An Introduction to Multiagent Systems

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 flexible autonomous action.

Three types of agent architectures:

- Symbolic/logical
- Reactive
- Hybrid

Problems with Symbolic Reasoning led to a reaction against this.

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. From 1990-present, a number of alternative proposals: Hybrid — the so-called Reactance agents movement, 1995-present. Problems with symbolic reasoning led to a reaction against this.

Maes defines an agent architecture as:

This is the area of agent architectures.

We want to build agents that enjoy the properties of autonomy, reactivity, pro-activeness, and social ability that we talked about earlier.

Kaelbling considers an agent architecture to be:

Kealbing considers an agent architecture to be:

Returning to the earlier issue of the agents: an agent's architecture encompasses how its reasoning, computation, and interaction abilities are combined into a single entity. The set of internal agents that make up the agent is to be understood in the context of the agent's methods and modules. How the algorithms, methods, and modules are made to work together is the construction of a set of modular modules and how formal theories should be made to work. How the control of the agent is to be decomposed is the total set of modules and their interaction, which must provide a solution to the problem of how the agent can be decomposed into the constituent parts.

Problems with symbolic reasoning led to a reaction against this — the so-called Agent oriented movement, 1995-present. From 1990-present, a number of alternatives proposed: Hybrid architectures, which attempt to combine the best of reasoning and reactive architectures.
The classical approach to building agents is to view them as symbolic agents. This paradigm is known as symbolic AI.

We define a deliberative agent or an agent architecture to be one that:

- contains an explicitly represented, symbolic model of the world;
- makes decisions (for example about what actions to perform) using a symbolic representation (not an algorithm);
- operates in an environment represented symbolically.

This paradigm is known as symbolic AI. It makes decisions in the form of symbolic expressions that are operated upon by symbolic reasoning and manipulation algorithms. A symbolic reasoning agent encodes a theory for reasoning about its environment.

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 algorithms are highly intractable.

...how can an agent decide what to do using the information provided?

1. The transduction problem:
   - that of transforming the real world into an accurate, adequate symbolic description in time for the results to be useful.

2. The representation/reasoning problem:
   - that of how to symbolically represent information about the agent's environment.

Basic idea is to use logic to encode a theory stating the best action to perform in any given situation.

If we aim to build an agent in this way, there are 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 the results to be useful.

2. The representation/reasoning problem:
   - that of how to symbolically represent information about the agent's environment.

The classical approach to building agents is to view them as symbolic agents.

...vision, speech understanding, learning.
In this exercise:

Use 3 domain predicates in this exercise:

- **In**
  - \( x, y \): The agent is at \( (x, y) \).
- **Dirt**
  - \( x, y \): There is dirt at \( (x, y) \).
- **Facing**
  - \( d \): The agent is facing direction \( d \).

Possible actions:

- \( \text{turn} \) means "turn right".
- \( \text{forward} \) means "move forward".

\( \text{Do} \) -- actions:

- \( \text{Do} \text{forward} \)
- \( \text{Do} \text{turn} \)

Rules for determining what to do:

- If the agent is facing north and there is dirt at \( (x, y) \), do forward.
- If the agent is facing east and there is dirt at \( (x, y) \), do forward.
- If the agent is facing north and there is dirt at \( (x, y) \), do turn.
- If the agent is facing south and there is dirt at \( (x, y) \), do suck.

...and soon!

Using these rules (+ other obvious ones), the robot will clear up all dirt.

An example: The Vacuum World.

The goal is for the robot to clear up all dirt.

\( AC = \{ \text{turn, forward, suck} \} \)

NB: \( \text{turn right} \) means "turn right".
Problems:

- how to convert video camera input to Dirt?

- decision making assumes a static environment: calculative
  - decision making using first-order logic is undecidable

- how to convert video camera input to Pnin(1,1,2)?

