1985–present.

- From 1990-present, a number of alternatives proposed: *hybrid* architectures, which attempt to combine the best of reasoning and reactive architectures.

2 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 (discredited?!) 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.

- If we aim to build an agent in this way, there two key problems to be solved:

  1. *The transduction problem*:

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

     . . . vision, speech understanding, learning.

  2. *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.

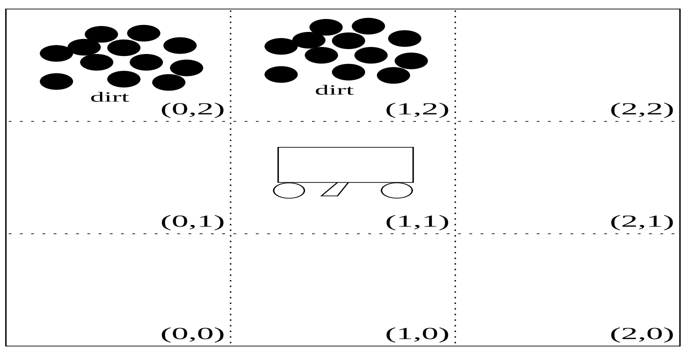
- Most researchers accept that neither problem is anywhere near solved.
- Underlying problem lies with the complexity of symbol manipulation algorithms in general: many (most) search-based symbol manipulation algorithms of interest are *highly intractable*.
- Because of these problems, some researchers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

2.1 Deductive Reasoning Agents

- How can an agent decide what to do using theorem proving?
- Basic idea is to use logic to encode a theory stating 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.

/* *try to find an action explicitly prescribed* */
for each $a \in Ac$ do
    if $\Delta \vdash_\rho Do(a)$ then
        return a
    end-if
end-for
/* *try to find an action not excluded* */
for each $a \in Ac$ do
    if $\Delta \not\vdash_\rho \neg Do(a)$ then
        return a
    end-if
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:

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

- ... 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 $Dirt(0,1)$?
  - decision making assumes a *static* environment: *calculative* rationality.
  - decision making using first-order logic is *undecidable*!
- Even where we use *propositional* logic, decision making in the worst case means solving co-NP-complete problems.

  (NB: co-NP-complete = bad news!)
- Typical solutions:
  - weaken the logic;
  - use symbolic, non-logical representations;
  - shift the emphasis of reasoning from *run time* to *design time*.
- We now look at some examples of these approaches.

2.2 AGENT0 and PLACA

- Much of the interest in agents from the AI community has arisen from Shoham's notion of *agent oriented programming* (AOP).
- AOP a 'new programming paradigm, based on a societal view of computation'.
- The key idea that informs AOP is that of directly programming agents in terms of intentional notions like belief, commitment, and intention.
- The motivation behind such a proposal is that, as we humans use the intentional stance as an *abstraction* mechanism for representing the properties of complex systems.

  In the same way that we use the intentional stance to describe humans, it might be useful to use the intentional stance to program machines.

- Shoham suggested that a complete AOP system will have 3 components:
  - a logic for specifying agents and describing their mental states;
  - an interpreted programming language for programming agents;
  - an 'agentification' process, for converting 'neutral applications' (e.g., databases) into agents.

  Results only reported on first two components.

  Relationship between logic and programming language is *semantics*.
- We will skip over the logic(!), and consider the first AOP language, AGENT0.

- AGENT0 is implemented as an extension to LISP.
  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*.
- The key component, which determines how the agent acts, is the commitment rule set.

