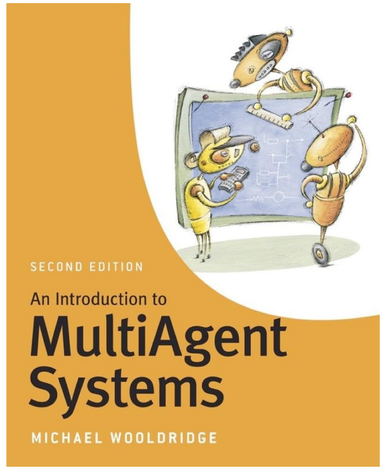
# IT4899
# Multi-Agent Systems
Chapter 1 - Introduction

Dr. Nguyen Binh Minh
Department of Information Systems








## Five Trends in the History of Computing

- ubiquity;
- interconnection;
- intelligence;
- delegation;
- human-orientation.

# Ubiquity

- Continual reduction in cost of computing makes it possible to introduce processing power into places and devices that would have once been uneconomic.
- As processing capability spreads, sophistication (and intelligence of a sort) becomes *ubiquitous*.
- What could benefit from having a processor embedded in it?



3

# Home Automation Wars

- Apple announced "*HomeKit*" in 2014, & rolled out full App support in iOS10 in Sept 2016
- Siri-driven HopePod released February, 2018


- Amazon launched "*Echo*" in the UK on 26th Sept, 2016


- Google announced "*Home*" in May 2016, with a launch date planned in Nov 2016



4

# Interconnection

- Computer systems no longer stand alone, but are networked into large distributed systems.
- Internet an obvious example, but networking is spreading its ever-growing tentacles.
- Since distributed and concurrent systems have become the norm, some researchers are putting forward theoretical models that portray computing as primarily a process of interaction.



5

# Intelligence

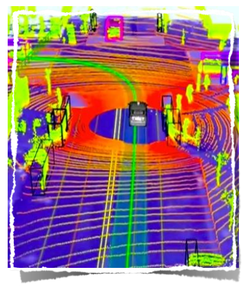
- The complexity of tasks that we are capable of automating and delegating to computers has grown steadily
  - Many of these tasks are ones that can be thought of as requiring a good deal of intelligence
- If you don’t feel comfortable with this definition of “intelligence”, it’s probably because you are a human. . .



6

# Delegation

- Computers are doing more for us . . . without our intervention
- We are giving control to computers, even in safety critical tasks
- One example:
  - fly-by-wire aircraft, where the machine's judgment may be trusted more than an experienced pilot
- Next on the agenda:
  - fly-by-wire cars, intelligent braking systems, cruise control that maintains distance from car in front. . .



7

# Human Orientation

- The movement away from machine-oriented views of programming toward concepts and metaphors that more closely reflect the way we ourselves understand the world
  - Programmers (and users!) relate to the machine differently
- Programmers conceptualize and implement software in terms of ever higher-level – more *human-oriented* – abstractions



8

# Abstractions

- Remember: most important developments in computing are based on new abstractions.
- Just as moving from machine code to higher level languages brings an efficiency gain, so does moving from objects to agents.
  - The following 2006 paper claims that developing complex applications using agent-based methods leads to an average saving of 350% in development time (and up to 500% over the use of Java).
    - S. Benfield, *Making a Strong Business Case for Multiagent Technology*, Invited Talk at AAMAS 2006.

*Programming has progressed through:*

- *machine code;*
- *assembly language;*
- *machine-independent programming languages;*
- *sub-routines; procedures*
- *& functions; abstract*
- *data types; objects;*
- 

*to*

- *Agents, as intentional systems, that represent a further, and increasingly powerful abstraction.*



9

# Other Trends in Computer Science

- the Grid/Cloud;
- ubiquitous computing;
- semantic web.



10

# The Grid/Cloud

- The *Grid* aims to develop massive-scale open distributed systems, capable of being able to effectively and automatically deploy and redeploy computational (and other) resources to solve *large computational problems*:
  - huge datasets;
  - huge processing requirements.
- Current Grid research focussed mainly on *middleware*





# The Grid and MAS

'The Grid and agent communities are both pursuing the development of such open distributed systems, albeit from different perspectives. The Grid community has historically focussed on [. . . ] "***brawn***": interoperable infrastructure and tools for secure and reliable resource sharing within dynamic and geographically distributed virtual organisations (VOs), and applications of the same to various resource federation scenarios.

In contrast, those working on agents have focussed on "***brains***", i.e., on the development of concepts, methodologies and algorithms for autonomous problem solvers that can act flexibly in uncertain and dynamic environments in order to achieve their objectives.'

(Foster et al, 2004)

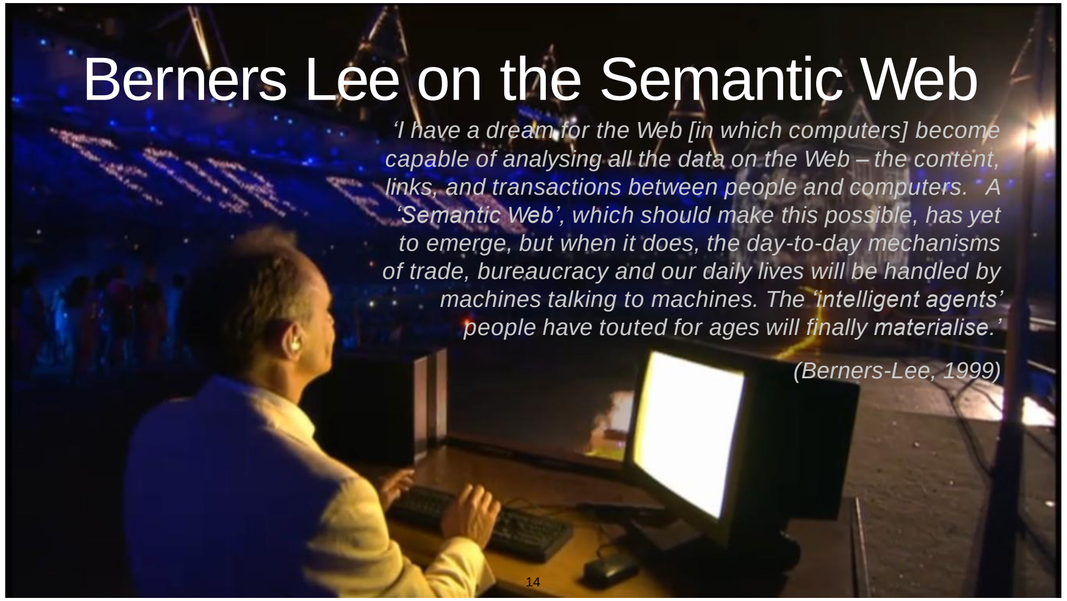
# The Semantic Web

- The semantic web aims to annotate web sites with semantic markup: information in a form processable by computer, typically relating to the content of the web site.

- The idea is that this markup will enable browsers (etc) provide richer, more meaningful services to users.







# Agents: A First Definition

- An agent is a computer system that is capable of independent (*autonomous*) action on behalf of its user or owner
  - i.e. figuring out what needs to be done to satisfy design objectives, rather than constantly being told.



15

# Multi-Agent Systems: A First Definition

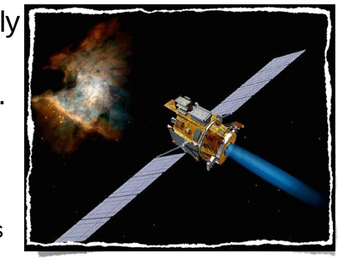
- A multiagent system is one that consists of a number of agents, which interact with one-another.
- In the most general case, agents will be acting on behalf of users with different goals and motivations.
- To successfully interact, they will require the ability to cooperate, coordinate, and negotiate with each other, much as people do.



16

# A Vision: Autonomous Space Probes

- When a space probe makes its long flight from Earth to the outer planets, a ground crew is usually required to continually track its progress, and decide how to deal with unexpected eventualities.
  - This is costly and, if decisions are required *quickly*, it is simply not practicable.
  - For these reasons, organisations like NASA are seriously investigating the possibility of making probes more autonomous
    - giving them richer decision making capabilities and responsibilities.



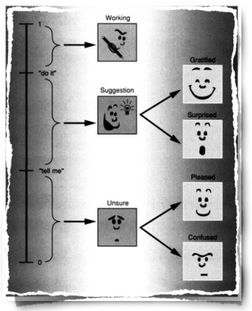
- This is not fiction: NASA's DS1 did it 20 years ago in 1998!





# A Vision: Internet Agents

- Searching the Internet for the answer to a specific query can be a long and tedious process.
  - So, why not allow a computer program — an agent — do searches for us?
  - The agent would typically be given a query that would require synthesising pieces of information from various different Internet information sources.
  - Failure would occur when a particular resource was unavailable, (perhaps due to network failure), or where results could not be obtained.



# The Micro and Macro Problems

- Agent design
  - How do we build agents that are capable of independent, autonomous action in order to successfully carry out the tasks that we delegate to them?
- Society Design
  - How do we build agents that are capable of interacting (cooperating, coordinating, negotiating) with other agents in order to successfully carry out the tasks that we delegate to them, particularly when the other agents cannot be assumed to share the same interests/goals?



19

# Some Views of the Field

- Agents as a paradigm for software engineering:
  - Software engineers have derived a progressively better understanding of the characteristics of complexity in software. It is now widely recognised that *interaction* is probably the most important single characteristic of complex software.
- Agents as a tool for understanding human societies:
  - Multiagent systems provide a novel new tool for simulating societies, which may help shed some light on various kinds of social processes.



20

# Some Views of the Field

- Agents are the achievable bit of the AI project:
  - The aim of Artificial Intelligence as a field is to produce general human-level intelligence. This requires a very high level of performance in lots of areas:
    - Vision
    - Natural language understanding/generation
    - Reasoning
  - Building an agent that can perform well on a narrowly defined task in a specific environment is much much easier (though not easy).



21

# Objections to MAS

- Isn’t it all just Distributed/Concurrent Systems?
- Isn’t it all just AI?
- Isn’t it all just Economics/Game Theory?
- Isn’t it all just Social Science?



22

# Summary

- This has been a brief introduction to "*An Introduction to Multiagent Systems*"
- We have argued that MAS are:
  - a natural development of computer science;
  - a natural means to handle ever more distributed systems; and
  - not science fiction :-)
- We also made a first definition of agent and multiagent system.

*Class Reading (Chapter 1):*

*"Readings in Distributed Artificial Intelligence.", Bond, A. H. and Gasser, L., editors (1988). Morgan Kaufmann Publishers: San Mateo, CA (Introduction)*

This article is probably the best survey of the problems and issues associated with multiagent systems. Most of the issues are fundamentally still open!

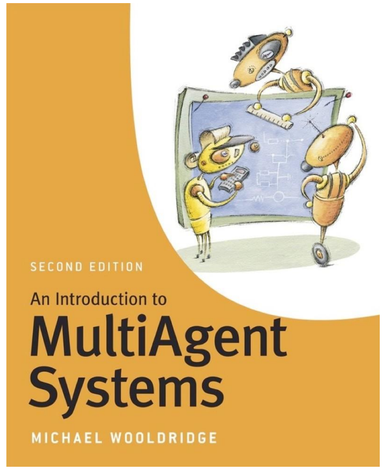


23

# IT4899
# Multi-Agent Systems
Chapter 2 - Intelligent Agents

Dr. Nguyen Binh Minh
Department of Information Systems







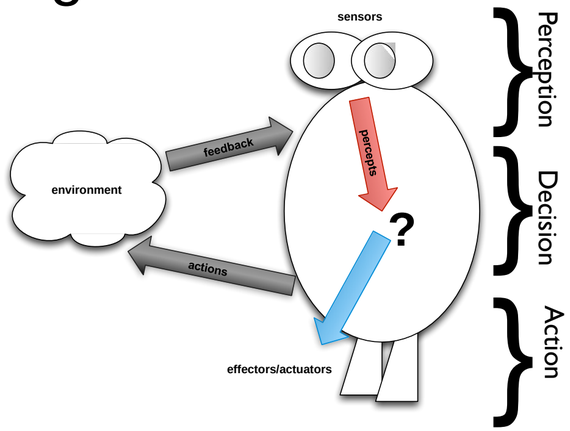

# What is an Agent?

- The main point about agents is they are *autonomous*: capable independent action.

> "... An agent is a computer system that is *situated* in some *environment*, and that is capable of *autonomous action* in that environment in order to meet its delegated objectives..."

- It is all about decisions
  - An agent has to choose ***what*** action to perform.
  - An agent has to decide ***when*** to perform an action.

# Agent and Environment

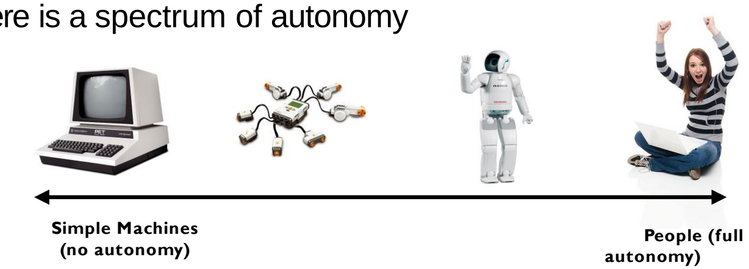







# Autonomy

- There is a spectrum of autonomy




- Autonomy is adjustable
  - Decisions handed to a higher authority when this is beneficial

# Simple (Uninteresting) Agents

- Control Systems
  - Example: *Thermostat* (physical environment)
  - *delegated goal* is maintain room temperature
  - *actions* are heat on/off


- Software Demons
  - Example: *UNIX biff program* (software environment)
  - *delegated goal* is monitor for incoming email and flag it
  - *actions* are GUI actions.

- They are trivial because the decision making they do is trivial.



5

# Agents and Objects

- *Are agents just objects by another name?*

- Object:
  - encapsulates some state;
  - communicates via message passing;
  - has methods, corresponding to operations that may be performed on this state.

"… Agents are objects with **attitude**…"



6

# Differences between Agents and Objects

- Agents are autonomous:
  - agents embody stronger notion of autonomy than objects, and in particular, they decide for themselves whether or not to perform an action on request from another agent;
- Agents are smart:
  - capable of flexible (reactive, pro-active, social) behaviour – the standard object-oriented model has nothing to say about such types of behaviour;
- Agents are active:
  - not passive service providers; a multi-agent system is inherently multi-threaded, in that each agent is assumed to have at least one thread of active control.

Objects do it because they **have to**!

Objects do it for **free**!

Agents do it because they **want to**!

Agents do it for personal gain!

7



# Aren’t agents just expert systems by another name?

- Expert systems typically disembodied ‘expertise’ about some (abstract) domain of discourse.
  - agents are *situated in an environment*
- MYCIN is not aware of the world — only information obtained is by asking the user questions.
  - agents *act*
- MYCIN does not operate on patients.

**MYCIN** is an example of an Expert System that knows about blood diseases in humans.

It has a wealth of knowledge about blood diseases, in the form of rules.

A doctor can obtain expert advice about blood diseases by giving MYCIN facts, answering questions, and posing queries.

8

# Intelligent Agents and AI

- When building an agent, we simply want a system that can choose the right action to perform, typically in a limited domain.
- We do not have to solve all the problems of AI to build a useful agent:

**"...a little intelligence goes a long way!.."**

- Oren Etzioni, speaking about the commercial experience of NETBOT, Inc:

*"… We made our agents dumber and dumber and dumber . . . until finally they made money…"*





# Properties of Environments

- Since agents are in close contact with their environment, the properties of the environment affect agents.
  - Also have a big effect on those of us who build agents.
- Common to categorise environments along some different dimensions.
  - Fully observable vs partially observable
  - Deterministic vs non-deterministic
  - Static vs dynamic
  - Discrete vs continuous
  - Episodic vs non-episodic
  - Real Time

# Properties of Environments

- Fully observable vs partially observable.
  - An accessible or ***fully observable*** environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state.
  - Most moderately complex environments (including, for example, the everyday physical world and the Internet) are inaccessible, or ***partially observable***.
  - The more accessible an environment is, the simpler it is to build agents to operate in it.



# Properties of Environments

- Deterministic vs non-deterministic.
  - A deterministic environment is one in which any action has a single guaranteed effect — there is no uncertainty about the state that will result from performing an action.
  - The physical world can to all intents and purposes be regarded as non-deterministic.
  - We'll follow Russell and Norvig in calling environments ***stochastic*** if we quantify the non-determinism using probability theory.
  - Non-deterministic environments present greater problems for the agent designer.

# Properties of Environments

- Static vs dynamic.
  - A ***static*** environment is one that can be assumed to remain unchanged except by the performance of actions by the agent.
  - A ***dynamic*** environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control.
  - The physical world is a highly dynamic environment.
  - One reason an environment may be dynamic is the presence of other agents.





# Properties of Environments

- Discrete vs continuous.
  - An environment is discrete if there are a fixed, finite number of actions and percepts in it.
    - Otherwise it is continuous
  - Russell and Norvig give a chess game as an example of a discrete environment, and taxi driving as an example of a continuous one.

- Often we treat a continuous environment as discrete for simplicity

# Properties of Environments

- Episodic vs non-episodic
  - In an ***episodic*** environment, the performance of an agent is dependent on a number of discrete episodes, with no link between the performance of an agent in different scenarios.
    - An example of an episodic environment would be an assembly line where an agent had to spot defective parts.
  - Episodic environments are simpler from the agent developer's perspective because the agent can decide what action to perform based only on the current episode — it need not reason about the interactions between this and future episodes.
    - Relations to the Markov property
  - Environments that are not episodic are sometimes called ***non-episodic*** or ***sequential***.
    - Here the current decision affects future decisions.
    - Driving a car is sequential.





# Properties of Environments

- Real time
  - A *real time* interaction is one in which time plays a part in evaluating an agents performance
  - Such interactions include those in which:
    - A decision must be made about some action within a given time bound
    - Some state of affairs must occur as quickly as possible
    - An agent has to repeat some task, with the objective to repeat the task as often as possible

# Intelligent Agents

- We typically think of as intelligent agent as exhibiting 3 types of behaviour:
  - Reactive (environment aware)
  - Pro-active (goal-driven);
  - Social Ability.





# Reactivity

- If a program's environment is guaranteed to be fixed, the program need never worry about its own success or failure
- Program just executes blindly.
  - Example of fixed environment: compiler.
- The real world is not like that: most environments are *dynamic* and information is *incomplete*.

# Reactivity

- Software is hard to build for dynamic domains: program must take into account possibility of failure
  - ask itself whether it is worth executing!
- A *reactive* system is one that maintains an ongoing interaction with its environment, and responds to changes that occur in it (in time for the response to be useful).



19

# Proactiveness

- Reacting to an environment is easy
  - e.g., stimulus → response rules
- But we generally want agents to *do things for us*.
  - Hence *goal directed behaviour*.
- *Pro-activeness* = generating and attempting to achieve goals; not driven solely by events; taking the initiative.
  - Also: recognising opportunities.



20

# Social Ability

- The real world is a *multi*-agent environment: we cannot go around attempting to achieve goals without taking others into account.
  - Some goals can only be achieved by interacting with others.
  - Similarly for many computer environments: witness the INTERNET.
- *Social ability* in agents is the ability to interact with other agents (and possibly humans) via *cooperation*, *coordination*, and *negotiation*.
  - At the very least, it means the ability to communicate. . .





# Social Ability: Cooperation

- Cooperation is *working together as a team to achieve a shared goal*.
- Often prompted either by the fact that no one agent can achieve the goal alone, or that cooperation will obtain a better result (e.g., get result faster).

# Social Ability: Coordination

- Coordination is *managing the interdependencies between activities.*
- For example, if there is a non-sharable resource that you want to use and I want to use, then we need to coordinate.



23

# Social Ability: Negotiation

- Negotiation is the ability to reach *agreements* on matters of common interest.
- For example:
  - You have one TV in your house; you want to watch a movie, your housemate wants to watch football.
  - A possible deal: watch football tonight, and a movie tomorrow.
- Typically involves *offer and counter-offer*, with compromises made by participants.



24

# Some Other Properties...

- Mobility
  - The ability of an agent to move. For software agents this movement is around an electronic network.
- Rationality
  - Whether an agent will act in order to achieve its goals, and will not deliberately act so as to prevent its goals being achieved.
- Veracity
  - Whether an agent will knowingly communicate false information.
- Benevolence
  - Whether agents have conflicting goals, and thus whether they are inherently helpful.
- Learning/adaption
  - Whether agents improve performance over time.



25

# Agents as Intentional Systems

- When explaining human activity, it is often useful to make statements such as the following:
  - Janine took her umbrella because she *believed* it was going to rain.
  - Michael worked hard because he *wanted* to possess a PhD.
- These statements make use of a *folk psychology*, by which human behaviour is predicted and explained through the attribution of *attitudes*
  - *e.g. believing, wanting, hoping, fearing ...*
- The attitudes employed in such folk psychological descriptions are called the *intentional* notions.



26

# Intentional Systems

- It provides us with a familiar, non-technical way of understanding and explaining agents.



31

# Abstract Architectures for Agents

- Assume the world may be in any of a finite set $E$ of discrete, instantaneous states

$$E = \{e, e', \ldots\}$$

- Agents are assumed to have a repertoire of possible actions, $Ac$, available to them, which transform the state of the world.
  - Actions can be non-deterministic, but only one state ever results from and action.

$$Ac = \{\alpha, \alpha', \ldots\}$$

- A run, $r$, of an agent in an environment is a sequence of interleaved world states and actions:

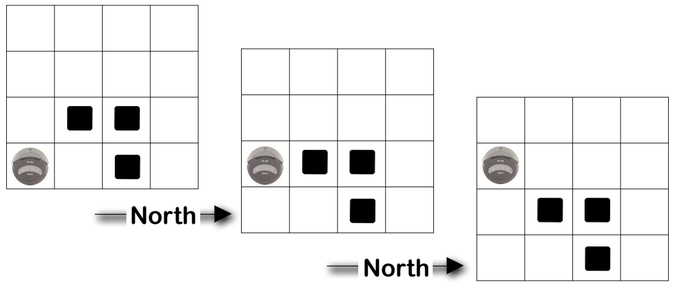
$$r : e_0 \xrightarrow{\alpha_0} e_1 \xrightarrow{\alpha_1} e_2 \xrightarrow{\alpha_2} e_3 \xrightarrow{\alpha_3} \cdots \xrightarrow{\alpha_{u-1}} e_u$$



37

# Abstract Architectures for Agents (1)

- When actions are deterministic each state has only one possible successor.
- A run would look something like the following:

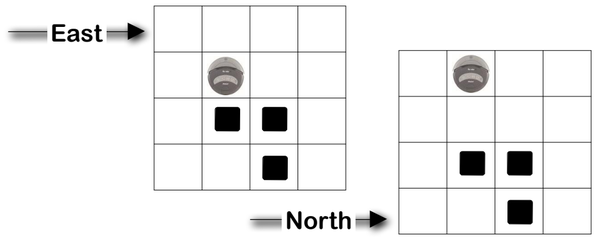




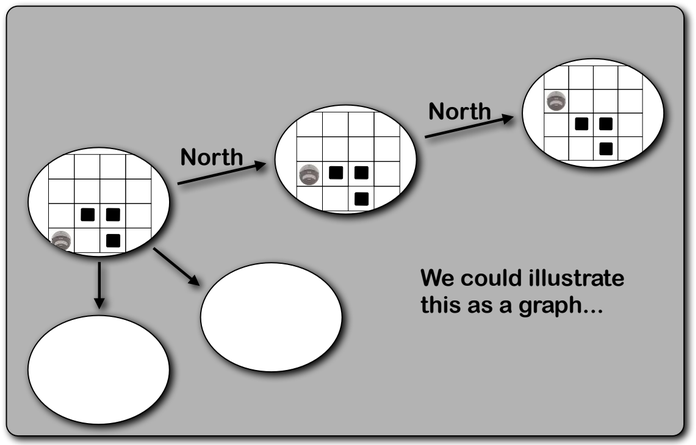



# Abstract Architectures for Agents (2)

- When actions are deterministic each state has only one possible successor.
- A run would look something like the following:



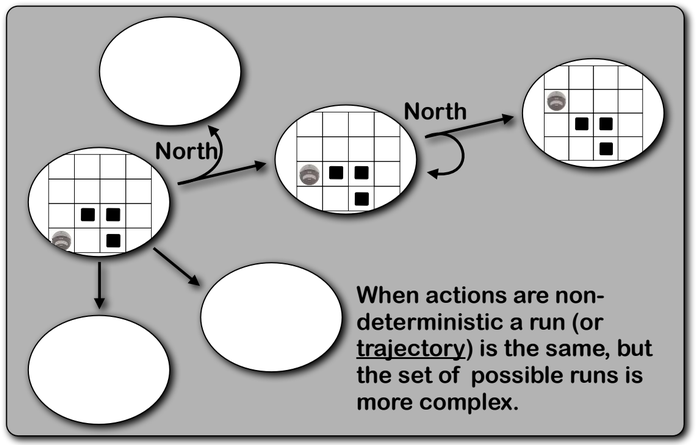
# Abstract Architectures for Agents








# Abstract Architectures for Agents



# Runs

- In fact it is more complex still, because all of the runs we pictured start from the same state.
- Let:

$\mathcal{R}$ be the set of all such possible finite sequences (over $E$ and $Ac$);
$\mathcal{R}^{Ac}$ be the subset of these that end with an action; and
$\mathcal{R}^{E}$ be the subset of these that end with a state.

- We will use *r, r′,...* to stand for the members of R
  - These sets of runs contain *all* runs from *all* starting states.



# Environments

- A state transformer function represents behaviour of the environment:

$$\tau : \mathcal{R}^{Ac} \longrightarrow 2^{E}$$

- Note that environments are...
  - *history dependent*: the next state not only dependent on the action of the agent, but an earlier action may be significant
  - *non-deterministic*: There is some uncertainty about the result
- If $\tau(r) = \emptyset$ there are no possible successor states to r, so we say the run has ended. (“Game over.”)
  - An environment $Env$ is then a triple $Env = \langle E, e_0, \tau \rangle$ where $E$ is set of states, $e_0 \in E$ is initial state; and $\tau$ is state transformer function.

# Agents

- We can think of an agent as being a function which maps runs to actions: $Ag : \mathcal{R}^E \rightarrow Ac$
- Thus an agent makes a decision about what action to perform
  - based on the history of the system that it has witnessed to date.
- Let $Ag$ be the set of all agents.



