

# PROGRAMMING GROUPS OF RATIONAL AGENTS

Michael Fisher

*Department of Computer Science, University of Liverpool, UK*

michael@csc.liv.ac.uk

<http://www.csc.liv.ac.uk/~michael>

Fourth International Workshop on  
Computational Logic in Multi-Agent Systems (CLIMA-IV)

Fort Lauderdale, USA — 6th/7th January, 2004

Supported by



funded through EU Framework V



*To provide appropriate logic-based abstractions allowing us to program both individual rational agents and more complex organisational structures.*

- Motivation:
  - future → logic-based rational agent programming.
- Programming individual agents:
  - logics of time, belief and ability;
  - direct execution of individual agent specifications.
- Organising multiple agents:
  - concurrency, communication and semantics;
  - agents and grouping.
- Programming groups of rational agents:
  - developing groups and teams;
  - current work.

# PART I: MOTIVATION

# Evolution of Software

As computational platforms and wireless networking become ubiquitous, so software artefacts will increasingly be able to

- migrate through large physical/virtual distances,
- access vast amounts of information,
- spawn new computations on a wide variety of platforms,
- etc.

In addition, the requirement for autonomous and ‘intelligent’ behaviour will allow such software artefacts to

- evolve unexpected autonomous behaviour (aka *emergent* behaviour)

# Problems

The big problems will not concern making this vision of ubiquitous computation happen, but

a) *programming* such software so that it can do what we want,

and

b) *verifying* that software does, indeed, do what we expect.

Since computational elements are

- evolving,
- interacting, and
- everywhere,

we need new ways of controlling the behaviour of both

- a) individual elements, and
- b) fluid organisations of such elements.

It is here that the concept of a *rational agent* provides an useful abstraction for describing such systems.

# Rational Agents

object  $\longrightarrow$  *encapsulation*

agent  $\longrightarrow$  object + *autonomy*

However, require

rational agent  $\longrightarrow$  agent + *flexible autonomous action*

Rational agents must be able to adapt their autonomous behaviour to cater for their dynamic environment, requirements and knowledge.

Typically, they involve:

- pro-activeness
- social ability
- deliberation

# Aside: Representing Rational Agents

It is interesting to note that many models of rational agency share similar elements, in particular

- a *dynamic* element, allowing the representation of the agent's basic dynamic activity,
- an *informational* element, representing the agent's database of information,
- a *motivational* element, often representing the agent's goals, and
- a mechanism for *deliberation* that characterises the way in which motivations develop dynamically.

# Example: Spacecraft Landing

Imagine a rational agent controlling a spacecraft that is attempting to land on a planet. The agent has:

- dynamic activity concerning speed, direction, etc;
- information about the planet terrain and target landing sites;
- motivations, such as to land soon, and to remain aloft until safe to land.

# Example: Spacecraft Landing

Imagine a rational agent controlling a spacecraft that is attempting to land on a planet. The agent has:

- dynamic activity concerning speed, direction, etc;
- information about the planet terrain and target landing sites;
- motivations, such as to land soon, and to remain aloft until safe to land.

The agent must dynamically

- assess, and possibly revise, the information held
- decide what to do, i.e. deliberate over motivations,
- generate new motivations or revise its current ones

# PART II: INDIVIDUAL AGENTS

# Executing Agent Specifications

There are many rational agent theories, and agent programming languages. However, the two rarely match.

Our approach is to attempt to directly execute rational agent specifications — in this way, we can be more confident that the required behaviour is being exhibited.

Here, execution of a formula,  $\varphi$ , of a logic,  $L$ , is taken to mean constructing a model,  $\mathcal{M}$ , for  $\varphi$ , i.e.  $\mathcal{M} \models_L \varphi$ .

This not only provides a close link between the theory and implementation, but also provides high-level concepts within the programming language.

# Why Temporal Logic?

Temporal logic is an extension of classical logic with the notion of temporal order built in.

With such logics we can describe many dynamic properties, but they all boil down to describing

- what we must do *now*,
- what we must do *next*, and
- what we guarantee to do at *some* point in the future.

This, seemingly simple, view gives us the flexibility to represent a wide range of computational activities.

# Programming using Temporal Logic

With temporal logic as a basis, we are able to program individual agents:

- their dynamic behaviour;
- how their knowledge evolves;
- how their goals evolve.

# Programming using Temporal Logic

With temporal logic as a basis, we are able to program individual agents:

- their dynamic behaviour;
- how their knowledge evolves;
- how their goals evolve.

We are also able to describe fluid organisations:

- their communication structure;
- how the structures evolve;
- how the computation within each structure evolves.

# Temporal Logic: Intuition

In its simplest form, temporal logic can be seen as an extension of classical logic, incorporating additional operators relating to temporal order.

These operators are typically:

- ‘ $\bigcirc$ ’ ..... “in the next moment in time”,
- ‘ $\square$ ’ ..... “at every future moment”
- ‘ $\diamond$ ’ ..... “at some future moment”

These operators give us significant expressive power.

# Temporal Logic: History

Temporal logic was originally developed in order to represent tense in natural language.

Within Computer Science, it has achieved a significant role in the formal specification and verification of concurrent and distributed systems.

Much of this popularity has been achieved as a number of useful concepts can be formally, and concisely, specified using temporal logics, e.g.

# Temporal Logic: History

Temporal logic was originally developed in order to represent tense in natural language.

Within Computer Science, it has achieved a significant role in the formal specification and verification of concurrent and distributed systems.

Much of this popularity has been achieved as a number of useful concepts can be formally, and concisely, specified using temporal logics, e.g.

- *safety*

$$\square \neg bad$$

# Temporal Logic: History

Temporal logic was originally developed in order to represent tense in natural language.

Within Computer Science, it has achieved a significant role in the formal specification and verification of concurrent and distributed systems.

Much of this popularity has been achieved as a number of useful concepts can be formally, and concisely, specified using temporal logics, e.g.

- *safety*

$\square \neg bad$

- *liveness*

$\diamond good$

# Temporal Logic: History

Temporal logic was originally developed in order to represent tense in natural language.

Within Computer Science, it has achieved a significant role in the formal specification and verification of concurrent and distributed systems.

Much of this popularity has been achieved as a number of useful concepts can be formally, and concisely, specified using temporal logics, e.g.

● *safety*

$\square \neg bad$

● *liveness*

$\diamond good$

● *fairness*

$\square \diamond ask \Rightarrow \diamond receive$

# Logical Agent Theories

Recall that rational agent theories typically consist of

Dynamism —

Information —

Motivation —

# Logical Agent Theories

Recall that rational agent theories typically consist of

Dynamism — temporal or dynamic logic;

Information —

Motivation —

# Logical Agent Theories

Recall that rational agent theories typically consist of

Dynamism — temporal or dynamic logic;

Information — modal logics of belief or knowledge;

Motivation —

# Logical Agent Theories

Recall that rational agent theories typically consist of

Dynamism — temporal or dynamic logic;

Information — modal logics of belief or knowledge;

Motivation — modal logics of goals, intentions, desires.

