Using *Jason* to Implement a Team of Gold Miners

Rafael H. Bordini[1], Jomi F. Hübner[2], and Daniel M. Tralamazza[3]

[1] Department of Computer Science
University of Durham
Durham DH1 3LE, U.K.
`R.Bordini@durham.ac.uk`

[2] Departamento de Sistemas e Computação
Universidade Regional de Blumenau
Blumenau, SC 89035-160, Brazil
`jomi@inf.furb.br`

[3] School of Computer & Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland
`daniel.tralamazza@epfl.ch`

Abstract. This paper describes a team of agents that took part in the second CLIMA Contest. The team was implemented in a logic-based language for BDI agents and was run in a Java-based interpreter that makes it easy for legacy code to be invoked from within the agents' practical reasoning. Even though the implementation was not completely finished in time, the team won the competition, and the experience also allowed us to improve various aspects of the interpreter.

1 Introduction

In this paper, we describe the development of a team of agents created to enter the second CLIMA Contest, which took place with CLIMA VII. We quote bellow the general description of the scenario (Figure 1 shows a screenshot).

> Recently, rumours about the discovery of gold scattered around deep Carpathian woods made their way into the public. Consequently hordes of gold miners are pouring into the area in the hope to collect as much of gold nuggets as possible. Two small teams of gold miners find themselves exploring the same area, avoiding trees and bushes and competing for the gold nuggets spread around the woods. The gold miners of each team coordinate their actions in order to collect as much gold as they can and to deliver it to the trading agent located in a depot where the gold is safely stored.
> (`http://cig.in.tu-clausthal.de/CLIMAContest/`)

The first important characteristic of the team described here is that the agents were programmed in AgentSpeak, an agent-oriented programming language based on logic programming and suitable for (BDI) reactive planning systems (the language is briefly explained in Section 2). An interpreter for an extended version of AgentSpeak called ***Jason*** was used to run the agents (see Section 3). Section 4 presents the overall team specification and Section 5 gives further details of the implementation.

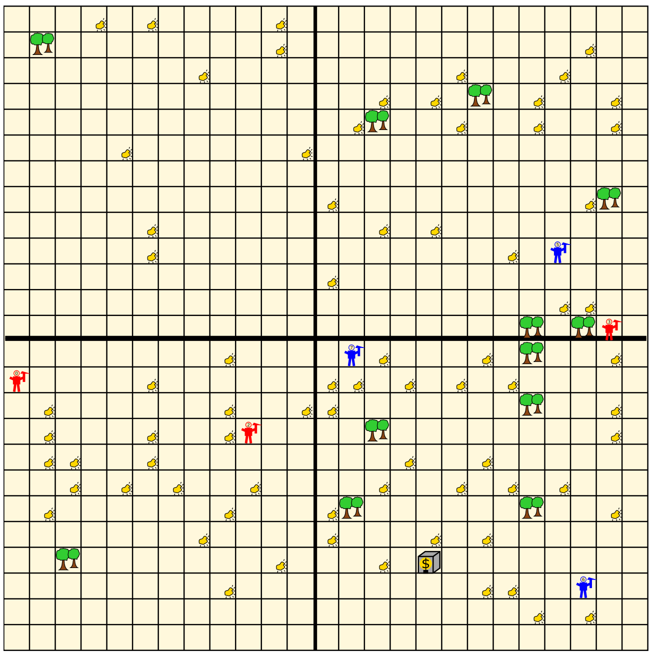

Fig. 1. The Contest Scenario and the Quadrants Used by our Team.

2 AgentSpeak

The AgentSpeak(L) programming language was introduced in [7]. It is based on logic programming and provides an elegant abstract framework for programming BDI agents. The BDI architecture is, in turn, the predominant approach to the implementation of *intelligent* or *rational* agents [8], and a number of commercial applications have been developed using this approach.