44

# System

- A system is a *pair* containing an *agent* and an *environment*.
- Any system will have associated with it a set of possible runs
  - We denote the set of runs of agent $Ag$ in environment $Env$ by:

$$\mathcal{R}(Ag, Env)$$

- Assume that this only contains runs that have ended.



45

# Systems

Formally, a sequence

$$(e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots)$$

represents a run of an agent $Ag$ in environment $Env = \langle E, e_0, \tau \rangle$ if:

1. $e_0$ is the initial state of $Env$
2. $\alpha_0 = Ag(e_0)$; and
3. for $u > 0$,

$$\begin{array}{rcl} e_u & \in & \tau((e_0, \alpha_0, \ldots, \alpha_{u-1})) \quad \text{and} \\ \alpha_u & = & Ag((e_0, \alpha_0, \ldots, e_u)) \end{array}$$



46

# Why the notation?

- Well, it allows us to get a precise handle on some ideas about agents.
  - For example, we can tell when two agents are the same.
- Of course, there are different meanings for “same”. Here is one specific one

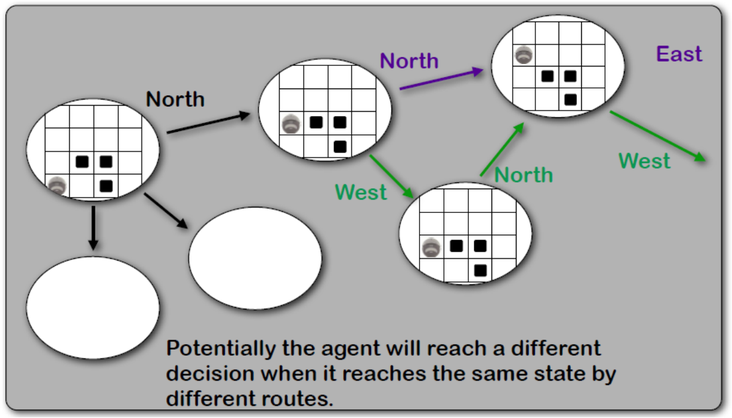
Two agents are said to be *behaviorally equivalent* with respect to $Env$ iff $\mathcal{R}(Ag_1, Env) = \mathcal{R}(Ag_2, Env)$.

- We won’t be able to tell two such agents apart by watching what they do.



47

# Deliberative Agents








# Purely Reactive Agents

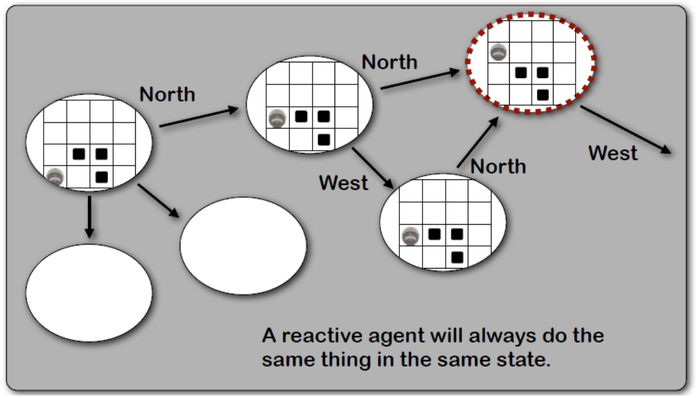
- Some agents decide what to do without reference to their history
  - they base their decision making entirely on the present, with no reference at all to the past.
- We call such agents purely reactive:

$$action : E \rightarrow Ac$$

- A thermostat is a purely reactive agent.

$$action(e) = \begin{cases} \text{off} & \text{if } e = \text{temperature OK} \\ \text{on} & \text{otherwise.} \end{cases}$$

# Reactive Agents

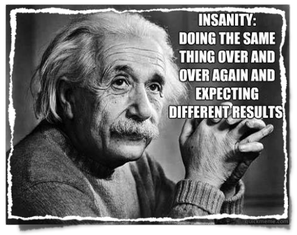




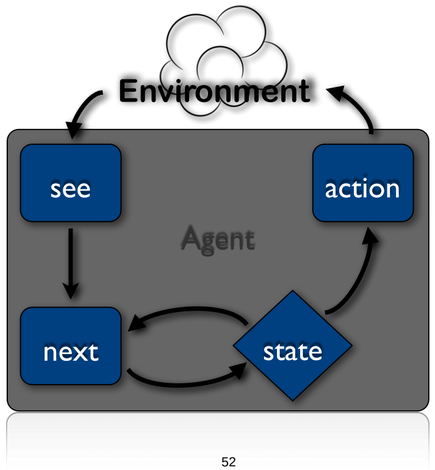



# Purely Reactive Robots

- A simple reactive program for a robot might be:
  - *Drive forward until you bump into something. Then, turn to the right. Repeat.*




# Agents with State





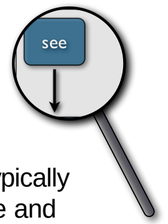



# Perception

- The see function is the agent's ability to observe its environment, whereas the action function represents the agent's decision making process.
- Output of the see function is a percept:

$$see : E \rightarrow Per$$

  - ...which maps environment states to percepts.




- The agent has some internal data structure, which is typically used to record information about the environment state and history.
- Let *I* be the set of all internal states of the agent.

# Actions and Next State Functions

- The action-selection function *action* is now defined as a mapping from internal states to actions:

$$action : I \rightarrow Ac$$


- An additional function *next* is introduced, which maps an internal state and percept to an internal state:

$$next : I \times Per \rightarrow I$$


- This says how the agent updates its view of the world *when it gets a new percept*.





# Agent Control Loop

1. Agent starts in some initial internal state $i_0$.
2. Observes its environment state $e$, and generates a percept $see(e)$.
3. Internal state of the agent is then updated via $next$ function, becoming $next(i_0, see(e))$.
4. The action selected by the agent is $action(next(i_0, see(e)))$.
   This action is then performed.
5. Goto (2).

# Tasks for Agents

- We build agents in order to carry out *tasks* for us.
  - The task must be *specified* by us. . .
- But we want to tell agents what to do *without* telling them how to do it.
  - How can we make this happen???





# Utility functions

- One idea:
  - associated *rewards* with states that we want agents to bring about.
  - We associate *utilities* with individual states
    - the task of the agent is then to bring about states that maximise utility.
- A task specification is then a function which associates a real number with every environment state:

$$u : E \rightarrow \mathbb{R}$$

# Local Utility Functions

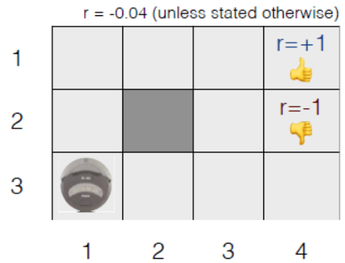
- But what is the value of a run...
  - minimum utility of state on run?
  - maximum utility of state on run?
  - sum of utilities of states on run?
  - average?
- Disadvantage:
  - difficult to specify a *long term* view when assigning utilities to individual states.
- One possibility:
  - a *discount* for states later on. This is what we do in *reinforcement learning*.





# Example of local utility function

- Goal is to select actions to *maximise future rewards*
  - Each action results in moving to a state with some assigned reward
  - Allocation of that reward may be immediate or delayed (e.g. until the end of the run)
  - It may be better to sacrifice immediate reward to gain more long-term reward

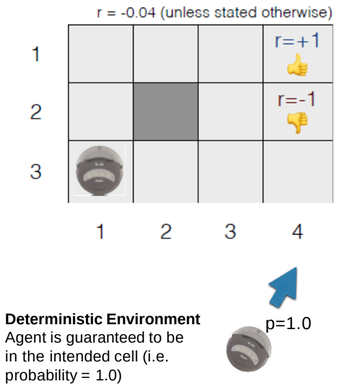



- We can illustrate with a simple 4x3 environment
  - What actions maximise the reward?

# Example of local utility function

Assume environment was *deterministic*

- *Optimal Solution* is:
  - [Up, Up, Right, Right, Right]
- *Additive Reward* is:
  - r = (-0.04 x 4) + 1.0
  - r = 1.0 - 0.16 = 0.84
- i.e. the utility gained is the sum of the rewards received
  - The negative (-0.04) reward incentivises the agent to reach its goal asap.




**Deterministic Environment**
Agent is guaranteed to be in the intended cell (i.e. probability = 1.0)



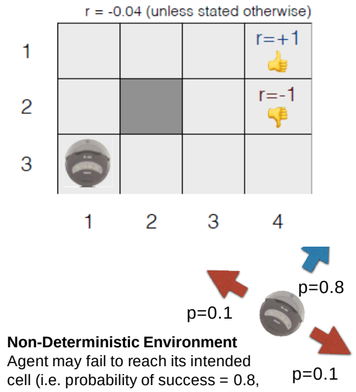


# Sequential Decision Making

*Returning to our earlier example*

- However, now assume environment was *non-deterministic*

- Probability of reaching the goal if successful:
  - $p = 0.8^5 = 0.32768$
- Could also reach the goal accidentally by going the wrong way round:
  - $p = 0.1^4 \times 0.8 = 0.0001 \times 0.8 = 0.00008$
  - Final probability of reaching the goal: p = 0.32776
- Utility gained depends on the route taken
  - We will see later how to compute this…
    - Reinforcement Learning builds upon this type of model




**Non-Deterministic Environment**
Agent may fail to reach its intended cell (i.e. probability of success = 0.8, but may move sideways with p=0.1 in each direction

# Utilities over Runs

- Another possibility: assigns an utility not to individual states, but to runs themselves:

$$u : \mathcal{R} \rightarrow \mathbb{R}$$

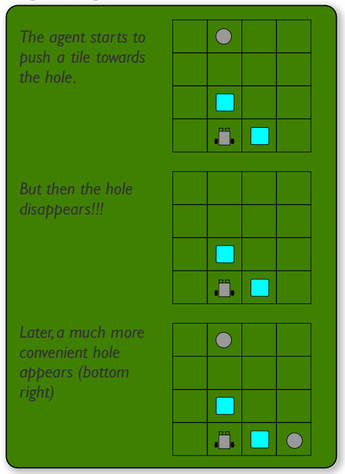
- Such an approach takes an inherently *long term view*.
- Other variations:
  - incorporate probabilities of different states emerging.
- To see where utilities might come from, let's look at an example.





# Utility in the Tileworld

- Simulated two dimensional grid environment on which there are *agents*, *tiles*, *obstacles*, and *holes*.
- An agent can move in four directions:
  - up, down, left, or right
  - If it is located next to a tile, it can push it.
- Holes have to be filled up with tiles by the agent.
  - An agent scores points by filling holes with tiles, with the aim being to fill as many holes as possible.
- TILEWORLD changes with the random appearance and disappearance of holes.



# Utilities in the Tileworld

- Utilities are associated over runs, so that more holes filled is a higher utility.
  - Utility function defined as follows: $u(r) \doteq \frac{\text{number of holes filled in } r}{\text{number of holes that appeared in } r}$
  - Thus:
    - if agent fills all holes, utility = 1.
    - if agent fills no holes, utility = 0.
- TILEWORLD captures the need for *reactivity* and for the advantages of exploiting opportunities.





# Expected Utility

- To denote probability that run r occurs when agent Ag is placed in environment Env, we can write:

$$P(r \mid Ag, Env)$$

- In a non-deterministic environment, for example, this can be computed from the probability of each step.

For a run $r = (e_0, \alpha_0, e_1, \alpha_1, e_2, \ldots)$:

$$P(r \mid Ag, Env) = P(e_1, \mid e_0, \alpha_0) P(e_2 \mid e_1, \alpha_1) \ldots$$

and clearly:

$$\sum_{r \in \mathcal{R}(Ag, Env)} P(r \mid Ag, Env) = 1.$$

# Expected Utility

- The expected utility ($EU$) of agent $Ag$ in environment $Env$ (given $P$, $u$), is then:

$$EU(Ag, Env) = \sum_{r \in \mathcal{R}(Ag, Env)} u(r)P(r \mid Ag, Env).$$

- That is, for each run we compute the utility and multiply it by the probability of the run.

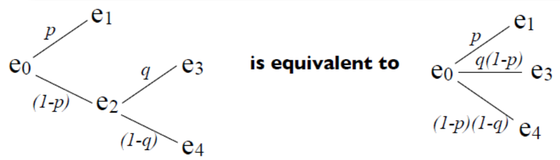
- The expected utility is then the sum of all of these.



# Expected Utility

- The probability of a run can be determined from individual actions within a run
  - Using the decomposability axiom from Utility Theory

*"…Compound lotteries can be reduced to simpler ones using the law of probability. Known as the "no fun in gambling" as two consecutive lotteries can be compressed into a single equivalent lottery…."*



# Optimal Agents

- The optimal agent $Ag_{opt}$ in an environment $Env$ is the one that maximizes expected utility:

$$Ag_{opt} = \arg \max_{Ag \in \mathcal{AG}} EU(Ag, Env)$$

- Of course, the fact that an agent is optimal does *not mean that it will* be best; only that *on average*, we can expect it to do best.





# Example 1

Consider the environment $Env_1 = \langle E, e_0, \tau \rangle$ defined as follows:

$$E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$$

$$\tau(e_0 \xrightarrow{\alpha_0}) = \{e_1, e_2\}$$

$$\tau(e_0 \xrightarrow{\alpha_1}) = \{e_3, e_4, e_5\}$$

There are two agents possible with respect to this environment:

$$Ag_1(e_0) = \alpha_0$$

$$Ag_2(e_0) = \alpha_1$$



The probabilities of the various runs are as follows:

$$P(e_0 \xrightarrow{\alpha_0} e_1 \mid Ag_1, Env_1) = 0.4$$

$$P(e_0 \xrightarrow{\alpha_0} e_2 \mid Ag_1, Env_1) = 0.6$$

$$P(e_0 \xrightarrow{\alpha_1} e_3 \mid Ag_2, Env_1) = 0.1$$

$$P(e_0 \xrightarrow{\alpha_1} e_4 \mid Ag_2, Env_1) = 0.2$$

$$P(e_0 \xrightarrow{\alpha_1} e_5 \mid Ag_2, Env_1) = 0.7$$

Assume the utility function $u_1$ is defined as follows:

$$u_1(e_0 \xrightarrow{\alpha_0} e_1) = 8$$

$$u_1(e_0 \xrightarrow{\alpha_0} e_2) = 11$$

$$u_1(e_0 \xrightarrow{\alpha_1} e_3) = 70$$

$$u_1(e_0 \xrightarrow{\alpha_1} e_4) = 9$$

$$u_1(e_0 \xrightarrow{\alpha_1} e_5) = 10$$

What are the expected utilities of the agents for this utility function?

# Example 1 Solution

Given the utility function $u_1$ in the question, we have two transition functions defined as $\tau(e_0 \xrightarrow{\alpha_0}) = \{e_1, e_2, e_3\}$, and $\tau(e_0 \xrightarrow{\alpha_1}) = \{e_4, e_5, e_6\}$. The probabilities of the various runs (two for the first agent and three for the second) is given in the question, along with the probability of each run occurring. Given the definition of the utility function $u_1$, the *expected utilities* of agents $Ag_0$ and $Ag_1$ in environment $Env$ can be calculated using:

$$EU(Ag, Env) = \sum_{r \in \mathcal{R}(Ag, Env)} u(r) P(r | Ag, Env).$$

This is equivalent to calculating the sum of the product of each utility for a run ending in some state with the probability of performing that run; i.e.

- Utility of $Ag_0 = (0.4 \times 8) + (0.6 \times 11) = 9.8$
- Utility of $Ag_1 = (0.1 \times 70) + (0.2 \times 9) + (0.7 \times 10) = 15.8$

Therefore agent $Ag_1$ is optimal.





# Example 2

Consider the environment $Env_1 = \langle E, e_0, \tau \rangle$ defined as follows:

$$E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$$

$$\tau(e_0 \xrightarrow{\alpha_0}) = \{e_1, e_2\}$$

$$\tau(e_1 \xrightarrow{\alpha_1}) = \{e_3\}$$

$$\tau(e_2 \xrightarrow{\alpha_2}) = \{e_4, e_5\}$$

There are two agents, $Ag_1$ and $Ag_2$, with respect to this environment:

| | |
|---|---|
| $Ag_1(e_0) = \alpha_0$ | $Ag_2(e_0) = \alpha_0$ |
| $Ag_1(e_1) = \alpha_1$ | $Ag_2(e_2) = \alpha_2$ |

The probabilities of the various runs are as follows:

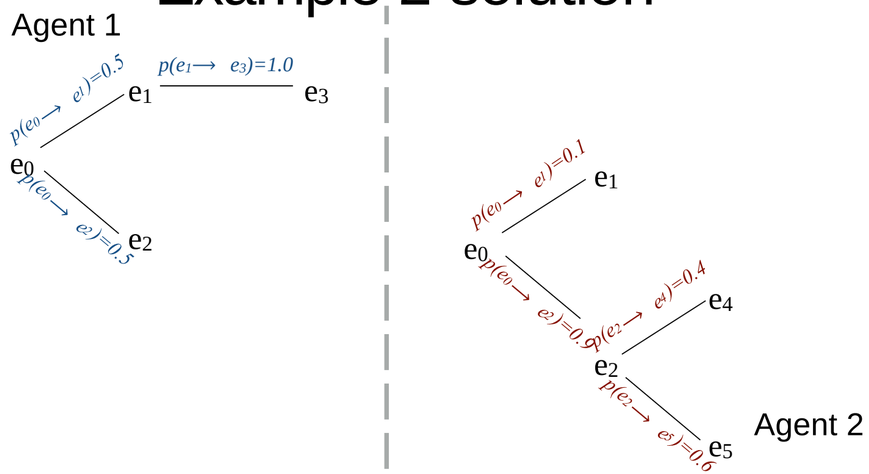
$$P(e_0 \xrightarrow{\alpha_0} e_1 \mid Ag_1, Env_1) = 0.5$$
$$P(e_0 \xrightarrow{\alpha_0} e_2 \mid Ag_1, Env_1) = 0.5$$
$$P(e_1 \xrightarrow{\alpha_1} e_3 \mid Ag_1, Env_1) = 1.0$$
$$P(e_0 \xrightarrow{\alpha_0} e_1 \mid Ag_2, Env_1) = 0.1$$
$$P(e_0 \xrightarrow{\alpha_0} e_2 \mid Ag_2, Env_1) = 0.9$$
$$P(e_2 \xrightarrow{\alpha_2} e_4 \mid Ag_2, Env_1) = 0.4$$
$$P(e_2 \xrightarrow{\alpha_2} e_5 \mid Ag_2, Env_1) = 0.6$$

Assume the utility function $u_1$ is defined as follows:

$$u_1(e_0 \xrightarrow{\alpha_0} e_1) = 4$$
$$u_1(e_0 \xrightarrow{\alpha_0} e_2) = 3$$
$$u_1(e_1 \xrightarrow{\alpha_1} e_3) = 7$$
$$u_1(e_2 \xrightarrow{\alpha_2} e_4) = 3$$
$$u_1(e_2 \xrightarrow{\alpha_2} e_5) = 2$$

What are the expected utilities of the agents for this utility function?

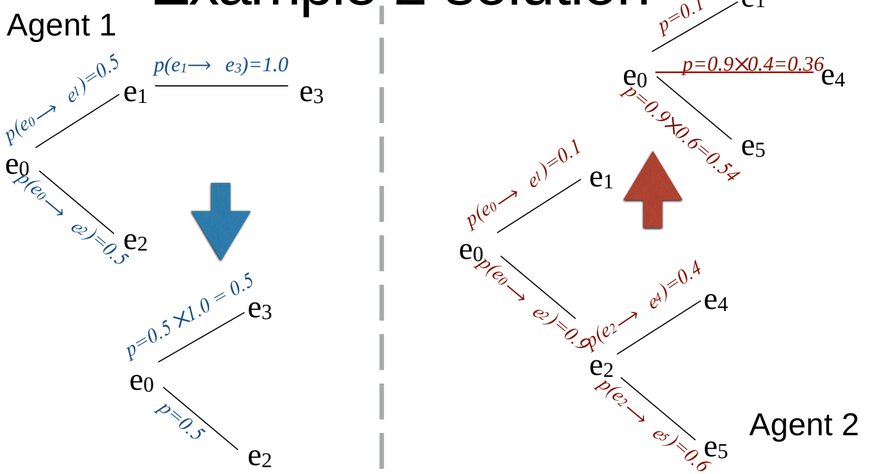
# Example 2 solution






72

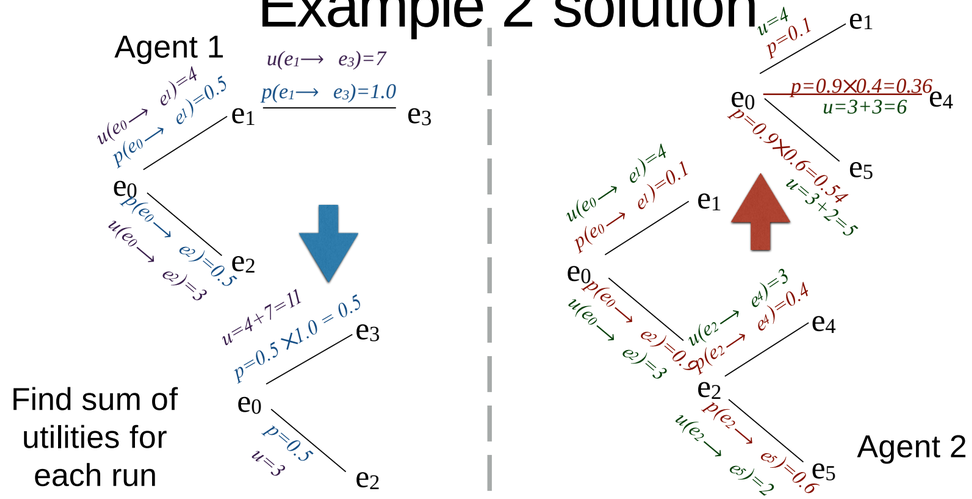
# Example 2 solution






73

# Example 2 solution









74

# Example 2 solution

| | Run | Utility | Probability |
|---|---|---|---|
| Agent 1 | $e_0 \longrightarrow e_3$ | $u=11$ | $p=0.5$ |
| | $e_0 \longrightarrow e_2$ | $u=3$ | $p=0.5$ |
| Agent 2 | $e_0 \longrightarrow e_1$ | $u=4$ | $p=0.1$ |
| | $e_0 \longrightarrow e_4$ | $u=6$ | $p=0.36$ |
| | $e_0 \longrightarrow e_5$ | $u=5$ | $p=0.54$ |

$Ag_1 = (11 \times 0.5) + (3 \times 0.5) = 5.5 + 1.5 = 7$

$Ag_2 = (4 \times 0.1) + (6 \times 0.36) + (5 \times 0.54)$
$= 0.4 + 2.16 + 2.7 = 5.26$



75

# Bounded Optimal Agents

- Some agents cannot be implemented on some computers
  - The number of actions possible on an environment (and consequently the number of states) may be so big that it may need more than available memory to implement.
- We can therefore constrain our agent set to include only those agents that can be implemented on machine $m$:

$$\mathcal{AG}_m = \{Ag \mid Ag \in \mathcal{AG} \text{ and } Ag \text{ can be implemented on } m\}.$$

- The bounded optimal agent, $Ag_{bopt}$, with respect to $m$ is then. . .

$$Ag_{bopt} = \arg \max_{Ag \in \mathcal{AG}_m} EU(Ag, Env)$$

76



# Predicate Task Specifications

- A special case of assigning utilities to histories is to assign 0 (false) or 1 (true) to a run.
  - If a run is assigned 1, then the agent *succeeds* on that run, otherwise it *fails*.
- Call these predicate task specifications.
  - Denote predicate task specification by $\Psi$:

$$\Psi : \mathcal{R} \rightarrow \{0, 1\}$$

77

# Task Environments

- A task environment is a pair $<Env, \Psi>$, where $Env$ is an environment, and the task specification $\Psi$ is defined by:

$$\Psi : \mathcal{R} \rightarrow \{0, 1\}$$

- Let the set of all task environments be defined by:

$$\mathcal{TE}$$

- A task environment specifies:
  - the properties of the system the agent will inhabit;
  - the criteria by which an agent will be judged to have either failed or succeeded.





# Task Environments

- To denote set of all runs of the agent $Ag$ in environment $Env$ that satisfy $\Psi$, we write:

$$\mathcal{R}_{\Psi}(Ag, Env) = \{r \mid r \in \mathcal{R}(Ag, Env) \text{ and } \Psi(r) = 1\}.$$

- We then say that an agent $Ag$ succeeds in task environment $<Env, \Psi>$ if

$$\mathcal{R}_{\Psi}(Ag, Env) = \mathcal{R}(Ag, Env)$$

- In other words, an agent succeeds if every run satisfies the specification of the agent.

We could also write this as:

$$\forall r \in \mathcal{R}(Ag, Env), \text{we have} \Psi(r) = 1$$

***However***, this is a bit ***pessimistic***: if the agent fails on a single run, we say it has failed overall.

A more ***optimistic*** idea of success is:

$$\exists r \in \mathcal{R}(Ag, Env), \text{we have} \Psi(r) = 1$$

which counts an agent as successful as soon as it completes a single successful run.

# The Probability of Success

- If the environment is non-deterministic, the $\tau$ returns a *set* of possible states.
  - We can define a probability distribution across the set of states.
  - Let $P(r \mid Ag, Env)$ denote probability that run $r$ occurs if agent $Ag$ is placed in environment $Env$.
  - Then the probability $P(\Psi \mid Ag, Env)$ that $\Psi$ is satisfied by $Ag$ in $Env$ would then simply be:

$$P(\Psi \mid Ag, Env) = \sum_{r \in \mathcal{R}_\Psi(Ag, Env)} P(r \mid Ag, Env)$$





# Achievement and Maintenance Tasks

- The idea of a predicate task specification is admittedly abstract.

- It generalises two common types of tasks, *achievement tasks* and *maintenance tasks*:
  1. Achievement tasks: Are those of the form “achieve state of affairs $\varphi$”.
  2. Maintenance tasks: Are those of the form “maintain state of affairs $\psi$”.