# Typical (BDI) Example

The behaviour of an agent may be specified in terms of its beliefs, desires and intentions:

$$B_{me} \diamond D_{you} \text{attack}(you, me) \Rightarrow I_{me} \bigcirc \text{attack}(me, you)$$

i.e.

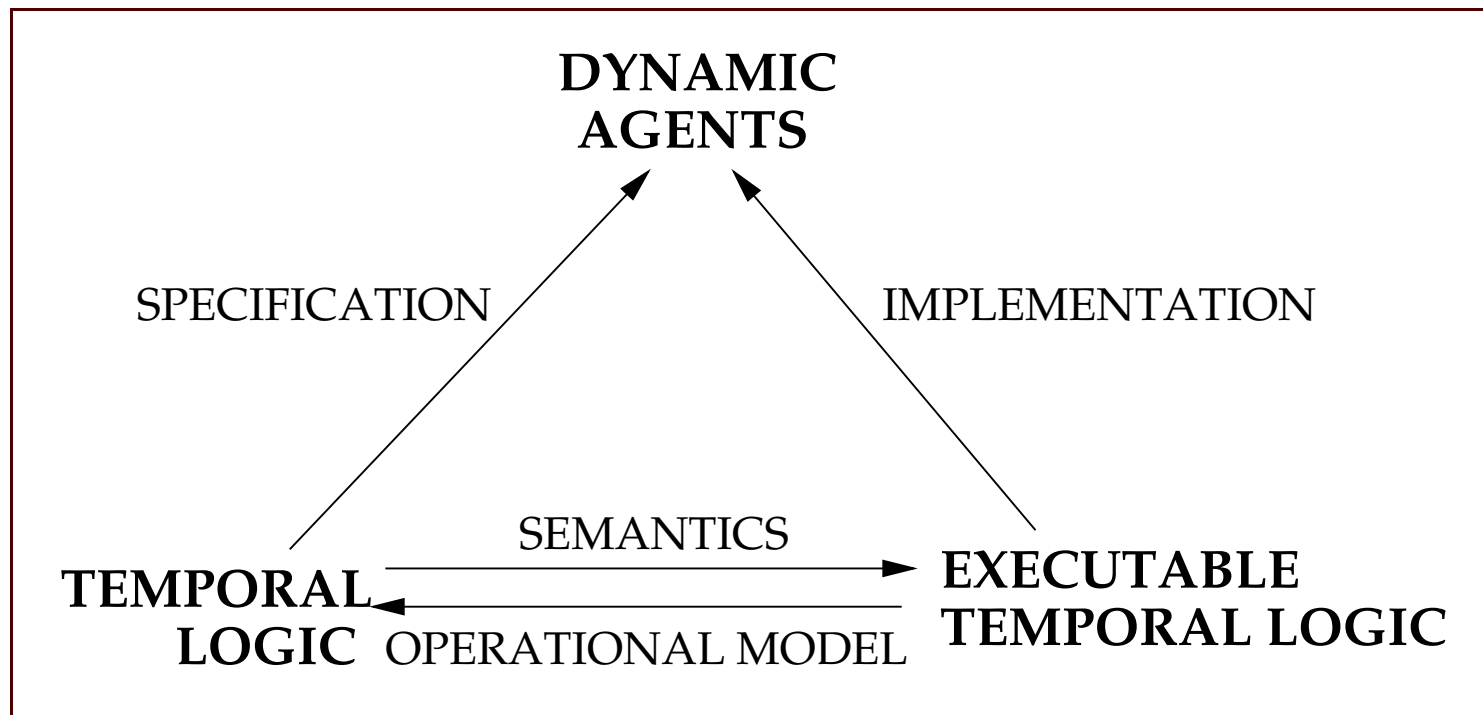
*if I believe that you desire to attack me, then I intend to attack you at the next moment in time*

Alternatively, using just belief and time:

$$B_{me} \diamond B_{you} \text{attack}(you, me) \Rightarrow B_{me} \bigcirc \text{attack}(me, you)$$

# Single Agent Execution

We begin by executing the basic temporal logic described earlier and then extend the logic in various ways.



# Execution Mechanism

We use the *Imperative Future* approach:

- Transform the temporal specification into normal form.
- From the initial constraints, *forward chain* through the set of temporal rules representing the agent.
- Constrain the execution by attempting to satisfy goals, such as  $\diamond g$  (i.e.  $g$  eventually becomes true).

Need strategy for handling *outstanding* eventualities.

# Execution Example

Imagine a 'car' agent which can *go*, *turn* and *stop*, but can also run out of fuel (*empty*) and *overheat*.

$$\begin{aligned} \mathbf{start} &\Rightarrow \neg \mathit{moving} \\ \mathit{go} &\Rightarrow \diamond \mathit{moving} \\ (\mathit{moving} \wedge \mathit{go}) &\Rightarrow \bigcirc (\mathit{overheat} \vee \mathit{empty}) \end{aligned}$$

# Execution Example

Imagine a 'car' agent which can *go*, *turn* and *stop*, but can also run out of fuel (*empty*) and *overheat*.

$$\mathbf{start} \Rightarrow \neg moving$$
$$go \Rightarrow \diamond moving$$
$$(moving \wedge go) \Rightarrow \bigcirc (overheat \vee empty)$$

- Thus, *moving* is false at the beginning of time.

# Execution Example

Imagine a 'car' agent which can *go*, *turn* and *stop*, but can also run out of fuel (*empty*) and *overheat*.

$$\mathbf{start} \Rightarrow \neg \mathit{moving}$$

$$\mathit{go} \Rightarrow \diamond \mathit{moving}$$

$$(\mathit{moving} \wedge \mathit{go}) \Rightarrow \bigcirc (\mathit{overheat} \vee \mathit{empty})$$

- Thus, *moving* is false at the beginning of time.
- Whenever *go* is true, a commitment to eventually make *moving* true is given.

# Execution Example

Imagine a 'car' agent which can *go*, *turn* and *stop*, but can also run out of fuel (*empty*) and *overheat*.

$$\mathbf{start} \Rightarrow \neg moving$$

$$go \Rightarrow \diamond moving$$

$$(moving \wedge go) \Rightarrow \bigcirc (overheat \vee empty)$$

- Thus, *moving* is false at the beginning of time.
- Whenever *go* is true, a commitment to eventually make *moving* true is given.
- Whenever both *go* and *moving* are true, then either *overheat* or *empty* will be made true in the next moment in time.

# Extension: Deliberation

Aim: to allow agents to modify/adapt their goals dynamically.

In our framework, deliberation concerns deciding in which order eventualities/goals (e.g.  $\diamond g_1$ ,  $\diamond g_2$ ,  $\diamond g_3$ ) should be attempted.

Given a list of outstanding eventualities/goals, we can re-order this list at every execution step given a set of situation-specific priority functions.

These functions are programmed by the user.

# Deliberation Example

[◇be\_famous, ◇sleep, ◇eat\_lunch, ◇make\_lunch]

Standard approach would be to execute these oldest-first.

# Deliberation Example

[◇be\_famous, ◇sleep, ◇eat\_lunch, ◇make\_lunch]

Standard approach would be to execute these oldest-first.