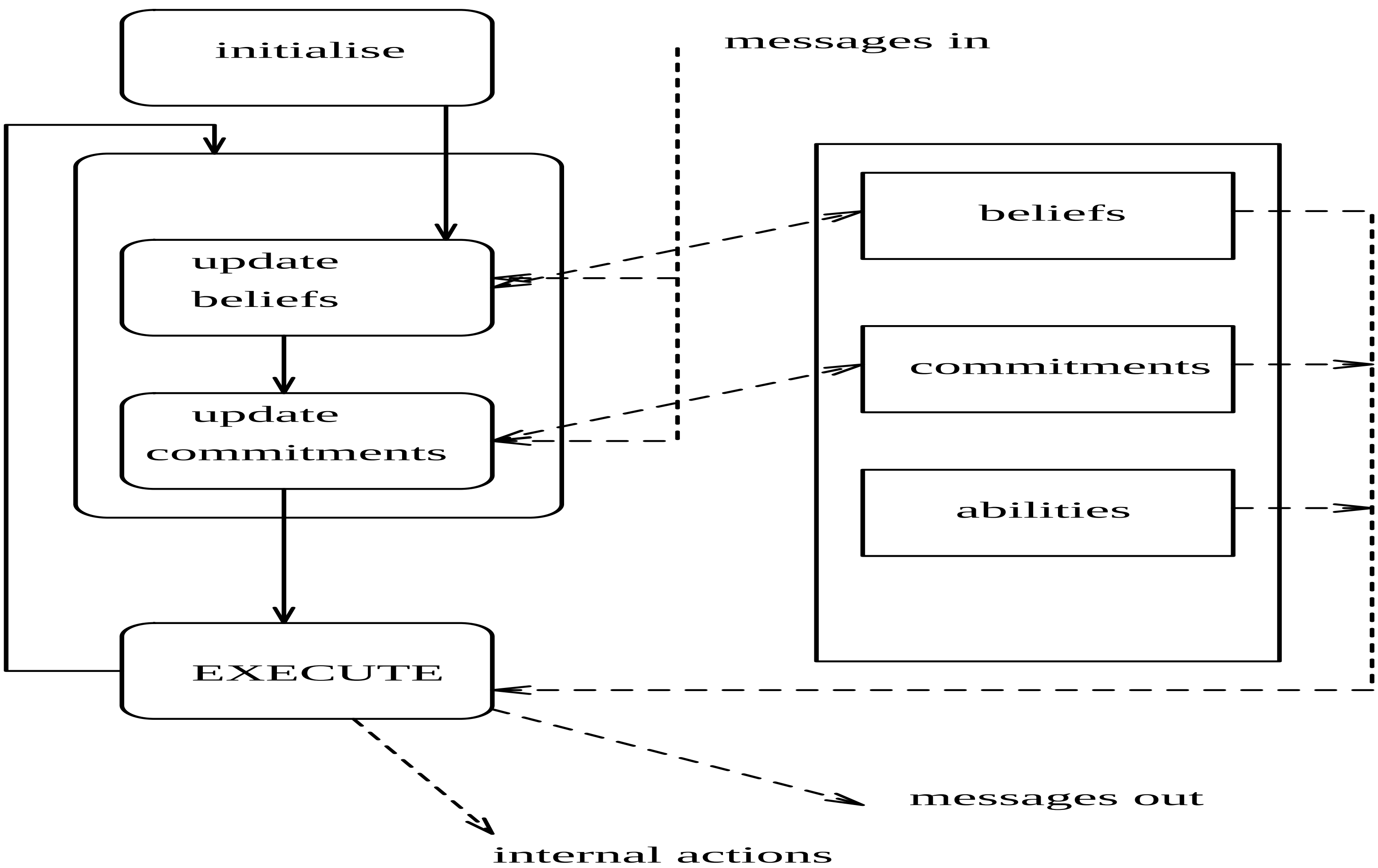
- Each commitment rule contains
  - a *message condition*;
  - a *mental condition*; and
  - an action.
- On each 'agent cycle' . . .

  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.


initialise
messages in
update
beliefs
update
commitments
beliefs
commitments
abilities
EXECUTE
messages out
internal actions

- 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*.

- AGENT0 provides support for multiple agents to cooperate and communicate, and provides basic provision for debugging. . .
- . . . it is, however, a *prototype*, that was designed to illustrate some principles, rather than be a production language.
- 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.

2.3 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...

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

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"

$$\blacklozenge \text{important(Prolog)}$$

means "sometime in the past it was true that Prolog was 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 is a framework for *directly executing* temporal logic specifications.
- The root of the MetateM concept is Gabbay's *separation theorem*:

  Any arbitrary temporal logic formula can be rewritten in a logically equivalent *past* $\Rightarrow$ *future* form.
- This *past* $\Rightarrow$ *future* form can be used as *execution rules*.
- A MetateM program is a set of such 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.

- Execution is thus is process of iteratively generating a model for the formula made up of the program rules.
- The future-time parts of instantiated rules represent *constraints* on this model.
- An example MetateM program: the resource controller. . .

$$\forall x \quad ⊚ ask(x) \Rightarrow \Diamond give(x)$$
$$\forall x,y \quad give(x) \wedge give(y) \Rightarrow (x=y)$$

- 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.
- There are algorithms for executing MetateM programs that appear to give reasonable performance.
- There is also *separated normal form*.

- ConcurrentMetateM provides an operational framework through which societies of MetateM processes can operate and communicate.
- It is based on a new model for concurrency in executable logics: the notion of executing a logical specification to generate individual agent behaviour.
- A ConcurrentMetateM system contains a number of agents (objects), each object has 3 attributes:
  - a name;
  - an interface;
  - a MetateM program.

- An object's interface contains two sets:
  - environment predicates — these correspond to messages the object will accept;
  - component predicates — correspond to messages the object may send.
- For example, a 'stack' object's interface:

  stack(pop, push)[popped, stackfull]

  {pop, push} = environment preds

  {popped, stackfull} = component preds
- If an agent receives a message headed by an environment predicate, it accepts it.
- If an object satisfies a commitment corresponding to a component predicate, it broadcasts it.

- 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]:

$$\circledcirc\ ask(x) \Rightarrow \Diamond\ give(x)$$
$$give(x) \wedge give(y) \Rightarrow (x = y)$$

- 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]:

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

- 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]:

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

- Summary:
  - an(other) experimental language;
  - very nice underlying theory...
  - ... but unfortunately, lacks many desirable features — could not be used in current state to implement 'full' system.
  - currently prototype only, full version on the way!

2.4 Planning agents

- Since the early 1970s, the AI planning community has been closely concerned with the design of artificial agents.
- Planning is essentially automatic programming: the design of a course of action that will achieve some desired goal.
- Within the symbolic AI community, it has long been assumed that some form of AI planning system will be a central component of any artificial agent.
- Building largely on the early work of Fikes & Nilsson, many planning algorithms have been proposed, and the theory of planning has been well-developed.
- But in the mid 1980s, Chapman established some theoretical results which indicate that AI planners will ultimately turn out to be unusable in any time-constrained system.