CHAPTER 3: DEDUCTIVE REASONING AGENTS

An Introduction to Multiagent Systems

http://www.csc.liv.ac.uk/~mjw/pubs/imas/
Chapter 3
An Introduction to Multiagent Systems

agent Architectures

Agent Architectures

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

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

We have already seen a top-level decomposition, into:

Agent Architectures

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

• We have already seen a top-level decomposition, into:

Agent Architectures

• An agent architecture is a software design for an agent.
A particular methodology for building agents. An agent architecture encompasses techniques for determining the actions ... and future internal state of the agent. The current internal state of the agent and the data from the sensor provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions. The total set of modules and their interactions has to specify how these modules should be made to interact. The construction of a set of component modules and the construction of agent can be decomposed into Agent Architectures – Pattie Maes (1991)
A more abstract view of an architecture is as a general methodology for designing particular modular decompositions for particular tasks.

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

• 1990–present: Hybrid Agents
  Hybrid architectures attempt to combine the best of symbolic and reactive architectures.

• 1995–present: Symbolic Reasoning Agents

• 1985–present: Reactive Agents

http://www.csc.liv.ac.uk/~mjw/pubs/imas/
An Introduction to Multiagent Systems 2e

Chapter 3

Symbolic Reasoning Agents

The classical approach to building agents is to view them as a particular type of knowledge-based system, an introduction to Multiagent Systems 2e

Symbolic Reasoning Agents...
Representing the Environment Symbolically
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.

...
Therepresentation/reasoning problem

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

The Representation/Reasoning Problem

An Introduction to Multiagent Systems 2e
Problems with Symbolic Approaches

• Most researchers accept that neither problem is anywhere near solved.

• Because of these problems, some researchers have looked to alternative techniques for building agents.

• Underlying problem lies with the complexity of symbolic algorithms in general: many (most) search-based symbolic manipulation algorithms are highly intractable.

• We look at these later.
Deductive Reasoning Agents

• Use logic to encode a theory defining the best action

Let: •

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

\[ \phi \vdash_{d} \psi \]

\(\rho\) be this theory (typically a set of rules);
\(\Delta\) be a logical database that describes the current state of the world;
\(\Delta\) be the set of actions the agent can perform;
\(\Delta\) be the set of actions the agent can perform;
\(\Delta\) be the set of actions the agent can perform.

