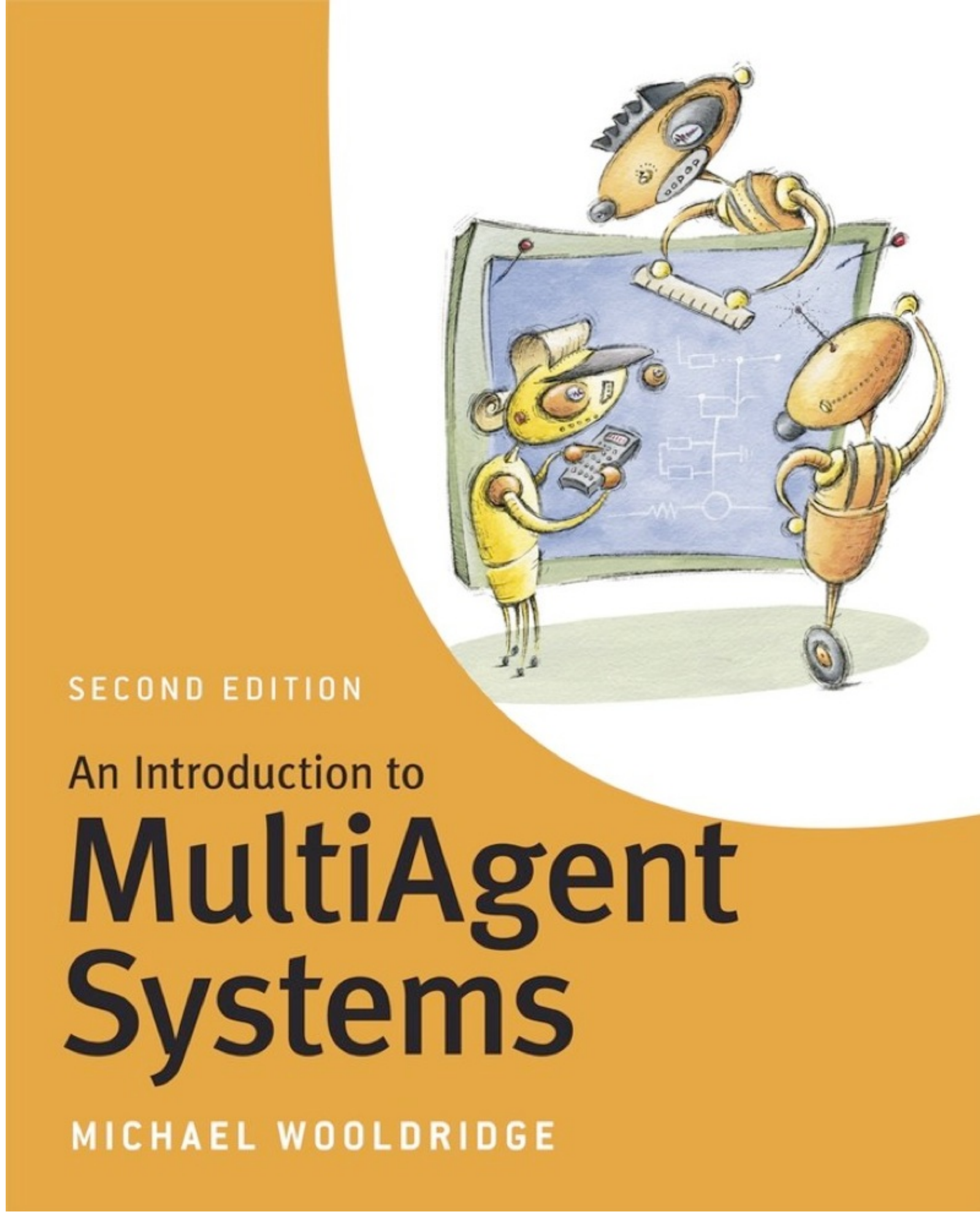
COMP310
Multi-Agent Systems
Chapter 8 - Working Together

Dr Terry R. Payne
Department of Computer Science



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..."

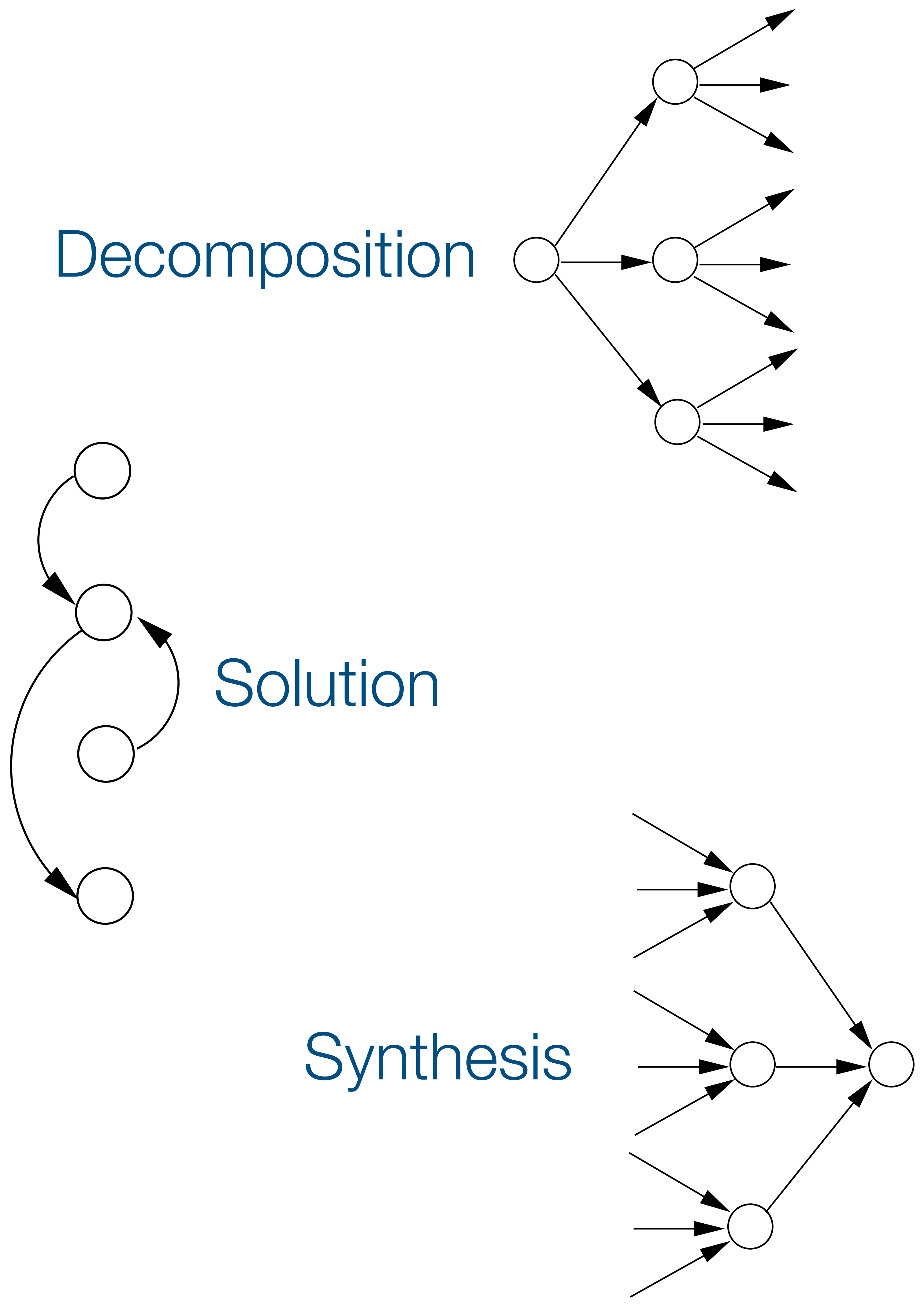
(Bond and Gasser, 1988).

Task Sharing and Result Sharing

- How does a group of agents work together to solve problems?