An AgentSpeak agent is defined by a set of *beliefs* giving the initial state of the agent's *belief base*, which is a set of ground (first-order) atomic formulæ, and a set of plans which form its *plan library*. An AgentSpeak plan has a *head* which consists of a triggering event (specifying the events for which that plan is *relevant*), and a conjunction of belief literals representing a *context*. The conjunction of literals in the context must be a logical consequence of that agent's current beliefs if the plan is to be considered *applicable* when the triggering event happens (only applicable plans can be chosen for execution). A plan also has a *body*, which is a sequence of basic actions or (sub)goals that the agent has to achieve (or test) when the plan is triggered. *Basic actions* represent the atomic operations the agent can perform so as to change the environment. Such actions are also written as atomic formulæ, but using a set of *action symbols* rather than predicate symbols. AgentSpeak distinguishes two types of *goals*: achievement goals and test goals. Achievement goals are formed by an atomic formulæ prefixed with the '**!**' operator, while test goals are prefixed with the '**?**' operator. An *achievement goal* states that the agent wants to achieve a state of the world where the associated atomic formulæ is true. A *test goal* states that the agent wants to test whether the associated atomic formulæ is (or can be unified with) one of its beliefs.

An AgentSpeak agent is a *reactive planning system*. Plans are triggered by the *addition* ('**+**') or *deletion* ('**-**') of beliefs due to perception of the environment, or to the addition or deletion of goals as a result of the execution of plans triggered by previous events.

```
+green_patch(Rock)
   :  not battery_charge(low)
   <- ?location(Rock,Coordinates);
      !traverse(Coordinates);
      !examine(Rock).
+!traverse(Coords)
   :  safe_path(Coords)
   <- move_towards(Coords).
+!traverse(Coords)
   :  not safe_path(Coords)
   <- ...
```

Fig. 2. Example of AgentSpeak Plans.

A simple example of an AgentSpeak program for a Mars robot is given in Figure 2. The robot is instructed to be especially attentive to "green patches" on rocks it observes while roving on Mars. The AgentSpeak program consists of three plans. The first plan says that whenever the robot perceives a green patch on a certain rock (a belief addition), it should try and examine that particular rock. However this plan can only be used (i.e., it is only applicable) if the robot's batteries are not too low. To examine the rock, the robot must retrieve, from its belief base, the coordinates it has associated with that rock (this is the reason for the test goal in the beginning of the plan's body), then achieve the goal of traversing to those coordinates and, once there, examining the rock. Recall that each of these achievement goals will trigger the execution of some other plan.

The two other plans (note the last one is only an excerpt) provide alternative courses of action that the rover should take to achieve a goal of traversing towards some given coordinates. Which course of action is selected depends on its beliefs about the environment at the time the goal-addition event is handled. If the rover believes that there is a safe path in the direction to be traversed, then all it has to do is to take the action of moving towards those coordinates (this is a basic action which allows the rover to effect changes in its environment, in this case physically moving itself). The alternative plan (not shown here) provides an alternative means for the agent to reach the rock when the direct path is unsafe.

3 *Jason*

The ***Jason*** interpreter implements the operational semantics of AgentSpeak as given in, e.g., [4]. ***Jason*** [4] is written in Java, and its IDE supports the development and execution of distributed multi-agent systems [3, 2]. Some of the features of ***Jason*** are:

[4] ***Jason*** is *Open Source* (GNU LGPL) and is available from `http://jason.sourceforge.net`

- speech-act based inter-agent communication (and annotation of beliefs with information sources);
- the possibility to run a multi-agent system distributed over a network (using SACI or some other middleware);
- fully customisable (in Java) selection functions, trust functions, and overall agent architecture (perception, belief-revision, inter-agent communication, and acting);
- straightforward extensibility and use of legacy code by means of user-defined “internal actions” implemented in Java;
- clear notion of *multi-agent environments*, which can be implemented in Java (this can be a simulation of a real environment, e.g., for testing purposes before the system is actually deployed).

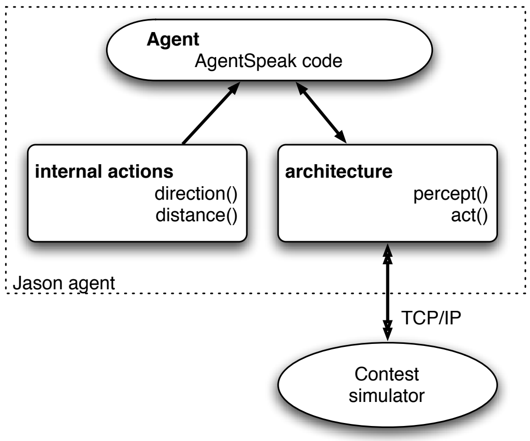

Fig. 3. Agent Extensibility and Customisation.