Re-order based on importance:

[◇be\_famous, ◇eat\_lunch, ◇sleep, ◇make\_lunch]

# Deliberation Example

[◇be\_famous, ◇sleep, ◇eat\_lunch, ◇make\_lunch]

Standard approach would be to execute these oldest-first.

Re-order based on importance:

[◇be\_famous, ◇eat\_lunch, ◇sleep, ◇make\_lunch]

Re-order based on plans:

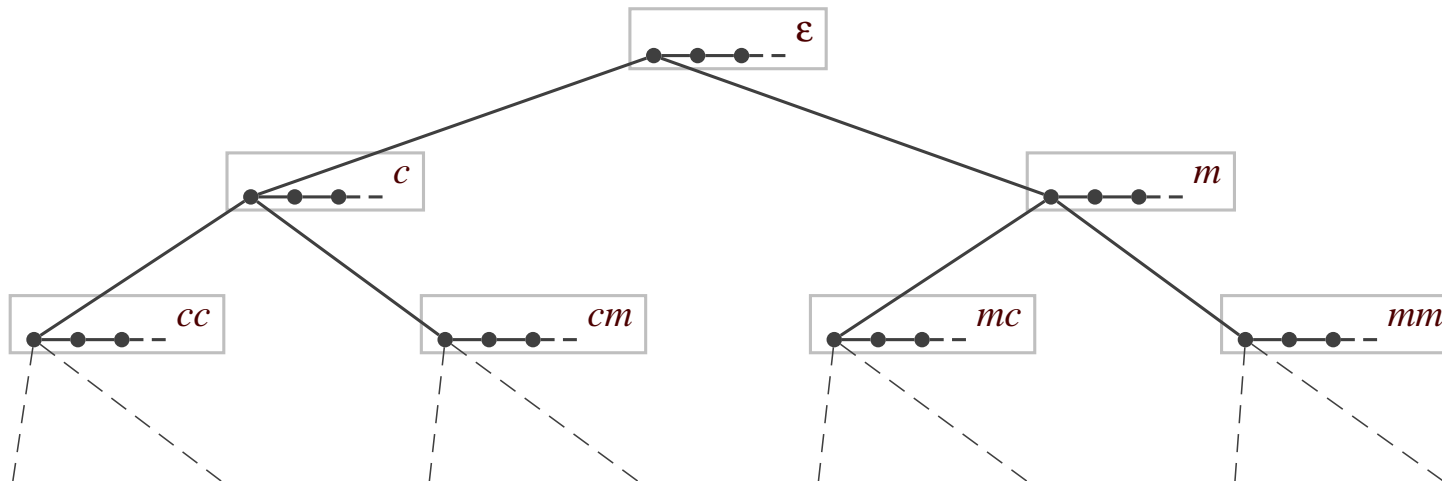
[◇make\_lunch, ◇eat\_lunch, ◇sleep, ◇be\_famous]

# Extension: Bounded Belief

Aim: to model the nesting of belief operators using contexts.

Language:  $B_i$  represents the beliefs of agent  $i$ .

$B_i\psi$  intuitively means ' $i$  believes  $\psi$ '.



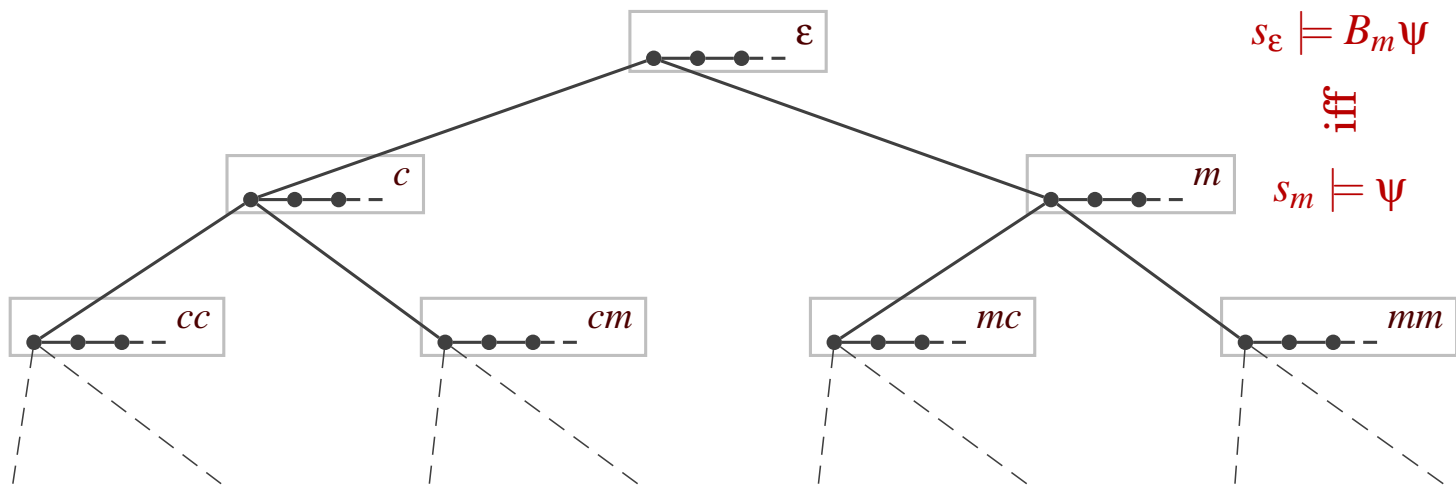
# Extension: Bounded Belief

Aim: to model the nesting of belief operators using contexts.

Language:  $B_i$  represents the beliefs of agent  $i$ .

$B_i\psi$  intuitively means ' $i$  believes  $\psi$ '.

Semantics:  $\left[ \begin{array}{l} B_m\psi \text{ in } \varepsilon \\ \psi \text{ in } m \end{array} \right]$  :  $\varepsilon$  believes that  $m$  believes that  $\psi$ .