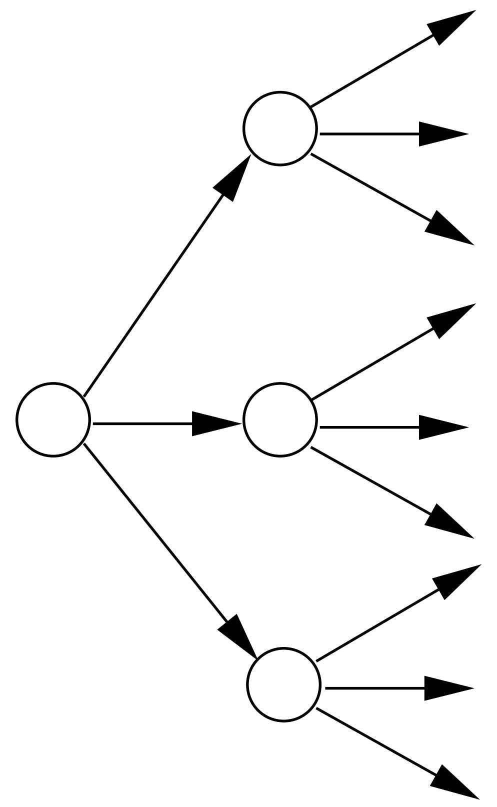
- CPDS addresses the following:
  - Problem decomposition
    - How can a problem be ***divided into smaller tasks*** for distribution amongst agents?
  - Sub-problem solution
    - How can the overall problem-solving activities of the agents be optimised so as to produce a solution that ***maximises 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.

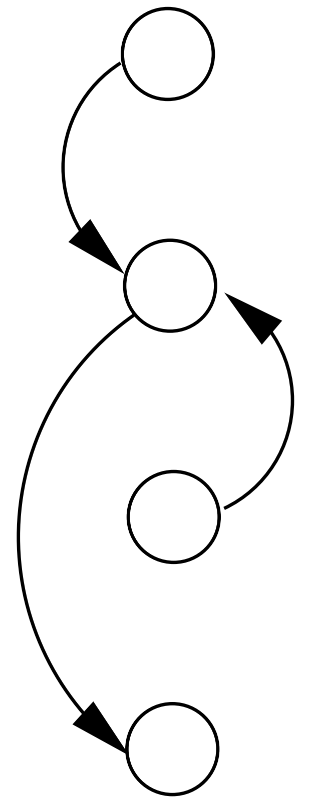


Problem Decomposition

- 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.

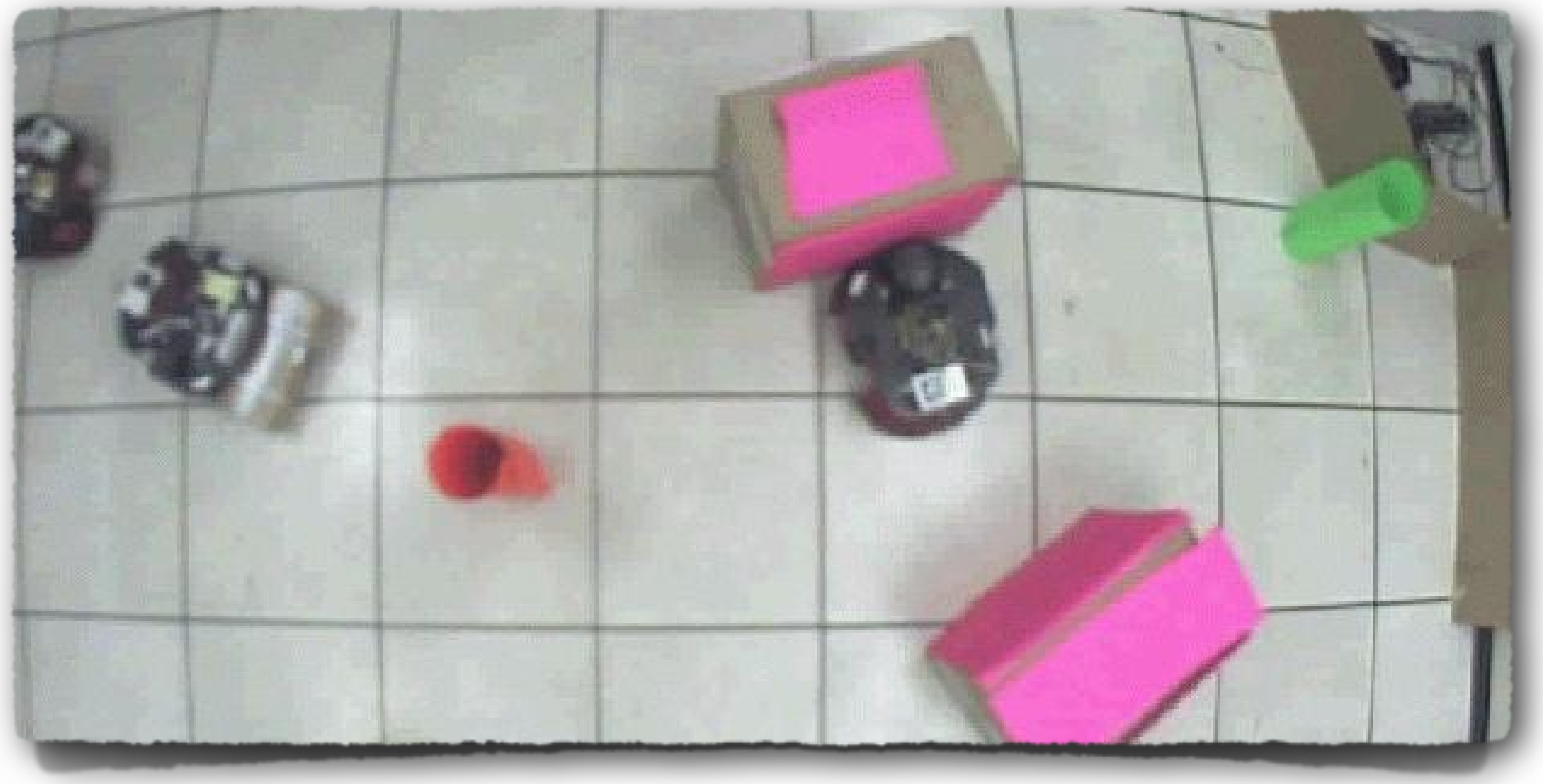


- Clearly there is some processing to do the division.
  - How this is done is one design choice.
- Another choice is ***who*** does the division.
  - Is it centralised?
  - Which agents have knowledge of task structure?
  - Who is going to solve the sub-problems?