To implement our agent team, two of these features were specially useful: architecture customisation and internal actions (see Figure 3). A customisation of the agent architecture is used to interface between the agent and its environment. The environment for the CLIMA Contest was implemented by the contest organisers in a remote server that simulates the mining field, sending perception to the agents and receiving requests for action execution. Therefore, when an agent attempts to perceive the environment, the architecture sends to it the information provided by the simulator, and when the agent chooses an action to be performed, the architecture sends the action execution request to the simulator. For example, the plan

```
+pos(X,Y) : Y > 0 <- up.
```

is triggered when the agent perceives its position and its current line in the world grid is greater than zero. The `+pos(X,Y)` percept is produced by the architecture from the messages sent by the simulator, and `up` is an action that the architecture sends to the simulator.

Although most of the agent is coded in AgentSpeak, some parts were implemented in Java, in this case because we wanted to use legacy code, in particular, we already had a Java implementation of the A* search algorithm, which we used to find paths in instances of the simulated scenario (it is interesting to note that in one of the “maze” scenarios used in the competition, our team was the only to successfully find a path to the depot). This algorithm was made accessible to the agents by means of *internal actions*. These were used in AgentSpeak plans as shown in the example below:

```
+gold(X,Y): pos(X,Y) & depot(DX,DY) & carrying_gold
   <- pick; jia.direction(X,Y,DX,DY,Dir); Dir.
```

In this plan, when the agent perceives some gold in its current position, it picks up the gold and calls the `direction` internal action of the `jia` library. This internal action receives two locations as parameters ($\langle$X,Y$\rangle$ and $\langle$DX,DY$\rangle$), computes a path between them using A* (using the Manhattan distance as heuristic, as usual in scenarios such as this), and instantiates `Dir` with the first action (`up`, `down`, `left`, or `right`) according to the path it found from the first to the second coordinate. The plan then says that the agent should perform the action instantiated to variable `Dir`. Note that this plan is illustrative, it does not generate the behaviour of carrying the gold to the depot; only one step towards it is performed in the excerpt above. Also, as this is a cooperative team and each agent has only a partial view of the environment, the underlying architecture ensures that the agents share all the information about obstacles which the A* algorithms uses for navigation.

4 Overall Team Strategy

The team is composed of two roles enacted by five agents. The miner role is played by four agents who will have the goal of finding gold and carrying it to the depot. The team also has one agent playing the leader role; its goals are to allocate agents' quadrants and allocate free agents to a piece of gold that has been found. The leader agent helps the team, but each team must have exactly four miner agents that log into the simulation server as official contestants, so the leader is not registered with the server. The diagrams in Figures 4, 5, and 6 give an overview of the system and the two roles using the Prometheus methodology [6].

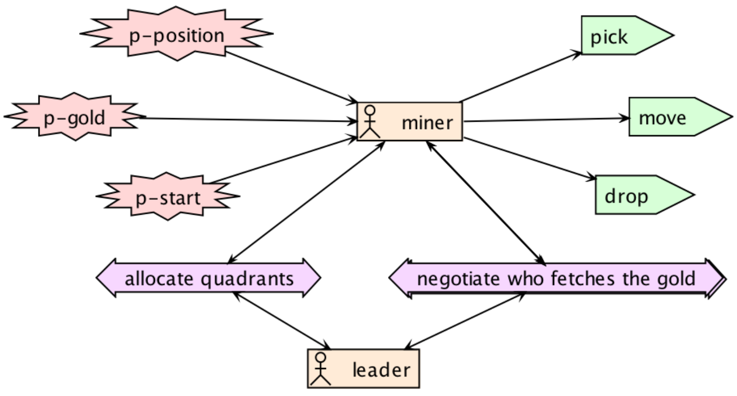

Fig. 4. System overview diagram.