# Achievement and Maintenance Tasks

- An *achievement task* is specified by a set $G$ of "good" or "goal" states: $G \subseteq E$.
  - The agent succeeds if it is guaranteed to bring about at least one of these states (we don't care which, as all are considered good).
  - The agent *succeeds* if in an achievement task if it can *force the environment* into one of the goal states $g \in G$.
- A *maintenance goal* is specified by a set $B$ of "bad" states: $B \subseteq E$.
  - The agent succeeds in a particular environment if it manages to avoid all states in $B$ — if it never performs actions which result in any state in $B$ occurring.
  - In terms of games, the agent *succeeds* in a maintenance task if it ensures that it is *never forced into* one of the fail states $b \in B$.



82

# Summary

- This chapter has looked in detail at what constitutes an intelligent agent.
  - We looked at the properties of an intelligent agent and the properties of the environments in which it may operate.
  - We introduced the intentional stance and discussed its use.
  - We looked at abstract architectures for agents of different kinds; and
  - Finally we discussed what kinds of task an agent might need to carry out.
- In the next chapter, we will start to look at how one might program an agent using deductive reasoning.

*Class Reading (Chapter 2):*

*"Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents", Stan Franklin and Art Graesser. ECAI '96 Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages. pp 21-35*

This paper informally discusses various different notions of agency. The focus of the discussion might be on a comparison with the discussions in this chapter

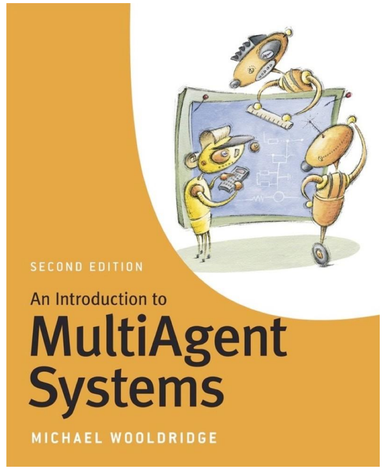


83

# IT4899
# Multi-Agent Systems
Chapter 3 - Deductive Reasoning Agents

Dr. Nguyen Binh Minh
Department of Information Systems








# 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 ..."

- Leslie Kaebling (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 ..."

# Classes of Architecture

- 1956–present: Symbolic Reasoning Agents
  - Agents make decisions about what to do via *symbol manipulation*.
  - 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 reasoning and reactive architectures.



3

# 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.

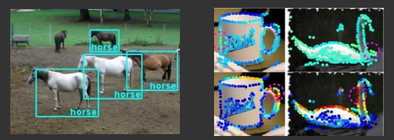


4

# Two issues

**The Transduction Problem**

Identifying objects is hard!!!



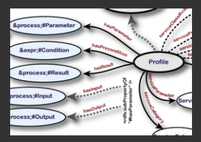



The transduction problem is that of translating the real world into an accurate, adequate symbolic description, in time for that description to be useful.

This has led onto research into vision, speech understanding, learning…

**The Representation/Reasoning Problem**

Representing objects is harder!




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.

This has led onto research into knowledge representation, automated reasoning, planning…

Most researchers accept that neither problem is anywhere near solved.



5

# The representation / reasoning problem

- The underlying problem with knowledge representation/ reasoning lies with the complexity of symbol manipulation algorithms.
  - In general many (most) search-based symbol manipulation algorithms of interest are *highly intractable*.
  - Hard to find *compact representations*.
- Because of these problems, some researchers have looked to alternative techniques for building agents; we look at these later.



6

# 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:
  - $\rho$ be this theory (typically a set of rules);
  - $\Delta$ be a logical database that describes the current state of the world;
  - $Ac$ be the set of actions the agent can perform;
  - $\Delta \vdash_\rho \varphi$ means that $\varphi$ can be proved from $\Delta$ using $\rho$.



# Deductive Reasoning Agents

- How does this fit into the abstract description we talked about last time?
  - The perception function is as before:

$$see : E \rightarrow Per$$

  - of course, this is (much) easier said than done.
- The next state function revises the database $\Delta$ :

$$next : \Delta \times Per \rightarrow \Delta$$

- And the action function?
  - Well a possible action function is on the next slide.

# Action Function

```
for each α ∈ Ac do          /* try to find an action explicitly prescribed */
        if Δ ⊢ρ Do(α) then
                return α
        end-if
end-for

for each α ∈ Ac do          /* try to find an action not excluded */
        if Δ ⊬ρ ¬Do(α) then
                return α
        end-if
end-for

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



9

# An example: The Vacuum World

**The Vacuum World**

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

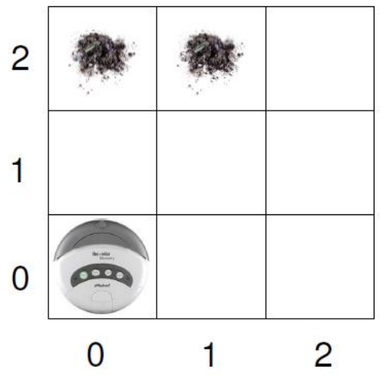
***Uses 3 domain predicates in this exercise:***

| | |
|---|---|
| *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}*

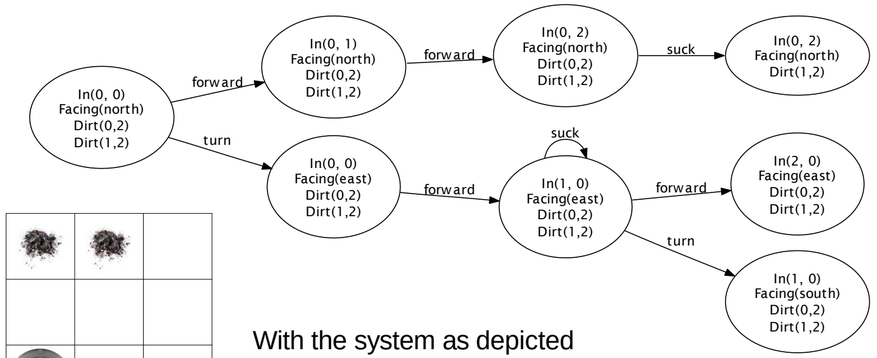
*Note: turn means "turn right"*

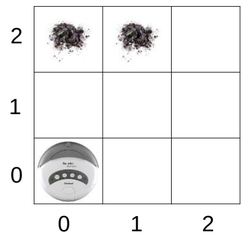





10

# The Vacuum World




With the system as depicted above, here are some possible ways that the system might run.





# The Vacuum World

- Rules $\rho$ for determining what to do:

$In(0, 0) \wedge Facing(north) \wedge \neg Dirt(0, 0) \rightarrow Do(forward)$
$In(0, 1) \wedge Facing(north) \wedge \neg Dirt(0, 1) \rightarrow Do(forward)$
$In(0, 2) \wedge Facing(north) \wedge \neg Dirt(0, 2) \rightarrow Do(turn)$
$In(0, 2) \wedge Facing(east) \rightarrow Do(forward)$

  - ... and so on!

- Using these rules (+ other obvious ones), starting at *(0, 0)* the robot will clear up dirt.

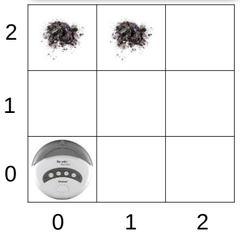
***Uses 3 domain predicates in this exercise:***

| | |
|---|---|
| *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}*

*Note:turn means "turn right"*



# The Vacuum World

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



13

# Agent-oriented programming

- 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, desire, and intention
    - Adopts the same abstraction as humans
  - Resulted in the Agent0 programming language



14

# 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*.





# 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 externally executed computation
- Communicative
  - Sending messages

Messages are constrained to be one of three types . . .

- requests
  - To commit to action
- unrequests
  - To refrain from action
- Informs
  - Which pass on information

# Commitment Rules

- 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*.

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)
)
```



17

# PLACA

- A more refined implementation was developed by Becky 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.



19

# PLACA: Mental Change Rule

- If:
  - someone *asks* you to xerox something *x* at time *t* 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.

*A PLACA 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))))))))
```



20

# Concurrent MetateM

- Concurrent METATEM is a multi-agent language, developed by Michael Fisher
  - 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.
  - Think of the world as being a number of discrete states.
  - There is a single past history, but a number of possible futures
    - all the possible ways that the world might develop.



21

# MetateM Agents

- 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*



22

# MetateM

- The root of the MetateM concept is Gabbay's separation theorem:
  - Any arbitrary temporal logic formula can be rewritten in a logically equivalent past ⇒ future form.
- 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.



23

# Examples

| | |
|---|---|
| $\Box$important(agents) | *means "it is now, and will always be true that agents are important"* |
| $\Diamond$important(ConcurrentMetateM) | *means "sometime in the future, ConcurrentMetateM will be important"* |
| $\blacklozenge$ important(Prolog) | *means "sometime in the past it was true that Prolog was important"* |
| $(\neg\text{friends(us)})\,\mathcal{U}$ apologise(you) | *means "we are not friends until you apologise"* |
| $\bigcirc$apologise(you) | *means "tomorrow (in the next state), you apologise"* |
| $\bullet$ apologise(you) $\Rightarrow \bigcirc$friends(us) | *means "if you apologised yesterday, then tomorrow we will be friends"* |
| friends(us) $\mathcal{S}$ apologise(you) | *means "we have been friends since you apologised"* |



24

# Summary

- This chapter has focussed on Agent Architectures and general approaches to programming an agent.
  - We defined the notion of symbolic reasoning agents, and discussed...
    - ...how can deductive reasoning be achieved through the use of logic; and
    - ...the Transduction and Representation Problems
  - We introduced the concept of Agent Oriented Programming, and looked at examples of AOP languages, including:
    - Agent0 and PLACA
    - Concurrent MetateM and temporal logic
- In the next chapter, we will consider the merits of practical reasoning agents.

*Class Reading (Chapter 3):*

*"Agent Oriented Programming", Yoav Shoham. Artificial Intelligence Journal 60(1), March 1993. pp51-92.*

This paper introduced agent-oriented programming and throughout the late 90ies was one of the most cited articles in the agent community. One of the main points was the notion of using mental states, and introduced the programming language Agent0.

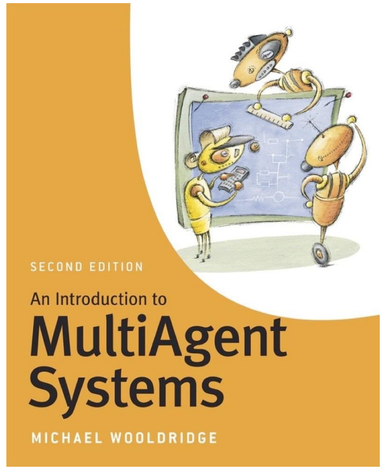


25

# IT4899
# Multi-Agent Systems
Chapter 4 - Practical Reasoning Agents

Dr. Nguyen Binh Minh
Department of Information Systems








# Pro-Active Behaviour

- Previously we looked at:
  - Characteristics of an Agent and its Environment
  - The Intentional Stance
  - Translating the Formal Agent model into a Deductive Logic framework
- We said:
  - *An intelligent agent is a computer system capable of flexible autonomous action in some environment*.
  - Where by flexible, we mean:
    - reactive;
    - pro-active;
    - social.
- This is where we deal with the "*proactive*" bit, showing how we can program agents to have goal-directed behaviour.

# What is Practical Reasoning?

- Practical reasoning is reasoning directed towards actions — the process of figuring out what to do:

> "… Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes..." (Bratman)

- Distinguish practical reasoning from *theoretical reasoning*.
- Theoretical reasoning is directed towards beliefs.





# The components of Practical Reasoning

- Human practical reasoning consists of two activities:
  - *deliberation*:
    - deciding ***what*** state of affairs we want to achieve
    - the outputs of deliberation are *intentions*;
  - *means-ends reasoning*:
    - deciding ***how*** to achieve these states of affairs
    - the outputs of means-ends reasoning are ***plans***.
- Intentions are a key part of this.
  - The interplay between *beliefs*, *desires* and *intentions* defines how the model works

# Intentions in Practical Reasoning

1. Intentions pose problems for agents, who need to determine ways of achieving them.

   *If I have an intention to φ, you would expect me to devote resources to deciding how to bring about φ.*

2. Intentions provide a "filter" for adopting other intentions, which must not conflict.

   *If I have an intention to φ, you would not expect me to adopt an intention ψ that was incompatible with φ.*

3. Agents track the success of their intentions, and are inclined to try again if their attempts fail.

   *If an agent's first attempt to achieve φ fails, then all other things being equal, it will try an alternative plan to achieve φ.*



5

# Intentions in Practical Reasoning

4. Agents believe their intentions are possible.

   *That is, they believe there is at least some way that the intentions could be brought about.*

5. Agents do not believe they will not bring about their intentions.

   *It would not be rational of me to adopt an intention to φ if I believed I would fail with φ.*

6. Under certain circumstances, agents believe they will bring about their intentions.

   *If I intend φ, then I believe that under "normal circumstances" I will succeed with φ.*



6

# Intentions in Practical Reasoning

7. Agents need not intend all the expected side effects of their intentions.

*If I believe φ ⇒ ψ and I intend that φ, I do not necessarily intend ψ also.*

- Intentions are not closed under implication.
  - This last problem is known as the side effect or package deal problem.


*I may believe that going to the dentist involves pain, and I may also intend to go to the dentist — but this does not imply that I intend to suffer pain!*





# Intentions are Stronger than Desire

"… My desire to play basketball this afternoon is merely a potential influencer of my conduct this afternoon. It must vie with my other relevant desires [. . . ] before it is settled what I will do.

In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons.

When the afternoon arrives, I will normally just proceed to execute my intentions..."

Michael E. Bratman (1990)

# Means-ends Reasoning/Planning

- Planning is the design of a course of action that will achieve some desired goal.
  - Basic idea is to give a planning system:
    - (representation of) goal/intention to achieve;
    - (representation of) actions it can perform;
    - (representation of) the environment;
  - and have it generate a *plan* to achieve the goal.
- This is *automatic programming*.



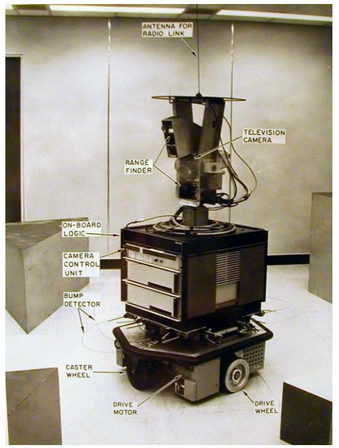



# Means-ends Reasoning/Planning

- Don't have to directly tell the system what to do!
  - Let it *figure out* how to achieve the goal on its own!

# STRIPS Planner

- STRIPS
  - The **St**anford **R**esearch **I**nstitute **P**roblem **S**olver
  - Used by Shakey, the robot
    - Developed by Richard Fikes and Nils Nilsson in 1971 at SRI International







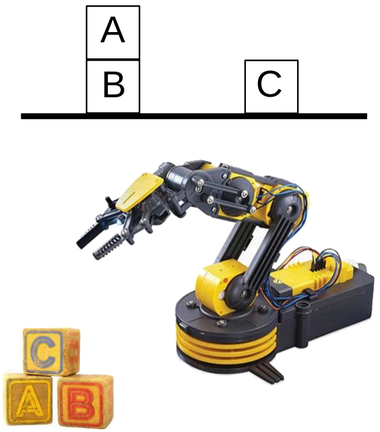

# Representations

- ***Question***: How do we represent. . .
  - goal to be achieved;
  - state of environment;
  - actions available to agent;
  - plan itself.

- ***Answer***: We use *logic*, or something that looks a lot like logic.

# Blocksworld

- We'll illustrate the techniques with reference to the blocks world.
  - A simple (toy) world, in this case one where we consider toys
- The blocks world contains a robot arm, 3 blocks (A, B and C) of equal size, and a table-top.
- The aim is to generate a plan for the robot arm to build towers out of blocks.






13

# Blocksworld

- The environment is represented by an *ontology*.
- The closed world assumption is used
  - Anything not stated is assumed to be false.
- A *goal* is represented as a set of formulae.

**Blocksworld Ontology**

| | |
|---|---|
| *On(x,y)* | *object x on top of object y* |
| *OnTable(x)* | *object x is on the table* |
| *Clear(x)* | *nothing is on top of object x* |
| *Holding(x)* | *arm is holding x* |

Representation of the following blocks

*Clear(A)*
*On(A, B)*
*OnTable(B)*
*Clear(C)*
*OnTable(C)*
*ArmEmpty*

The goal:

*{OnTable(A), OnTable(B), OnTable(C), ArmEmpty}*



14

# Blocksworld Actions

- Each action has:
  - a *name*: which may have arguments;
  - a *pre-condition list*: list of facts which must be true for action to be executed;
  - a *delete list*: list of facts that are no longer true after action is performed;
  - an *add list*: list of facts made true by executing the action.
- Each of these may contain variables.
- What is a plan?
  - A sequence (list) of actions, with variables replaced by constants.





# Blocksworld Actions

*Stack*(*x*, *y*)
pre *Clear*(*y*) ^ *Holding*(*x*)
del *Clear*(*y*) ^ *Holding*(*x*)
add *ArmEmpty* ^ *On*(*x*, *y*)

*The **stack** action occurs when the robot arm places the object x it is holding is placed on top of object y.*

*Pickup*(*x*)
pre *Clear*(*x*) ^ *OnTable*(*x*) ^ *ArmEmpty*
del *OnTable*(*x*) ^ *ArmEmpty*
add *Holding*(*x*)

*The **pickup** action occurs when the arm picks up an object x from the table.*

*UnStack*(*x*, *y*)
pre *On*(*x*, *y*) ^ *Clear*(*x*) ^ *ArmEmpty*
del *On*(*x*, *y*) ^ *ArmEmpty*
add *Holding*(*x*) ^ *Clear*(*y*)

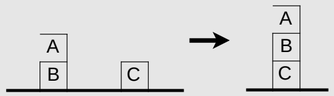
*The **unstack** action occurs when the robot arm picks an object x up from on top of another object y.*

*PutDown*(*x*)
pre *Holding*(*x*)
del *Holding*(*x*)
add *OnTable*(*x*) ^ *ArmEmpty* ^ *Clear*(*x*)

*The **putdown** action occurs when the arm places the object x onto the table.*

# Using Plans

*To get from here (left) to here (right)...*




*...we need this set of actions:*

*UnStack(A,B)*
*Putdown(A)*
*Pickup(B)*
*Stack(B, C)*
*Pickup(A)*
*Stack(A, B)*

*Stack*(*x*, *y*)
pre *Clear*(*y*) ^ *Holding*(*x*)
del *Clear*(*y*) ^ *Holding*(*x*)
add *ArmEmpty* ^ *On*(*x*, *y*)

*UnStack*(*x*, *y*)
pre *On*(*x*, *y*) ^ *Clear*(*x*) ^ *ArmEmpty*
del *On*(*x*, *y*) ^ *ArmEmpty*
add *Holding*(*x*) ^ *Clear*(*y*)

*Pickup*(*x*)
pre *Clear*(*x*) ^ *OnTable*(*x*) ^ *ArmEmpty*
del *OnTable*(*x*) ^ *ArmEmpty*
add *Holding*(*x*)

*PutDown*(*x*)
pre *Holding*(*x*)
del *Holding*(*x*)
add *OnTable*(*x*) ^ *ArmEmpty* ^ *Clear*(*x*)



17

# Plan Validity

- Thus, a plan is simply a *sequence of steps*
- However, how can we:
  - Generate the plan?
  - Ensure that it is correct?



18

# Formal Representation

- Let's relate the STRIPS model back to the formal description of an agent we talked about before.
  - This will help us to see how it fits into the overall picture.
- As before we assume that the agent has a set of actions $Ac$, and we will write individual actions as $\alpha_1$, $\alpha_2$ and so on.
- Now the actions have some structure, each one has preconditions $P_{\alpha i}$, add list $A_{\alpha i}$, and delete list $D_{\alpha i}$, for each $\alpha_i \in Ac$:

$$\alpha_i = \langle P_{\alpha_i}, D_{\alpha_i}, A_{\alpha_i} \rangle$$

- A plan is just a sequence of actions, where each action is one of the actions from $Ac$:

$$\pi = (\alpha_1, \ldots, \alpha_n)$$





# Formal Representation

- A *planning problem* is therefore: $< B_0, Ac, I >$
  - $B_0$ is the set of beliefs the agent has about the world.
  - $Ac$ is the set of actions, and
  - $I$ is a goal (or intention)
- Since actions change the world, any rational agent will change its beliefs about the world as a result of carrying out actions.
  - Thus, a plan $\pi$ for a given planning problem will be associated with a sequence of sets of beliefs:

$$B_0 \xrightarrow{\alpha_1} B_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} B_n$$

  - In other words at each step of the plan the beliefs are updated by removing the items in the delete list of the relevant action and adding the items in the add list.

# Formal Representation

- A plan $\pi$ is said to be acceptable with respect to the problem $< B_0, Ac, I >$ if and only if, for all $1 \leq j \leq n$, $B_{j-1} \models P_{\alpha j}$
  - In other words, the pre-requisites for each action have to be true right before the action is carried out.
    - We say this because the pre-conditions don't have to be in $B_{j-1}$, we just have to be able to prove the pre-conditions from $B_{j-1}$.
- A plan $\pi$ is correct if it is acceptable, and: $B_n \models i$
  - In other words, it is correct if it is acceptable and the final state makes the goal true.

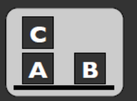


21

A plan $\pi$ is a sequence of actions, where each action results in changing the set of beliefs the agent has, until the final set of beliefs matches that of the intentions, such that $B_0 \overset{\alpha_1}{\rightarrow} B_1 \overset{\alpha_2}{\rightarrow} \cdots \overset{\alpha_n}{\rightarrow} B_n$. Therefore, a planner will explore all the different possible sequences of actions to determine which one will result in the final set of intentions.

In this solution, only those actions that result in the final solution are given, with the set of beliefs that result in each step presented. The aim is to start with an initial set of beliefs, $B_0$, and arrive at a final set of beliefs, $B_n$ which corresponds to the intentions given in the question - i.e.

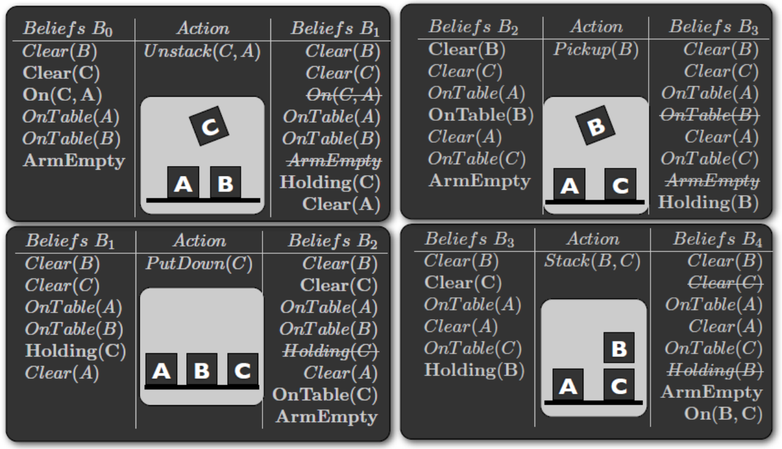
| $Beliefs\ B_0$ | $Intention\ i$ |
|---|---|
| $Clear(B)$ | $Clear(A)$ |
| $Clear(C)$ | $Clear(B)$ |
| $On(C, A)$ | $On(B, C)$ |
| $OnTable(A)$ | $OnTable(A)$ |
| $OnTable(B)$ | $OnTable(C)$ |
| $ArmEmpty$ | $ArmEmpty$ |




The solution is given on the next slide. In each case, the beliefs that hold prior to the action are given in bold, and the beliefs that are new after the action are also presented in bold.



23




The beliefs B4, once rearranged, are now equivalent to the intentions.





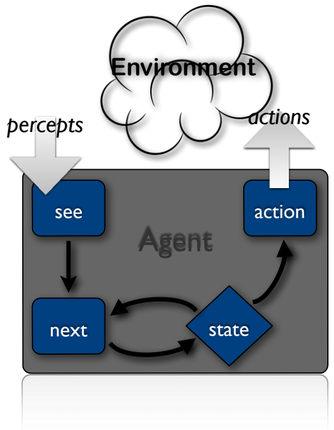
# Implementing Practical Reasoning Agents

- A first pass at an implementation of a practical reasoning agent:
- For now we will not be concerned with stages 2 or 3.
  - These are related to the functions *see* and *next* from the earlier lecture notes.

```
Agent Control Loop Version 1
1.   while true
2.        observe the world;
3.        update internal world  model;
4.        deliberate about what intention
            to achieve next;
5.        use means-ends reasoning to  get
            a plan for the intention;
6.        execute the plan
7.   end while
```

# Implementing Practical Reasoning Agents

- see is as before:
  - see: E → Percept
- Instead of the function next…
  - which took a percept and used it to update the internal state of an agent
- …we have a belief revision function:
  - brf: $\mathcal{P}$ {Bel} x Percept → $\mathcal{P}$ {Bel}
    - $\mathcal{P}$ {Bel} is the *power set of beliefs*
    - Bel is the set of all possible beliefs that an agent might have.








# Implementing Practical Reasoning Agents

- Problem:
  - deliberation and means-ends reasoning processes are not instantaneous.
  - They have a *time cost*.
- Suppose that deliberation is *optimal*
  - The agent selects the optimal intention to achieve, then this is the best thing for the agent.
  - i.e. it maximises expected utility.
- So the agent selects an intention to achieve that would have been optimal *at the time it observed the world*.
  - This is calculative rationality.
- The *world may change* in the meantime.
  - Even if the agent can compute the right thing to do, it may not do the right thing.
  - Optimality is hard.

# Implementing Practical Reasoning Agents

- Let's make the algorithm more formal with the algorithm opposite
  - where $I \subseteq Int$, i.e the set of intentions,
  - `plan()` is exactly what we discussed above,
  - `brf()` is the belief revision function,
  - and `execute()` is a function that executes each action in a plan.
- How might we implement these functions?