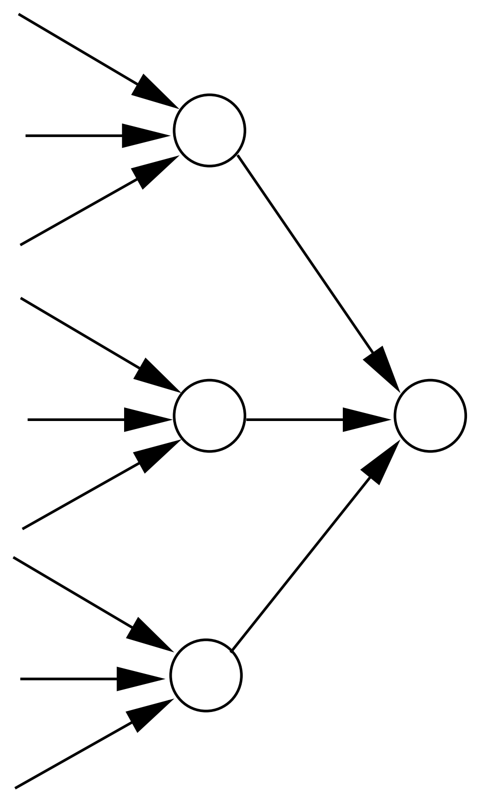


Sub-problem Solution

- 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

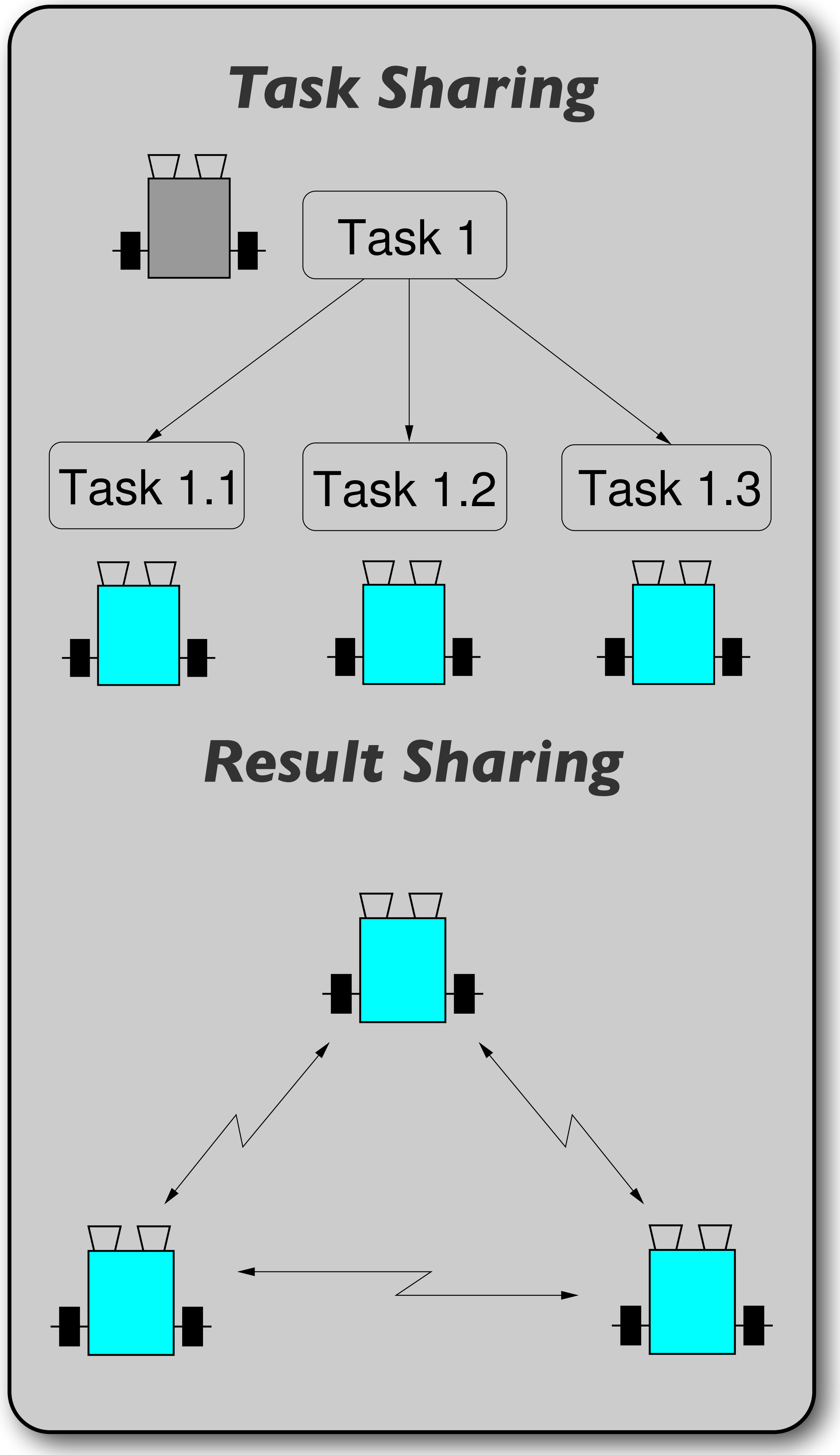


Solution Synthesis

- 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

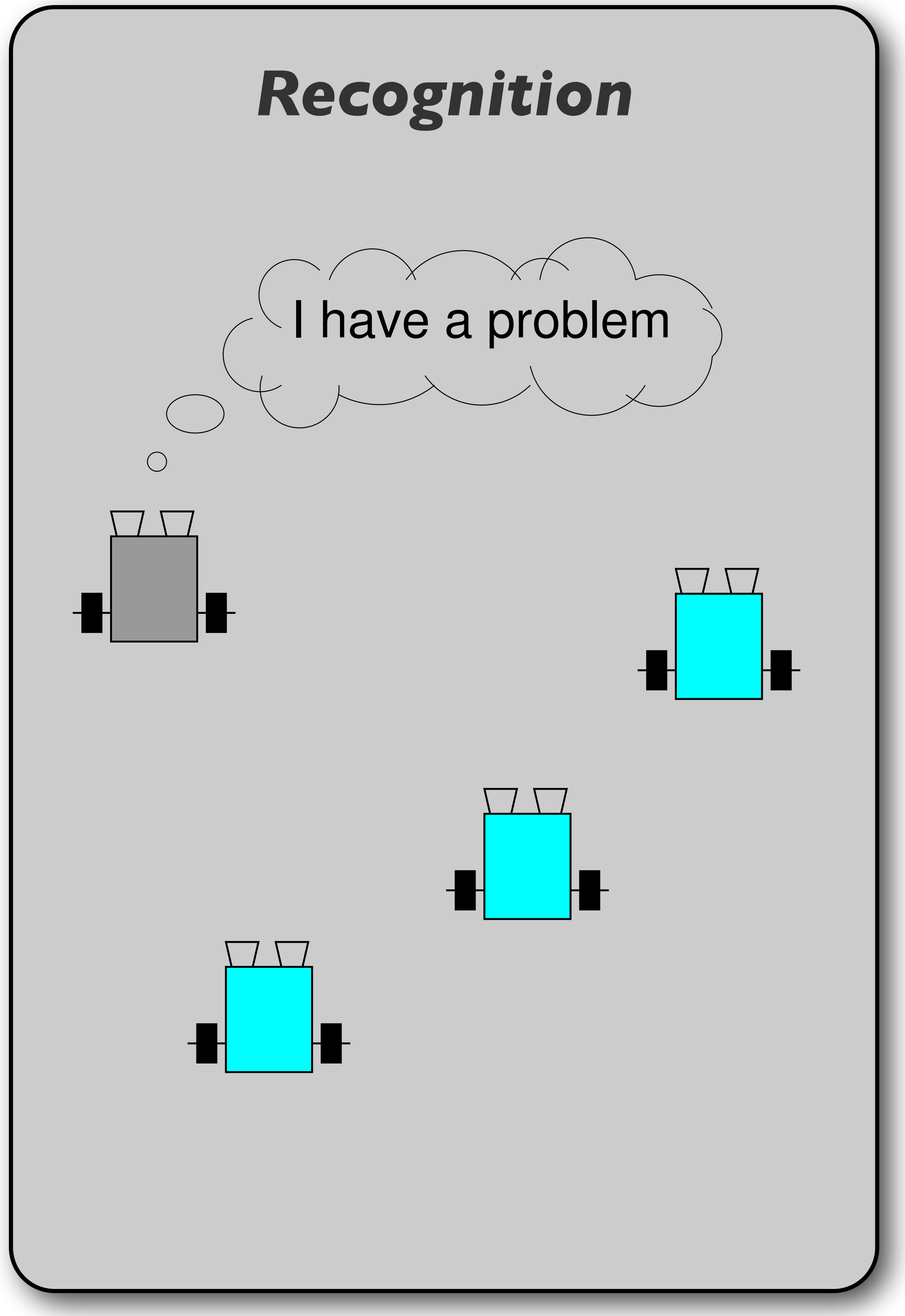
- Well known task-sharing protocol for task allocation is the ***contract net***.

- 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.