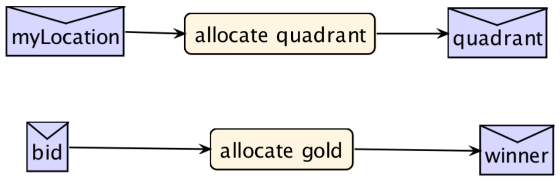

Fig. 5. Leader agent specification.

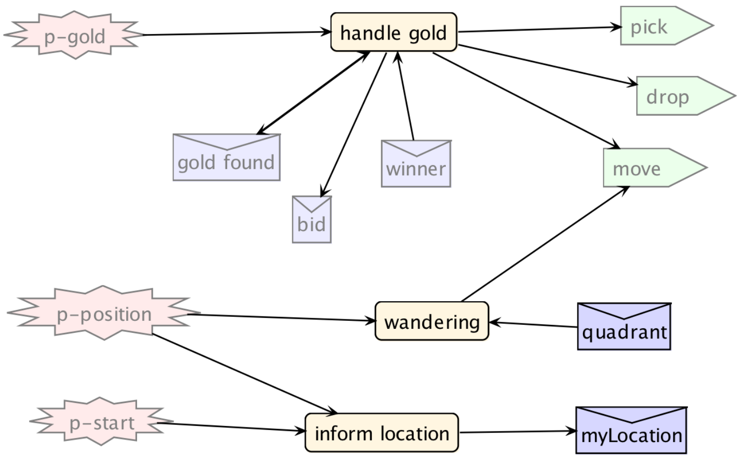

Fig. 6. Miner agent specification.

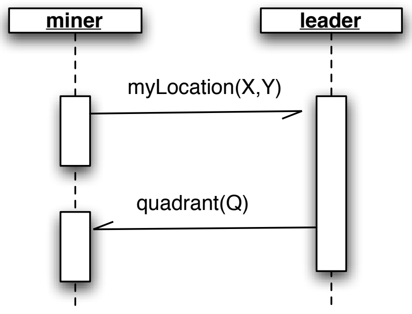

Fig. 7. Quadrant allocation protocol.

The overall strategy of the ***Jason*** team is as follows. Each miner is responsible for systematically (rather than randomly) searching gold *within* one quadrant of the environment (see Figure 9). Since the initial positions of the agents are only known when the game starts, the allocation of the agents to quadrants depends on such positions. The team uses the protocol in figure 7 for the leader to allocate each agent to a quadrant. At the beginning of each game, the four miner agents send their location to the leader agent, the leader allocates each quadrant to an agent by checking which agent is nearest to that quadrant, and sends a message to each agent saying which quadrant they have been allocated. We have decided to centralise some decisions in a leader agent so as to decrease the number of required messages in a distributed negotiation; even though all agents were run in the same machine in the actual competition, this is particularly important if in future competitions we decide to run agents in different machines, in case agents become too elaborate to run them all in one machine.

Another (simple) protocol is used to decide which agent will commit to gold found by a miner that is already carrying gold (Figure 8). When a miner (e.g., miner1 in Figure 8) sees a piece of gold and cannot pick it up (e.g., because it is already carrying gold), it broadcasts the location of the piece of gold just found to all agents. The other agents bid to take on the task of fetching that piece of gold; such bid is computed based on its availability and distance to that gold. All bids are sent to the leader who then chooses the best agent to commit to collect that piece of gold.

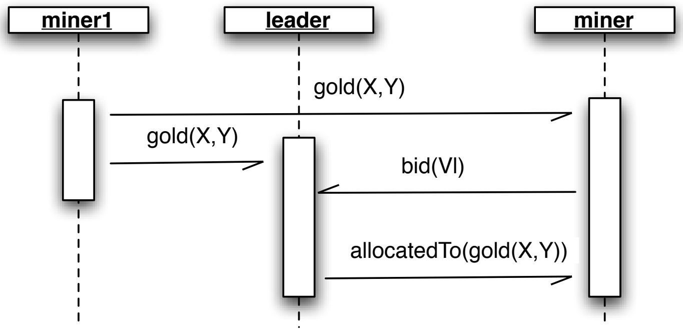

Fig. 8. Gold Allocation Protocol.