Typical solutions:

- weakens the logic;

- uses symbolic, non-logical representations;

- shift the emphasis of reasoning from run time to design time;

- even where we use propositional logic, decision making in the
  worst case means solving co-NP-complete problems.

- decision making using first-order logic is undecidable

Result on only reported on first two components.

Relationship between logic and programming language is

semanitics.

Each agent in AGENT_0 has 4 components:

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

AGENT_0 is implemented as an extension to LISP.

The key component, which determines how the agent acts, is the
commitment rule set.

Each agent in AGENT_0 has 4 components:

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

The key component behind such a proposal is that, as we humans
represent the properties of complex systems, so we need
to use the intentional stance as an abstraction mechanism for
reasoning about the properties of complex systems.

The key idea that informs AOP is that of directly programming
computation, a new programming paradigm based on a societal view of
agents in terms of intentional notions like belief, commitment, and intention.

Much of the interest in agents from the AI community has arisen
from Shoham’s notion of agent oriented programming (AOP).

We now look at some examples of these approaches.

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.

AGENT_0 is implemented as an extension to LISP.

Each agent in 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); and
- a set of commitment rules.

The key component, which determines how the agent acts, is the
commitment rule set.

2.5 AGENT_0 and PLAQA

We will skip over the logic(!) and consider the first AOP
language, AGENT_0.
A commitment rule:

1. Each commitment rule contains:
   - a message condition;
   - a mental condition;
   - an action.

On each "agent cycle"...

The message condition is matched against the messages the agent has received:
- On each agent cycle...
  - a mental condition; and
  - a message condition; and

The action gets added to the agent's commitment set.

Messages are constrained to be one of three types:
- "requests" to commit to action;
- "unrequests" to refrain from action;
- "requests" to commit to action:
  - to an internally executed computation, or
  - to an internally executed computation, or
  - to a private action.

Messages which pass on information:
- "informs" which pass on information.

A commitment rule:

COMMIT((agent, REQUEST, DO(time, action)),
  [now, Friend agent] AND 
  CAN(agent, action) AND
  NOT[time, CMT(agent, any action)]
)
This rule may be paraphrased as follows:

If I receive a message from an agent which requests me to do action at time \( t \), and I believe that:

1. the agent is a friend;
2. I can do the action;
3. at time \( t \), I am not committed to doing any other action,

then commit to doing action at time \( t \).

AGENT provides support for multiple agents to cooperate and communicate, and communicative requests for action.

Temporal logic is classical logic augmented by modal operators that describe the behavior of the agent.

These specifications are executed directly in order to generate the behavior of the agent.

Concurrent MetaAgent is a multi-agent language in which each agent is programmed by giving it a temporal logic specification of the behavior it should exhibit.

An example mental change rule:

\[
( Self \equiv \text{agent} ) \land ( \text{REQUEST}(?t(xeroxed?x)) ) \land ( \text{CAN-ACHIEVE}(?txeroxed?x) ) \land ( \text{NOT}(\text{BEL}(\text{*now*shelving})) ) \land ( \text{NOT}(\text{BEL}(\text{*now*(VIP?agent)))) ) \Rightarrow ( \text{ADOPT}(\text{INTEND}(5\text{pm}(xeroxed?x))) ) \land ( \text{INFORM}(\text{*now*(\text{INTEND}(5\text{pm}(xeroxed?x))))))
\]

Paraphrased:

- If someone asks you to xerox something, and you can and 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.

Temporal logic is classical logic augmented by modal operators that describe how the truth of propositions changes over time.

Temporal logic is classical logic augmented by modal operators that describe the behavior of the agent.

These specifications are executed directly in order to generate the behavior of the agent.

Concurrent MetaAgent is a multi-agent language in which each agent is programmed by giving it a temporal logic specification of the behavior it should exhibit.

2.3 Concurrent MetaAgent

Each agent in PRIVX is programmed in much the same way as in the high-level goals.

The specifications are executed directly in order to generate the behavior of the agent.

Temporal logic is classical logic augmented by modal operators that describe how the truth of propositions changes over time.