# Extension: Bounded Belief

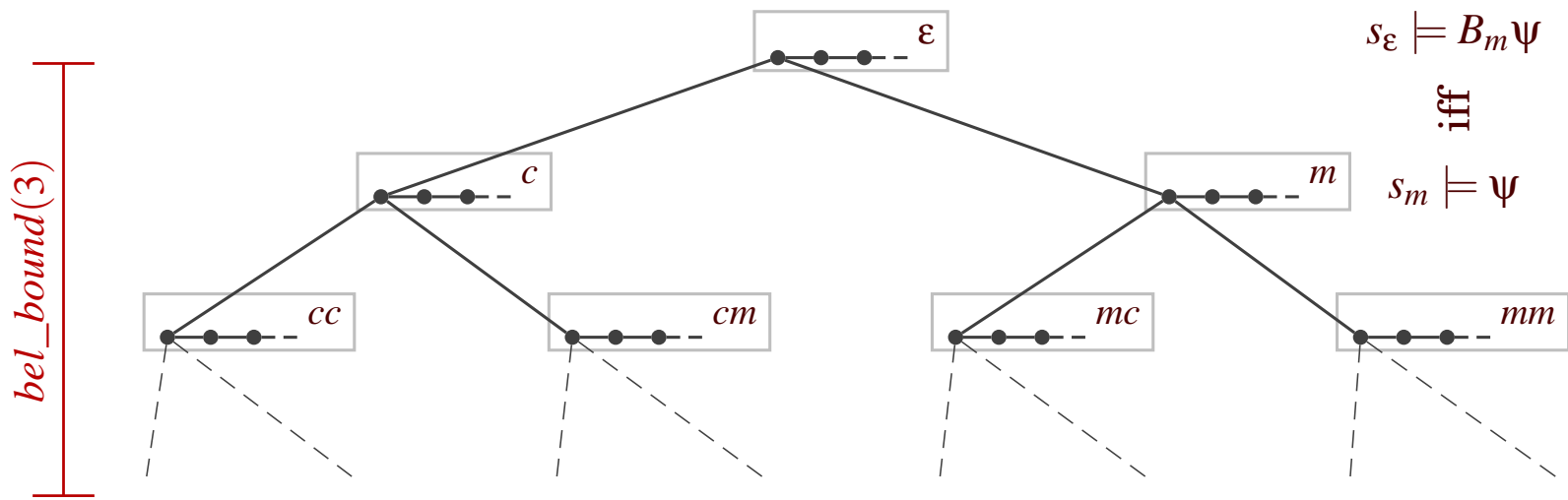
Aim: to model the nesting of belief operators using contexts.

Language:  $B_i$  represents the beliefs of agent  $i$ .

$B_i\psi$  intuitively means ' $i$  believes  $\psi$ '.

$bel\_bound(x)$ : amount of resources for reasoning about belief.

Semantics:  $\left[ \begin{array}{l} B_m\psi \text{ in } \varepsilon \\ \psi \text{ in } m \end{array} \right]$  :  $\varepsilon$  believes that  $m$  believes that  $\psi$ .



# Extension: Bounded Belief

Aim: to model the nesting of belief operators using contexts.

Language:  $B_i$  represents the beliefs of agent  $i$ .

$B_i\psi$  intuitively means 'i believes  $\psi$ '.

$bel\_bound(x)$ : amount of resources for reasoning about belief.

Semantics:  $\left[ \begin{array}{l} B_m\psi \text{ in } \varepsilon \\ \psi \text{ in } m \end{array} \right] : \varepsilon \text{ believes that } m \text{ believes that } \psi.$

Why add beliefs?

$buy\_ticket \Rightarrow B_{me} \diamond lottery\_winner$

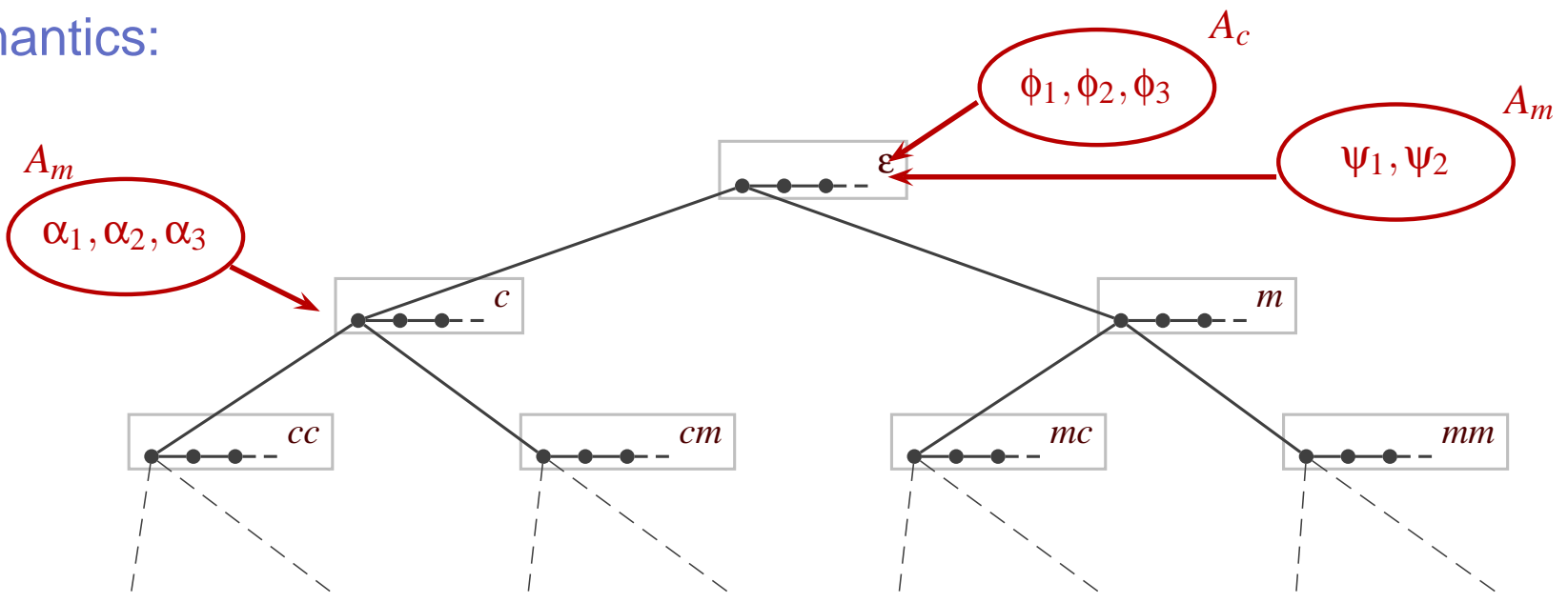
# Extension: Ability

Aim: represent the abilities of each agent.

Language:  $A_i$  represents the ability agent  $i$ .

$A_i\phi$  intuitively means ' $i$  is able to do  $\phi$ '.

Semantics:



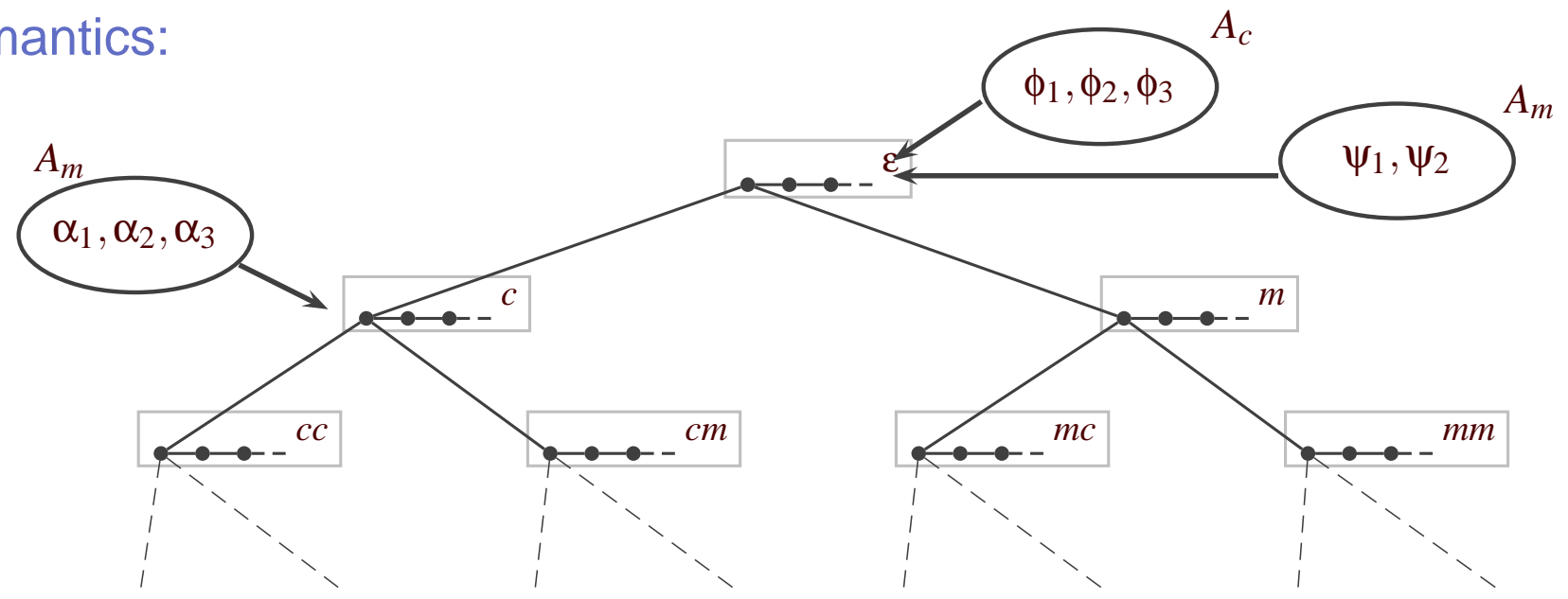
# Extension: Ability

Aim: represent the abilities of each agent.

Language:  $A_i$  represents the ability agent  $i$ .

$A_i\phi$  intuitively means ' $i$  is able to do  $\phi$ '.

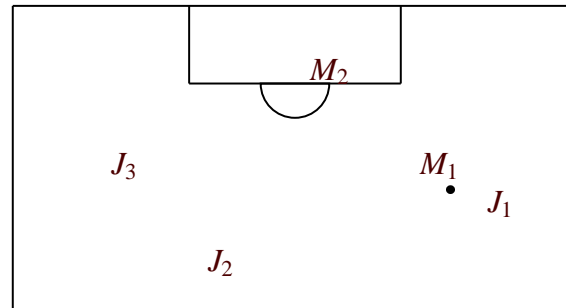
Semantics:



Why add ability?

$$A_{me}buy\_ticket \Rightarrow \bigcirc buy\_ticket$$

# Example: Resource-Bounded Deliberation



**Problem:** agent  $J_1$  has to decide what to do next.

**Possibilities:**  $J_1$  has one motivation (to score) and two abilities (pass or shoot).

**Constraint:**  $J_1$  might have resource bounds concerning the amount of reasoning it is able to carry out.

**Question:** what happens under such resource bounds?

# Beyond Individual Agents

Common rule:

$$(B_i \diamond \phi \wedge A_i \phi) \Rightarrow \diamond \phi$$

But what if:

$$B_i \diamond \phi \wedge \neg A_i \phi?$$

# PART III: MULTIPLE AGENTS

# Concurrent Operational Framework

We now consider two elements:

1. The representation of an individual agent's behaviour using a temporal/modal specification.
2. An operational framework providing both concurrency and communication.

Agents here execute independently and asynchronously.

The use of *broadcast* message-passing links the logical and operational aspects.

# Agent Interfaces

Each agent has an *interface*, consisting of:

- a unique *identifier*, which names the agent;
- a set of symbols defining what messages will be accepted by the agent — *environment predicates*;
- a set of symbols defining messages that the agent may send — *component predicates*.

The interface definition of a ‘car’ agent might be:

```
car ( )  
  in:  go , stop , turn  
  out: empty , overheat
```

# Example: Restaurant Tactics!

waiter()

in: ask  
out: serve

$\text{ask}(X) \Rightarrow \blacklozenge \text{serve}(X);$

$\text{serve}(X) \wedge \text{serve}(Y) \Rightarrow X=Y.$

---

eager()

out: ask

**start**  $\Rightarrow \square \text{ask}(\text{eager}).$

---

mimic()

in: ask  
out: ask

$\text{ask}(\text{eager}) \Rightarrow \bigcirc \text{ask}(\text{mimic}).$

---

jealous()

in: serve  
out: ask

$\text{serve}(\text{eager}) \Rightarrow \bigcirc \text{ask}(\text{jealous}).$

Semantics of multi-agent system can be given by combining semantics of individual agents.

Variety of combination depends on type of concurrency.

For example, although semantics of each agent may be given using *discrete* temporal logics, semantics of a collection of such agents executing asynchronously may require *dense* temporal logics.

But, so far, agent space is unstructured.....

# Grouping

Agents occur in *groups* — this is a simple and intuitive structuring mechanism with diverse applications.

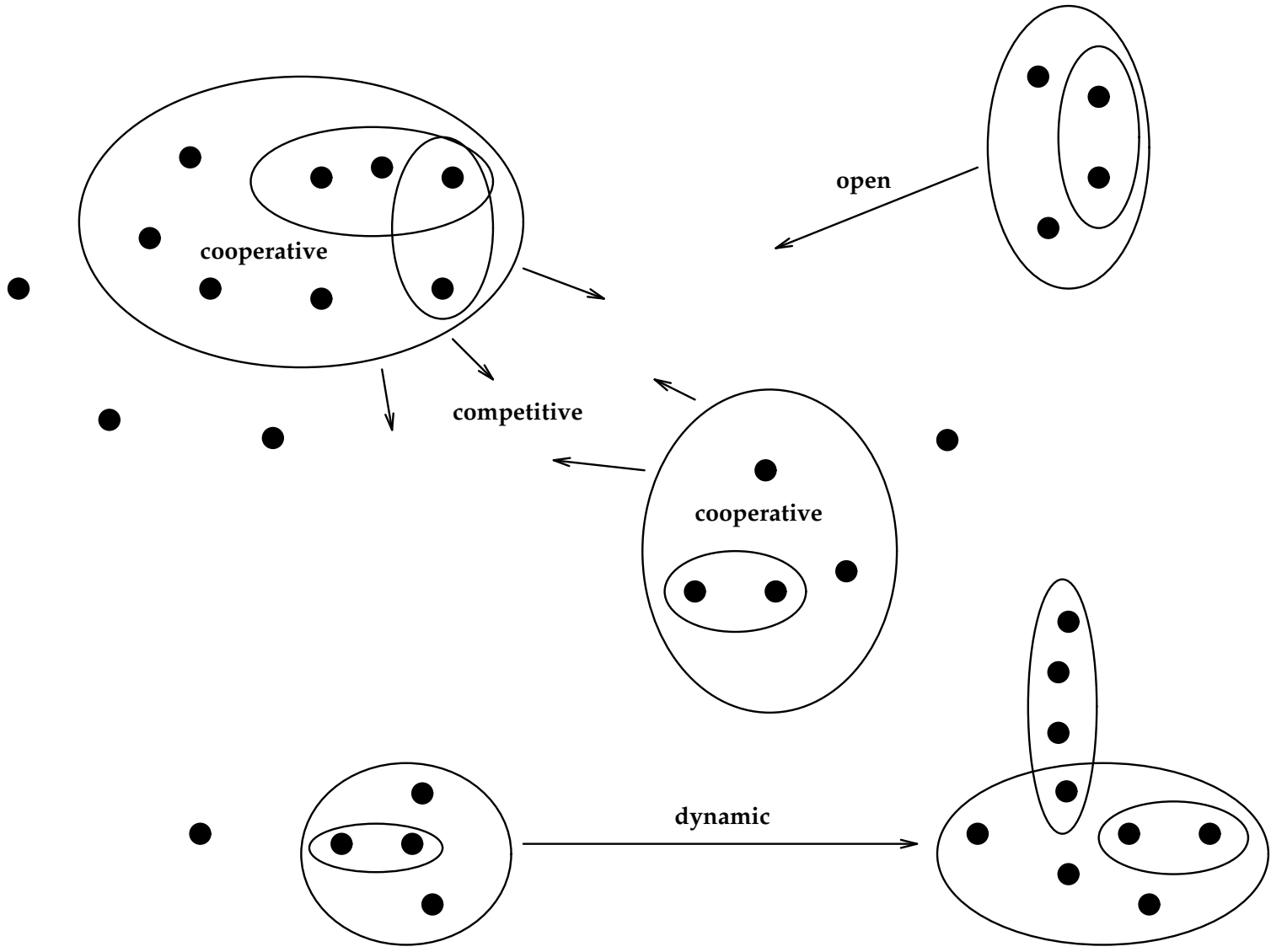
Groups are dynamic and open.