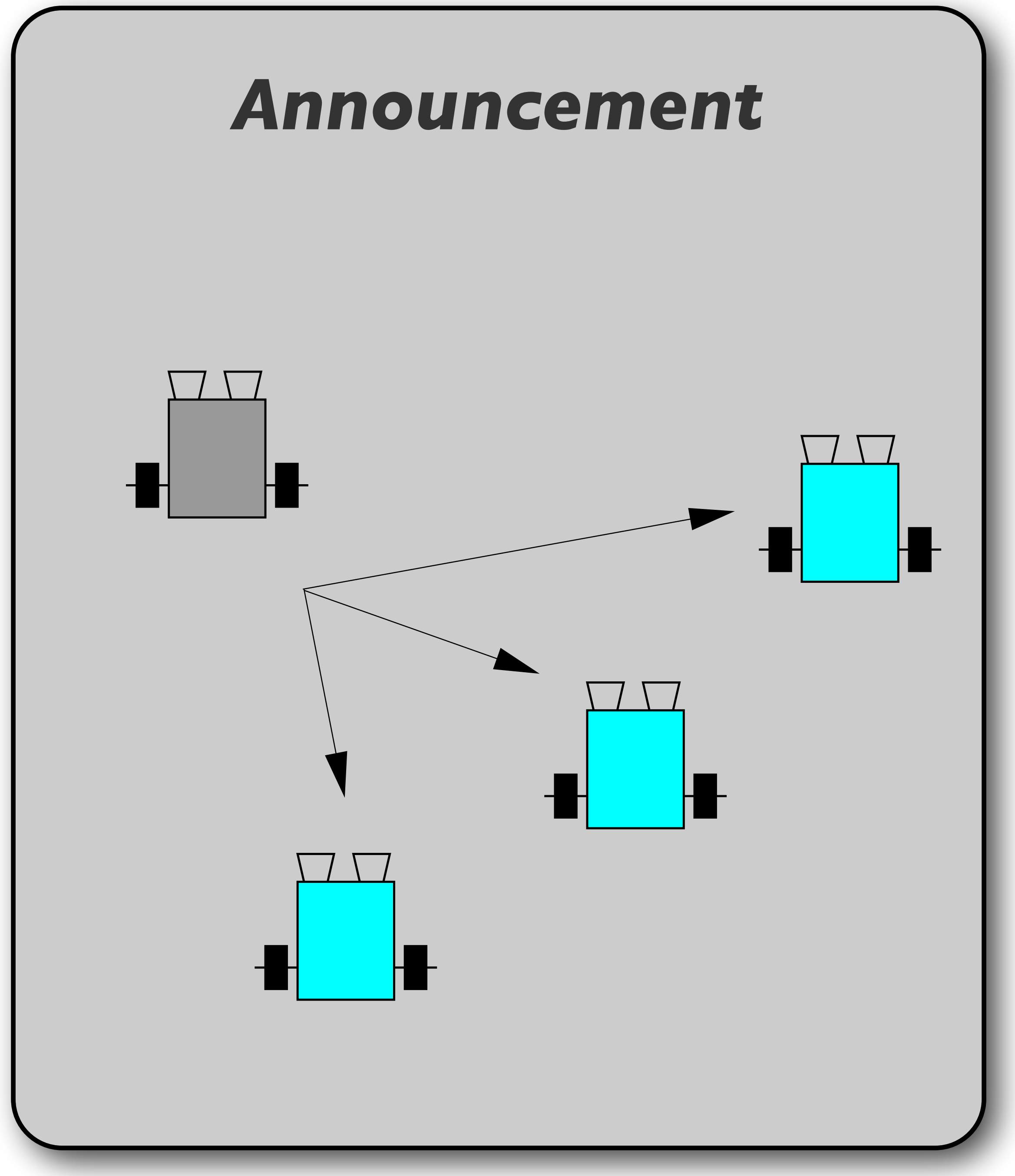


Announcement

- 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.

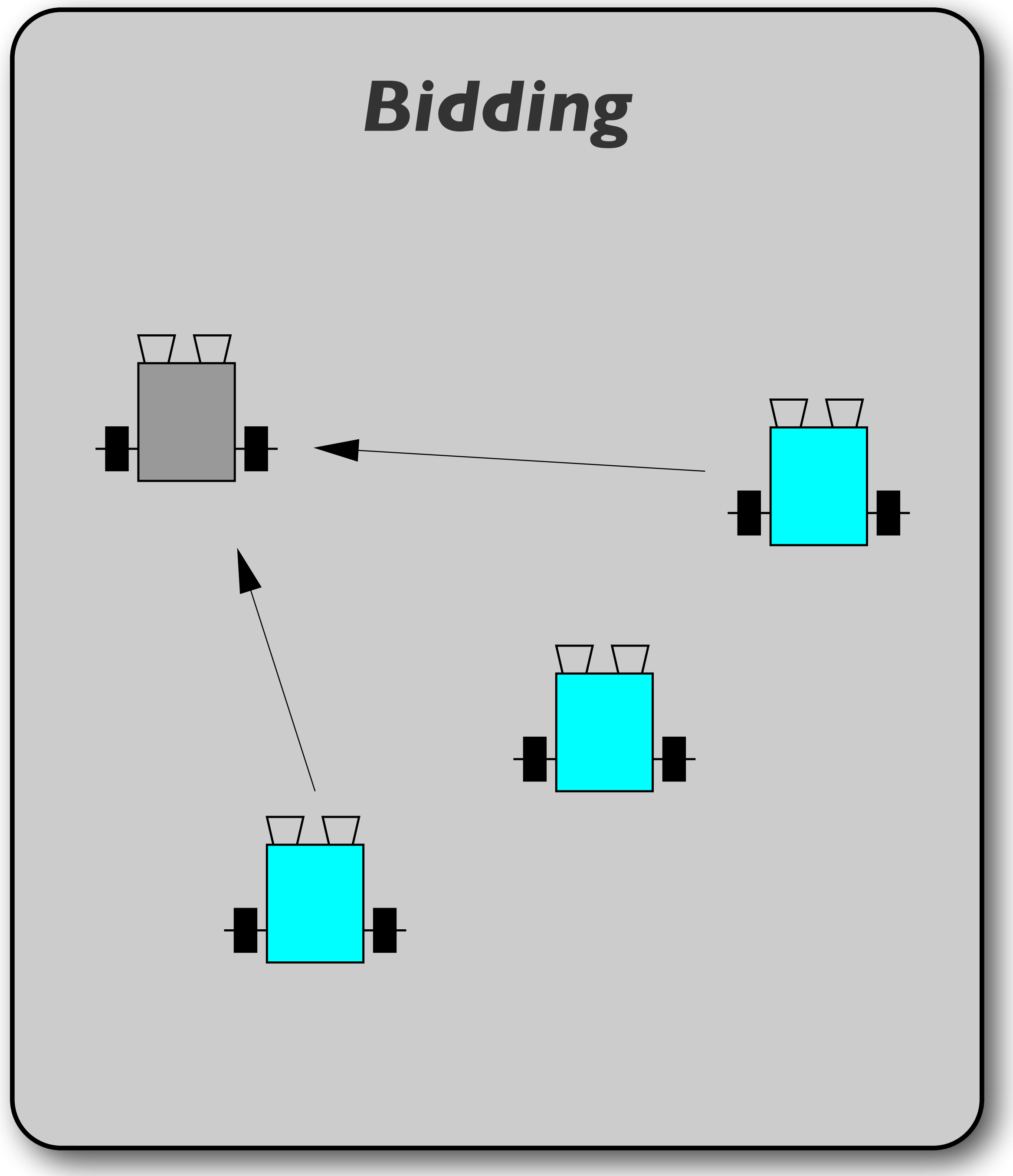
- Specification must encode:
    - description of task itself (maybe executable)
    - any constraints (e.g., deadlines, quality constraints)
    - meta-task information (e.g., " . . . bids must be submitted by . . . ")

- The ***announcement is then broadcast***.



