Developing a Team of Gold Miners Using *Jason*

Jomi F. Hübner[1] and Rafael H. Bordini[2]

[1] G2I – ENS Mines Saint-Etienne
158 Cours Fauriel
42023 Saint-Etienne Cedex, France
Jomi.Hubner@emse.fr

[2] Department of Computer Science
University of Durham
Durham DH1 3LE, UK
R.Bordini@durham.ac.uk

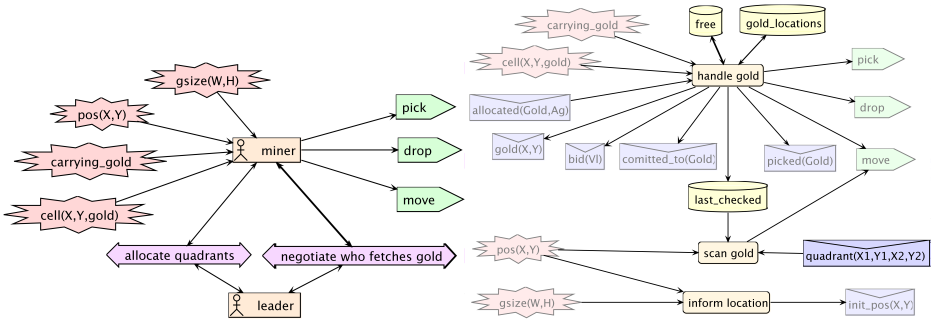
1 Introduction

This document gives an overview of a multi-agent system formed by a team of gold miners to compete in the Multi-Agent Programming Contest 2007 (the "gold miners" scenario). One of the main objectives has been to test and improve ***Jason***, the interpreter for an agent programming language used to implement the MAS. ***Jason*** [2, 4] is an agent platform based on an extension of an agent-oriented programming language called AgentSpeak(L) [6]. The language is inspired by the BDI architecture [7], hence based on notions such as beliefs, goals, plans, intentions, etc.

2 System Analysis and Design

One of the existing software engineering methodologies which we find particularly suitable for BDI agents is the Prometheus methodology [5]. Figures 1(a) and 1(b) are use the notation of that methodology to briefly give an idea of the overall system and the miner agent design, respectively. The analysis and design of the system is based on our previous team that won the CLIMA Contest in 2006 [3]. There are two kinds of agents in the team: miners and leader. Miners are the agents that interact with the contest simulator and the leader helps the coordination of some activities.

The leader helps the miners to coordinate themselves in two situations. It initially divides the grid representing the environment into four quadrants and then allocates miners to them; the miners will therefore look for gold in different places. Since we have six agents and only four quadrants, the two agents without a specific quadrant will search for gold anywhere in the grid, preferring the places least visited by the others. The second situation of coordination is the negotiation process that is started when a miner sees a piece of gold and is not able to collect it (because its container is full). This miner broadcasts the gold location to other miners who then send bids to the leader. The leader chooses the best offer and allocate the corresponding agent to collect that piece of gold (Figure 2). The protocol also states that whenever some agent decides to go to some gold location, it should announce it to others (so that they can reconsider their intentions). Similarly, they should announce whenever they collect a piece of gold.



(a) System Overview (b) Miner Agent Overview

Fig. 1. ***Jason*** Team Design Diagrams.

All miners have the same individual goals:

search gold : search for gold in the environment. This goal is the initial goal of these agents and is also adopted when there is nothing else to do. Two strategies were used to achieve this goal. The first is used by agents that have a quadrant allocated to them and consists of scanning (i.e., searching systematically rather than randomly) for gold in the miner's quadrant. The second is used by "quadrant-less" agents and consists of always going the nearest least-visited location. For this latter strategy to work properly, all agents should inform the others about the places they are visiting.

fetch gold : go to the location of some known piece of gold and pick it up. This goal is adopted when the agent both has space in its container and knows of a "worthwhile" piece of gold. The piece of gold is known when the miner sees it or is informed about it by other miners (recall the gold negotiation protocol discussed above). The evaluation of the worth of a piece of gold is based on the path length from the agent to its location and that of the other agent possibly committed to the same piece. If there is no other agent committed, the piece is considered worthwhile. Otherwise, the distance to the piece, considering the agent's fatigue, must be less than the distance of the committed agent to the gold. This evaluation is also used to choose the gold to be fetched. To evaluate the other agents' distances to the gold, each agent should maintain the others informed of its location.

go to depot : go to the "depot" to drop there all pieces of gold being carried. This goal can only be adopted when the miner is carrying at least one piece of gold.