Groups may contain sub-groups and each agent may be a member of several groups.

Groups are useful both for restricting the extent of broadcast and structuring the agent space.

Broadcast, plus *grouping* mechanism provides *multicast* message-passing.

# Groups and Agents



# Example: Distributed Problem-Solving

Individual agents represent simple problem-solving components, while grouping defines the local problem-solving activities.

Agents cooperate (e.g. by sharing or subcontracting) with their fellow group members, but groups themselves might compete with each other.

# Example: Distributed Problem-Solving

Individual agents represent simple problem-solving components, while grouping defines the local problem-solving activities.

Agents cooperate (e.g. by sharing or subcontracting) with their fellow group members, but groups themselves might compete with each other.

Successful groups may spawn more agents, providing a form of adaptability.

# Example: Distributed Problem-Solving

Individual agents represent simple problem-solving components, while grouping defines the local problem-solving activities.

Agents cooperate (e.g. by sharing or subcontracting) with their fellow group members, but groups themselves might compete with each other.

Successful groups may spawn more agents, providing a form of adaptability.

Broadcast communication allows many different agents to be undertaking similar tasks, and facilitates the dynamic introduction of new problem-solvers.

# Example: Distributed Problem-Solving

Individual agents represent simple problem-solving components, while grouping defines the local problem-solving activities.

Agents cooperate (e.g. by sharing or subcontracting) with their fellow group members, but groups themselves might compete with each other.

Successful groups may spawn more agents, providing a form of adaptability.

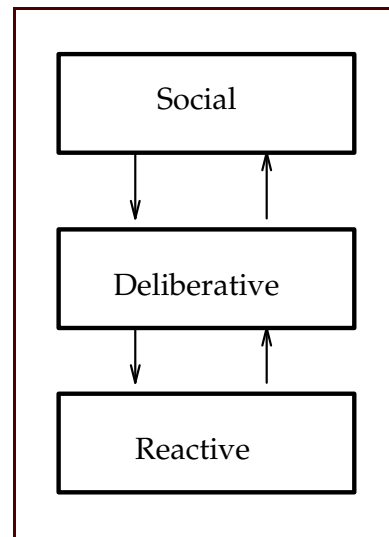
Broadcast communication allows many different agents to be undertaking similar tasks, and facilitates the dynamic introduction of new problem-solvers.

No explicit global control — groups are self-organising.

# Example: Layered Agent Architectures (1)

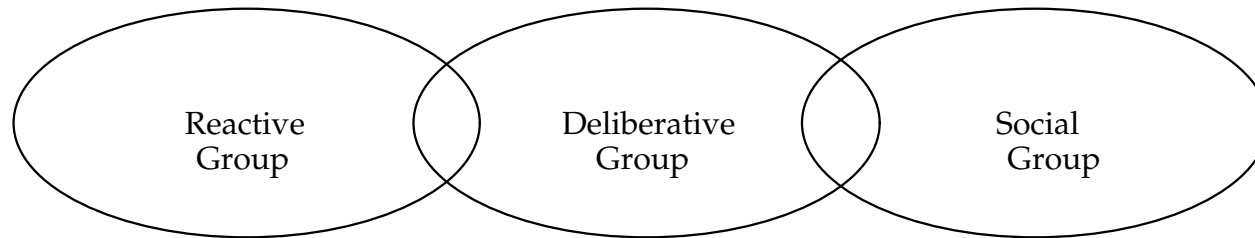
In developing complex agents, layered architectures are commonly used, often comprising

- a *reactive* layer,
- a *deliberative* layer, and
- a *cognitive/modelling/social* layer.

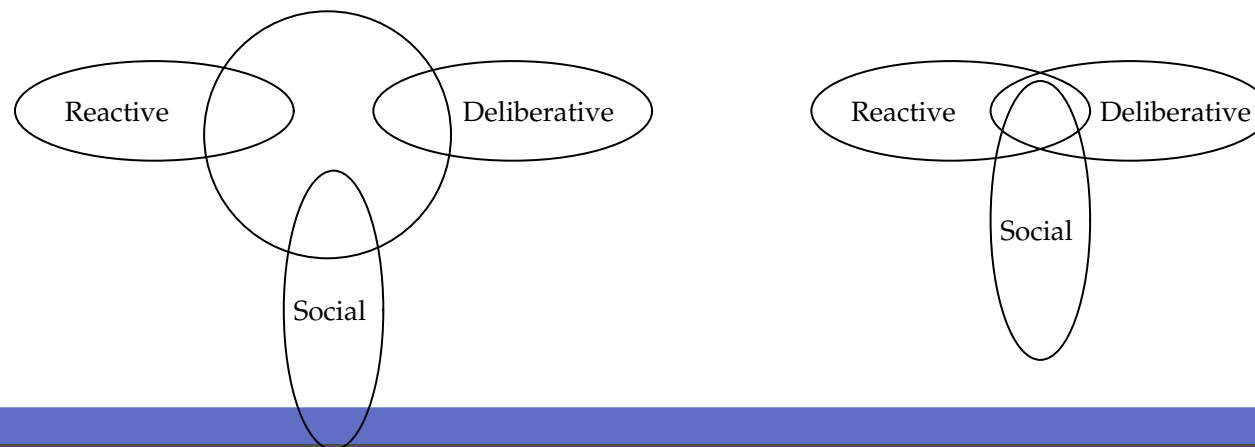


# Example: Layered Agent Architectures (2)

We can simulate layering by grouping together similar types of behaviours.