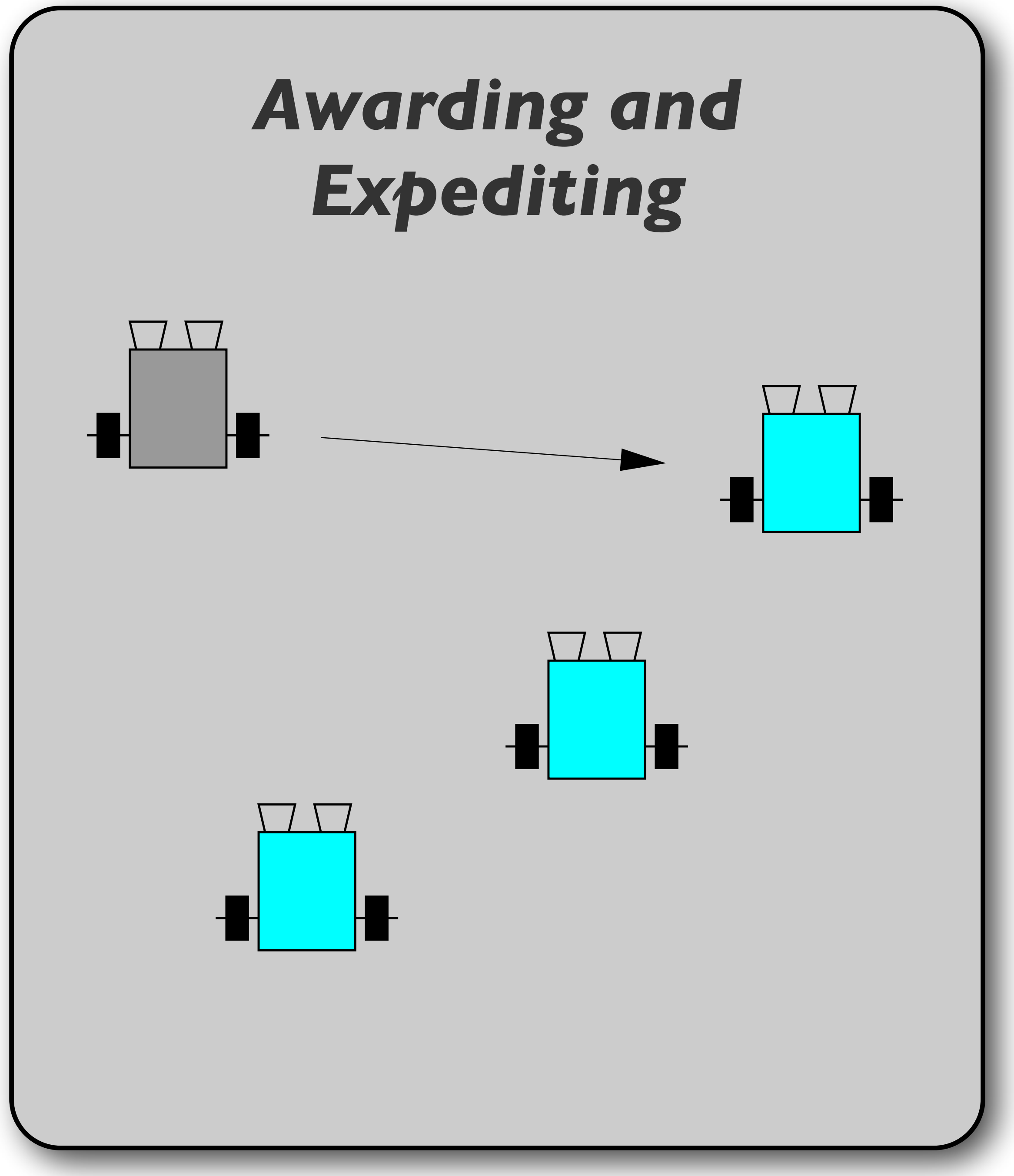
Bidding

- 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***.



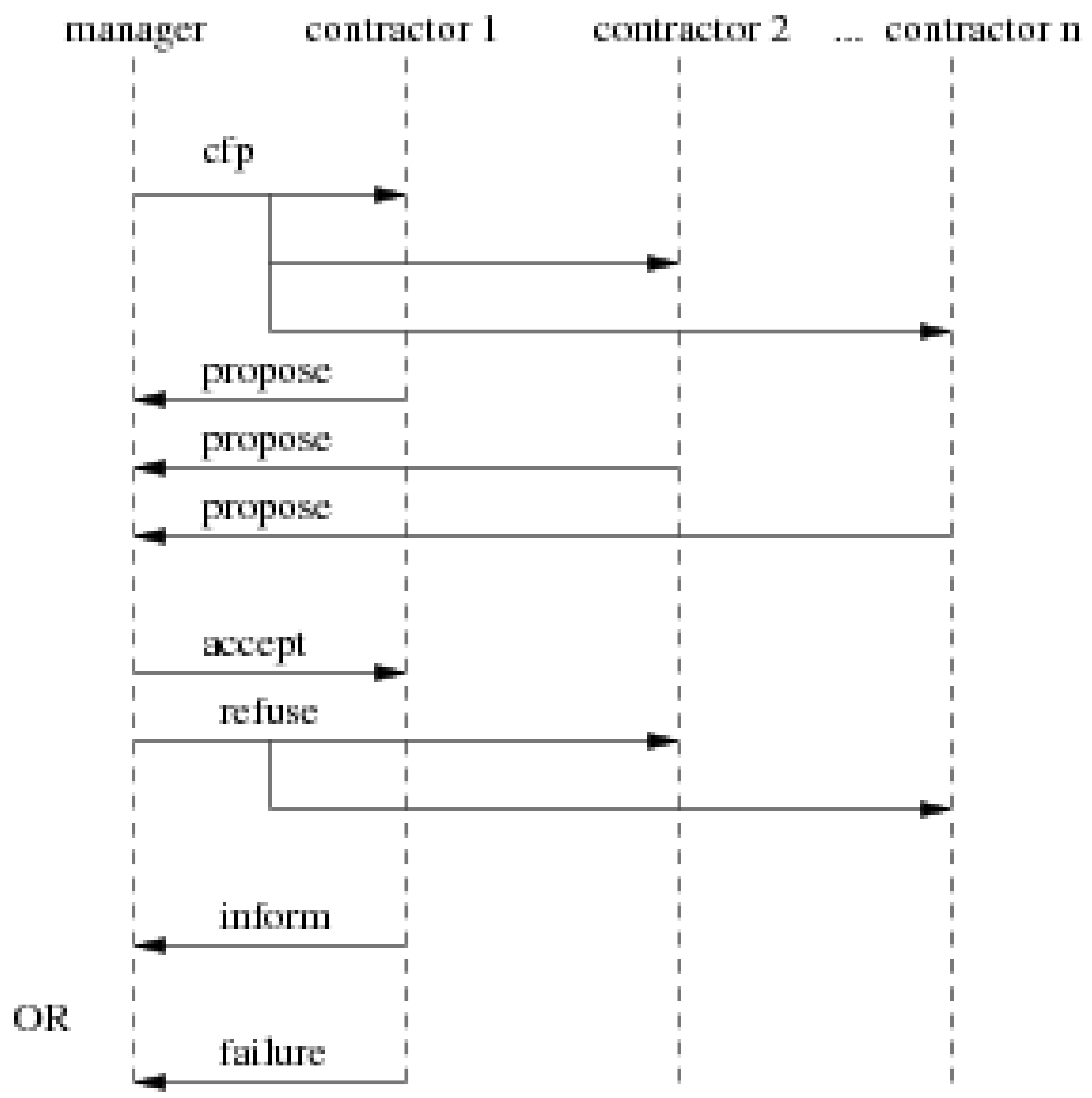
Awarding & Expediting

- 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.

```
MAS contractNetProtocol {
        infrastructure: Centralised

 agents:
        contractor              // The CNP Initiator
              [mindinspector="gui(cycle,html,history)"];
        participant #3;         // The 3 service providers
        refusenik;              // Participant who always refuse
        silentpartner;          // A Participant that doesn't answer

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

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,_,_).
```