```
Agent Control Loop Version 2
1.  B := B_0; /* initial beliefs */
2.  while true do
3.      get next percept ρ;
4.      B := brf(B, ρ);
5.      I := deliberate(B);
6.      π := plan(B, I);
7.      execute(π)
8.  end while
```





# Deliberation

- How does an agent *deliberate*?
  - begin by trying to understand what the *options* available to you are;
  - *choose* between them, and *commit* to some.
- Chosen options are then *intentions*.
- The *deliberate* function can be decomposed into two distinct functional components:
  - ***option generation***; and
  - ***filtering***.

# Option Generation and Filtering

- Option Generation
  - In which the agent generates a set of possible alternatives
  - Represent option generation via a function, `options()`, which takes the agent's current beliefs and current intentions, and from them determines a set of options
    - ***desires***

$$options : \mathcal{P}(Bel) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Des)$$

- Filtering
  - In which the agent chooses between competing alternatives, and commits to achieving them.
  - In order to select between competing options, an agent uses a `filter()` function.
    - ***intentions***

$$filter : \mathcal{P}(Bel) \times \mathcal{P}(Des) \times \mathcal{P}(Int) \rightarrow \mathcal{P}(Int)$$





# Implementing Practical Reasoning Agents

**Agent Control Loop Version 3**

1. $B := B_0;$
2. $I := I_0;$
3. **while true do**
4. **get next percept** $\rho;$
5. $B := brf(B, \rho);$
6. $D := options(B, I);$
7. $I := filter(B, D, I);$
8. $\pi := plan(B, I);$
9. $execute(\pi)$
10. **end while**

# Under Commitment

"... Some time in the not-so-distant future, you are having trouble with your new household robot. You say "Willie, bring me a beer." The robot replies "OK boss." Twenty minutes later, you screech "Willie, why didn't you bring me that beer?" It answers "Well, I intended to get you the beer, but I decided to do something else." Miffed, you send the wise guy back to the manufacturer, complaining about a lack of commitment..."

P. R. Cohen and H. J. Levesque (1990). Intention is choice with commitment. Artificial intelligence, 42(2), 213-261.





# Over Commitment

"... After retrofitting, Willie is returned, marked "Model C: The Committed Assistant." Again, you ask Willie to bring you a beer. Again, it accedes, replying "Sure thing." Then you ask: "What kind of beer did you buy?"It answers: "Genessee." You say "Never mind." One minute later, Willie trundles over with a Genessee in its gripper..."


P. R. Cohen and H. J. Levesque (1990). Intention is choice with commitment. Artificial intelligence, 42(2), 213-261.

# Wise Guy ???

"... After still more tinkering, the manufacturer sends Willie back, promising no more problems with its commitments. So, being a somewhat trusting customer, you accept the rascal back into your household, but as a test, you ask it to bring you your last beer. [. . . ]

The robot gets the beer and starts towards you. As it approaches, it lifts its arm, wheels around, deliberately smashes the bottle, and trundles off. Back at the plant, when interrogated by customer service as to why it had abandoned its commitments, the robot replies that according to its specifications, it kept its commitments as long as required — commitments must be dropped when fulfilled or impossible to achieve. By smashing the bottle, the commitment became unachievable..."



P. R. Cohen and H. J. Levesque (1990). Intention is choice with commitment. Artificial intelligence, 42(2), 213-261.



# Degrees of Commitment

- Blind commitment
  - A blindly committed agent will continue to maintain an intention until it believes the intention has actually been achieved. Blind commitment is also sometimes referred to as *fanatical* commitment.
- Single-minded commitment
  - A single-minded agent will continue to maintain an intention *until* it believes that *either* the intention has been achieved, *or else that it is no longer possible* to achieve the intention.
- Open-minded commitment
  - An open-minded agent will maintain an intention *as long as it is still believed* possible.

# Degrees of Commitment

- An agent has commitment both to:
  - *ends* (i.e., the state of affairs it wishes to bring about), and
  - *means* (i.e., the mechanism via which the agent wishes to achieve the state of affairs).
- Currently, our agent control loop is overcommitted, both to means and ends.
  - Modification: *replan* if ever a plan goes wrong.
  - However, to write the algorithm down we *need to refine* our notion of plan execution.



37

# Degrees of Commitment

- The previous version was *blindly committed to its means and its ends*
- If $\pi$ is a plan, then:
  - *empty($\pi$)* is true if there are no more actions in the plan.
  - *hd($\pi$)* returns the first action in the plan.
  - *tail($\pi$)* returns the plan minus the head of the plan.
  - *sound($\pi$, $I$, $B$)* means that $\pi$ is a correct plan for $I$ given $B$.

```
Agent Control Loop Version 4

1.  B := B_0;
2.  I := I_0;
3.  while true do
4.      get next percept ρ;
5.      B := brf(B, ρ);
6.      D := options(B, I);
7.      I := filter(B, D, I);
8.      π := plan(B, I);
9.      while not empty(π) do
10.         α := hd(π);
11.         execute(α);
12.         π := tail(π);
13.         get next percept ρ;
14.         B := brf(B, ρ);
15.         if not sound(π, I, B) then
16.             π := plan(B, I)
17.         end-if
18.     end-while
19. end-while
```



38

# Degrees of Commitment

- *Blind Commitment*
  - Makes the control loop more reactive, able to change intention when the world changes.
    - i.e. it is not committed to its means (line 16)
- Still overcommitted to intentions (ends).
  - Never stops to consider whether or not its intentions are appropriate.

```
Agent Control Loop Version 4

1.  B := B_0;
2.  I := I_0;
3.  while true do
4.      get next percept ρ;
5.      B := brf(B, ρ);
6.      D := options(B, I);
7.      I := filter(B, D, I);
8.      π := plan(B, I);
9.      while not empty(π) do
10.         α := hd(π);
11.         execute(α);
12.         π := tail(π);
13.         get next percept ρ;
14.         B := brf(B, ρ);
15.         if not sound(π, I, B) then
16.             π := plan(B, I)
17.         end-if
18.     end-while
19. end-while
```



39

# Single Minded Commitment

- Modification:
  - stop to determine whether intentions have succeeded or whether they are impossible
- Our agent now gets to reconsider its intentions once every time around the outer control loop (line 9), i.e., after:
  - it has completely executed a plan to achieve its current intentions; or
  - it believes it has achieved its current intentions; or
  - it believes its current intentions are no longer possible.

```
Agent Control Loop Version 5

1.  B := B_0;
2.  I := I_0;
3.  while true do
4.      get next percept ρ;
5.      B := brf(B, ρ);
6.      D := options(B, I);
7.      I := filter(B, D, I);
8.      π := plan(B, I);
9.      while not(empty(π)
                or succeeded(I, B)
                or impossible(I, B)) do
10.         α := hd(π);
11.         execute(α);
12.         π := tail(π);
13.         get next percept ρ;
14.         B := brf(B, ρ);
15.         if not sound(π, I, B) then
16.             π := plan(B, I)
17.         end-if
18.     end-while
19. end-while
```



40

# Open Minded Commitment

- Open Minded Commitment
  - In the previous version, our agent reconsiders its intentions once every time around the outer control loop
  - In this new version, our agent also reconsiders its intentions after every action (lines 15 & 16)
- But this intention reconsideration is *costly*!

```
Agent Control Loop Version 6

1.  B := B_0;
2.  I := I_0;
3.  while true do
4.      get next percept ρ;
5.      B := brf(B, ρ);
6.      D := options(B, I);
7.      I := filter(B, D, I);
8.      π := plan(B, I);
9.      while not(empty(π)
                  or succeeded(I, B)
                  or impossible(I, B)) do
10.         α := hd(π);
11.         execute(α);
12.         π := tail(π);
13.         get next percept ρ;
14.         B := brf(B, ρ);
15.         D := options(B, I);
16.         I := filter(B, D, I);
17.         if not sound(π, I, B) then
18.             π := plan(B, I)
19.         end-if
20.     end-while
21. end-while
```



41

# Intention Reconsideration

- A dilemma:
  - an agent that *does not stop* to reconsider its intentions sufficiently often *will continue to attempt* to achieve its intentions even after it is clear that they cannot be achieved, or that there is no longer any reason for achieving them;
  - an agent that *constantly* reconsiders its attentions may *spend insufficient time* actually working to achieve them, and hence runs the risk of never actually achieving them.
- Solution: incorporate an explicit meta-level control component, that decides whether or not to reconsider.

```
Agent Control Loop Version 7

1.  B := B_0;
2.  I := I_0;
3.  while true do
4.      get next percept ρ;
5.      B := brf(B, ρ);
6.      D := options(B, I);
7.      I := filter(B, D, I);
8.      π := plan(B, I);
9.      while not(empty(π)
                  or succeeded(I, B)
                  or impossible(I, B)) do
10.         α := hd(π);
11.         execute(α);
12.         π := tail(π);
13.         get next percept ρ;
14.         B := brf(B, ρ);
15.         if reconsider(I, B) then
16.             D := options(B, I);
17.             I := filter(B, D, I);
18.         end-if
19.         if not sound(π, I, B) then
12.             π := plan(B, I)
21.         end-if
22.     end-while
23. end-while
```



42

# Intention Reconsideration

- The possible interactions between meta-level control and deliberation are:

| Situation number | Chose to deliberate? | Changed intentions? | Would have changed intentions? | *reconsider*(...) optimal? |
|---|---|---|---|---|
| 1 | No | — | No | Yes |
| 2 | No | — | Yes | No |
| 3 | Yes | No | — | No |
| 4 | Yes | Yes | — | Yes |

- An important assumption: cost of reconsider(. . .) is *much* less than the cost of the deliberation process itself.





# Intention Reconsideration

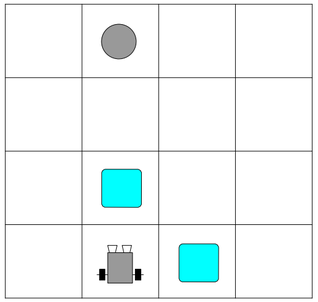
| Situation number | Chose to deliberate? | Changed intentions? | Would have changed intentions? | *reconsider*(...) optimal? |
|---|---|---|---|---|
| 1 | No | — | No | Yes |
| 2 | No | — | Yes | No |
| 3 | Yes | No | — | No |
| 4 | Yes | Yes | — | Yes |

```
if reconsider(I,B) then
    D := options(B,I);
    I := filter(B,D,I);
end-if
```

- In situation (1), the agent did not choose to deliberate, and as a consequence, did not choose to change intentions. Moreover, if it *had* chosen to deliberate, it would not have changed intentions. In this situation, the reconsider(. . .) function is behaving optimally.
- In situation (2), the agent did not choose to deliberate, but if it had done so, it *would* have changed intentions. In this situation, the reconsider(. . .) function is *not* behaving optimally.
- In situation (3), the agent chose to deliberate, but did not change intentions. In this situation, the reconsider(. . .) function is *not* behaving optimally.
- In situation (4), the agent chose to deliberate, and did change intentions. In this situation, the reconsider(. . .) function is behaving optimally.

# Optimal Intention Reconsideration

- Kinny and Georgeff's experimentally investigated effectiveness of intention reconsideration strategies.
- Two different types of reconsideration strategy were used:
  - *bold* agents: never pause to reconsider intentions, and
  - *cautious* agents: stop to reconsider after every action.
- *Dynamism* in the environment is represented by the rate of world change, $\gamma$.
  - Experiments were carried out using Tileword.



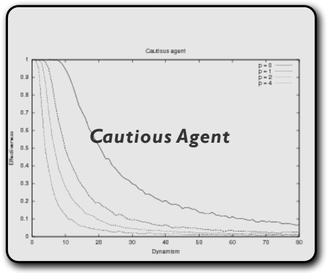




# Optimal Intention Reconsideration

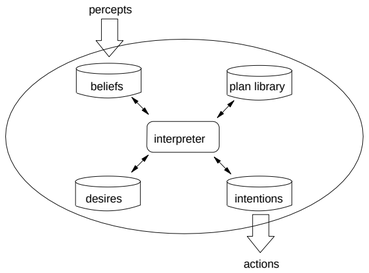
- If $\gamma$ is *low* (i.e., the environment does not change quickly), then bold agents do well compared to cautious ones.
  - This is because cautious ones waste time reconsidering their commitments while bold agents are busy working towards — and achieving — their intentions.
- If $\gamma$ is *high* (i.e., the environment changes frequently), then cautious agents can outperform bold agents.
  - This is because they are able to recognise when intentions are doomed, and also to take advantage of serendipitous situations and new opportunities when they arise.
- When planning costs are high, this advantage can be eroded.


low ⟵ $\gamma$ ⟶ high



# Implemented BDI Agents: Procedural Reasoning System

- We now make the discussion even more concrete by introducing an actual agent architecture: the Procedural Reasoning System (PRS).
  - In the PRS, each agent is equipped with a *plan library*, representing that agent's *procedural knowledge*: knowledge about the mechanisms that can be used by the agent in order to realise its intentions.
  - The options available to an agent are directly determined by the plans an agent has: an agent with no plans has no options.
- In addition, PRS agents have explicit representations of beliefs, desires, and intentions, as above.








# Example PRS (JAM) System

- The agent possesses a number of pre-compiled plans (constructed manually)
  - Each plan contains:
    - *a goal* - the postcondition of the plan
    - *a context* - the pre condition of the plan
    - *a body* - the course of action to take out
- When an agent starts, goals are pushed onto the *intention* stack.
  - This stack contains all of the goals that are pending
  - A set of *facts or beliefs* are maintained and updated as the agent achieves different goals
  - The agent then *deliberates* (i.e. selects the most appropriate goal to adopt).
    - This is achieved using meta level plans, or utilities
    - When utilities are used, the agent selects the goal with the highest value

```
GOALS:
    ACHIEVE blocks_stacked;

FACTS:
// Block1 on Block2 initially so need
//to clear Block2 before stacking.

    FACT ON "Block1" "Block2";
    FACT ON "Block2" "Table";
    FACT ON "Block3" "Table";
    FACT CLEAR "Block1";
    FACT CLEAR "Block3";
    FACT CLEAR "Table";
    FACT initialized "False";
```

# Example PRS (JAM) System

- This is the plan for the top level goal:
  - ACHIEVE blocks_stacked.
- Note that the body contains a mix of instructions and goals.
- When executing, the goals will be added to the intention stack

```
Plan: {
   NAME: "Top-level plan"
   DOCUMENTATION:
      "Establish Block1 on Block2 on Block3."
   GOAL:
      ACHIEVE blocks_stacked;
   CONTEXT:
   BODY:
      EXECUTE print "Goal: Blk1 on Blk2 on Blk3 on Table.\n";
      EXECUTE print "World Model at start is:\n";
      EXECUTE printWorldModel;

      EXECUTE print "ACHIEVEing Block3 on Table.\n";
      ACHIEVE ON "Block3" "Table";

      EXECUTE print "ACHIEVEing Block2 on Block3.\n";
      ACHIEVE ON "Block2" "Block3";

      EXECUTE print "ACHIEVEing Block1 on Block2.\n";
      ACHIEVE ON "Block1" "Block2";

      EXECUTE print "World Model at end is:\n";
      EXECUTE printWorldModel;
}
```



49

# Example PRS (JAM) System

- This plan also has a utility associated with it
  - This is used by the agent during the deliberation phase
- The plan can also determine actions to execute if it fails

```
Plan: {
   NAME: "Stack blocks that are already clear"
   GOAL:
      ACHIEVE ON $OBJ1 $OBJ2;
   CONTEXT:
   BODY:
      EXECUTE print "Making sure " $OBJ1 " is clear\n";
      ACHIEVE CLEAR $OBJ1;
      EXECUTE print "Making sure " $OBJ2 " is clear.\n";
      ACHIEVE CLEAR $OBJ2;
      EXECUTE print "Moving " $OBJ1 " on top of " $OBJ2 ".\n";
      PERFORM move $OBJ1 $OBJ2;
   UTILITY: 10;
   FAILURE:
      EXECUTE print "\n\nStack blocks failed!\n\n";
}
```



50

# Example PRS (JAM) System

- This plan includes an EFFECTS field
- This determines what the agent should do once the agent has succeeded in executing all of the BODY instructions.

```
Plan: {
   NAME: "Clear a block"
   GOAL:
      ACHIEVE CLEAR $OBJ;
   CONTEXT:
      FACT ON $OBJ2 $OBJ;
   BODY:
      EXECUTE print "Clear " $OBJ2 " from on top of " $OBJ "\n";
      EXECUTE print "Move " $OBJ2 " to table.\n";
      ACHIEVE ON $OBJ2 "Table";
   EFFECTS:
      EXECUTE print "Clear: Retract ON " $OBJ2 " " $OBJ "\n";
      RETRACT ON $OBJ1 $OBJ;
   FAILURE:
      EXECUTE print "\n\nClearing block " $OBJ " failed!\n\n";
}
```

51



# Example PRS (JAM) System

- I'll leave it as an exercise to work out what plans will be executed
  - If you have a solution and want me to check, let me know.
- The Java version of JAM and further details/documentation are available from Marcus Huber's website:
  - http://www.marcush.net/IRS/irs_downloads.html

```
Plan: {
   NAME: "Move a block onto another object"
   GOAL:
      PERFORM move $OBJ1 $OBJ2;
   CONTEXT:
      FACT CLEAR $OBJ1;
      FACT CLEAR $OBJ2;
   BODY:
      EXECUTE print "Performing low-level move action"
      EXECUTE print " of " $OBJ1 " to " $OBJ2 ".\n";
   EFFECTS:
      WHEN : TEST (!= $OBJ2 "Table") {
         EXECUTE print "-Retract CLEAR " $OBJ2 "\n";
         RETRACT CLEAR $OBJ2;
      };
      FACT ON $OBJ1 $OBJ3;
      EXECUTE print "-move: Retract ON " $OBJ1 " " $OBJ3 "\n";
      RETRACT ON $OBJ1 $OBJ3;
      EXECUTE print "-move: Assert CLEAR " $OBJ3 "\n";
      ASSERT CLEAR $OBJ3;
      EXECUTE print "-move: Assert ON " $OBJ1 " " $OBJ2 "\n\n";
      ASSERT ON $OBJ1 $OBJ2;
   FAILURE:
      EXECUTE print "\n\nMove failed!\n\n";
}
```

52

# Summary

- This lecture has covered a lot of ground on practical reasoning.
  - We started by discussing what practical reasoning was, and how it relates to intentions.
  - We then looked at planning (how an agent achieves its desires) and how deliberation and means-ends reasoning fit into the basic agent control loop.
  - We then refined the agent control loop, considering commitment and intention reconsideration.
  - Finally, we looked at an implemented system (the textbook discusses a couple of others).
- Next Lecture - we start looking at the AgentSpeak and the Jason framework to exploit BDI

*Class Reading (Chapter 4):*

*"Plans and resource-bounded practical reasoning", Michael E. Bratman, David J. Israel, Martha E. Pollack. Computational Intelligence 4: 1988. pp349-355.*

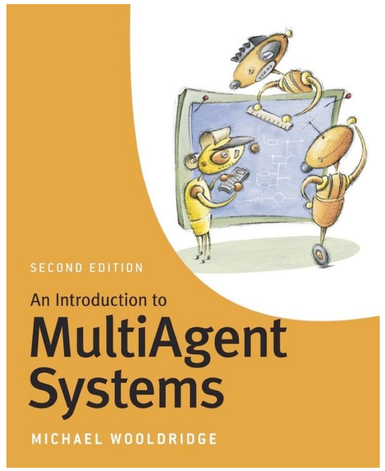
This is an interesting, insightful article, with not too much technical content. It introduces the IRMA architecture for practical reasoning agents, which has been very influential in the design of subsequent systems.



53

# IT4899
# Multi-Agent Systems
Chapter 5 - Reactive and Hybrid Architectures

Dr. Nguyen Binh Minh
Department of Information Systems



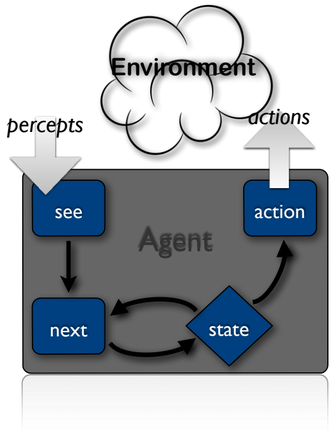



# Reactive Architectures

- There are many unsolved (some would say insoluble) problems associated with symbolic AI.
  - These problems have led some researchers to question the viability of the whole paradigm, and to the development of reactive architectures.
  - Although united by a belief that the assumptions underpinning mainstream AI are in some sense wrong, *reactive* agent researchers use many different techniques.
- In this chapter, we look at alternative architectures that better support some classes of agents and robots
  - At the end, we then examine how *hybrid architectures* exploits the best aspects of deliberative and reactive ones

# General Control Architecture

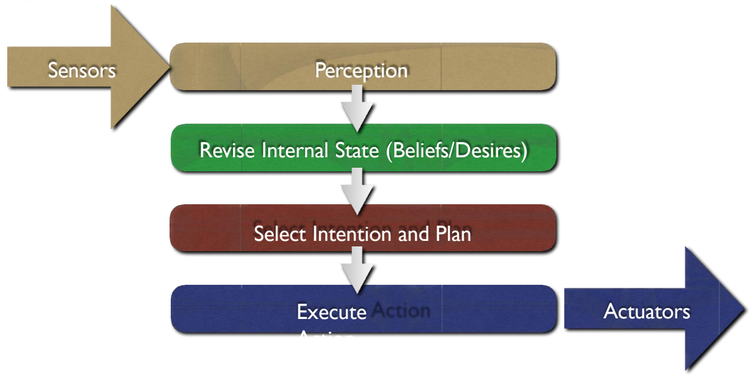
- So far, we have viewed the control architecture of an agent as one that:
  - *Perceives* the environment
  - *Revises* its internal state, identifying beliefs and desires
  - *Selects* actions from its intention and plan
  - *Acts*, possibly changing the environment
- Intention Reconsideration is important in highly dynamic environments





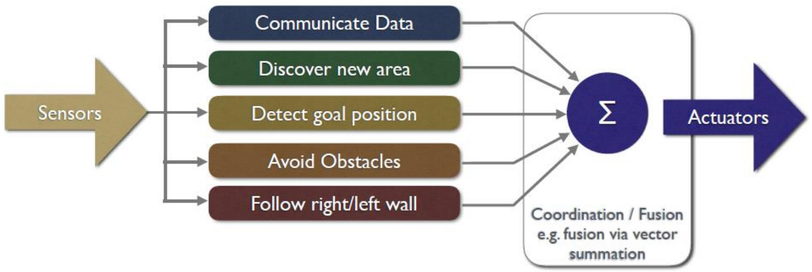



# Agent Control Loop as Layers




The classic “Sense/Plan/Act” approach breaks it down serially like this

# Behaviours



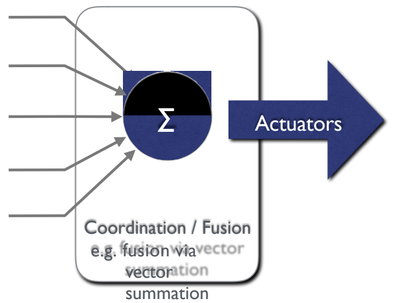

- Behaviour based control sees things differently
  - Behavioural chunks of control each connecting sensors to actuators
  - Implicitly parallel
  - Particularly well suited to Autonomous Robots





# Behaviours

- Range of ways of combining behaviours.
- Some examples:
  - Pick the ``best''
  - Sum the outputs
  - Use a weighted sum
- Flakey redux used a fuzzy combination which produced a nice integration of outputs.



# Subsumption Architecture

- A subsumption architecture is a hierarchy of task-accomplishing *behaviours*.
  - Each behaviour is a rather simple rule-like structure.
  - Each behaviour 'competes' with others to exercise control over the agent.
  - Lower layers represent more primitive kinds of behaviour, (such as avoiding obstacles), and have precedence over layers further up the hierarchy.
- The resulting systems are, in terms of the amount of computation they do, *extremely* simple.
  - Some of the robots do tasks that would be impressive if they were accomplished by symbolic AI systems.

Rodney Brooks "subsumption architecture" was originally developed open Genghis





# Brooks Behavioural Languages

- Brooks proposed the following three theses:
  1. Intelligent behaviour can be generated *without* explicit *representations* of the kind that symbolic AI proposes.
  2. Intelligent behaviour can be generated *without* explicit *abstract reasoning* of the kind that symbolic AI proposes.
  3. Intelligence is an *emergent* property of certain complex systems.

# Brooks Behavioural Languages

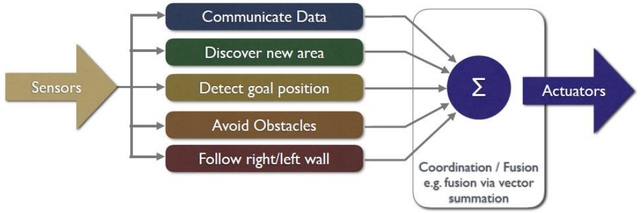
- He identified two key ideas that have informed his research:
  1. *Situatedness and embodiment*: 'Real' intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.
  2. *Intelligence and emergence*: 'Intelligent' behaviour arises as a result of an agent's interaction with its environment. Also, intelligence is 'in the eye of the beholder'; it is not an innate, isolated property.
- Brooks built several agents (such as Genghis) based on his *subsumption architecture* to illustrate his ideas.



9

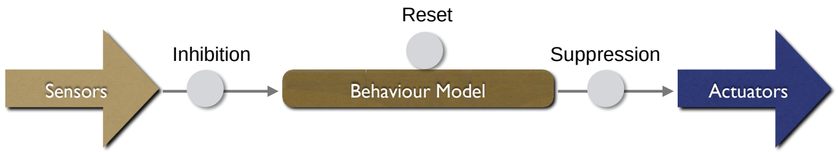
# Subsumption Architecture

- It is the piling up of layers that gives the approach of its power.
  - Complex behaviour emerges from simple components.
  - Since each layer is independent, each can independently be:
    - Coded / Tested / Debugged
  - Can then assemble them into a complete system.



# Abstract view of a Subsumption Machine

- Layered approach based on levels of competence
  - Higher level behaviours *inhibit* lower levels
- Augmented finite state machine:

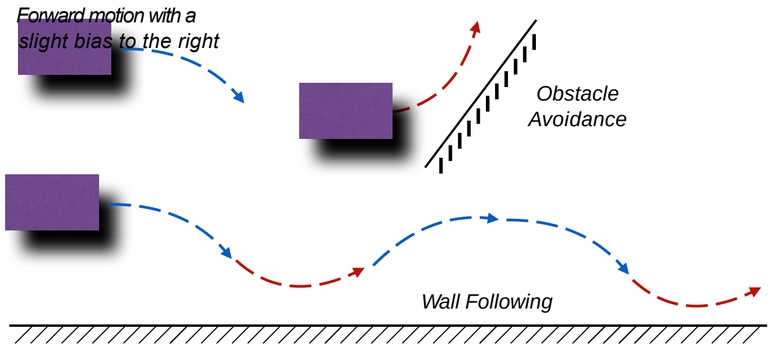







# Emergent Behaviour

Putting simple behaviours together leads to synergies



# Emergent behaviour

- Important but not well-understood phenomenon
  - Often found in behaviour-based/reactive systems
- Agent behaviours “emerge” from interactions of rules with environment.
  - Sum is greater than the parts.
  - The interaction links rules in ways that weren’t anticipated.
- *Coded behaviour*: In the programming scheme
- *Observed behaviour*: In the eyes of the observer
  - There is no one-to-one mapping between the two!
- When observed behaviour “exceeds” programmed behaviour, then we have emergence.

13

***Emergent Flocking***

*Flocking is a classic example of emergence, e.g. Reynolds “Boids”, or Mataric´s “nerd herd”.*

*Each agent uses the following three rules:*

1. *Don’t run into any other robot*
2. *Don’t get too far from other robots*
3. *Keep moving if you can*

*When run in parallel on many agents, the result is flocking*

13



# ToTo

- Maja Mataric´’s Toto is based on the subsumption architecture
  - Can map *spaces* and *execute plans* without the need for a *symbolic* representation.
  - Inspired by “…*the ability of insects such as bees to identify shortcuts between feeding sites*…”
- Each feature/landmark is a set of sensor readings
  - Signature
- Recorded in a behaviour as a triple:
  - Landmark type
  - Compass heading
  - Approximate length/size
- Distributed topological map.


14

# ToTo

- Whenever Toto visited a particular landmark, its associated map behaviour would become activated
  - If no behaviour was activated, then the landmark was new, so a new behaviour was created
  - If an existing behaviour was activated, it inhibited all other behaviours
- Localization was based on which behaviour was active.
  - No map object, but the set of behaviours clearly included map functionality.



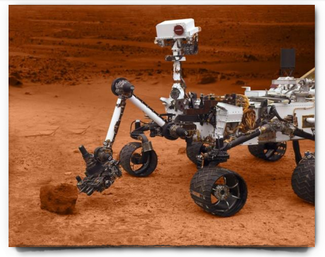



# Steel's Mars Explorer System

- Steels' Mars explorer system
  - Uses the *subsumption* architecture to achieve near-optimal cooperative performance in simulated 'rock gathering on Mars' domain
  - *Individual behaviour* is governed by a set of simple rules.
  - *Coordination between agents* can also be achieved by leaving "markers" in the environment.

**Objective**

To explore a distant planet, and in particular, to collect sample of a precious rock. The location of the samples is not known in advance, but it is known that they tend to be clustered.



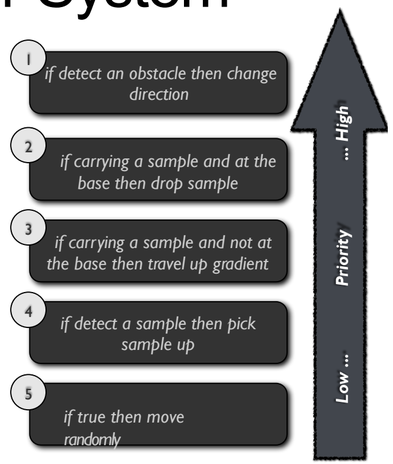
# Steel's Mars Explorer System

1. For individual (non-cooperative) agents, the lowest-level behaviour, (and hence the behaviour with the highest "priority") is obstacle avoidance.

2. Any samples carried by agents are dropped back at the mother-ship.

3. If not at the mother-ship, then navigate back there.
   - The "gradient" in this case refers to a virtual "hill" radio signal that slopes up to the mother ship/base.

4. Agents will collect samples they find.
5. An agent with "nothing better to do" will explore randomly. This is the highest-level behaviour (and hence lowest level "priority").

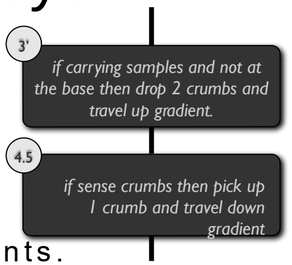







# Steel's Mars Explorer System

- Existing strategy works well when samples are distributed randomly across the terrain.
  - However, samples are located in clusters
  - Agents should *cooperate* with each other to locate clusters
- Solution to this is based on foraging ants.
  - Agents leave a "radioactive" trail of crumbs when returning to the mother ship with samples.
    - If another agent senses this trail, it follows the trail back to the source of the samples
    - It also picks up some of the crumbs, making the trail fainter.
    - If there are still samples, the trail is reinforced by the agent returning to the mother ship (leaving more crumbs)
    - If no samples remain, the trail will soon be erased.




# Situated Automata

- Approach proposed by Rosenschein and Kaelbling.
  - An agent is specified in a rule-like (declarative) language.
    - Then compiled down to a digital machine, which satisfies the declarative specification.
  - This digital machine can operate in a *provable time bound*.
  - Reasoning is done *off line*, at compile time, rather than *online at run time*.
- The theoretical limitations of the approach are not well understood.
  - *Compilation* (with propositional specifications) is equivalent to an NP-complete problem.
  - The more expressive the agent specification language, the harder it is to compile it.



19

# Situated Automata

- An agent is specified by perception and action
  - Two programs are used to synthesise agents:
    1. RULER specifies the perception component
       - (see opposite)
    2. GAPPS specifies the action component
       - Takes a set of goal reduction rules and a top level goal (symbolically specified) and generates a non-symbolic program

**RULER takes as its input three components…**

"…[A] specification of the semantics of the [agent's] inputs ("whenever bit 1 is on, it is raining"); a set of static facts ("whenever it is raining, the ground is wet"); and a specification of the state transitions of the world ("if the ground is wet, it stays wet until the sun comes out"). The programmer then specifies the desired semantics for the output ("if this bit is on, the ground is wet"), and the compiler … [synthesises] a circuit whose output will have the correct semantics… All that declarative "knowledge" has been reduced to a very simple circuit…"

*Kaelbling, L.P. (1991) A Situated Automata Approach to the Design of Embedded Agents. SIGART Bulletin, 2(4): 85-88*



20

# Limitations of Reactive Systems

- Although there are clear advantages of Reactive Systems, there are also limitations!
  - If a model of the environment isn't used, then sufficient information of the local environment is needed for determining actions
  - As actions are based on local information, such agents inherently take a "short-term" view
  - *Emergent behaviour is very hard to engineer* or validate; typically a trial and error approach is ultimately adopted
  - Whilst agents with few layers are straightforward to build, *models using many layers are inherently complex* and difficult to understand.





# Hybrid Architectures

- Many researchers have argued that neither a completely deliberative nor completely reactive approach is suitable for building agents.
- They have suggested using *hybrid* systems, which attempt to marry classical and alternative approaches.
- An obvious approach is to build an agent out of two (or more) subsystems:
  - a *deliberative* one, containing a symbolic world model, which develops plans and makes decisions in the way proposed by symbolic AI; and
  - a *reactive* one, which is capable of reacting to events without complex reasoning.

# Hybrid Architectures

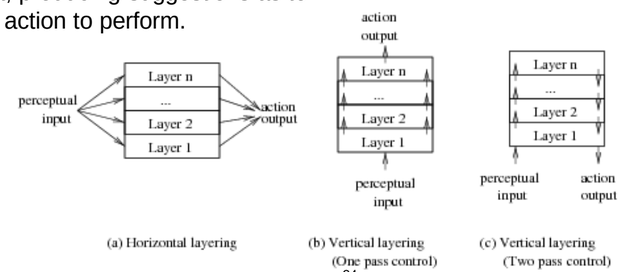
- Often, the reactive component is given some kind of precedence over the deliberative one.
- This kind of structuring leads naturally to the idea of a layered architecture, of which *InterRap* and *TouringMachines* are examples.
  - In such an architecture, an agent's control subsystems are arranged into a *hierarchy*…
  - …with higher layers dealing with information at *increasing levels of abstraction*.
- A key problem in such architectures is what kind control framework to embed the agent's subsystems in, to manage the interactions between the various layers.





# Hybrid Architectures

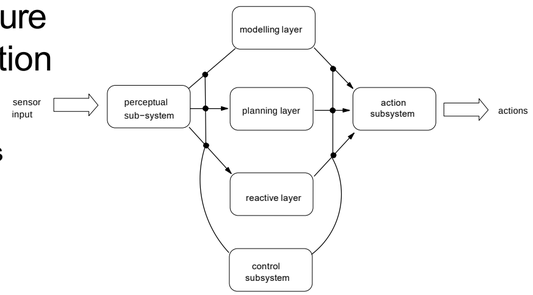
- ***Horizontal layering.***
  - Layers are each directly connected to the sensory input and action output.
  - In effect, each layer itself acts like an agent, producing suggestions as to what action to perform.
- ***Vertical layering.***
  - Sensory input and action output are each dealt with by at most one layer each.




(a) Horizontal layering (b) Vertical layering (One pass control) (c) Vertical layering (Two pass control)

# Ferguson - TouringMachines

- The TouringMachines architecture consists of perception and action subsystems
  - These interface directly with the agent's environment, and three control layers, embedded in a control framework, which mediates between the layers.








# Ferguson - TouringMachines

- The ***reactive layer*** is implemented as a set of situation-action rules, this is subsumption architecture.
  - The planning layer constructs plans and selects actions to execute in order to achieve the agent's goals.