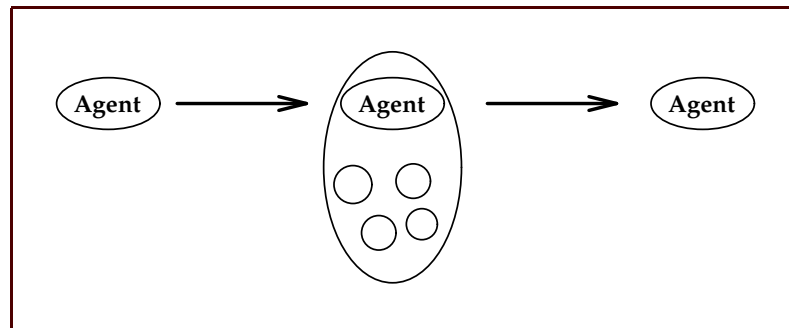
In order to facilitate communication between layers, certain 'linking' agents must be members of more than one group. The power of the grouping aspects allows us to represent a range of different configurations, e.g.



# Example: Groups in general computation

In general, why might groups be useful?

1. To provide further structure in the agent space.
2. Spawn a group to perform sub-computations.



N.B., these groups can either be transient or persistent.

# PART IV: RATIONAL AGENT GROUPS

# Agent $\equiv$ Group (1)

So far the notion of group is essentially just a container.

In essence we have

$$\textit{Agent} ::= \textit{Behaviour} : \textit{Spec}$$
$$\textit{Group} ::= \textit{Contents} : \mathcal{P}(\textit{Agent})$$

But, now we want

$$\textit{Agent} ::= \textit{Behaviour} : \textit{Spec}$$
$$\textit{Contents} : \mathcal{P}(\textit{Agent})$$
$$\textit{Context} : \mathcal{P}(\textit{Agent})$$

# Agent $\equiv$ Group (2)

Once we have this view, we can examine varieties of agent.

- A simple agent:  $Contents = \emptyset$ .
- A simple group:  $Behaviour = \emptyset$
- A more complex group:

$$Contents \neq \emptyset \quad Behaviour \neq \emptyset$$

Groups, rather than being mere *containers*, can now have behaviours, captured by their internal policies and rules.

In particular, agents can control the communication policies within their Contents.

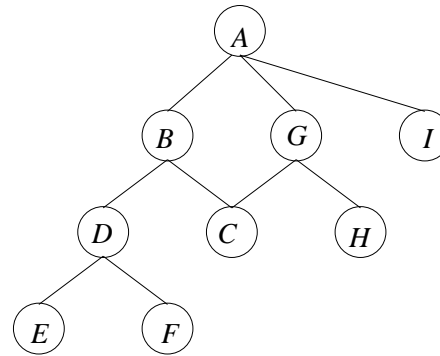
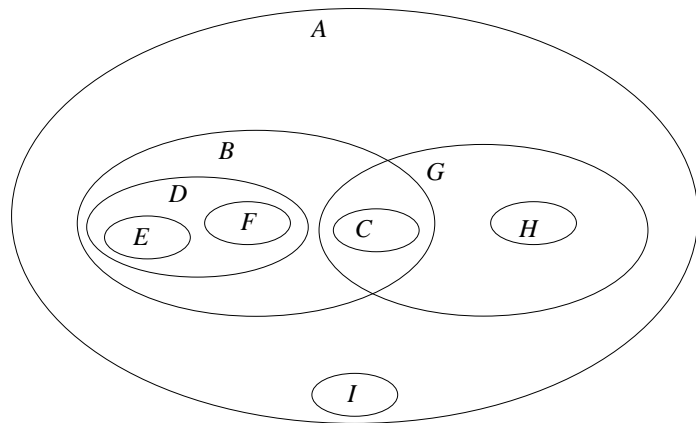
# Group Behaviour

Agents are opaque, in that their internal structure is hidden from other agents — it is not obvious whether agents have certain abilities natively, or merely make use of other (internal) agents.

Every agent has the potential to become a group!

- all agents respond to 'addToContent' and 'addToContext' messages
- some 'clone' messages also copy Contents and Context

# Example Agent/Group Structure



Agent	CN	CX
A	{B,G,I}	$\emptyset$
B	{D,C}	{A}
C	$\emptyset$	{B,G}
D	{E,F}	{B}
E	$\emptyset$	{D}
F	$\emptyset$	{D}
G	{C,H}	{A}
H	$\emptyset$	{G}
I	$\emptyset$	{A}

# Example: Collecting Agents (1)

Suppose that agent  $i$  wishes for  $\varphi$  to occur ( $B_i \diamond \varphi$ ), but does not have the corresponding ability ( $\neg A_i \varphi$ ).

We can add to the description of this agent a rule

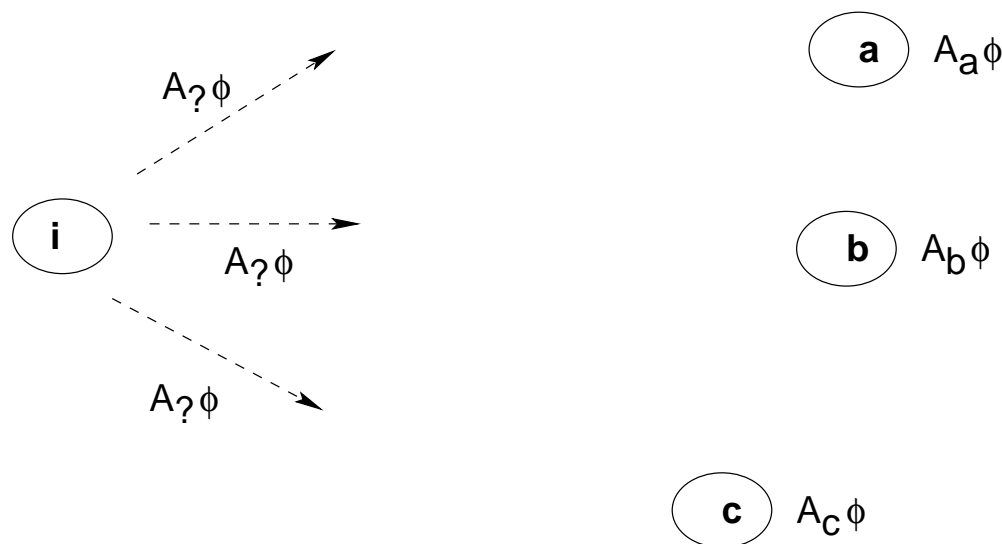
$$B_i \diamond \varphi \wedge \neg A_i \varphi \Rightarrow \bigcirc \text{send}_i(A? \varphi)$$

# Example: Collecting Agents (1)

Suppose that agent  $i$  wishes for  $\varphi$  to occur ( $B_i \diamond \varphi$ ), but does not have the corresponding ability ( $\neg A_i \varphi$ ).