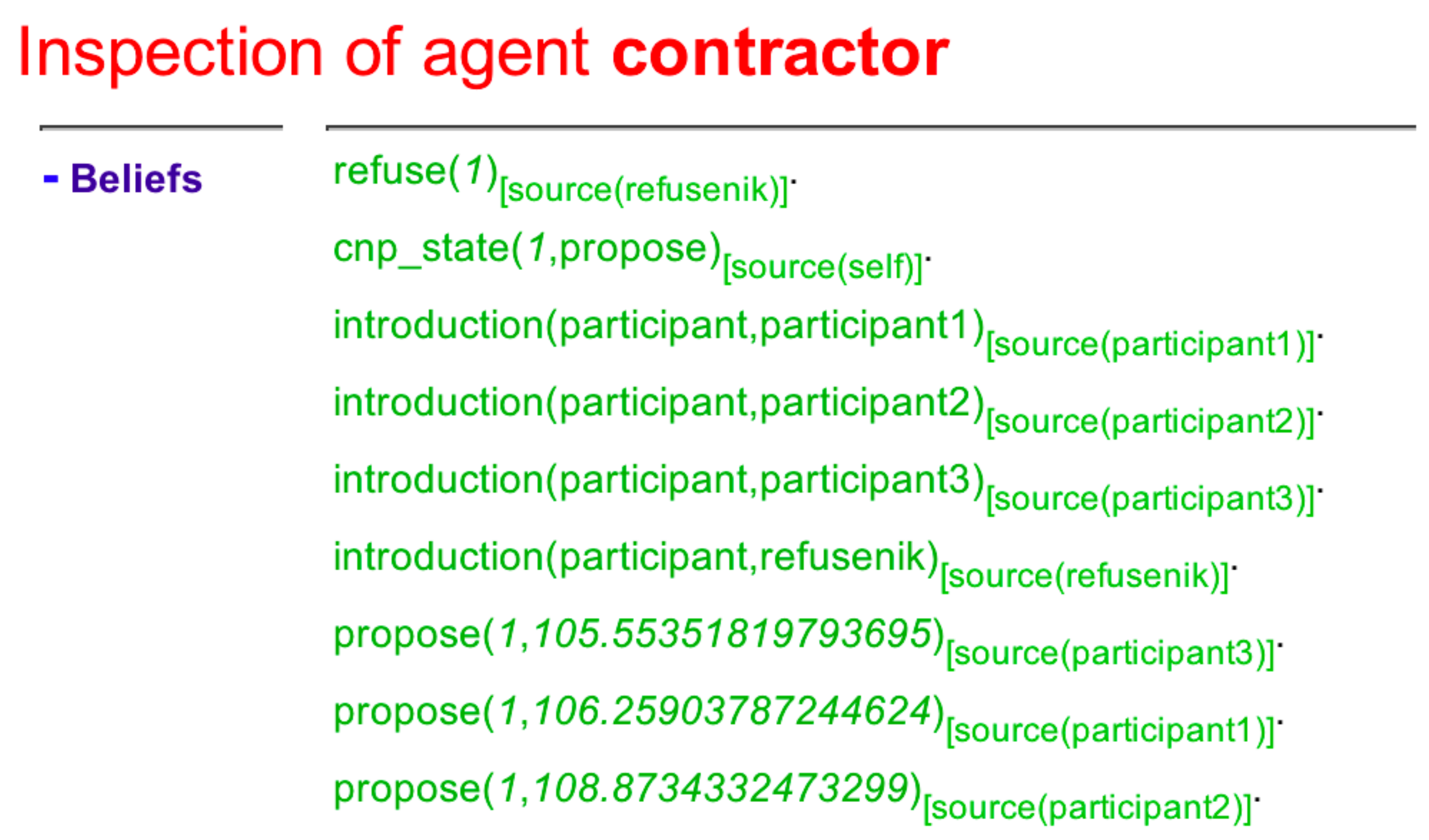
CNP in Jason: contractor

- 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)

```
// Agent contractor in project contractNetProtocol

// Initial beliefs and rules
all_proposals_received(CNPId)
        :-      .count(introduction(participant,_),NP) &
                            // number of participants
                .count(propose(CNPId,_)[source(_)], NO) &
                        // number of proposes received
                .count(refuse(CNPId)[source(_)], NR) &
                    // number of refusals received
                NP = NO + NR.

// Initial goals
!startCNP(1,fix(computer)).

//!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!!!



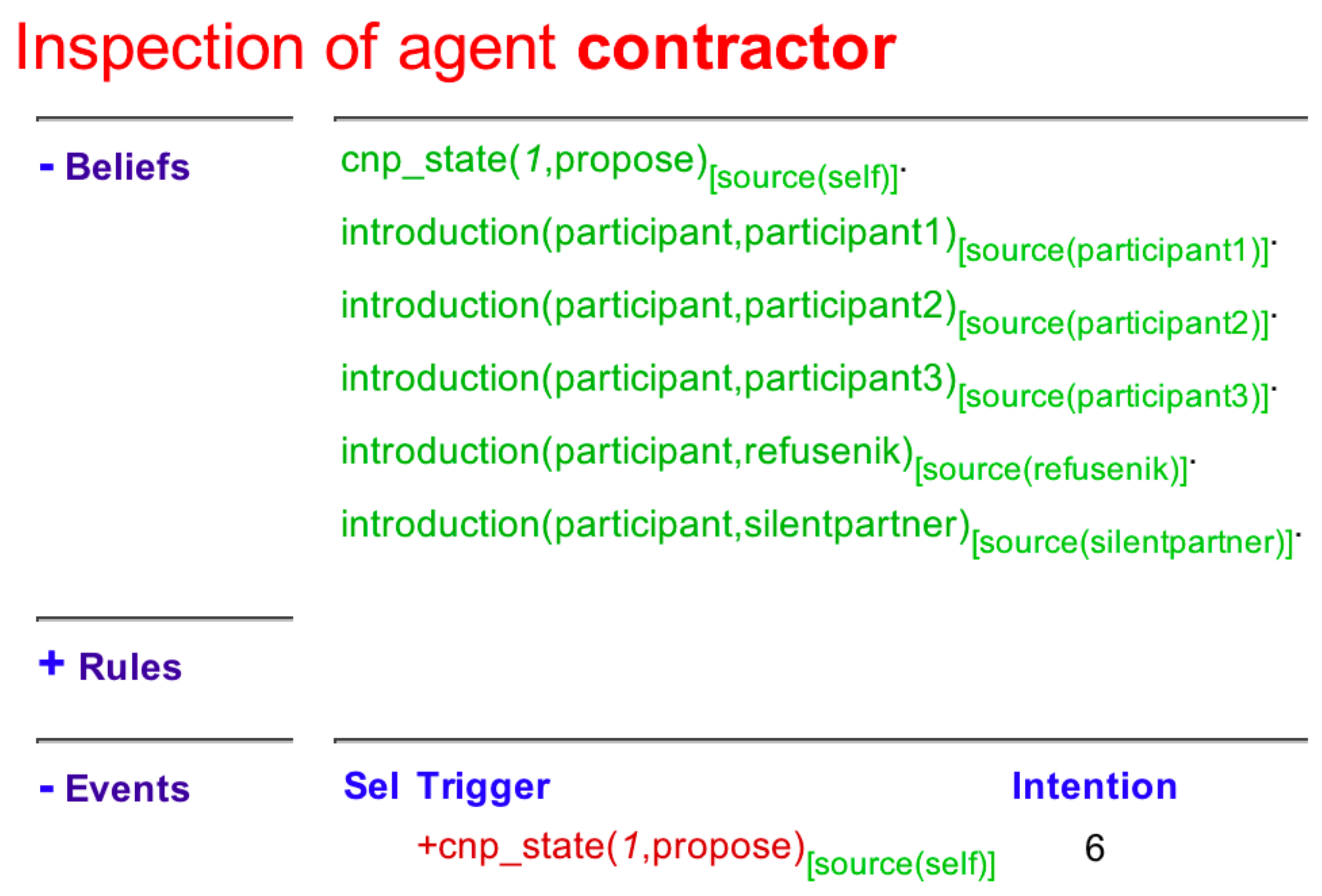
Inspection of agent **contractor**

- 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

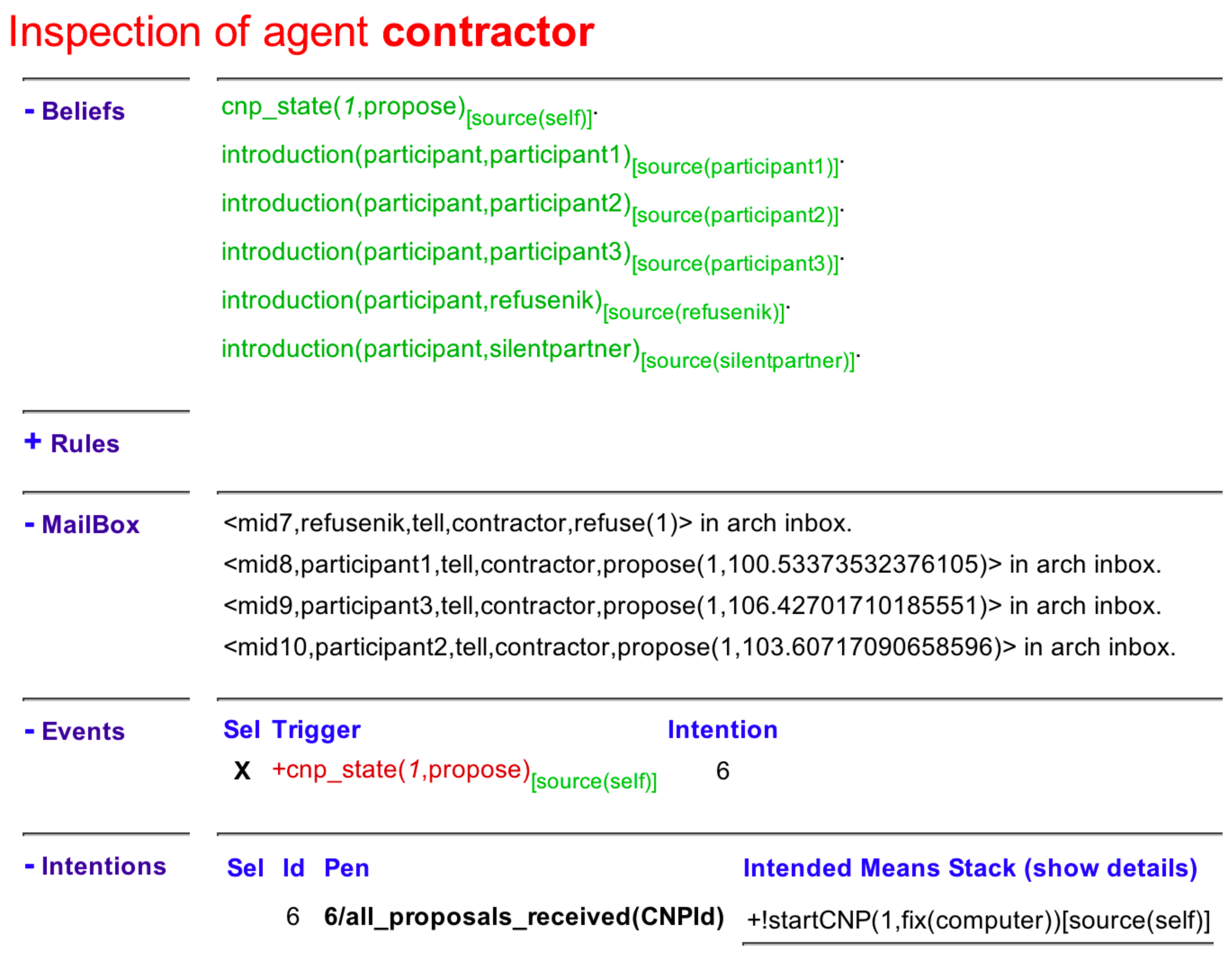
```
1.   // Agent contractor in project contractNetProtocol
2.   ...