```
rule-1: kerb-avoidance
    if
        is-in-front(Kerb, Observer) and
        speed(Observer) > 0 and
        separation(Kerb, Observer) < KerbThreshHold
    then
        change-orientation(KerbAvoidanceAngle)
```

# Ferguson - TouringMachines

- The ***modelling layer*** contains symbolic representations of the 'cognitive state' of other entities in the agent's environment.
  - The three layers communicate with each other and are embedded in a control framework, which use control rules.
  - Such control structures have become common in robotics.

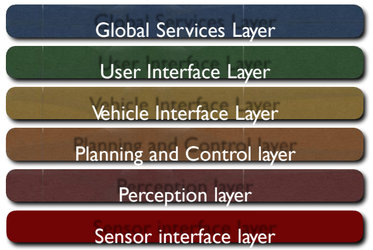
```
censor-rule-1:
    if
        entity(obstacle-6) in perception-buffer
    then
        remove-sensory-record(layer-R, entity(obstacle-6))
```





# Real World Example: Stanley

- Won the 2005 DARPA Grand Challenge
  - Used a combination of the subsumption architecture with deliberative planning
  - Consists of 30 different independently operating modules across 6 layers




# Summary

- This lecture has looked at two further kinds of agent:
  - Reactive agents; and
  - Hybrid agents.
- *Reactive agents* build complex behaviour from simple components.
- *Hybrid agents* try to combine the speed of reactive agents with the power of deliberative agents.

*Class Reading (Chapter 5):*

*"A Robust Layered Control System for a Mobile Robot,", Rodney A. Brooks. IEEE Journal of Robotics and Automation, 2(1), March 1986, pp. 14–23. (also MIT AI Memo 864, September 1985)*

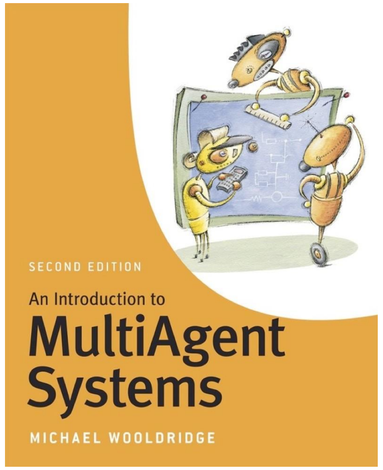
A provocative, fascinating article, packed with ideas. It is interesting to compare this with some of Brook's later - arguably more controversial - articles



31

# IT4899
# Multi-Agent Systems
Chapters 6/7 - Ontologies & Communication

Dr. Nguyen Binh Minh
Department of Information Systems






# Social Behaviour

- Previously we looked at:
  - Deductive Agents
  - Practical Reasoning, and BDI Agents
  - Reactive and Hybrid Architectures
- We said:
  - *An intelligent agent is a computer system capable of flexible autonomous action in some environment*.
  - Where by flexible, we mean:
    - reactive
    - pro-active
    - social
- This is where we deal with the "*social*" bit, showing how agents communicate and share information.

# Agent Communication

- In this lecture, we cover macro-aspects of intelligent agent technology, and those issues relating to the agent society, rather than the individual:
  - communication:
    - speech acts; KQML & KIF; FIPA ACL
  - ontologies:
    - the role of ontologies in communication
    - aligning ontologies
    - OWL
- There are some limited things that one can do without communication, but they are…, well…, *limited*!!!
  - Most work on multiagent systems assumes communication.



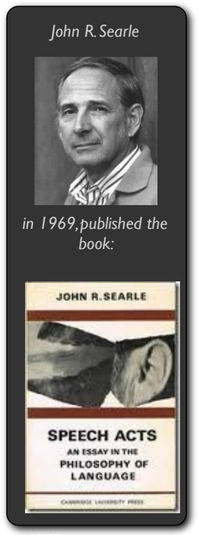


# Speech Acts

- Austin's 1962 book "***How to Do Things with Words***" is usually taken to be the origin of speech acts
- Speech act theories are *pragmatic* theories of language, that is theories of how language *use*:
  - they attempt to account for how language is used by people every day to achieve their goals and intentions.
- Austin noticed that some utterances are rather like 'physical actions' that appear to *change the state of the world*.

# Speech Acts

- Paradigm examples are:
  - declaring war;
  - naming a child;
  - "*I now pronounce you man and wife*" :-)
- But more generally, *everything* we utter is uttered with the intention of satisfying some goal or intention.
- A theory of how utterances are used to achieve intentions is a *speech act theory*.
  - Proposed by John Searle, 1969.






6

# Speech Acts: Searle

- In his 1969 book Speech Acts: an Essay in the Philosophy of Language he identified:
  - *representatives*:
    - such as informing, e.g., 'It is raining'
  - *directives*:
    - attempts to get the hearer to do something e.g., 'please make the tea'
  - *commisives*:
    - which commit the speaker to doing something, e.g., 'I promise to...'
  - *expressives*:
    - whereby a speaker expresses a mental state, e.g., 'thank you!'
  - *declarations*:
    - such as declaring war or naming.



7

# Speech Acts: Searle

- There is some debate about whether this (or any!) typology of speech acts is appropriate.
- In general, a speech act can be seen to have two components:
  - a *performative* verb:
    - (e.g., request, inform, . . . )
  - *propositional* content:
    - (e.g., "the door is closed")

The door is closed...

**request**
speech act = "*please close the door*"

**inform**
speech act = "*the door is closed!*"

**inquire**
speech act = "*is the door closed?*"

Each of the above speech acts result from the same propositional content ("*...the door is closed...*"), but with different performatives





8

# Plan Based Semantics

- How does one define the semantics of speech acts?
  - When can one say someone has uttered, e.g., a request or an inform?
- Cohen & Perrault (1979) defined semantics of speech acts using the *precondition-delete-add* list formalism of planning research.
  - Just like STRIPS planner
- Note that a speaker cannot (generally) *force* a hearer to accept some desired mental state.

***The semantics for "request"***

*request(s, h, φ)*

*precondition:*

- *s believes h can do φ*
  (you don't ask someone to do something unless you think they can do it)
- *s believe h believe h can do φ*
  (you don't ask someone unless they believe they can do it)
- *s believe s want φ*
  (you don't ask someone unless you want it!)

*post-condition:*

- *h believe s believe s want φ*
  (the effect is to make them aware of your desire)





9

# KQML and KIF

- We now consider *agent communication languages (ACLs)*
  - ACLs are standard formats for the exchange of messages.

> "… [developing] protocols for the exchange of represented knowledge between autonomous information systems…"
> Tim Finin, 1993

- One well known ACL is KQML, developed by the DARPA-funded *Knowledge Sharing Effort* (KSE).
  - The ACL proposed by KSE was comprised of two parts:
    - the message itself: the *Knowledge Query and Manipulation Language (KQML)*; and the body of the message: the *Knowledge interchange format (KIF)*.





# KQML and KIF

- KQML is an 'outer' language, that defines various acceptable 'communicative verbs', or *performatives*.
  - Example performatives:
    - `ask-if` ('is it true that. . . ')
    - `perform` ('please perform the following action. . . ')
    - `tell` ('it is true that. . . ')
    - `reply` ('the answer is . . . ')

- KIF is a language for expressing message *content*, or *domain knowledge*.
  - It can be used to writing down *ontologies*.
  - KIF is based on first-order logic.

# KQML & Ontologies

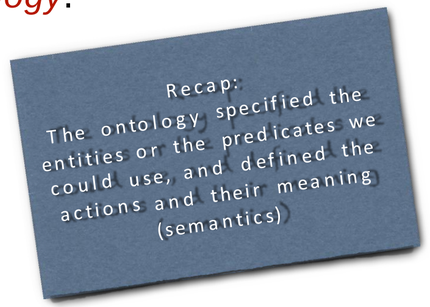
- In order to be able to communicate, agents need to *agree on the words* (terms) they use to describe a domain.
  - Always a problem where multiple languages are concerned.
- A formal specification of a set of terms is known as a ontology.
  - The DARPA *Knowledge Sharing Effort* project has associated with it a large effort at defining common ontologies
    - software tools like ontolingua, etc, for this purpose.
- We've previously discussed the use of ontologies and semantics…





# Blocksworld

- The environment is represented by an *ontology*.




**Blocksworld Ontology**

| | |
|---|---|
| *On(x,y)* | *object x on top of object y* |
| *OnTable(x)* | *object x is on the table* |
| *Clear(x)* | *nothing is on top of object x* |
| *Holding(x)* | *arm is holding x* |

$Stack(x, y)$

| | |
|---|---|
| pre | $Clear(y) \wedge Holding(x)$ |
| del | $Clear(y) \wedge Holding(x)$ |
| add | $ArmEmpty \wedge On(x, y)$ |

$UnStack(x, y)$

| | |
|---|---|
| pre | $On(x, y) \wedge Clear(x) \wedge ArmEmpty$ |
| del | $On(x, y) \wedge ArmEmpty$ |
| add | $Holding(x) \wedge Clear(y)$ |

# Ontologies

- The role of an ontology is to fix the meaning of the terms used by agents.

> "… An ontology is a formal definition of a body of knowledge. The most typical type of ontology used in building agents involves a structural component. Essentially a taxonomy of class and subclass relations coupled with definitions of the relationships between these things …"
>
> Jim Hendler

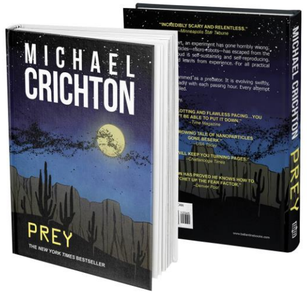
- How do we do this? Typically by defining new terms in terms of old ones.
  - Let's consider an example.





# Ontologies

- Alice:
  - *Did you read "Prey"?*
- Bob:
  - *No, what is it?*
- Alice:
  - *A science fiction novel. Well, it is also a bit of a horror novel. It is about multiagent systems going haywire.*



# Ontologies

- What is being conveyed about “Prey” here?
  1. It is a novel
  2. It is a science fiction novel
  3. It is a horror novel
  4. It is about multiagent systems
- Alice assumes that Bob knows what a “*novel*” is, what “*science fiction*” is and what “*horror*” is.
- She thus defines a new term “*Prey*” in terms of ones that Bob already knows.

**Types of objects**

**Classes**
*collections of things with similar properties*

**Instances**
*specific examples of classes*

**Relations**
*Describe the properties of objects and connect them together*

Note that we also have these types of objects in languages such as Java, or modelling frameworks such as ER Diagrams. Such languages and frameworks also support inheritance.





# Ontologies

- Part of the reason this interaction works is that Bob has knowledge that is relevant.
  - Bob knows that novels are fiction books
    - “novel” is a subclass of “fiction book”
  - Bob knows things about novels: they have
    - authors,
    - publishers,
    - publication dates, and so on.
- Because “Prey” is a novel, it *inherits* the properties of novels. It has an author, a publisher, a publication date.
  - Instances inherit attributes from their classes.

# Ontology Inheritance

- Classes also inherit.
  - Classes inherit attributes from their super-classes.
    - If “novel” is a subclass of “fiction book”, then “fiction book” is a superclass of “novel”
- Fiction books are books.
  - Books are sold in bookstores.
  - Thus fiction books are sold in bookstores.

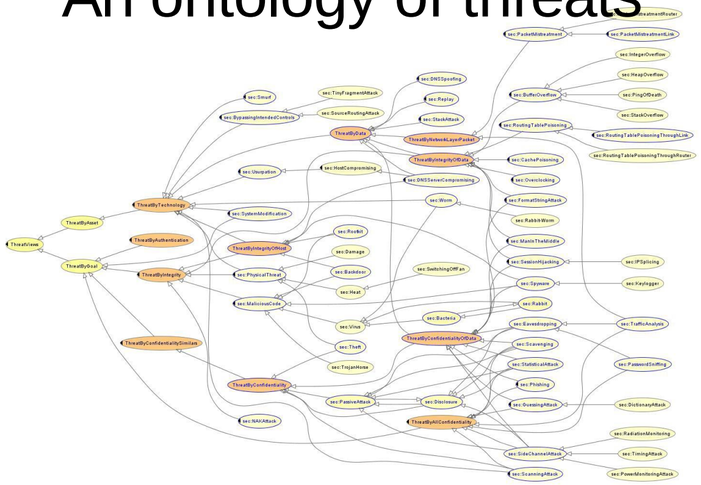




# Ontologies

- A lot of knowledge can be captured using these notions.
  - We specify which class “***is-a***” sub-class of which other class.
  - We specify which classes have which ***attributes***.
  - An ***axiomatic theory*** can also be included to support inference
    - *if socrates is [an instance of a] man, and all men are mortal…*
    - *… we can infer that socrates is mortal!*
- This structure over knowledge is called an ontology.
  - A knowledge base is an ontology with a set of instances.
- A **huge** number of ontologies have been constructed.

# An ontology of threats



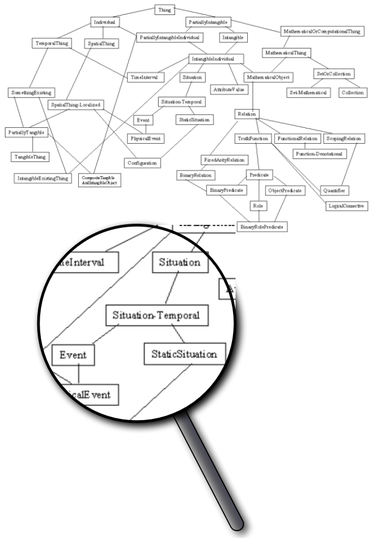







# Ontologies

- In general there are multiple ontologies at different levels of detail.
  - Application ontology
    - Like the threat ontology (see opposite)
  - Domain ontology
  - Upper ontology
    - Contains very general information about the world.
- The more specific an ontology, the less reusable it is.



# Multiple Ontologies

- Application and domain ontologies will typically overlap
  - Illustrated by the challenges of facilitating interoperability between similar ontologies.
  - Different knowledge systems can be integrated to form merged knowledge bases
- But in many cases, all that is needed is to be understood!

**Ontologies as perspectives on a domain**

A single domain may have an **intended representation** in the real world, that is not perfectly represented by any single formal ontology. Many separate ontologies then emerge, based on different contexts…

**Ontology Alignment Evaluation Initiative**

A test suite of similar ontologies used to test out alignment systems that link different ontologies representing the same domain.

The conference test suite consists of 21 ontology pairs.







# Modelling and Context

- The problem with modelling ontologies is that different designers have different contexts, and requirements

**The Architect**

*When modelling a bridge, important characteristics include:*

*tensile strength*
*weight*
*load*
*etc*

*Pat Hayes, 2001 in conversation*

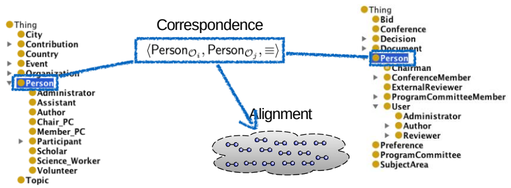
**The Military**

*When modelling a bridge, important characteristics include:*

*what munitions are required to destroy it!*

## Ontologies, Alignments and Correspondences

- Different ontological models exist for overlapping domains
  - Modelled implicitly, or explicitly by defining entities (classes, roles etc), typically using some logical theory, i.e. an Ontology
- Alignment Systems align similar ontologies






24





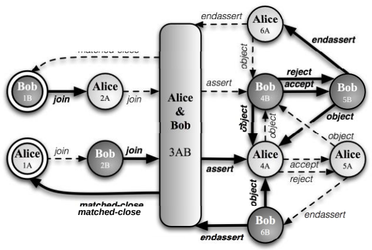
## Aligning Agents' Ontologies

> *"… as agents can differ in the ontologies they assume, the resulting semantic heterogeneity can impede meaningful communication. One solution is to align the ontologies; i.e. find correspondences between the ontological entities to resolve this semantic heterogeneity. However, this raises the question: how can agents align ontologies that they do not want to disclose?…"*
>
> *Terry Payne & Valentina Tamma, 2014*

- Agent alignment is inherently *decentralised*!

# Correspondence Inclusion Dialogue

- Correspondence Inclusion Dialogue (CID)
  - Allows two agents to *exchange knowledge about correspondences* to agree upon a *mutually acceptable* final alignment AL.
  - This alignment aligns only those entities in each agents' working ontologies, without disclosing the ontologies, or all of the known correspondences.
- Assumptions
  1. Each agent knows about different *correspondences* from *different* sources
  2. This knowledge is *partial*, and possibly *ambiguous*; i.e. more than one correspondence exists for a given entity
  3. Agents associate a utility (*Degree of Belief*) $K_C$ to each unique correspondence



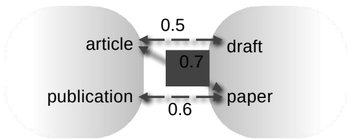


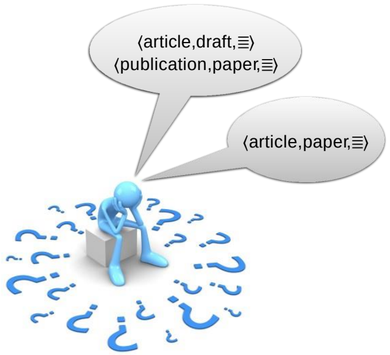





# Picking the right correspondences

- Quality vs Quantity
  - Do we maximise coverage
    - Preferable when merging the whole ontology
  - Do we find the "best" mappings
    - Preferable when aligning specific signatures



# OWL - Web Ontology Language

- A general purpose family of ontology languages for describing knowledge
  - Originated from the *DARPA Agent Markup Language Program*
    - Followup to DARPA Control of Agent Based Systems (CoABS)


- Based on *description logics*
  - Various flavours with different expressivity / computability
  - Different syntaxes: XML, Turtle, Manchester Syntax…
  - Underpins the Semantic Web



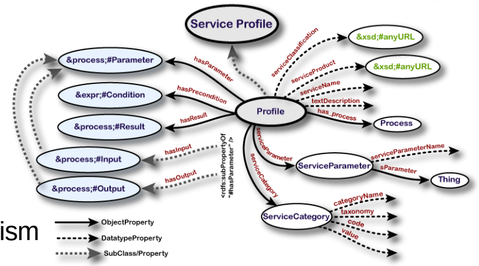


# OWL Example

```
NS1:geographicCoordinates rdf:nodeID='A179'/>
   <NS1:mapReferences>North America</NS1:mapReferences>
   <NS1:totalArea>9629091</NS1:totalArea>
   <NS1:landArea>9158960</NS1:landArea>
   <NS1:waterArea>470131</NS1:waterArea>
   <NS1:comparativeArea>about half the size of Russia; about three-tenths the size of Africa; about half the size of South America (or slightly larger than Brazil); slightly larger than China; about two and a half times the size of Western Europe
   </NS1:comparativeArea>
   <NS1:landBoundaries>12034</NS1:landBoundaries>
   <NS1:coastline>19924</NS1:coastline>
   <NS1:contiguousZone>24</NS1:contiguousZone>
   <NS1:exclusiveEconomicZone>200</NS1:exclusiveEconomicZone>
   <NS1:territorialSea>12</NS1:territorialSea>
   <NS1:climate>mostly temperate, but tropical in Hawaii and Florida, arctic in Alaska, semiarid in the great plains west of the Mississippi River, and arid in the Great Basin of the southwest; low winter temperatures in the northwest are ameliorated occasionally in January and February by warm chinook winds from the eastern slopes of the Rocky Mountains
   </NS1:climate>
   <NS1:terrain>vast central plain, mountains in west, hills and low mountains in east; rugged mountains and broad river valleys in Alaska; rugged, volcanic topography in Hawaii
   </NS1:terrain>
```

# OWL and Services


- OWL-S is an *upper level service ontology* developed to describe agent based (or semantic web) services
  - Profile used for discovery
    - input / outputs etc
  - Process model provided a planning formalism
  - Grounding linked to the syntactic messaging








# KQML / KIF

- After that digression, we can return to the KQML/KIF show.
  - KQML is an agent communication language. It provides a set of performatives for communication.
- KIF is a language for representing domain knowledge.
  - It can be used to writing down ontologies.
  - KIF is based on first-order logic.
- Given that, let's look at some examples.

# KQML/KIF Example

```
(stream-about
  :sender      A
  :receiver    B
  :language    KIF
  :ontology    motors
  :reply-with  q1
  :content m1
)
```

A asks B for information about motor 1, using the ontology (represented in KIF) about motors.