\(\phi\) mean that can be proved from using.
Action Selection via Theorem Proving

```
foreach α ∈ Ac do
  if ∆ ⊢ ρ Do(α) then return α
end-for

foreach α ∈ Ac do
  if ∆ ̸⊢ ρ ¬Do(α) then return α
end-for

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

http://www.csc.liv.ac.uk/~mjw/pubs/imas/11
An Example: The Vacuum World

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

http://www.csc.liv.ac.uk/~mjw/pubs/imas/
An Introduction to Multiagent Systems

Chapter 3

Use 3 domain predicates in this exercise:

- \( \text{Dirt}(x, y) \) means there is dirt at \((x, y)\).
- \( \text{Facing}(d) \) means the agent is facing direction \(d\).
- \( \text{AgentIsAt}(x) \) means the agent is at \((x)\).

Possible actions:

- \( \text{Turn} \)
- \( \text{Forward} \)
- \( \text{Suck} \)

NB: \text{Turn} means "turn right". \( \{ \text{turn, forward, suck} \} = \text{AC} \)
• Rules for determining what to do:

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

- how to convert video camera input to $Dirt(0, 1)$?
- decision making assumes a static environment: calculative rationality.
- (maybe even undecidable!) decision making via theorem proving is complex
Approaches to Overcoming these Problems

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

• weaken the logic;
• use symbolic, non-logical representations;
• shift the emphasis of reasoning from run time to design time.
Yoav Shoham introduced "agent-oriented programming" in 1990. The key ideas:

- A societal view of computation.
- "New programming paradigm", based on a societal view of computation.
- "Agent-oriented programming".

Yoav Shoham introduced "agent-oriented programming" in 1990.
Each agent in $\text{AGENT}_0$ 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);
- A set of commitment rules.

And

Each agent in $\text{AGENT}_0$ has 4 components.
Commitment Rules

- an action.
- a mental condition; and
- a message condition.

Each commitment rule contains

acts, is the commitment rule set.

The key component, which determines how the agent


Commitment Rules
On each decision cycle...

On each decision cycle...

agent decision cycle
• **Actions may be**

  • **private**
  
  • **an internally executed computation, or**
  
  • **communicative**
  
  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:

http://www.csc.liv.ac.uk/~mjw/pubs/imas/23

\[
\begin{align*}
\text{COMMIT} & \quad \text{COMMIT} \\
\text{agent, REQUEST, DO(time, action)} & \quad \text{COMMIT} \\
\text{self, [now, Friendagent] AND CAN(self, action) AND NOT [time, CMT(self, anyaction)]} & \quad \text{self, DO(time, action)} \\
\text{B, [now, Friendagent] AND NOT [time, CMT(self, anyaction)]} & \quad \text{B, [now, Friendagent] AND NOT [time, CMT(self, anyaction)]} \\
\text{msg condition} & \quad \text{msg condition} \\
\text{agent, REQUEST, DO(time, action)} & \quad \text{agent, REQUEST, DO(time, action)} \\
\end{align*}
\]
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;
  - agent is currently a friend;
  - at time, I am not committed to doing any other action;
  - at time, I am not committed to doing any other action;
  - I can do the action;
  - I can do the action;

  then commit to doing action at time.
• Amor 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. A more refined implementation was developed by Thomas.
An example mental change rule:

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

If someone asks you to xerox something, and you’re supposed to be shelving books, then:

Paraphrased:

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

http://www.csc.liv.ac.uk/~mjw/pubs/imas/26
Concurrency over time.

- Temporal logic is classical logic augmented by modal operators for describing how the truth of propositions changes over time.

- These specifications are executed directly in order to generate the behaviour of the agent. Each agent is programmed by giving it a temporal logic specification of the behaviour it should exhibit.

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. Each agent is programmed by giving it a temporal logic specification of the behaviour it should exhibit.

An Introduction to Multiagent Systems 2e
important(agents)

means "itisnow, and will always be true that agents"

important(ConcurrentMetateM)

means "sometime in the future, ConcurrentMetateM"

friends(us)(you)

means "we are not friends until you apologise"

apologise(you)

means "in the next state, you apologise"

apologise(you)

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

will be important

important(ConcurrentMetateM)

are important

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

For example...
A MetateM program is a collection of commitments, which must subsequently be satisfied.

- 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.

- MetateM is a collection of past ↔ future.
An example MetateM program: the Resource Controller.

Firstruleensures that an 'ask' is eventually followed by a 'give'.

\( \forall y ( \text{give}(x) \land \text{give}(y) \Rightarrow (x = y) ) \)

\( \forall x ( \text{ask}(x) \land \Diamond \text{give}(x) \iff \Box \text{give}(x) ) \)

Second rule ensures that only one 'give' is ever performed at any one time.
A Concurrent Metaltem system contains a number of agents (objects), each object has 3 attributes:

- a name;
- an interface;
- a Metaltem program.

http://www.csc.liv.ac.uk/~mjw/pubs/tmas/
An agent's interface contains two sets:

- messages the agent will accept;
- messages the agent may send.

For example, a 'stack' object's interface:

\[
\text{stack}(\text{pop, push})[\text{popped, stackfull}]
\]

\{ \text{pop, push} \}

\{ \text{popped, stackfull} \} = \text{messages received}

\{ \text{pop, push} \} = \text{messages sent}
To illustrate the language Concurrent MetaM:

\[ x = (\forall y \ (\text{give}(x) \lor \text{give}(y)) \land \exists y \ (\text{give}(x) \land \text{ask}(y)) \}\]

Snow-White(ask)[give]:

Here is Snow White, written in Concurrent MetaM:

- She 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.
- She will always eventually give to a dwarf that asks.
- She will give to the Dwarves (resource consumers).

Snow White has some sweets (resources), which she will give to the Dwarves (resource consumers).

\begin{align*}
\text{Snow-White}(\text{ask})[\text{give}] & : \text{give}(x) \land \text{give}(y) \\
\quad & \Rightarrow (x = y)
\end{align*}

\begin{align*}
\text{ask}(x) & \Leftrightarrow \Box \text{give}(x) \\
\text{give}(x) & \land \text{give}(y) \\
\Rightarrow & (x = y)
\end{align*}

In Introduction to Multiagent Systems 2e
The dwarf 'eager' asks for a sweet initially, and then whenever he has just received one, asks again.

\[
\text{eager} \xleftrightarrow{\text{give}} \text{ask} \quad \text{ask} \xleftarrow{\text{start}} \text{give} \text{eager}
\]

The dwarf 'eager' asks for a sweet initially, and then
Some dwarves are even less polite: 'greedy' just asks...

\[
\text{start} \quad \begin{array}{c}
\text{ask}\text{(greedy)}
\end{array}
\leftarrow \begin{array}{c}
\text{greedy}\text{(give)}\text{(ask)}
\end{array}
\]

every time.

http://www.csc.liv.ac.uk/~mjw/pubs/tamas/
Fortunately, some have better manners; courteous

\[\begin{align*}
&\text{ask(}\text{giv}e\text{}(\text{courte}ous)) \\
&\iff (\neg \text{ask(}c\text{ourte}ous) \lor \text{give}(\text{greedy})) \\
&\lor (\neg \text{ask(}c\text{ourte}ous) \land \text{give}(\text{eager}))
\end{align*}\]

only asks when 'eager' and 'greedy' have eaten.
And finally, ‘shy’ will only ask for a sweet when no-one else has just asked.

\[
\begin{align*}
\text{give(shy)} & \iff \neg \text{ask(shy)} \\
\text{ask(x)} & \iff \neg \text{ask(shy)} \\
\text{start} & \iff \text{shy(give)(ask)} \\
\end{align*}
\]