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(1,106.42701710185551)> 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)]

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

// start the CNP
+!startCNP(Id,Task)
        <-   .print("Waiting participants for task ",Task,"...");
             .wait(2000);  // wait participants introduction
             // remember the state of the CNP
             +cnp_state(Id,propose);
             .findall(Name,introduction(participant,Name),LP);
             .print("Sending CFP to ",LP);
             .send(LP,tell,cfp(Id,Task));
             // the deadline of the CNP is now + 4 seconds
             // (or all proposals were received)
             .wait(all_proposals_received(CNPId), 4000, _);
             !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!").
```

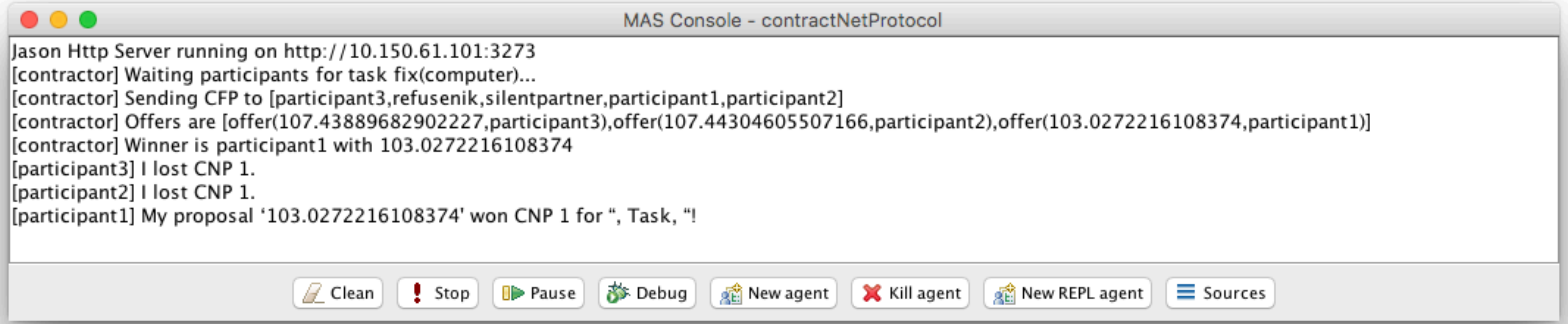
CNP in Jason: contractor

- 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).
```

CNP in Jason

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



```
Jason Http Server running on http://10.150.61.101:3273
[contractor] Waiting participants for task fix(computer)...
[contractor] Sending CFP to [participant3,refusenik,silentpartner,participant1,participant2]
[contractor] Offers are [offer(107.43889682902227,participant3),offer(107.44304605507166,participant2),offer(103.0272216108374,participant1)]
[contractor] Winner is participant1 with 103.0272216108374
[participant3] I lost CNP 1.
[participant2] I lost CNP 1.
[participant1] My proposal '103.0272216108374' won CNP 1 for ", Task, "!
```

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 τ 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

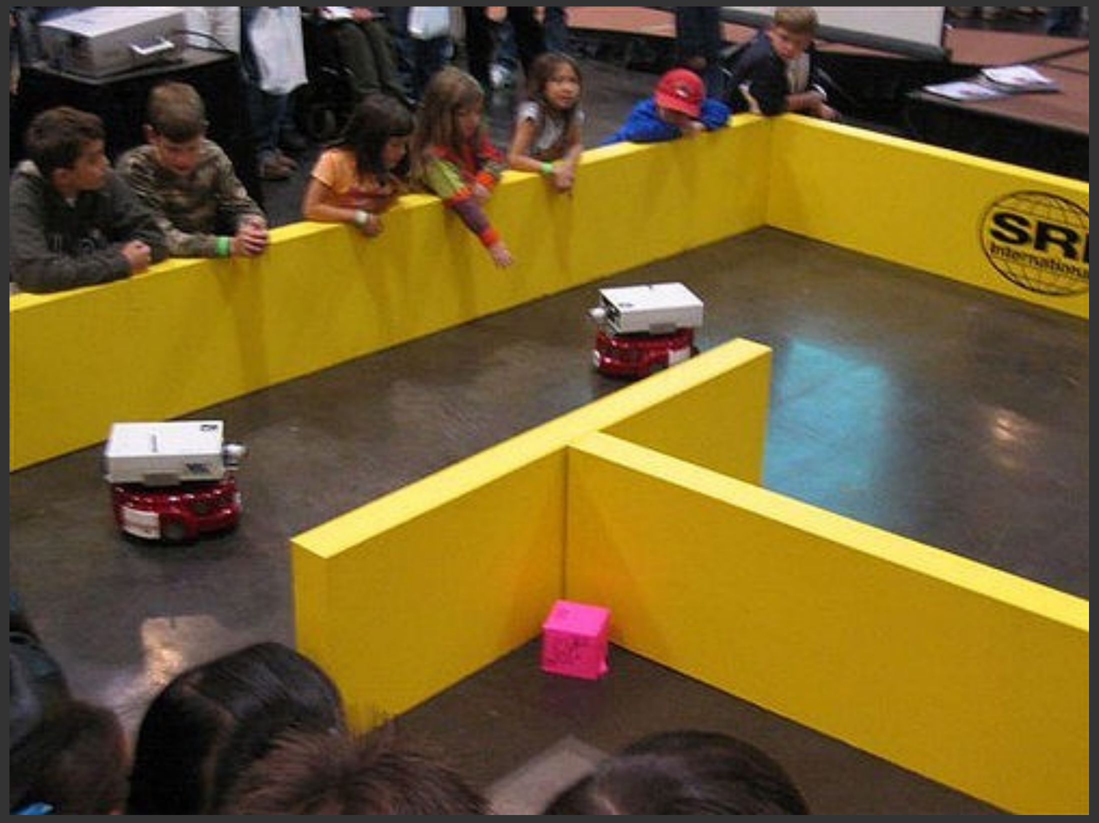
- In results sharing, agents provide each other with information as they work towards a solution.
- It is generally accepted that results sharing improves problem solving by:
  - Independent pieces of a solution can be cross-checked.
  - Combining local views can achieve a better overall view.
  - Shared 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.

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*.

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.



The Centibots robots collaborate to map a space and find objects.

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.

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?

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.

$Ag : \mathcal{R}^E \rightarrow Ac$

- 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 α cannot be done in any state in E'.