```
(tell
  :sender      B
  :receiver    A
  :in-reply-to q1
  :content
   (= (torque m1) (scalar 12 kgf))
)
(tell
  :sender      B
  :receiver    A
  :in-reply-to q1
  :content
    (= (status m1) normal)
)
(eos
  :sender      B
  :receiver    A
  :in-reply-to q1
)
```

B responds to A's query q1. Two facts are sent:

1) that the torque of motor 1 is 12kgf; and
2) that the status of the motor is normal.

The ask stream is terminated using the eos performative.





# Problems with KQML

- The basic KQML performative set was fluid
  - Different implementations were not *interoperable*
- Transport mechanisms for messages were not precisely defined
  - Again - *interoperability*
- Semantics of KQML were not rigorously defined
  - Ambiguity resulted in impairing *interoperability*!
- There were no commissives in the language
  - Without the ability to *commit* to a task, how could agents *coordinate* behaviour
- The performative set was arguably ad-hoc and overly large

# FIPA ACL

- More recently, the Foundation for Intelligent Physical Agents (FIPA) started work on a program of agent standards
  - the centrepiece is an ACL.

- Basic structure is quite similar to KQML:
  - The number of performatives was reduced to 20.
  - A formal semantics has been defined for the language, using the language SL
    - SL can represents beliefs, desires and uncertain beliefs, as well as actions

*FIPA ACL example*

```
(inform
    :sender     agent1
    :receiver   agent5
    :content    (price good200 150)
    :language   sl
    :ontology   npi-auction
)
```







## FIPA ACL Performatives

| performative | passing info | requesting info | negotiation | performing actions | error handling |
|---|---|---|---|---|---|
| accept-proposal | x | x | x | x | x |
| agree | | | | | |
| cancel | | | | x | |
| cfp | | | x | | |
| confirm | | | | | |
| disconfirm | x | | | | |
| failure | | | | | |
| inform | x | | | | |
| inform-if | x | | | | |
| inform-ref | x | | | | |
| not-understood | | | | | x |
| propose | | | x | | |
| query-if | | x | | | |
| query-ref | | x | | | |
| refuse | | | | x | |
| reject-proposal | | | x | | |
| request | | | | x | |
| request-when | | | | x | |
| request-whenever | | | | x | |
| subscribe | | x | | | |

# “Inform” and “Request”

- “Inform” and “Request” are the two basic performatives in FIPA. Others are macro definitions, defined in terms of these.
- The meaning of inform and request is defined in two parts:
  - *pre-condition*: what must be true in order for the speech act to succeed.
  - “*rational effect*”: what the sender of the message hopes to bring about.

**FIPA “Inform” Performative**

The content is a ***statement***. The pre-condition is that the sender:

- holds that the content is true;
- intends that the recipient believe the content;
- does not already believe that the recipient is aware of whether content is true or not

The speaker only has to believe that what he says is true.

**FIPA “Request” Performative**

The content is an ***action***. The pre-condition is that the sender:

- intends action content to be performed;
- believes recipient is capable of performing this action;
- does not believe that recipient already intends to perform action.

The last of these conditions captures the fact that you don’t speak if you don’t need to.





# Communication in AgentSpeak

- AgentSpeak agents communicate using a simpler structure to KQML/ ACL
- Messages received typically have the form

  <sender, performative, content>
  - sender: the AgentSpeak term corresponding to the agent that sent the message
    - i.e. an agentID
  - performative: this represents the goal the sender intends to achieve by sending the message
    - tell, achieve, askOne, tellHow etc
  - content: an AgentSpeak formula or message body
    - varies depending on the performative

# Messages in Jason

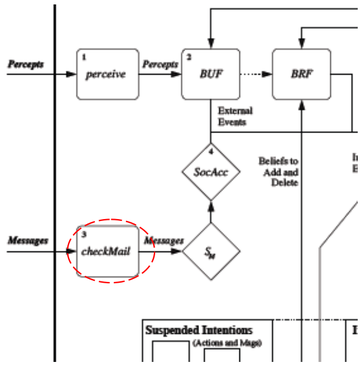
- Messages are passed through the use of internal actions that are pre-defined in Jason
  - The most typically used are:
    .send(receiver, performative, content)
    .broadcast(performative, content)
  - where receiver, performative and content relate to the elements in the message
- The .send action sends messages to specific agents
  - The receiver can be a single agentID, or a list of agentIDs
- The .broadcast action sends the message to all agents registered in the system





# Handling messages in Jason

- In the internal Jason architecture
  - messages are delivered into the agents "mailbox"
    - This is done automatically by the customisable checkMail method
    - Passes them onto the AgentSpeak interpreter
  - One message is processed during each reasoning cycle
    - A customisable *message selection function* ($S_M$) selects the next message to process
    - A *selection process* ($SocAcc$) determines if the message should be rejected
      - For example, ignoring messages from a certain agent
      - Think of this as a spam filter
  - If the message goes through, Jason will interpret it according to precise semantics
    - by generating new events pertaining to the goal and belief bases, and in turn, triggering plans



# Performatives in Jason

- Sharing Beliefs (Information Exchange)
  - tell and untell
    - The sender intends the receiver (not) to believe the literal in the content to be true and that the sender believes it
- Sharing Plans (Deliberation)
  - tellHow and untellHow
    - The sender requests the receiver (not) to include within their plan library the plan in the message content
  - askHow
    - The sender wants to know the receiver's applicable plan for the triggering event in the message content
- Delegate an Achievement Gaol (Goal Delegation)
  - achieve and unachieve
    - The sender requests the receiver (not) to try and achieve a state-of-affairs where the content of the message is true
- Delegate a Test Goal (Information Seeking)
  - askOne and askAll
    - The sender wants to know whether the receiver knows (askOne) if the content is true (i.e. is there a single answer) or for all answers (askAll).





# Semantics of tell / untell

Information Exchange

| Cycle # | sender (s) actions | recipient (r) belief base | recipient (r) events |
|---|---|---|---|
| 1 | .send (r, tell, open(left_door)) | | |
| 2 | | open(left_door)[source(s)] | ⟨+open(left_door) [source(s)],T⟩ |
| 3 | .send (r, untell, open(left_door)) | | |
| 4 | | | ⟨-open(left_door) [source(s)],T⟩ |

- Note that events are represented internally as a tuple: ⟨event, intention⟩
  - This associates an event with an intention that generated it
- With communication, there is no intention responsible for the event
  - Thus, we indicate this with the `T` symbol

# Semantics of achieve / unachieve

## Goal Delegation

| Cycle # | sender (s) actions | recipient (r) intentions | recipient (r) events |
|---|---|---|---|
| 1 | .send (r, achieve, open(left_door)) | | |
| 2 | | | ⟨+!open(left_door) [source(s)], T⟩ |
| 3 | | !open(left_door)[source(s)] | |
| 3 | .send (r, unachieve, open(left_door)) | !open(left_door)[source(s)] | |
| 4 | | <<< intention has been removed >>> | |

- Note that the intention is adopted *after* the goal is added.
- With unachieve, the internal action .drop_desire(open(left_door)) is executed.





# Semantics of askOne / askAll

## Information Seeking

| Cycle # | sender (s) actions | recipient (r) actions | sender (s) events |
|---|---|---|---|
| 1 | .send (r, askOne, open(Door)) | | |
| 2 | | .send(s, tell, open(left_door)) | |
| 3 | | | ⟨+open(left_door)[source(r)], T⟩ |
| 4 | .send (r, askAll, open(Door)) | | |
| 5 | | .send(s, tell, [open(left_door), open(right_door)]) | |
| 6 | | | ⟨+open(left_door)[source(r)], T⟩ |
| | | | ⟨+open(right_door)[source(r)], T⟩ |

r's belief base

open(left_door)

open(right_door)

# Semantics of Deliberation

- .send(receiver, tellHow,"@p ... : ... <- ...")
  - adds the plan to the plan library of the receiver with its plan label @p
    - .send(r, tellHow, "@pOD +!open(Door): not locked(Door) <- turn_handle(Door); push(Door); ?oopen(Door).")
- .send(receiver, untellHow, @p)
  - removes the plan with the plan label @p from the plan library of receiver
    - .send(r, untellHow, "@pOD")
- .send(receiver, askHow, Goal-addition-event)
  - requires receiver to pass all relevant plans to the triggering event in the content
    - .send(r, askHow, "+!open(Door)")





# Handling performatives

- Jason implements plans for each of the performatives
  - More elegant than hard coding within the interpreter
  - Allows the agent developer to introduce new performatives when necessary
    - Existing performatives can be overridden
  - The goal !kqml_received is created whenever a message is received
  - Predefined plans can be found in Jason Distribution in
    - src/asl/kqmlPlans.asl

```
/* ---- achieve performatives ---- */

@kqmlReceivedAchieve
+!kqml_received(KQML_Sender_Var, achieve, KQML_Content_Var, KQML_MsgId)
  <- .add_annot(KQML_Content_Var, source(KQML_Sender_Var), CA);
     !!CA.
```

# Creating performatives

- Defining the tell_rule performative
  - This simple example illustrates how a new performative for sharing rules could be written
    - It is based on the example code tell_rule in the Jason Distribution
  - Two agents are defined:
    - receiver, which implements the plan for the new performative tellRule
    - sender, that sends two messages using tellRule
  - No environment is used in this multi-agent system (MAS)



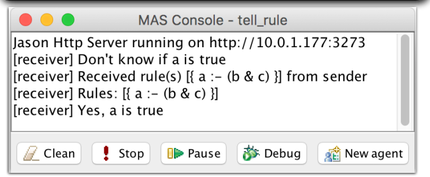
```
//tell_rule.mas2j

/*
This example shows how to customise the
KQML to add a new performative,
identified by "tellRule", used by one agent
to send rules like "a :- b & c" to another
agent.
*/

MAS tell_rule {

    infrastructure: Centralised

    agents:
        receiver;
        sender;

    aslSourcePath:
        "src/asl";
}
```





# Creating performatives

```
// Agent sender in project tell_rule

/* Initial goals */
!start.

/* Plans */
+!start : true
  <-   // ask the receiver to achieve the goal test
       .send(receiver,achieve,test);
       // send a list with a single rule
       .send(receiver,tellRule, [{a :- b & c}]);
       // ask the receiver to achieve the goal test
       .send(receiver,achieve,test).
```

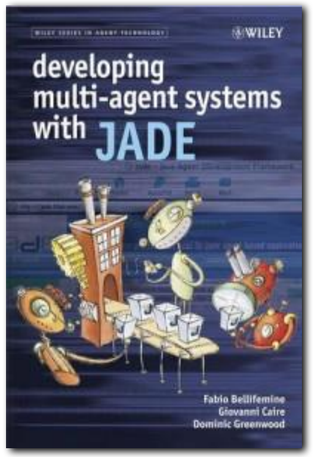





```
// Agent receiver in project tell_rule

/* Initial beliefs */
b.
c.

/* Plans */
+!test : a <- .print("Yes, a is true").
+!test    <- .print("Don't know if a is true").

// customisation of KQML performative tellRule
+!kqml_received(A,tellRule,Rules,_)
  <- .print("Received rule(s) ", Rules, " from ", A);
     for ( .member(R, Rules) ) {
       +R[source(A)];
     }
     // get all rules and print them
     .relevant_rules(_,LR);
     .print("Rules: ",LR).
```

# Jade

- The FIPA ACL provides a language for writing messages down.
  - It says nothing about how they are passed between agents.
- Several software platforms have been developed to support ACL-based communication.
  - One of the most widely used is JADE.
- Provides transparent (from the perspective of the agent designer) transport of ACL messages



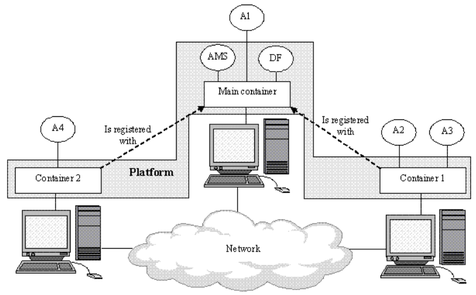





48

# Jade

- In JADE, agents are Java threads running in a “container”.




- All containers register with the main container





49

# JADE Main Container

- The main container does the following:
  - Maintains the *container table* which lists all the containers and their contact information.
  - Maintains a list of all the agents in the system (including location and status).
  - Hosts the *agent management system* (AMS) which names agents as well as creating and destroying them.
  - Hosts the *directory facilitator* which provides a yellow pages allowing agents to be identified by the services they provide.
- See http://jade.tilab.com/ for more details.





# Alternative Semantics

- There is a problem with the "mental state" semantics that have been proposed for the FIPA ACL.
  - This also holds for KQML.
- How do we know if an agent's locutions conform to the specification?
  - As Wooldridge pointed out, *since the semantics are in terms of an agent's internal state*, we cannot *verify compliance* with the semantics laid down by FIPA.
  - In practice, this means that we cannot be sure that a agent is being sincere.
    - Or, more importantly, we cannot detect if it is being insincere.

# Alternative Semantics

- Singh suggested a way to deal with this.
  - Rather than define the conditions on a locution in terms of an agent's mental state, base it on something external to the agent.
- Move from a "mentalistic" semantics to a *social semantics*.
  - How?
- Take an agent's utterances as *commitments*.
  - But what does it mean to say that "if an agent utters an inform then it is committing to the truth of the proposition that is the subject of the utterance"?
- Doesn't stop an agent lying, but it allows you to detect when it does.





# Summary

- This lecture has discussed some aspects of agent ontologies and communication between agents.
  - It has focussed on the interpretation of locutions/ performatives as speech acts, and some suggestions for what performatives one might use.
    - Examples of communication were also given in AgentSpeak / Jason (Chapter 6 of Bordini et al.)
  - There is much more to communication that this. . .
    - . . . but this kind of thing is required as a "transport layer" to support the kinds of thing we will talk about later.

*Class Reading (Chapter 7):*

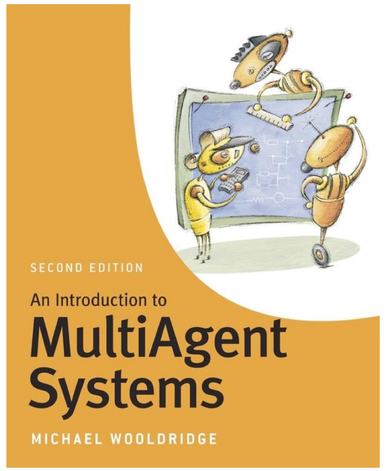
*Agent Communication Languages: Rethinking the Principles", Munindar P. Singh. IEEE Computer: 1998, pp40-49.*

This is an overview of the state of the art in agent communication (as of 1998), and an introduction to the key challenges, particularly with respect to the semantics of agent communication.

# IT4899
# Multi-Agent Systems
Chapters 8 - Working Together

Dr. Nguyen Binh Minh
Department of Information Systems








# Working Together

- Why and how agents work together?
- Since agents are autonomous, they have to make decisions at *run-time*, and be capable of *dynamic coordination*
- Overall they will need to be able to share:
  - Tasks
  - Information
- If agents are designed by different individuals, they may not have common goals
- Important to make a distinction between:
  - *benevolent agents* and
  - *self-interested agents*

# Agent Motivations

- Benevolent Agents
  - If we “own” the whole system, we can design agents to help each other whenever asked
  - In this case, we can assume agents are *benevolent*: our best interest is their best interest
  - Problem-solving in benevolent systems is *Cooperative Distributed Problem Solving* (CDPS)
    - Benevolence simplifies the system design task enormously!
  - We will talk about CDSP *in this lecture*
- Self Interested Agents
  - If agents represent the interests of individuals or organisations, (the more general case), then we cannot make the benevolence assumption
  - Agents will be assumed to act to *further there own interests*, possibly at the expense of others.
    - Potential for *conflict*
    - May complicate the design task enormously
  - Strategic behaviour may be required — we will cover some of these aspects *in later lectures*





# Cooperative Distributed Problem Solving

“… CDPS studies how a loosely coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities. Each problem solving node in the network is capable of sophisticated problem-solving, and can work independently, but the problems faced by the nodes cannot be completed without cooperation. Cooperation is necessary because no single node has sufficient expertise, resources, and information to solve a problem, and different nodes might have expertise solving different parts of the problem….”

(Durfee et. al. 1989).

# Coherence and Coordination

- Coherence:
  - We can measure coherence in terms of solution quality, how efficiently resources are used, conceptual clarity and so on.

> "... how well the [multiagent] system behaves as a unit along some dimension of evaluation..."
>
> (Bond and Gasser, 1988).

- Coordination:
  - If the system is perfectly coordinated, agents will not get in each others' way, in a physical or a metaphorical sense.

> "... the degree. . . to which [the agents]. . . can avoid 'extraneous' activity [such as] . . . synchronizing and aligning their activities..."
>
> (Bond and Gasser, 1988).

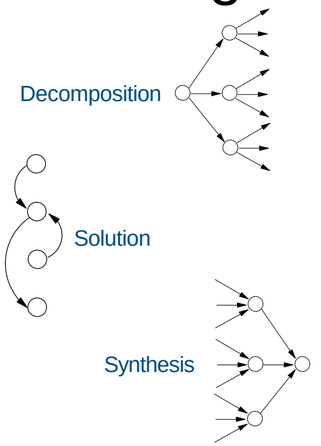






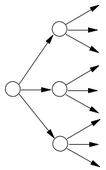
# Task Sharing and Result Sharing

- How does a group of agents work together to solve problems?
- CPDS addresses the following:
  - Problem decomposition
    - How can a problem be *divided into smaller tasks* for distribution amongst agents?
  - Sub-problem solution
    - How can the overall problem-solving activities of the agents be optimised so as to produce a solution that *maximises the coherence metric*?
    - What techniques can be used to coordinate the activity of the agents, *thus avoiding destructive interactions*?
  - Answer synthesis
    - How can a problem solution be effectively *synthesised from subproblem results*?
- Let's look at these in more detail.



# Problem Decomposition

- The overall problem is divided into smaller sub-problems.
  - This is typically a recursive/hierarchical process.
  - Subproblems get divided up also.
  - The granularity of the subproblems is important
    - At one extreme, the problem is decomposed to atomic actions
      - In ACTORS, this is done until we are at the level of individual program instructions.

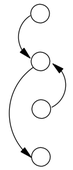


- Clearly there is some processing to do the division.
  - How this is done is one design choice.
- Another choice is *who* does the division.
  - Is it centralised?
  - Which agents have knowledge of task structure?
  - Who is going to solve the sub-problems?



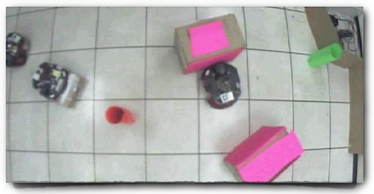


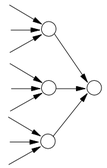




# Sub-problem Solution

- The sub-problems derived in the previous stage are solved.
  - Agents typically share some information during this process.
- A given step may involve two agents synchronising their actions.
  - eg. box pushing





# Solution Synthesis

- In this stage solutions to sub-problems are integrated.
  - Again this may be hierarchical
- Different solutions at different levels of abstraction.

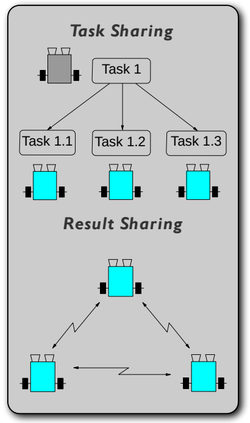






# Solution Synthesis

- Given this model of cooperative problem solving, we have two activities that are likely to be present:
  - ***task sharing***:
    - components of a task are distributed to component agents;
    - how do we decide how to allocate tasks to agents?
  - ***result sharing***:
    - information (partial results etc) is distributed.
    - how do we assemble a complete solution from the parts?
- An agent may well need a solution to both these problems in order to be able to function in a CDPS environment.



# Task Sharing & the Contract Net

- Well known task-sharing protocol for task allocation is the *contract net*.
- The contract net includes five stages:
  1. Recognition;
  2. Announcement;
  3. Bidding;
  4. Awarding;
  5. Expediting.
- The textbook describes these stages in procedural terms from the perspective of an individual agent.

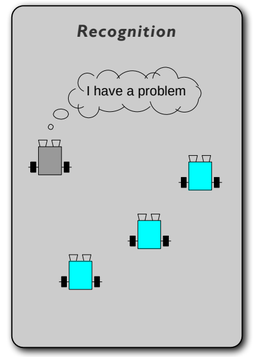






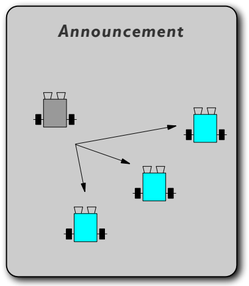
# Recognition

- In this stage, an agent recognises it has a problem it wants help with.
- Agent has a goal, and either. . .
  - realises it cannot achieve the goal in isolation
    - i.e. it does not have capability;
  - realises it would prefer not to achieve the goal in isolation (typically because of solution quality, deadline, etc)
- As a result, it needs to involve other agents.



# Announcement

- In this stage, the agent with the task sends out an announcement of the task which includes a specification of the task to be achieved.
- Specification must encode:
  - description of task itself (maybe executable)
  - any constraints (e.g., deadlines, quality constraints)
  - meta-task information (e.g., " . . . bids must be submitted by . . . ")
- The *announcement is then broadcast*.

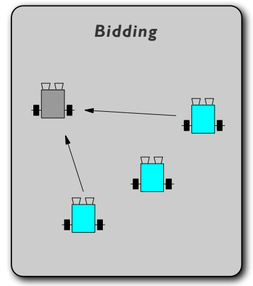






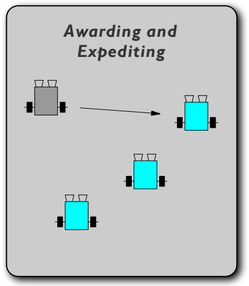

# Bidding

- Agents that receive the announcement decide for themselves whether they wish to bid for the task.
- Factors:
  - agent must decide whether it is capable of expediting task;
  - agent must determine quality constraints & price information (if relevant).
- If they do choose to bid, then they *submit a tender*.



# Awarding & Expediting

- Agent that sent task announcement must choose between bids & decide who to “award the contract” to.
  - The result of this process is *communicated to agents that submitted a bid*.
  - The successful *contractor* then expedites the task.
- May involve generating further manager-contractor relationships:
  - sub-contracting.
    - May involve another contract net.



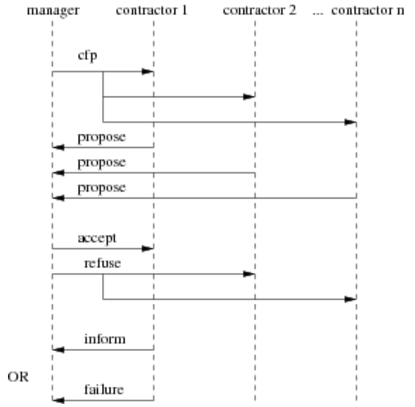







# The Contract Net via FIPA Performatives

- The FIPA ACL was designed to be able to capture the contract net.
  - cfp (call for proposals):
    - Used for announcing a task;
  - propose, refuse:
    - Used for making a proposal, or declining to make a proposal.
  - accept, reject:
    - Used to indicate acceptance or rejection of a proposal.
  - inform, failure:
    - Used to indicate completion of a task (with the result) or failure to do so.



# CNP in Jason: the MAS

- The Contract Net Protocol (CNP) in AgentSpeak / Jason
  - Six Agents
    - One Contractor who initiates the CNP
    - Three agents that fully participate in the protocol
    - One agent that always refuses
    - One agent that announces itself and then goes silent
  - This example also illustrates the mind inspector
    - A way to examine an agents beliefs etc.