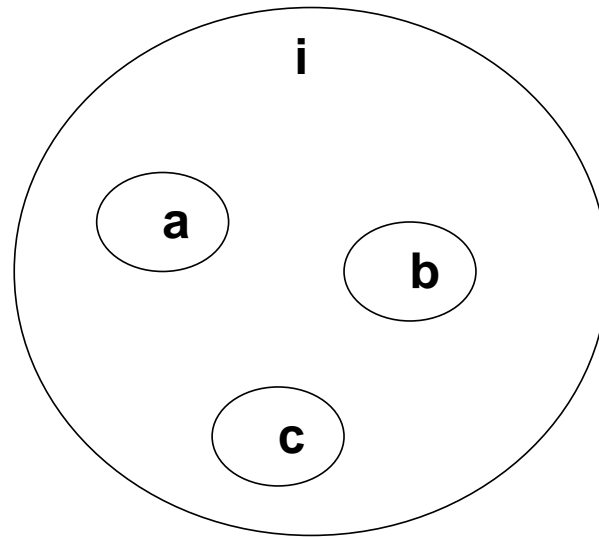
We can add to the description of this agent a rule

$$B_i \diamond \varphi \wedge \neg A_i \varphi \Rightarrow \bigcirc \text{send}_i(A? \varphi)$$



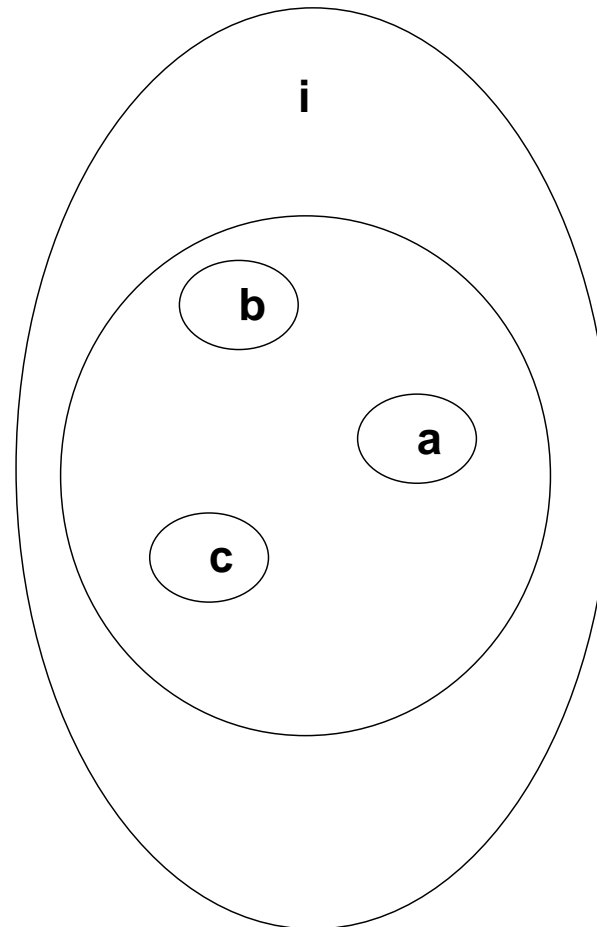
# Example: Collecting Agents (2)

- the sender may invite relevant agents to join its *Content*



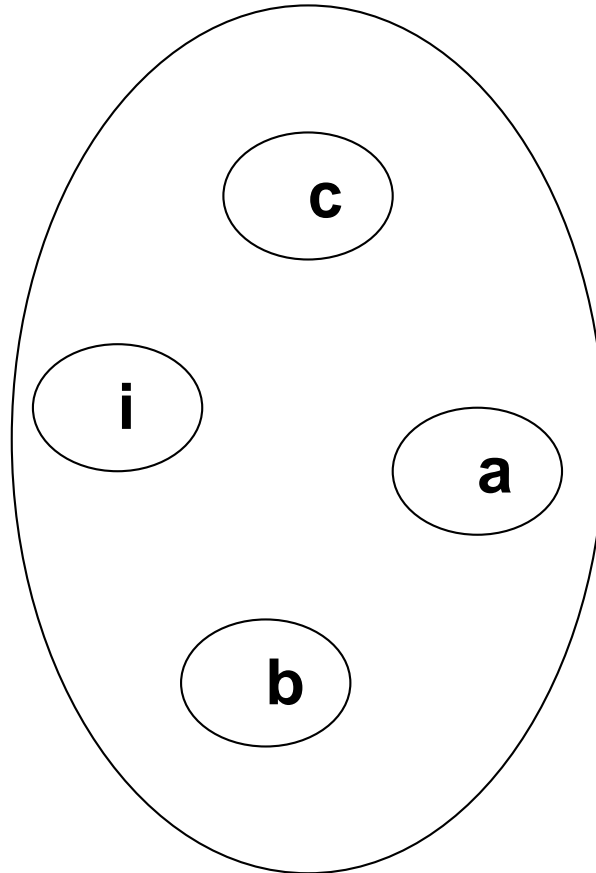
# Example: Collecting Agents (3)

- the sender may create a new “dedicated” agent to serve as a container for agents that share the relevant ability



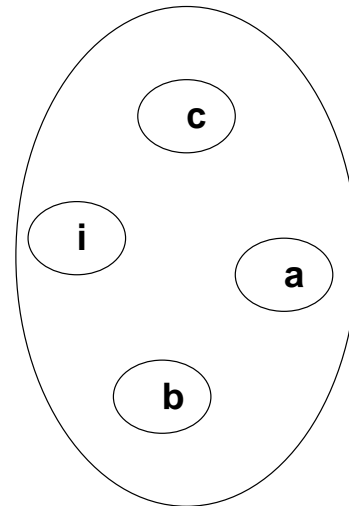
# Example: Collecting Agents (4)

- the sender may join a group that can help it solve its problem,



# Example: Teams

Consider an agent that will be used to collect other agents, e.g.



Using the policies captured within the agent, it might only allow certain agents to join (e.g, those that agree to some protocol).

Once we have such an agent, the “group agent” can spread information (and motivations) throughout the group — this can effectively provide common beliefs and common goals.

## Previously:

- individual agent execution implemented on top of Prolog;
- concurrent version (TL only) implemented in C++;
- prototype “agent≡group” framework in Java.

## Current work:

- Java implementation bringing together the above [Benjamin Hirsch]

# Conclusions

Where we are now:

- executable rational agent theory
- grouping as a mechanism for agent organisation
- group  $\equiv$  agent extends power of grouping

Where we are going:

- Java implementation
- exploring the power of *rational agent groups*

Choosing the appropriate logic provides a level of abstraction close to the key concepts of the software.

# Thanks

Work in collaboration with many colleagues, most recently:

- Chiara Ghidini [Trento, Italy]
- Benjamin Hirsch [Liverpool, UK]

This talk was sponsored by CologNet, the European Network of Excellence for Computational Logic.

For details of how to get involved, see

`http://www.colognet.org`