5 Implementation in *Jason*

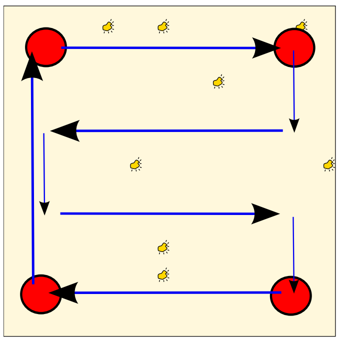

Fig. 9. Miner's Search Path Within its Quadrant.

The miner agents have two mutually exclusive goals: "find gold" or "handle gold". Whenever the agent has currently no other intention, it adopts the goal of exploring its own quadrant to find gold. When the agent either perceives some gold or was allocated to a piece of gold by the protocol in Figure 8, it gives up the "find gold" goal and commits to the goal of handling that particular piece of gold. When this latter goal is achieved, the agent commits again to the "find gold" goal.

To systematically find gold in its quadrant, the miner "scans" its quadrant as illustrated in Figure 9. The plans for achieving this goal determine that the agent should start from the place where it last stopped searching for gold, or start from the position in its quadrant which is closest to the depot. As the agent can perceive gold in all neighbouring cells, it can skip three lines when moving vertically.

When a miner sees a piece of gold, three relevant plans can be selected as applicable depending on the following conditions (the AgentSpeak code is shown in Figure 10):

1. The first plan is applicable when the miner is not carrying gold and is free. [5] The plan execution consist of removing the belief that it is free, adding a belief that there is gold at that location, and creating a goal to handle that gold.
2. The second plan is applicable when the miner is also not carrying gold but is not free because it is going to position `OldX,OldY` to collect some gold there. In this case, it prefers the gold just found, so the agent: (i)drops the previous intention; (ii) announces the availability of gold at the "old" location to the other agents (this will trigger again the allocation protocol in Figure 7); and (iii) creates a goal to handle the piece of gold it has just found.
3. If none of the above plans is applicable (i.e., the agent is carrying gold), the third alternative plan is used to announce the gold location to others agents, starting the allocation protocol (shown in Figure 7).

The last three plans in Figure 10 implement part of the allocation protocol. When the agent receives a message with some gold position, if it is free, it sends a bid based on its Manhattan distance to the gold; otherwise, it sends a very high bid. When some gold is allocated by the leader to the agent, it handles this gold if it is still free. Note that this is not an optimal strategy: we have not as yet dealt with the possibility that reallocating tasks to the agents that are already committed (i.e., no longer free) might lead to a better overall task allocation; in the future, we might use some DCOP (Distributed Constraint Optimisation Problem) algorithm for this.

The plan to achieve the goal of handling a found piece of gold[6] is shown in Figure 11. The plan initially drops the goal of finding gold (exploration behaviour), moves the agent to the gold position, picks the gold, announces to others that the gold was collected so they do not try to fetch this gold (to avoid agents moving to pieces of gold that are no longer there), retrieves the depot location from the belief base, moves the agent to the depot, drops the gold, and finally chooses another gold to pursue. In case the handle-gold plan fails (e.g., because the gold disappeared due to the environment being dynamic), the event `-!handle(G)` is created and the second plan is selected. This plan just removes the information about that gold location from the belief base and chooses another piece of gold to be collected. The `choose_gold` plans find the nearest known gold and create a gold to handle it; if no gold location is known, the agent is free and resumes the gold-searching behaviour.

It is important to note that AgentSpeak is the language used to define the high-level (practical) reasoning of the agents. The use of internal actions facilitates keeping the agent language at the right level of abstraction, even when legacy code needs to be invoked.

[5] The agent maintains a belief stating whether it is currently "free" or not. Being free means that the agent is not committed to handling any piece of gold.