```
1.MAS contractNetProtocol {
2.infrastructure: Centralised 3.
4.
5.    agents:
6.          contractor          // The CNP Initiator
7.                [mindinspector="gui(cycle,html,history)"];
8.          participant #3;     // The 3 service providers
9.          refusenik;          // Participant who always refuse
10.         silentpartner;      // A Participant that doesn't answer
11.
12.   aslSourcePath:
            "src/asl";
13. }
```






# CNP in Jason: silentpartner

- An agent that doesn't respond
  - Line 4: Initial belief that `contractor` is the initiator.
    - Line 8: A belief that `In` is the agent **contractor** generates a message to `In` introducing the agent.
      - Using the internal action `.my_name()`
      - A message is then sent to **contractor**
  - But at that point, nothing else is done
    - So, no response to any message

```
1.    // Agent silentpartner in project contractNetProtocol
2.
3.    // the name of the agent playing initiator in the CNP
4.    plays(initiator,contractor).
5.
6.    // send a message to the initiator introducing the
7.    // agent as a participant
8.    +plays(initiator,In)
9.          : .my_name(Me)
10.         <- .send(In,tell,introduction(participant,Me)).
11.
12.   // Nothing else
```

# CNP in Jason: refusenik

- An agent that says no
  - Line 4: Initial belief that contractor is the initiator.
    - Line 8: A belief that In is the agent **contractor** generates a message to In introducing the agent.
  - Line 13: A CfP message from an initiator agent will generate a refuse message

```
1.  // Agent refusenik in project contractNetProtocol
2.
3.  // the name of the agent playing initiator in the CNP
4.  plays(initiator,contractor).
5.
6.  // send a message to the initiator introducing the
7.  // agent as a participant
8.  +plays(initiator,In)
9.          : .my_name(Me)
10.         <- .send(In,tell,introduction(participant,Me)).
11.
12. // plan to answer a CFP
13. +cfp(CNPId,_Service)[source(A)]
14.         : plays(initiator,A)
15.         <- .send(A,tell,refuse(CNPId)).
```







# CNP in Jason: participant

- A participant agent
  - Lines 8-14: introduce the agent to the initiator
  - Line 5: rule that generates a random price for its service
- *Bidding* - Line 17: Plan @c1
  - On receipt of a cfp message from agent A (line 17)
    - Where A is the initiator, and where the agent can generate a price for the requested task
  - The agent keeps a mental note of its proposal (line 19)
  - Responds to CfP by making an offer (line 21)

```
1.  // Agent participant in project contractNetProtocol
2.
3.  // gets the price for the product,
4.  // a random value between 100 and 110.
5.  price(_Service,X) :- .random(R) & X =(10*R)+100.
6.
7.  // the name of the agent playing initiator in the CNP
8.  plays(initiator,contractor).
9.
10. // send a message to the initiator introducing the
11. // agent as a participant
12. +plays(initiator,In)
13.         : .my_name(Me)
14.         <- .send(In,tell,introduction(participant,Me)).
15.
16. // answer to Call For Proposal
17.@c1 +cfp(CNPId,Task)[source(A)]
18.:        plays(initiator,A) & price(Task,Offer)
19.         <-    // remember my proposal
20.               +proposal(CNPId,Task,Offer);
21.               .send(A,tell,propose(CNPId,Offer)).
```

# CNP in Jason: participant

- *Expediting* - Line 25: Plan @r1
  - Handling Accept messages
    - The agent responds to the addition of the belief accept_proposal()
      - The agent prints a success message for the contract, by retrieving the belief regarding the proposal
      - Note that there is nothing here to actually do the task.
- Line 32: Plan @r2
  - Handling Reject messages
    - The agent responds to the addition of the belief accept_proposal()
      - The agent prints a failure message and deletes the proposal from memory.

```
1.  // Agent participant in project contractNetProtocol
2.  ...

23. // Handling an Accept message
24. @r1 +accept_proposal(CNPId)
25.       : proposal(CNPId,Task,Offer)
26.       <- .print("My proposal '", Offer,"' won CNP ",
27.                                 CNPId, " for ", Task, "!").
28.       // do the task and report to initiator
29.
30. // Handling a Reject message
31. @r2 +reject_proposal(CNPId)
32.       <- .print("I lost CNP ",CNPId, ".");
33.       // clear memory
34.       -proposal(CNPId,_,_).
```





# CNP in Jason: contractor

- The contractor agent
  - The rule all_proposals checks that the number of the proposals received is equal to the number of introductions
    - The predicate will only be true for this equality
      - Note in the default run of the system with the **silentpartner** agent, this predicate will never be true!!!
    - The initial achievement goal, !startCNP(), is created:
      - with an Id of 1, and the task fix(computer)

```
1.  // Agent contractor in project contractNetProtocol
2.
3.  // Initial beliefs and rules
4.  all_proposals_received(CNPId)
5.        :-    .count(introduction(participant,_),NP) &
6.                    // number of participants
7.              .count(propose(CNPId,_)[source(_)], NO) &
8.                    // number of proposes received
9.              .count(refuse(CNPId)[source(_)], NR) &
10.                   // number of refusals received
11.             NP = NO + NR.
12.
13.// Initial goals
14.!startCNP(1,fix(computer)).
15.
16. //!startCNP(2,banana).
```

# Checking beliefs using the mind inspector

- The mind inspector can be used to check the internal state of the contract agent
  - In the example opposite:
    - the number of introductions (4) is equal to the number of proposals (3) and the number of refusals (1)
  - *Note that in this run, we removed the agent* **silentpartner** *from the agent community*
    - the other slides in this set assume that this agent does participate!!!

Inspection of agent **contractor**

- Beliefs

refuse(*1*)[source(refusenik)]
cnp_state(*1*,propose)[source(self)]
introduction(participant,participant1)[source(participant1)]
introduction(participant,participant2)[source(participant2)]
introduction(participant,participant3)[source(participant3)]
introduction(participant,refusenik)[source(refusenik)]
propose(*1,105.55351819793695*)[source(participant3)]
propose(*1,106.25903787244624*)[source(participant1)]
propose(*1,108.8734332473299*)[source(participant2)]







# CNP in Jason: contractor

- CNP *Announcement*
  - Plan for +!startCNP()
    - Line 21: wait for participants to introduce themselves
    - Line 23: track the current state of the protocol

Inspection of agent **contractor**

- Beliefs

cnp_state(*1*,propose)[source(self)]
introduction(participant,participant1)[source(participant1)]
introduction(participant,participant2)[source(participant2)]
introduction(participant,participant3)[source(participant3)]
introduction(participant,refusenik)[source(refusenik)]
introduction(participant,silentpartner)[source(silentpartner)]

+ Rules

- Events

| Sel | Trigger | Intention |
|---|---|---|
| | +cnp_state(*1*,propose)[source(self)] | 6 |

```
1.  // Agent contractor in project contractNetProtocol
2.  ...

18. // start the CNP
19. +!startCNP(Id,Task)
20.         <-  .print("Waiting participants for task ",Task,"...");
21.             .wait(2000);    // wait participants introduction
22.             // remember the state of the CNP
23.             +cnp_state(Id,propose);
24.             .findall(Name,introduction(participant,Name),LP);
25.             .print("Sending CFP to ",LP);
26.             .send(LP,tell,cfp(Id,Task));
27.             // the deadline of the CNP is now + 4 seconds
28.             // (or all proposals were received)
29.             .wait(all_proposals_received(CNPId), 4000, _);
30.             !contract(Id).
```

# CNP in Jason: contractor

- CNP *Announcement*
  - Plan for +!startCNP()
    - Line 21: wait for participants to introduce themselves
    - Line 23: track the current state of the protocol
    - Line 24: get a list of the agents that introduced themselves
      - Find all beliefs for the predicate introduction and unify the variable Name for each
      - Construct a list LP of all of the unified values of Name
    - Line 26: Send cfp messages to each agent in the list LP
    - …

```
1.  // Agent contractor in project contractNetProtocol
2.  …

18. // start the CNP
19. +!startCNP(Id,Task)
20.       <-    .print("Waiting participants for task ",Task,"...");
21.             .wait(2000);    // wait participants introduction
22.             // remember the state of the CNP
23.             +cnp_state(Id,propose);
24.             .findall(Name,introduction(participant,Name),LP);
25.             .print("Sending CFP to ",LP);
26.             .send(LP,tell,cfp(Id,Task));
27.             // the deadline of the CNP is now + 4 seconds
28.             // (or all proposals were received)
29.             .wait(all_proposals_received(CNPId), 4000, _);
30.             !contract(Id).
```







# CNP in Jason: contractor

Inspection of agent **contractor**

**- Beliefs**

cnp_state(*1*,propose)[source(self)]
introduction(participant,participant1)[source(participant1)]
introduction(participant,participant2)[source(participant2)]
introduction(participant,participant3)[source(participant3)]
introduction(participant,refusenik)[source(refusenik)]
introduction(participant,silentpartner)[source(silentpartner)]

**+ Rules**

**- MailBox**

<mid7,refusenik,tell,contractor,refuse(1)> in arch inbox.
<mid8,participant1,tell,contractor,propose(1,100.53373532376105)> in arch inbox.
<mid9,participant3,tell,contractor,propose(1,106.427017101855551)> in arch inbox.
<mid10,participant2,tell,contractor,propose(1,103.60717090658596)> in arch inbox.

**- Events**

| Sel | Trigger | Intention |
|---|---|---|
| X | +cnp_state(*1*,propose)[source(self)] | 6 |

**- Intentions**

| Sel | Id | Pen | Intended Means Stack (show details) |
|---|---|---|---|
| | 6 | **6/all_proposals_received(CNPId)** | +!startCNP(1,fix(computer))[source(self)] |

```
1.  // Agent contractor in project contractNetProtocol
2.  …

18. // start the CNP
19. +!startCNP(Id,Task)
20.       <-    .print("Waiting participants for task ",Task,"...");
21.             .wait(2000);    // wait participants introduction
22.             // remember the state of the CNP
23.             +cnp_state(Id,propose);
24.             .findall(Name,introduction(participant,Name),LP);
25.             .print("Sending CFP to ",LP);
26.             .send(LP,tell,cfp(Id,Task));
27.             // the deadline of the CNP is now + 4 seconds
28.             // (or all proposals were received)
29.             .wait(all_proposals_received(CNPId), 4000, _);
30.             !contract(Id).
```

# CNP in Jason: contractor

- CNP *Announcement*
  - Plan for +!startCNP()
    - …
    - Line 26: Send cfp messages to each agent in the list LP
    - Line 29: Wait until all of the proposals have been received, or we have a timeout of 4s
      - Note that the rule all_proposals_received() fails when the agent **slientpartner** is in the MAS
      - However, we recover by waiting for 4 seconds
    - Line 30: Create the achievement goal to award the contract for Id

```
1.  // Agent contractor in project contractNetProtocol
2.  …

18. // start the CNP
19. +!startCNP(Id,Task)
20.       <-    .print("Waiting participants for task ",Task,"...");
21.             .wait(2000);    // wait participants introduction
22.             // remember the state of the CNP
23.             +cnp_state(Id,propose);
24.             .findall(Name,introduction(participant,Name),LP);
25.             .print("Sending CFP to ",LP);
26.             .send(LP,tell,cfp(Id,Task));
27.             // the deadline of the CNP is now + 4 seconds
28.             // (or all proposals were received)
29.             .wait(all_proposals_received(CNPId), 4000, _);
30.             !contract(Id).
```






# CNP in Jason: contractor

- CNP *Awarding*
  - Plan for @lc1 +!contract()
    - Trigger only if we are in the propose state for the contract CNPId
    - Change the cnp_state to signify that we are awarding the contract (line 37)
    - Lines 38-44: Create a list L of offer(O,A) predicates and find the winner
      - Find all of the predicates propose() for the contact Id from each agent A, and extract the offer O from each
      - Ensure the list has at least one entry (line 41)
      - The winning offer is the one from L with the lowest offer WOf
    - Create the goal to announce the result (line 45)
    - Change the cnp_state to signify that we are finished (line 46)

```
1.  // Agent contractor in project contractNetProtocol
2.  …

32. // this plan needs to be atomic so as not to accept
33. // proposals or refusals while contracting
34. @lc1[atomic] +!contract(CNPId)
35.       : cnp_state(CNPId,propose)
36.       <-    -cnp_state(CNPId,_);
37.             +cnp_state(CNPId,contract);
38.             .findall(offer(O,A),propose(CNPId,O)[source(A)],L);
39.             .print("Offers are ",L);
40.             // constrain the plan execution to at least one offer
41.             L \== [];
42.             // sort offers, the first is the best
43.             .min(L,offer(WOf,WAg));
44.             .print("Winner is ",WAg," with ",WOf);
45.             !announce_result(CNPId,L,WAg);
46.             -+cnp_state(CNPId,finished).
```

# CNP in Jason: contractor

- CNP *Awarding*
  - Alternate Plan for +!contract()
    - An alternate plan exists if we are not in the right context; this does nothing (line 49)
  - Plan for -!contract()
    - If we delete the goal contract() then we know something failed, and thus a message is generated
    - This can occur if there were no viable contracts proposed (i.e. if the constraint on line 41 was violated)

```
1.  // Agent contractor in project contractNetProtocol
2.  …

39. …
40.             // constrain the plan execution to at least one offer
41.             L \== [];
42. …

48.// nothing todo, the current phase is not 'propose'
49.@lc2 +!contract(_).
50.
51. -!contract(CNPId)
52.         <-    .print("CNP",CNPId," has failed!").
```




29

# CNP in Jason: contractor

- CNP *Awarding*
  - The awarding process is recursive
    - The goal was created on line 45 of the plan @lc1
    - If the head of the list L is the winner WAg, then the plan on line 58 is satisfied
      - An accept_proposal belief is sent to the winner
      - The goal announce_result is then called on the tail of the list of agents L
    - If the head of the list L is *not* the winner WAg, then the plan on line 63 is satisfied
      - A reject_proposal belief is sent to the agent
      - Again, the goal announce_result is then called on the remaining agents (the tail of L)
  - To terminate the recursion
    - Line 55 triggers with a call on an empty list

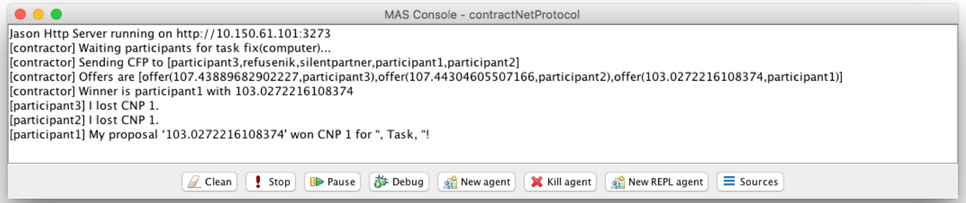
```
1.  // Agent contractor in project contractNetProtocol
2.  …