This rule may be paraphrased as follows:

- If I receive a message from an agent which requests me to do action at time \( t \), and I believe that:
  1. the agent is currently a friend;
  2. I can do the action;
  3. at time \( t \), I am not committed to doing any other action,

then inform them of your newly adopted intention.

AGENT provides support for multiple agents to cooperate and communicate, and communicative requests for action.

Temporal logic is classical logic augmented by modal operators that describe how the truth of propositions changes over time.
For example...

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

Important(ConcurrentMetateM) means "sometime in the future, ConcurrentMetateM will be important"

Important(Prolog) means "sometime in the past it was true that Prolog was important"

Friends(us) means "we are not friends until you apologize"

Apologise(you) means "tomorrow (in the next state), you apologize"

MetateMisa 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 form. This past 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. The instantiated future-time consequents become commitments which must subsequently be satisfied.

The instantiated future-time commitments become constraints on this model.

The future-time parts of instantiated rules represent constraints on the program rules.

Execution is thus a process of iteratively generalizing a model for the formulamade up of the program rules.

There is also separated normal form.

There are algorithms for executing MetateM programs that appear to give reasonable performance.

First rule ensures that an ask is eventually followed by a give.

\( \lambda \in \text{ask}(x) \Rightarrow \lambda \cdot \text{give}(x) \) \( x \wedge \text{give}(x) \Rightarrow \lambda \cdot \text{ask}(x) \wedge A \)

Second rule ensures that only one give is ever performed at any one time.

An example MetateM program: the resource controller...

\( x \cdot \text{give}(x) \Rightarrow x \cdot \text{give}(y) \)

\( x \cdot \text{give}(y) \Rightarrow (x=y) \)

Firstruleensuresthatan'ask'iseventuallyfollowedbyagive.

Secondruleensuresthatonlyone'give'iseverperformed at any onetime.

There are algorithms for executing MetateM programs that appear to give reasonable performance.

There is also separated normal form.

There are algorithms for executing MetateM programs that appear to give reasonable performance.

First rule ensures that an ask is eventually followed by a give.

\( \lambda \in \text{ask}(x) \Rightarrow \lambda \cdot \text{give}(x) \) \( x \wedge \text{give}(x) \Rightarrow \lambda \cdot \text{ask}(x) \wedge A \)

Second rule ensures that only one give is ever performed at any one time.

An example MetateM program: the resource controller...

\( x \cdot \text{give}(x) \Rightarrow x \cdot \text{give}(y) \)

\( x \cdot \text{give}(y) \Rightarrow (x=y) \)

Firstruleensuresthatan'ask'iseventuallyfollowedbyagive.

Secondruleensuresthatonlyone'give'iseverperformed at any onetime.

There are algorithms for executing MetateM programs that appear to give reasonable performance.

There is also separated normal form.

There are algorithms for executing MetateM programs that appear to give reasonable performance.

First rule ensures that an ask is eventually followed by a give.

\( \lambda \in \text{ask}(x) \Rightarrow \lambda \cdot \text{give}(x) \) \( x \wedge \text{give}(x) \Rightarrow \lambda \cdot \text{ask}(x) \wedge A \)

Second rule ensures that only one give is ever performed at any one time.

An example MetateM program: the resource controller...

\( x \cdot \text{give}(x) \Rightarrow x \cdot \text{give}(y) \)

\( x \cdot \text{give}(y) \Rightarrow (x=y) \)

Firstruleensuresthatan'ask'iseventuallyfollowedbyagive.

Secondruleensuresthatonlyone'give'iseverperformed at any onetime.
Lecture 3: An Introduction to Multiagent Systems

An object's interface contains two sets:
– environment predicates—these correspond to messages the object will accept;
– component predicates—these correspond to messages the object may send.