[6] Note that only the most important plans are included here; the complete code is available in the ***Jason*** web site.

```
+cell(X,Y,gold) : not carrying_gold & free
  <- -free; +gold(X,Y);
     !handle(gold(X,Y)).
+cell(X,Y,gold) : not carrying_gold & not free &
                  .desire(handle(gold(OldX,OldY))) &
  <- +gold(X,Y);
     .dropIntention(handle(gold(_,_)));
     .broadcast(tell,gold(OldX,OldY));
     !handle(gold(X,Y)).
+cell(X,Y,gold) : not committed(gold(X,Y))
  <- +gold(X,Y);
     .broadcast(tell,gold(X,Y)).

+gold(X1,Y1)[source(A)] : A \== self & free & pos(X2,Y2)
  <- jia.dist(X1,Y1,X2,Y2,Dist);
     .send(leader,tell,bidFor(gold(X1,Y1),Dist)).
+gold(X1,Y1)[source(A)] : A \== self
  <- .send(leader,tell,bidFor(gold(X1,Y1),1000)).

+allocatedTo(Gold,Me)[source(leader)]
  :  .myName(Me) & free // I am still free
     <- -free; !handle(Gold).
```

Fig. 10. Relevant Plans for When Gold is Perceived or Allocated.

```
+!handle(gold(X,Y)) : true
  <- .dropIntention(explore(_,_));
     !pos(X,Y);   !ensure(pick);
     .broadcast(tell,picked(gold(X,Y)));
     ?depot(_,DX,DY);
     !pos(DX,DY); !ensure(drop);
     -gold(X,Y);  !choose_gold.

-!handle(Gold) : true  <- -Gold; !choose_gold.

+!choose_gold : not gold(_,_) <- +free.
+!choose_gold : gold(_,_)
  <- .findall(gold(X,Y),gold(X,Y),LG);
     !calcGoldDistance(LG,LD);
     // LD is a list of terms d(Distance,gold(X,Y))
     !min(LD,d(Distance,NewGold));
     !handle(NewGold).

+!pos(X,Y) : pos(X,Y) <- true.
+!pos(X,Y) : not pos(X,Y) & pos(AgX,AgY)
  <- jia.getDirection(AgX, AgY, X, Y, D);
     D; !pos(X,Y).
```

Fig. 11. Plans to Handle a Piece of Gold.

6 Conclusion

The AgentSpeak code for the team of gold miners, in our opinion, is a quite elegant solution, being declarative, goal-based (based on the BDI architecture), and also neatly allowing agents to have long term goals while reacting to changes in the environment. The ***Jason*** interpreter provided good support for high-level (speech-act based) communication, transparent integration with the contest server, and for use of existing Java code (e.g., for the A* algorithm). Although not a "purely" declarative, logic-based approach, the combination of both declarative and legacy code was quite efficient without compromising the declarative level (i.e., the agent's practical reasoning which is the specific level for which AgentSpeak is an appropriate language).

On the other hand, using a new programming paradigm [1] is never easy, and we also faced difficulties with ***Jason*** being a new platform that had some features that had never been thoroughly tested before. The development of a ***Jason*** team was a good experience not only in the result of the competition but also for experimenting with multi-agent programming and the improvements of the ***Jason*** platform that ensued. The scenarios where our team did not do so well were the ones with highest uncertainty; we still need more work and experience taking this type of scenario into consideration. In future versions of this team, we plan to avoid the use of centralised negotiation (which has the leader as a single point of failure) and to use $\mathcal{M}$oise+ [5] to create an organisation with the specification of the roles in the system. In our original strategy, there was yet another role which was that of the "courier"; in case the depot happens to be in a position too far from the some of the quadrants, the courier would help carry to the depot pieces of gold from agents that are in the more distant quadrants. We also plan to experiment with DCOP algorithms for optimal allocation of agents to collect pieces of gold.

References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer, 2005.
2. R. H. Bordini, J. F. Hübner, et al. ***Jason***, manual version 0.8, Mar 2006. `http://jason.sourceforge.net/`.
3. R. H. Bordini, J. F. Hübner, and R. Vieira. ***Jason*** and the golden fleece of agent-oriented programming. In Bordini et al. [1], chapter 1, pages 3–37.
4. R. H. Bordini and Á. F. Moreira. Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226, Sept. 2004.
5. J. F. Hübner, J. S. Sichman, and O. Boissier. Using the Moise+ for a cooperative framework of MAS reorganisation. In *Proc. of 17th SBIA*, LNAI 3171, pages 506–515. Springer, 2004.
6. L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.
7. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of MAAMAW'96*, LNAI 1038, pages 42–55. Springer 1996.
8. M. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, MA, 2000.