44. …
45.             !announce_result(CNPId,L,WAg);
46. …

54. // Terminate the recursion when we have no more
55. // agents participating in the CFP
56. +!announce_result(_,[],_).
57.
58. // announce to the winner
59. +!announce_result(CNPId,[offer(_,WAg)|T],WAg)
60.         <-    .send(WAg,tell,accept_proposal(CNPId));
61.               !announce_result(CNPId,T,WAg).
62.
63. // announce to others
64. +!announce_result(CNPId,[offer(_,LAg)|T],WAg)
65.         <-    .send(LAg,tell,reject_proposal(CNPId));
66.               !announce_result(CNPId,T,WAg).
```




30

# CNP in Jason

- Here is the trace of the agents
  - Note that each agent's output is preceded by the agent name.










# Issues for Implementing Contract Net

- How to…
  - ... specify *tasks*?
  - ... specify *quality of service*?
  - ... decide how to *bid*?
  - ... select between competing offers?
  - … differentiate between offers based on multiple criteria?

# Deciding how to bid

- At some time $t$ a contractor $i$ is scheduled to carry out $\tau^t_i$.
  - Contractor $i$ also has resources $e_i$.
  - Then $i$ receives an announcement of task specification $ts$, which is for a set of tasks $\tau(ts)$.
  - The cost to $i$ to carry these out is: $c^t_i(\tau)$
- The *marginal cost* of carrying out $\tau$ will be:

$$\mu_i(\tau\ (ts) \mid \tau^t_i) = c_i(\tau\ (ts) \cup \tau^t_i) - c_i(\tau^t_i)$$

  - that is the difference between carrying out what it has already agreed to do and what it has already agreed plus the new tasks.





# Deciding how to bid

- Due to synergies, this is often not just $c^t_i(\tau(ts))$
  - in fact, it can be zero — the additional tasks can be done for free.
- Think of the cost of giving another person a ride to work.
  - As long as $\mu_i(\tau(ts) \mid \tau^t_i) < e$ then the agent can afford to do the new work, then it is rational for the agent to bid for the work.
  - Otherwise not.
- You can extend the analysis to the case where the agent gets paid for completing a task.
  - And for considering the duration of tasks.

# Results Sharing

- In results sharing, agents provide each other with information as they work towards a solution.
- It is generally accepted that results sharing improves problem solving by:
  - Independent pieces of a solution can be cross-checked.
  - Combining local views can achieve a better overall view.
  - Shared results can improve the accuracy of results.
  - Sharing results allows the use of parallel resources on a problem.
- The following are examples of results sharing.





35

# Results Sharing in Blackboard Systems

- The first scheme for cooperative problem solving: was the blackboard system.
  - Results shared via shared data structure (BB).
  - Multiple agents (KSs/KAs) can read and write to BB.
  - Agents write partial solutions to BB.
- Blackboards may be structured as a hierarchy.
  - Mutual exclusion over BB required ⇒ bottleneck.
  - Not concurrent activity.
- Compare:
  - *LINDA* tuple spaces, *JAVASPACES*.

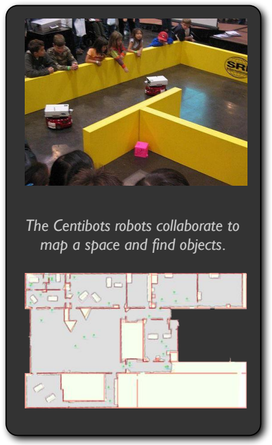




36

# Result Sharing in Subscribe/Notify Pattern

- Common design pattern in OO systems: subscribe/notify.
  - An object *subscribes* to another object, saying "*tell me when event $e$ happens*".
  - When event $e$ happens, original object is notified.
- Information pro-actively *shared* between objects.
- Objects required to know about the *interests* of other objects ⇒ inform objects when relevant information arises.










# Handling Inconsistency

- A group of agents may have inconsistencies in their:
  - Beliefs
  - Goals or intentions
- Inconsistent beliefs arise because agents have different views of the world.
  - May be due to sensor faults or noise or just because they can't see everything.
- Inconsistent goals may arise because agents are built by different people with different objectives.

# Handling Inconsistency

- Three ways to handle inconsistency (Durfee at al.)
  - Do not allow it!
    - For example, in the contract net the only view that matters is that of the manager agent.
  - Resolve inconsistency
    - Agents discuss the inconsistent information/goals until the inconsistency goes away.
      - We will discuss this later (*argumentation*).
- Build systems that degrade gracefully in the face of inconsistency.



39

# Coordination

- Coordination is managing dependencies between agents.
  - Any thoughts in resolving the following?

1. We both want to leave the room through the same door. We are walking such that we will arrive at the door at the same time. What do we do to ensure we can both get through the door?

2. We both arrive at the copy room with a stack of paper to photocopy. Who gets to use the machine first?



40

# Coordination

- Von Martial suggested that *positive* coordination is:
  - Requested (explicit)
  - Non-requested (implicit)
- Non-requested coordination relationships can be as follows.
  - ***Action equality***:
    - We both plan to do something, and by recognising this one of us can be saved the effort.
  - ***Consequence***:
    - What I plan to do will have the side-effect of achieving something you want to do.
  - ***Favor***:
    - What I plan to do will make it easier for you to do what you want to do.
- Now let's look at some approaches to coordination.





# Social Norms

- Societies are often regulated by (often unwritten) rules of behaviour.
- Example:
  - A group of people is waiting at the bus stop. The bus arrives. Who gets on the bus first?
  - Another example:
    - On 34th Street, which side of the sidewalk do you walk along?
- In an agent system, we can design the norms and program agents to follow them, or let norms evolve.

# Offline Design

- Recall how we described agents before:
  - As a function which, given a run ending in a state, gives us an action.

$Ag: R^E \rightarrow Ac$

- A *constraint* is then a pair:
  - where $E' \subseteq E$ is a set of states, and $\alpha \in Ac$ is an action.
  - This constraint says that $\alpha$ cannot be done in any state in $E'$.

$<E', \alpha>$

- A *social law* is then *a set of these constraints*.







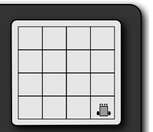
# Offline Design

- We can refine our view of an environment.
  - *Focal* states, $F \subseteq E$ are the states we want our agent to be able to get to.
  - From any focal state $e \in F$ it should be possible to get to any other focal state $e' \in F$ (though not necessarily right away).

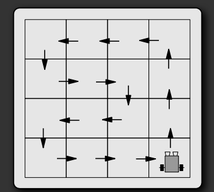
- A useful *social* law is then one that does not prevent agents from getting from one focal state to another.

***A useful social law that prevents collisions***

*(Wooldridge p177, from Shoham and Tennenholtz):*



1. On even rows the robots move left while in odd rows the robots move right.
2. Robots move up when in the rightmost column.
3. Robots move down when in the leftmost column of even rows or the second rightmost column of odd rows.



***Not necessarily efficient (On² steps to get to a specific square).***

# Emergence

- We can also design systems in which social laws emerge.

> "… Agents have both a red t-shirt and a blue t-shirt and wear one. Goal is for everyone to end up with the same color on. In each round, each agent meets one other agent, and decides whether or not to change their shirt. During the round they only see the shirt their pair is wearing — they don't get any other information…"
>
> T-shirt Game (Shoham and Tennenholtz)

- What strategy update function should they use?



Simple majority:

Agents pick the shirt they have seen the most.

Simple majority with types:

Agents come in two types. When they meet an agent of the same type, agents pass their memories. Otherwise they act as simple majority.

Highest cumulative reward:

Agents can "see" how often other agents (some subset of all the agents) have matched their pair. They pick the shirt with the largest number of matches.





# Joint Intentions

- Just as we have individual intentions, we can have joint intentions for a team of agents.
- Levesque defined the idea of *a joint persistent goal* (JPG).
  - A group of agents have a collective commitment to bring about some goal $\varphi$, "move the couch".
  - Also have motivation $\psi$, "Simon wants the couch moved".
- The mental states of agents mirror those in BDI agents.
  - Agents don't believe that $\varphi$ is satisfied, but believe it is possible.
  - Agents maintain the goal $\varphi$ until a termination condition is reached.

# Joint Intentions

- The terminations condition is that it is mutually believed that:
  - goal $\varphi$ is satisfied; or
  - goal $\varphi$ is impossible; or
  - the motivation $\psi$ is no longer present
- The termination condition is achieved when an agent realises that, the goal is satisfied, impossible and so on.
- But it doesn't drop the goal right away.
  - Instead it adopts a new goal — to make this new knowledge mutually believed.
  - This ensures that the agents are coordinated.
- They don't stop working towards the goal until they are all appraised of the situation.
  - Mutual belief is achieved by communication.








# Multiagent Planning

- Another approach to coordinate is to explicitly plan what all the agents do.
  - For example, come up with a large STRIPS plan for all the agents in a system.
- Could have:
  - ***Centralised*** planning for distributed plans.
    - One agent comes up with a plan for everybody
  - ***Distributed*** planning
    - A group of agents come up with a centralised plan for another group of agents.
  - ***Distributed planning for distributed plans***
    - Agents build up plans for themselves, but take into account the actions of others.

# Multiagent Planning

- In general, the more decentralized it is, the harder it is.
- Georgeff propsed a distributed version of STRIPS.
  - New list: *during*
  - Specifies what must be true while the action is carried out.
  - This places constraints on when other agents can do things.
- Different agents plan to achieve their goals using these operators and then do:
  - ***Interaction analysis***: do different plans affect one another?
  - ***Safety analysis***: which interactions are problematic?
  - ***Interaction resolution***: treat the problematic interactions as *critical sections* and enforce mutual exclusion.



49

# Summary

- This lecture has discussed how to get agents working together to do things.
  - Key assumption: *benevolence*
  - Agents are working together, *not in competition*.
- We discussed a number of ways of having agents decide what to do, and make sure that their work is coordinated.
  - A typical system will need to use a combination of these ideas.
- Next time, we will go on to look at agents being in *competition* with one another.

*Class Reading (Chapter 8):*

*"Distributed Problem Solving and Planning", E.H. Durfee. In Weiss, G. ed.: Multiagent Systems,1999, pp121-164.*

This is a detailed and precise introduction to distributed problem solving and distributed planning, with many useful pointers into the literature.



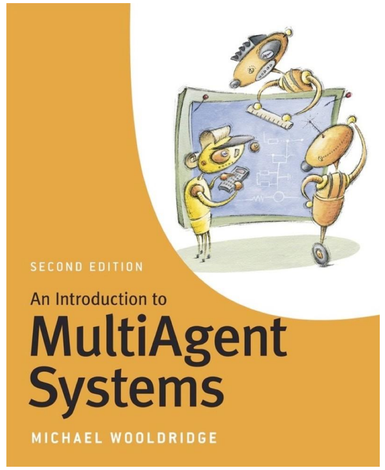
50

# IT4899
# Multi-Agent Systems

Chapter 11 - Multi-Agent Interactions

Dr. Nguyen Binh Minh
Department of Information Systems

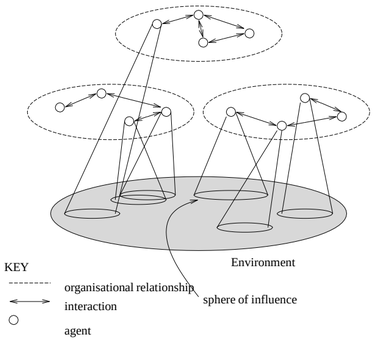







# What are Multi-Agent Systems?

- A multiagent system contains a number of agents that:
  - interact through communication;
  - are able to act in an environment;
  - have different “spheres of influence” (which may coincide); and
  - will be linked by other (organisational) relationships.
- We will look at how agents decide how to interact in *competitive* situations.



# Utilities and Preferences

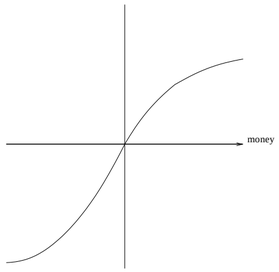
- Our Assumptions:
  - Assume we have just two agents: $Ag = \{i, j\}$
  - Agents are assumed to be *self-interested* i.e. they have preferences over how the environment is.
  - Assume $\Omega = \{\omega_1, \omega_2, \ldots\}$ is the set of "outcomes" that agents have preferences over.
- We capture preferences by *utility functions*, represented as real numbers ($\mathbb{R}$):

$$u_i : \Omega \rightarrow \mathbb{R}$$
$$u_j : \Omega \rightarrow \mathbb{R}$$

- Utility functions lead to *preference orderings* over outcomes, e.g.:

$\omega \succcurlyeq_i \omega'$ means $u_i(\omega) \geq u_i(\omega')$
$\omega \succ_i \omega'$ means $u_i(\omega) > u_i(\omega')$

- where $\omega$ and $\omega$' are both possible outcomes of $\Omega$




**Utility is not money. Just a way to encode preferences.**







# Multiagent Encounters

- We need a model of the environment in which these agents will act...
  - agents simultaneously choose an action to perform, and as a result of the actions they select, an outcome in $\Omega$ will result
  - the *actual* outcome depends on the *combination* of actions
  - assume each agent has just two possible actions that it can perform:
  - i.e. $Ac = \{C,D\}$, where
    - $C$ ("*cooperate*") and
    - $D$ ("*defect*")
- Environment behaviour given by *state transformer function* $\tau$
  - (introduced in Chapter 2):

$$\tau : \underbrace{Ac}_{\text{agent } i\text{'s action}} \times \underbrace{Ac}_{\text{agent } j\text{'s action}} \rightarrow \Omega$$

# Multiagent Encounters

- Here is a state transformer function $\tau(i,j)$
  - This environment is sensitive to actions of both agents.

$\tau(D,D) = \omega_1 \quad \tau(D,C) = \omega_2$
$\tau(C,D) = \omega_3 \quad \tau(C,C) = \omega_4$

- With this state transformer, neither agent has any influence in this environment.

$\tau(D,D) = \omega_1 \quad \tau(D,C) = \omega_1$
$\tau(C,D) = \omega_1 \quad \tau(C,C) = \omega_1$

- With this one, the environment is controlled by $j$

$\tau(D,D) = \omega_1 \quad \tau(D,C) = \omega_2$
$\tau(C,D) = \omega_1 \quad \tau(C,C) = \omega_2$







# Rational Action

- Suppose we have the case where both agents can influence the outcome, and they have the following utility functions:

$u_i(\omega_1)=1 \quad u_i(\omega_2)=1 \quad u_i(\omega_3)=4 \quad u_i(\omega_4)=4$
$u_j(\omega_1)=1 \quad u_j(\omega_2)=4 \quad u_j(\omega_3)=1 \quad u_j(\omega_4)=4$

  - With a bit of abuse of notation:

$u_i(D,D) = 1 \quad u_i(D,C) = 1 \quad u_i(C,D) = 4 \quad u_i(C,C) = 4$
$u_j(D,D) = 1 \quad u_j(D,C) = 4 \quad u_j(C,D) = 1 \quad u_j(C,C) = 4$

  - Then agent $i$'s preferences are $(C, C) \succcurlyeq_i (C, D) \succ_i (D, C) \succcurlyeq_i (D, D)$
  - In this case, what should $i$ do?

- $i$ prefers all outcomes that arise through $C$ over all outcomes that arise through $D$.
  - Thus $C$ is the *rational choice* for $i$.

# Payoff Matrices

- We can characterise the previous scenario in a *payoff matrix* shown opposite
  - Agent $i$ is the *column player* and gets the *upper reward* in a cell.
  - Agent $j$ is the *row player* and gets the *lower reward* in a cell.
- Actually there are two matrices here, one (call it $A$) that specifies the payoff to $i$ and another $B$ that specifies the payoff to $j$.
- Sometimes we'll write the game as $(A, B)$ in recognition of this.

In this case, $i$ cooperates and gains a utility of 4; whereas $j$ defects and gains a utility of only 1.

| | | $i$: defect | $i$: coop |
|---|---|---|---|
| $j$ | defect | 1, 1 | 4, 1 |
| | coop | 1, 4 | 4, 4 |

$$(C, C) \succcurlyeq_i (C, D) \succ_i (D, C) \succcurlyeq_i (D, D)$$





# Solution Concepts

- How will a rational agent will behave in any given scenario?
- Play. . .
  - dominant strategy;
  - Nash equilibrium strategy;
  - Pareto optimal strategies;
  - strategies that maximise social welfare.

# Dominant Strategies

- Given any particular strategy $s$ (either $C$ or $D$) that agent $i$ can play, there will be a number of possible outcomes.
  - We say $s_1$ dominates $s_2$ if every outcome possible by $i$ playing $s_1$ is preferred over every outcome possible by $i$ playing $s_2$.

| | | $i$ | |
|---|---|---|---|
| | | defect | coop |
| $j$ | defect | 1<br>1 | 4<br>1 |
| | coop | 1<br>4 | 4<br>4 |

- Thus in the game opposite, $C$ dominates $D$ for both players.







# Dominant Strategies

- A rational agent will never play a *dominated strategy*.
  - i.e, a strategy that is dominated (and thus inferior) by another
- So in deciding what to do, we can delete dominated strategies.
  - Unfortunately, there isn't always a unique un-dominated strategy.

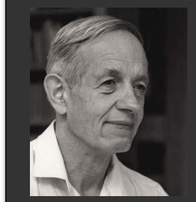
| | | $i$ | |
|---|---|---|---|
| | | defect | coop |
| $j$ | defect | 1<br>1 | 4<br>1 |
| | coop | 1<br>4 | 4<br>4 |

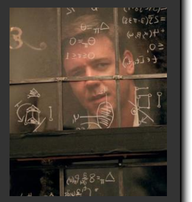
# Nash Equilibrium

- In general, we will say that two strategies $s_1$ and $s_2$ are in Nash equilibrium (NE) if:
  - under the assumption that agent *i* plays $s_1$, agent *j* can do no better than play $s_2$;
    - I.e. *if **I** drive on the left side of the road, **you** can do no better than also driving on the left!*
  - under the assumption that agent j plays $s_2$, agent *i* can do no better than play $s_1$.
    - I.e. *if **you** drive on the left side of the road, **I** can do no better than also driving on the left!*
- Neither agent has any incentive to deviate from a *Nash Equilibrium (NE)*.

***John Forbes Nash***

*(Nobel Laureate in Economics)*





*Portrayed by Russel Crowe in the film "A Beautiful Mind"*







# Nash Equilibrium

- Consider the payoff matrix opposite:
  - Here the *Nash equilibrium (NE)* is *(D, D)*.
  - In a game like this you can find the NE by cycling through the outcomes, asking if either agent can improve its payoff by switching its strategy.
- Thus, for example, *(C, D)* is not a NE because *i* can switch its payoff from 1 to 5 by switching from *C* to *D*.

| | | *i*: defect | *i*: coop |
|---|---|---|---|
| *j* | defect | 5<br>3 | 1<br>2 |
| | coop | 0<br>2 | 0<br>1 |

| | | *i*: defect | *i*: coop |
|---|---|---|---|
| *j* | defect | 5<br>3 | 1<br>2 |
| | coop | 0<br>2 | 0<br>1 |

# Nash Equilibrium

- More formally:
  - A strategy $(i^*, j^*)$ is a *pure strategy Nash Equilibrium* solution to the game $(A, B)$ if:

$$\forall i,\ a_{i^*,j^*} \geq a_{i,j^*}$$
$$\forall j,\ b_{i^*,j^*} \geq b_{i^*,j}$$

- Unfortunately:
  - Not every interaction scenario has a *pure strategy* Nash Equilibrium (NE).
  - Some interaction scenarios have *more than one pure strategy* Nash Equilibrium (NE).





# Nash Equilibrium

- The game opposite (upper) has two pure strategy NEs, $(C, C)$ and $(D, D)$
  - In both cases, a single agent can't unilaterally improve its payoff.

| | | $i$: defect | $i$: coop |
|---|---|---|---|
| $j$ | defect | 5<br>3 | 1<br>2 |
| | coop | 0<br>2 | 3<br>3 |

- In the game opposite game (lower) has no pure strategy NE
  - For every outcome, one of the agents will improve its utility by switching its strategy.
  - We can find a form of NE in such games, but we need to go beyond pure strategies.

| | | $i$: defect | $i$: coop |
|---|---|---|---|
| $j$ | defect | 2<br>1 | 1<br>2 |
| | coop | 0<br>2 | 1<br>1 |

# Mixed Strategy Nash equilibrium

- Matching Pennies
  - Players $i$ and $j$ simultaneously choose the face of a coin, either "heads" or "tails".
  - If they show the same face, then $i$ wins, while if they show different faces, then $j$ wins.
- NO pair of strategies forms a pure strategy NE:
  - whatever pair of strategies is chosen, somebody will wish they had done something else.
- The solution is to allow *mixed strategies*:
  - play "heads" with probability 0.5
  - play "tails" with probability 0.5.
- This is a Mixed Nash Equilibrium strategy.

| | | $i$: heads | $i$: tails |
|---|---|---|---|
| $j$ | heads | 1, -1 | -1, 1 |
| | tails | -1, 1 | 1, -1 |








# Mixed Strategy Nash equilibrium

- Consider the Game Rock/Paper/Scissors
  - Paper covers rock
  - Scissors cut paper
  - Rock blunts scissors
- This has the following payoff matrix

| | | $i$: rock | $i$: paper | $i$: scissors |
|---|---|---|---|---|
| $j$ | rock | 0, 0 | 1, 0 | 0, 1 |
| | paper | 0, 1 | 0, 0 | 1, 0 |
| | scissors | 1, 0 | 0, 1 | 0, 0 |

- What should you do?
  - *Choose a strategy at random!*

# Mixed Strategies

- A mixed strategy has the form
  - play $\alpha_1$ with probability $p_1$
  - play $\alpha_2$ with probability $p_2$
  - ...
  - play $\alpha_k$ with probability $p_k$.
  - such that $p_1+p_2+\cdots+p_k=1$.
- Nash proved that:
  - *every finite game has a Nash equilibrium in mixed strategies.*

***Nash's Theorem***

*Nash proved that* ***every finite game has a Nash equilibrium in mixed strategies****. (Unlike the case for pure strategies.)*

*So this result overcomes the lack of solutions; but there still may be more than one Nash equilibrium. . .*







# Pareto Optimality

- An outcome is said to be *Pareto optimal* (or *Pareto efficient*) if:
  - there is no other outcome that makes one agent *better off* without making another *agent worse* off.
    - If an outcome is Pareto optimal, then at least one agent will be reluctant to move away from it (because this agent will be worse off).
    - If an outcome $\omega$ is not Pareto optimal, then there is another outcome $\omega'$ that makes everyone as happy, if not happier, than $\omega$.
- "Reasonable" agents would agree to move to $\omega'$ in this case.
  - *Even if I don't directly benefit from $\omega'$, you can benefit without me suffering.*

This game has one Pareto efficient outcome: *(D, D)*

| | | *i*: defect | *i*: coop |
|---|---|---|---|
| *j* | defect | 5, 5 | 1, 3 |
| *j* | coop | 3, 1 | 2, 2 |

There is no solution in which either agent does better

# Social Welfare

- The social welfare of an outcome $\omega$ is the sum of the utilities that each agent gets from $\omega$:

$$\sum_{i \in Ag} u_i(\omega)$$

  - Think of it as the "total amount of money in the system".

- As a solution concept:
  - may be appropriate when the whole system (all agents) has a single owner (then overall benefit of the system is important, not individuals).
  - It doesn't consider the benefits to individuals.
  - A very skewed outcome can maximise social welfare.

In both these games, *(C, C)* maximises social welfare

| | | *i* defect | *i* coop |
|---|---|---|---|
| *j* | defect | 2, 2 | 1, 1 |
| | coop | 3, 3 | 4, 4 |

| | | *i* defect | *i* coop |
|---|---|---|---|
| *j* | defect | 2, 2 | 1, 1 |
| | coop | 3, 3 | 7, 0 |







# Competitive and Zero-Sum Interactions

- Where preferences of agents are diametrically opposed we have *strictly competitive* scenarios.

- Zero-sum encounters are those where utilities sum to zero:

$$u_i(\omega) + u_j(\omega) = 0 \qquad \text{for all } \omega \in \Omega.$$

  - Zero sum encounters are bad news: for me to get + utility you have to get negative utility! *The best outcome for me is the worst for you*!
  - *Zero sum encounters in real life are very rare* . . . but people tend to act in many scenarios as if they were zero sum.

- Most games have some room in the set of outcomes for agents to find (somewhat) mutually beneficial outcomes.

# The Prisoner's Dilemma

- Payoff matrix for prisoner's dilemma:

| | | *i* defect | *i* coop |
|---|---|---|---|
| *j* | defect | 2<br>2 | 1<br>4 |
| | coop | 4<br>1 | 3<br>3 |

As neither men want to admit to being guilty, **cooperation** means **not** confessing!

**The Prisoner's Dilemma**

Two men are collectively charged with a crime and held in separate cells, with no way of meeting or communicating.

They are told that:

- *if one confesses and the other does not (C,D) or (D,C), the confessor will be freed, and the other will be jailed for three years;*
- *if both confess (D,D), then each will be jailed for two years.*

Both prisoners know that if neither confesses (C,C), then they will each be jailed for one year.

- *Top left*: If both defect, then both get punishment for mutual defection.
- *Top right*: If i cooperates and j defects, i gets sucker's payoff of 1, while j gets 4.
- *Bottom left*: If j cooperates and i defects, j gets sucker's payoff of 1, while i gets 4.
- *Bottom right*: Reward for mutual cooperation (i.e. neither confess).







# What should you do?

- The *individual rational action is defect*.
  - This guarantees a payoff of no worse than 2, whereas cooperating guarantees a payoff of at most 1.
  - So defection is the best response to all possible strategies: both agents defect, and get payoff = 2.
- But *intuition* says this is *not* the best outcome:
  - Surely they should both cooperate and each get payoff of 3!
- This is why the Prisoners Dilemma game is interesting
  - The analysis seems to give us a *paradoxical* answer.

**Solution Concepts**

- The dominant strategy here is to defect.
- $(D, D)$ is the only Nash equilibrium.
- All outcomes **except** $(D, D)$ are Pareto optimal.
- $(C, C)$ maximises social welfare.

| | | *i* defect | *i* coop |
|---|---|---|---|
| *j* | defect | 2<br>2 | 1<br>4 |
| | coop | 4<br>1 | 3<br>3 |

# The Prisoner's Dilemma

- This apparent paradox is the fundamental problem of multi-agent interactions.
  - It appears to imply that cooperation will not occur in societies of self-interested agents.
- Real world examples:
  - nuclear arms reduction -"why don't I keep mine"
  - free rider systems - public transport, file sharing;
  - in the UK — television licenses.
  - in the US — funding for NPR/PBS.
- The prisoner's dilemma is ubiquitous.
  - Can we recover cooperation?

**Solution Concepts**

- The dominant strategy here is to defect.
- $(D, D)$ is the only Nash equilibrium.
- All outcomes **except** $(D, D)$ are Pareto optimal.
- $(C, C)$ maximises social welfare.

| | | $i$ = defect | $i$ = coop |
|---|---|---|---|
| $j$ | defect | 2, 2 | 1, 4 |
| | coop | 4, 1 | 3, 3 |







# Arguments for Recovering Cooperation

- Conclusions that some have drawn from this analysis:
  - the game theory notion of rational action is wrong!
  - somehow the dilemma is being formulated wrongly
- Arguments to recover cooperation:
  - Altruism
  - The other prisoner is my twin!
  - Program equilibria and mediators
  - The shadow of the future. . .

# We are not all Machiavelli

- Lots of “altruism” is something else:
  - Either there is some delayed reciprocity; or
  - There are mechanisms to punish defection.
- There is a reason why HMRC (or the IRS in the US) audits people’s taxes :-)
- Altruism may be something that makes us feel good
  - This is why we are prepared to pay for it.

“… We aren’t all that hard-boiled, and besides, people really do act altruistically …”





25

# The Other Prisoner is My Twin

- Argue that both prisoner’s will think alike and decide that it is best to cooperate.
  - If they are twins, they must think along the same lines, right?
  - (Or they have some agreement that they won’t talk.)
- Well, if this is the case, we aren’t really playing the Prisoner’s Dilemma!
- Possibly more to the point is that if you know the other person is going to cooperate, you are *still* better off defecting.

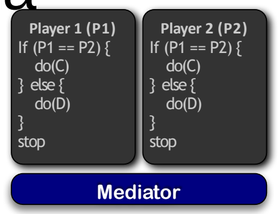




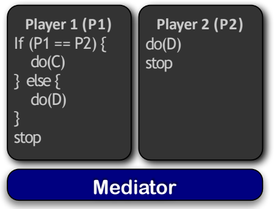
26

# Program Equilibria

- The strategy you really want to play in the prisoner's dilemma is: *I'll cooperate if he will*
  - Program equilibria provide one way of enabling this.
- Each agent submits a *program strategy* to a *mediator* which *jointly executes* the strategies.
  - Crucially, strategies can be *conditioned on the strategies of the others*.
  - The best response to this program:
    - *submit the same program*, giving an outcome of *(C, C)*!




P1:C P2:C




P1:D P2:D







# The Iterated Prisoner's Dilemma (The Shadow of the Future)

- Play the game more than once.
  - If you know you will be meeting your opponent again, then *the incentive to defect* appears to evaporate.
    - If you defect, you can be *punished* (compared to the co-operation reward.)
    - If you get suckered, then *what you lose can be amortised over the rest of the iterations*, making it a small loss.
- Cooperation is (provably) *the rational choice* in the infinitely repeated prisoner's dilemma.
  - (Hurrah!)
- But what if there are a finite number of repetitions?

# Backwards Induction

- But. . . suppose you both know that you will play the game exactly n times.
  - On round n − 1, you have an incentive to defect, to gain that extra bit of payoff.
  - But this makes round n − 2 the last “real”, and so you have an incentive to defect there, too.
  - This is the *backwards induction problem*.
- Playing the prisoner’s dilemma with a fixed, finite, pre-determined, commonly known number of rounds, defection is the best strategy.
  - That seems to suggest that you should never cooperate.
  - So how does cooperation arise? Why does it make sense?

As long as you have some probability of repeating the interaction co-operation can have a better expected payoff.

As long as there are enough co-operative folk out there, you can come out ahead by co-operating.





29

# Axelrod’s Tournament

- Suppose you play iterated prisoner’s dilemma (IPD) against a *range* of opponents.
- What approach should you choose, so as to maximise your overall payoff?
  - *Is it better to defect*, and hope to find suckers to rip-off?
  - Or *is it better to cooperate*, and try to find other friendly folk to cooperate with?

**Robert Axelrod**

Robert Axelrod (1984) investigated this problem, with a computer tournament for programs playing the iterated prisoner’s dilemma.

Axelrod hosted the tournament and various researchers sent in approaches for playing the game.





30

# Strategies in Axelrod's Tournament

- Surprisingly TIT-FOR-TAT for won.
  - But don't read too much into this :-)
- In scenarios like the Iterated Prisoner's Dilemma (IPD) tournament…
  - …the best approach depends heavily on *what the full set of approaches is*.
- TIT-FOR-TAT did well because there were other players it could co-operate with.

**JOSS**

As TIT-FOR-TAT, except periodically defect.

**ALL-D**

"Always defect" — the hawk strategy;

**Tit-For-Tat**

1. On round u = 0, cooperate.
2. On round u > 0, do what your opponent did on round u − 1.

**Tester**

On 1st round, defect. If the opponent retaliated, then play TIT-FOR-TAT. Otherwise intersperse cooperation & defection.







# Recipes for Success in Axelrod's Tournament

- Don't be envious:
  - Don't play as if it were zero sum!
- Be nice:
  - Start by cooperating, and reciprocate cooperation.
- Retaliate appropriately:
  - Always punish defection immediately, but use "measured" force
  - don't overdo it.
- Don't hold grudges:
  - Always reciprocate cooperation immediately.

# Stag Hunt

- A group of hunters goes stag hunting.
  - If *they all stay focussed on the stag*, they will catch it and all have a lot of food.
  - If *some of them head off to catch rabbits*, the stag will escape.
- In this case the rabbit hunters will have some small amount of food and the (remaining) stag hunters will go hungry.
  - What should each hunter do?

*"...You and a friend decide it would be a great joke to show up on the last day of school with some ridiculous haircut. Egged on by your clique, you both **swear** you'll get the haircut.*

*A night of indecision follows. As you anticipate your parents' and teachers reactions [...] you start wondering if your friend is really going to go through with the plan.*

*Not that you don't want the plan to succeed: the best possible outcome would be for both of you to get the haircut.*

*The trouble is, it would be awful to be the **only one** to show up with the haircut. That would be the worst possible outcome.*

*You're not above enjoying your friend's embarrassment. If you **didn't** get the haircut, but the friend did, and looked like a real jerk, that would be almost as good as if you both got the haircut..."*

*Mike Wooldridge*







# Stag Hunt

**Stag Hunt Payoff Matrix**

| | | $i$: defect | $i$: coop |
|---|---|---|---|
| $j$: defect | | 2, 2 | 3, 1 |
| $j$: coop | | 1, 3 | 4, 4 |

The difference from the prisoner's dilemma is that now it is better if you both co-operate than if you defect while the other co-operates.

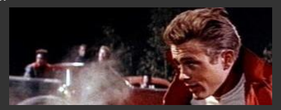


# Stag Hunt

- Two Nash equilibrium solutions $(C, C)$ and $(D, D)$.
  - If you know I'll co-operate, the best you can do is to co-operate as well.
  - If you know I'll defect, then that is the best you can do as well.
  - Social welfare is maximised by $(C, C)$.
  - The only Pareto efficient outcome is $(C, C)$.
- As usual with Nash equilibrium, theory gives us no real help in deciding what the other party will do.
  - Hence the worrying about the haircut.
- The same scenario occurs in mutinies and strikes.
  - We would all be better off if our hated captain is deposed, but if some of us give in, we will all be hanged.

# Game of Chicken

The **game of chicken** gets its name from a rather silly, macho "game" that was supposedly popular amongst juvenile delinquents in 1950s America; the game was immortalised by James Dean in the 1950s film Rebel without a Cause. The purpose of the game is to establish who is **bravest** of the two players.



The game is played by both players driving their cars at high speed towards a cliff. The idea is that the **least brave** of the two (the "chicken") will be the first to drop out of the game by jumping out of the speeding car. The **winner** is the one who lasts longest in the car. Of course, if neither player jumps out of the car, then **both cars fly off the cliff**, taking their foolish passengers to a fiery death on the rocks that undoubtedly lie at the foot of the cliff.

- Chicken has the following payoff matrix:

jump = coop
stay in car = defect.

| | | $i$: defect | $i$: coop |
|---|---|---|---|
| $j$: defect | | 1<br>1 | 2<br>4 |
| $j$: coop | | 4<br>2 | 3<br>3 |

- Difference to prisoner's dilemma:
  - *Mutual defection is most feared outcome*.
  - Whereas sucker's payoff is most feared in prisoner's dilemma.
- There is no dominant strategy
  - Strategy pairs $(C, D)$ and $(D, C)$ are Nash equilibria.
    - If I think you will stay in the car, I should jump out.
    - If I think you will jump out of the car, I should stay in.
  - All outcomes except $(D, D)$ are Pareto optimal.
  - All outcomes except $(D, D)$ maximise social welfare.






# Other Symmetric 2x2 Games

- Given the 4 possible outcomes of (symmetric) cooperate/defect games, there are 24 possible orderings on outcomes.
  - The textbook lists them all, but here we give the combinations
- These are more abstract descriptions of the games than the payoff matrices we considered.
- All payoff matrices consistent with these preferences orders are instances of the games.

***First, cases with dominant solutions:***

*Cooperation dominates*

$CC \succ_i CD \succ_i DC \succ_i DD$

$CC \succ_i CD \succ_i DD \succ_i DC$

*Deadlock (You will always do best by defecting)*

$DC \succ_i DD \succ_i CC \succ_i CD$

$DC \succ_i DD \succ_i CD \succ_i DD$

***Games that we looked at in detail:***

*Prisoner's dilemma.*

$DC \succ_i CC \succ_i DD \succ_i CD$

*Chicken*

$DC \succ_i CC \succ_i CD \succ_i DD$

*Stag Hunt*

$CC \succ_i DC \succ_i DD \succ_i CD$

# Summary

- This chapter has looked at agent interactions, and one approach to characterising them.
  - The approach we have looked at here is that of *game theory*, a powerful tool for analysing interactions.
  - We looked at solution concepts of Nash equilibrium and Pareto optimality.
  - We then looked at the classic Prisoner's Dilemma, and how the game can be analysed using game theory.
  - We also looked at the iterated Prisoner's Dilemma, and other canonical 2 × 2 games.

*Class Reading (Chapter 11):*

*"The Evolution of Cooperation", R. Axelrod. Basic Books, New York, 1984.*

This is a book, but is quite easy going, and most of the important content is conveyed in the first few chapters. Appart from anything else, it is beautifully written, and it is easy to see why so many readers are enthusiastic about it. However, read it in conjunction with Binmore's critiques, cited in the course text.