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

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

\[
\text{pop}, \text{push} = \text{environment preds}
\]
\[
\text{popped}, \text{stackfull} = \text{component preds}
\]

If an agent receives a message headed by an environment predicate, it accepts it.
If an agent receives a message headed by an environment predicate
– component predicates — corresponding to messages the object may send.
– environment predicates — those corresponding to messages the object will accept.

To illustrate the language Concurrent Metal, here are some example programs...

- Snow White has some sweets (resources), which she will give to the dwarfs (resource consumers).
- She will always give to a dwarf that asks.
- She will only give to one dwarf at a time.
- The dwarfs (resource consumers) will eventually give a sweet when none else has.

Fortunetely, some have better manners. Courteous, only asks when
- Snow White has some sweets (resources), which she will give to.
- She will only give to one dwarf at a time.
- It broadcasts if.
- She will always give to a dwarf that asks.
- The dwarfs (resource consumers) may send.

- Courteous (give) \[\text{ask}\]
  \[
  \text{ask}(\text{courteous}) \iff ( ( ( \text{give}(\text{eager}) \lor \neg \text{give}(\text{greedy}) ) \land \neg \text{give}(\text{eager}) ) \lor ( \text{give}(\text{greedy}) \land \neg \text{give}(\text{eager}) ))
  \]
  \[
  \text{ask}(\text{eager}) \iff \text{give}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]

- Some dwarves are even less polite. Greedy just asks every time.

- Greedy (give) \[\text{ask}\]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]
  \[
  \text{ask}(\text{greedy}) \iff \text{give}(\text{eager})
  \]
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]

- The dwarf, eager, asks for a sweet initially, and then whenever he
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]

- The dwarf, eager, asks for a sweet initially, and then whenever he
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]

- Snow White has some sweets (resources), which she will give to the dwarfs (resource consumers).
- She will always give to a dwarf that asks.
- She will only give to one dwarf at a time.
- The dwarfs (resource consumers) will eventually give a sweet when none else has.

Fortunetely, some have better manners. Courteous, only asks when

- Snow White has some sweets (resources), which she will give to.
- She will only give to one dwarf at a time.
- It broadcasts if.
- She will always give to a dwarf that asks.
- The dwarfs (resource consumers) may send.

- Courteous (give) \[\text{ask}\]
  \[
  \text{ask}(\text{courteous}) \iff ( ( ( \text{give}(\text{eager}) \lor \neg \text{give}(\text{greedy}) ) \land \neg \text{give}(\text{eager}) ) \lor ( \text{give}(\text{greedy}) \land \neg \text{give}(\text{eager}) ))
  \]
  \[
  \text{ask}(\text{eager}) \iff \text{give}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]

- Some dwarves are even less polite. Greedy just asks every time.

- Greedy (give) \[\text{ask}\]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]
  \[
  \text{ask}(\text{greedy}) \iff \text{give}(\text{eager})
  \]
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]

- The dwarf, eager, asks for a sweet initially, and then whenever he
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]

- The dwarf, eager, asks for a sweet initially, and then whenever he
  \[
  \text{give}(\text{eager}) \iff \text{ask}(\text{eager})
  \]
  \[
  \text{start} \iff \text{give}(\text{eager})
  \]
Lecture 3 An Introduction to Multiagent Systems

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!
- not be used in current state to implement 'full' system.

2.4 Planning agents

Since the early 1970's, the AI planning community has been focused on the design of automatically generating plans for artificial agents. Planning is essentially automatic program planning: the design of a course of action that will achieve some desired goal.

- Planning has been an active research area for many years.
- Within the symbolic AI community, it has long been assumed that a central component of any artificial agent will be a central component of any time-constrained system.
- But in the mid 1980's, Chapman established some theoretical planning results which indicate that AI planners will ultimately turn out to be unusable in any time-constrained system.
- Within the symbolic AI community, planning systems have been proposed, but lacking any particular system to be used in current state to implement 'full' system.
- Building largely on the early work of Fikes & Nilsson, many planning algorithms have been proposed, and the theory of planning has been well-developed.

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