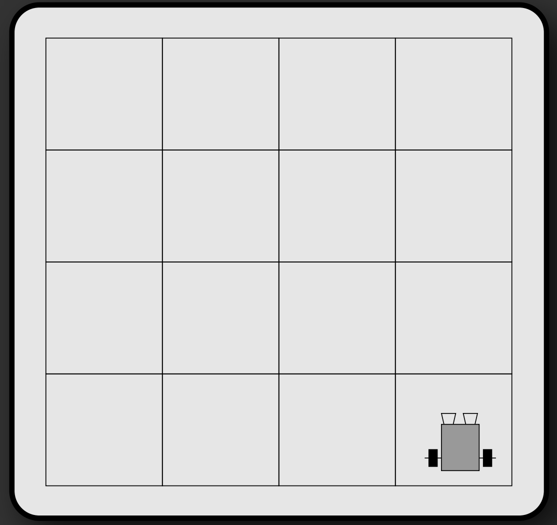
$<E', \alpha>$

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

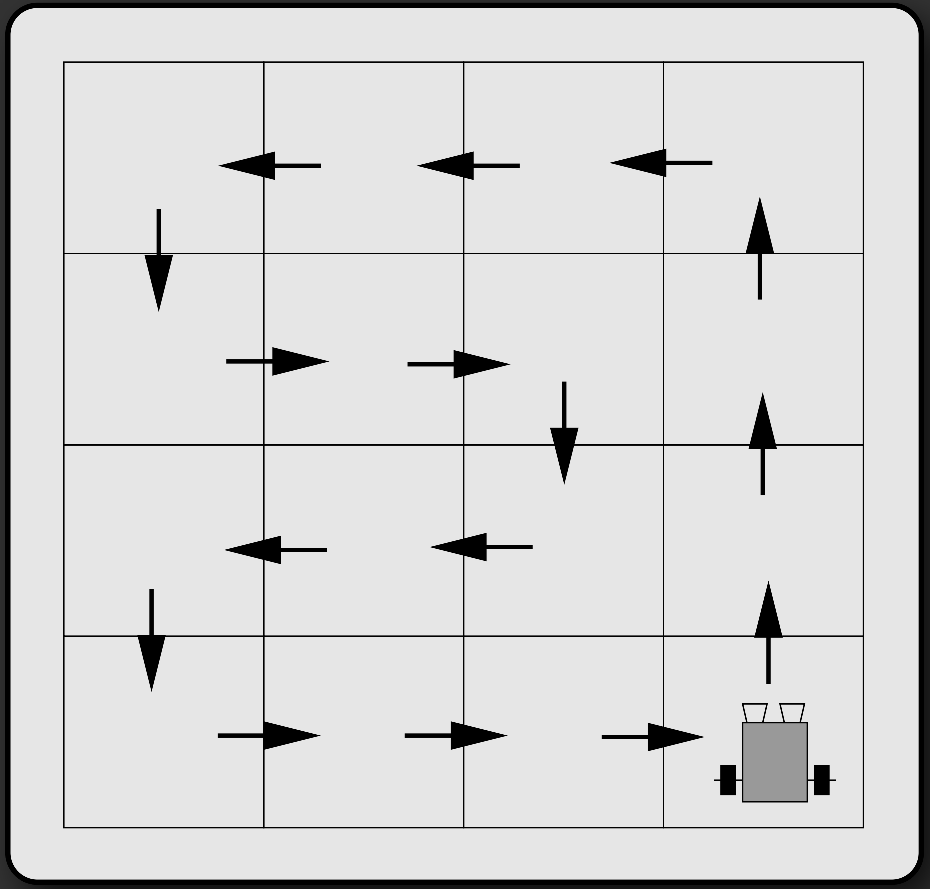
Offline Design

- 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 (On^2 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

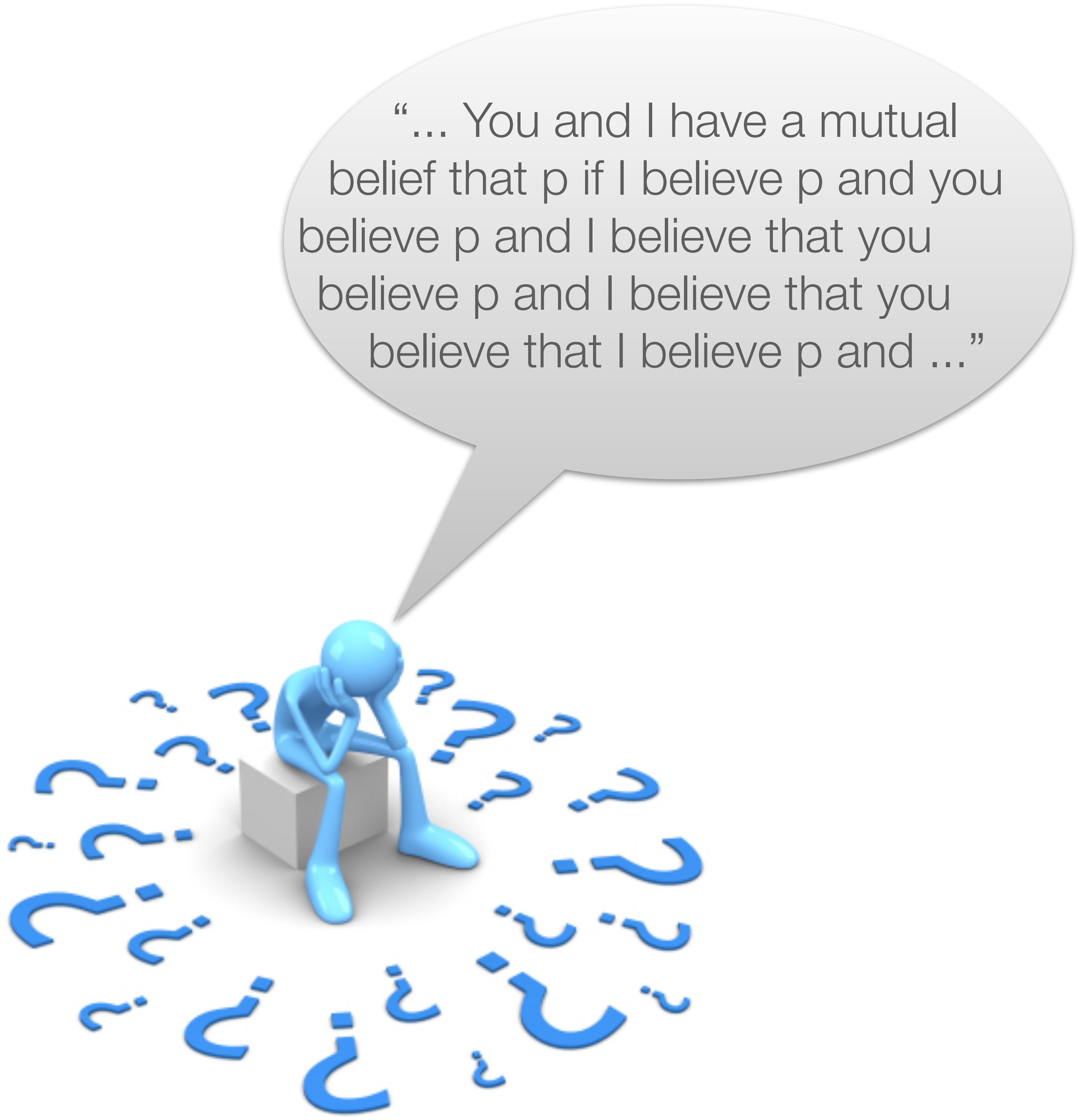
- Just as we have individual intentions, we can have joint intentions for a team of agents.

- Levesque defined the idea of ***a joint persistent goal*** (JPG).
  - A group of agents have a collective commitment to bring about some goal φ, “move the couch”.
  - Also have motivation ψ, “Simon wants the couch moved”.

- The mental states of agents mirror those in BDI agents.
  - Agents don’t believe that φ is satisfied, but believe it is possible.
  - Agents maintain the goal φ until a termination condition is reached.

Joint Intentions

- The terminations condition is that it is mutually believed that:
  - goal φ is satisfied; or
  - goal φ is impossible; or
  - the motivation ψ 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.


"... You and I have a mutual belief that p if I believe p and you believe p and I believe that you believe p and I believe that you believe that I believe p and ..."

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.

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.