These goals are mutually exclusive and there is a preference relation between them: fetch $>$ go to depot $>$ search. To choose a goal to achieve at a certain moment in time, a miner follows this preference order, checking the adopt conditions for each of these alternative goals. Figure 4 shows an excerpt of the AgentSpeak code that implements the choice of a new goal, when that is necessary. The following events trigger the process of choosing a new goal to achieve: a new piece of gold is discovered through perception or communication; a piece of gold is allocated to the miner by the leader; some agent has picked or committed to a piece of gold the agent is currently fetching. Notice that to be allocated to fetch some gold does not necessarily imply that the agent will fetch that

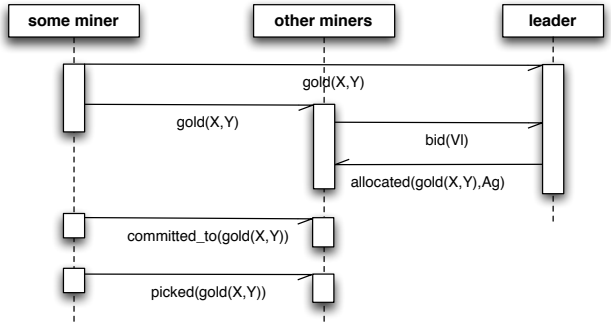


Fig. 2. Gold Allocation Protocol.

gold, it could be the case where the agent currently know that there is another better piece of gold for it to fetch than the one just allocated. The above events thus only trigger the attempt to choose a new goal to achieve and are not directly related to a particular goal adoption.

3 Software Architecture

To implement our team, two features of ***Jason*** 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 Agent Contest is implemented 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 customised architecture sends to the agent the information provided by the simulation server, and when the agent chooses an action to be performed, the architecture sends the action execution request also to the server. This architecture customisation also allow us to easily change between the contest simulation server and our (local) simulation by simply choosing another architecture; using a simulation running locally makes testing much faster and easier.

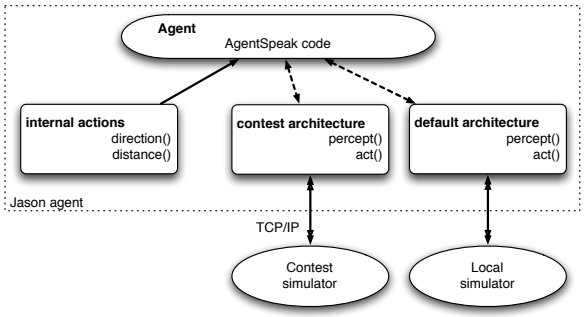


Fig. 3. Agent Architecture (reproduced from [2]).

Although most of the agent code was written in AgentSpeak, some parts were implemented in Java, in this case because we wanted to use some legacy code. In particular,

```
/* Plans to choose a new goal */
+!choose_goal
  :  container_has_space &               // I have space for more gold
     .findall(gold(X,Y),gold(X,Y),LG) &  // LG is all known golds
     evaluate_golds(LG,LD) &             // Evaluate golds in LG
     .length(LD,LLD) & LLD > 0 &         // Is there a gold to fetch?
     .min(LD,d(D,NewG,_)) &              // Get the nearest
      worthwhile(NewG)
  <- .print("Gold options are ",LD,". Next gold is ",NewG);
     !change_to_fetch(NewG).
+!choose_goal                            // there is no worthwhile gold
  :  carrying_gold(N) & N > 0
  <- !change_to_goto_depot.
+!choose_goal                // not carrying gold, is "free" to search gold
  <- !change_to_search.

/* Plans to change the goal to fetching some gold */
+!change_to_fetch(G)                     // nothing to do,
  :  .desire(fetch_gold(G)).             // I am already fetching that gold
+!change_to_fetch(G)
  :  .desire(goto_depot)                 // I am going to the depot
  <- .drop_desire(goto_depot);           // drop "goto_depot" first
     !change_to_fetch(G).
+!change_to_fetch(G)
  :  .desire(fetch_gold(OtherG))         // I am fetching another gold,
  <- .drop_desire(fetch_gold(OtherG));   // drop that goal
     !change_to_fetch(G).
+!change_to_fetch(G)                     // None of above conditions
  <- -free;                              // I am not free anymore
     !!fetch_gold(G).                    // Create the new goal fetch G
```

Fig. 4. Excerpt of AgentSpeak Code to Choose a New Goal to Achieve.

we already had a Java implementation of the A* search algorithm, which we use to find paths and calculate distances in the various scenarios of the competition. This algorithm was made accessible to the agents by means of *internal actions*. The more information (specially obstacles) about the scenario is available for A*, the better it performs. So when an agent sees an obstacle, it broadcasts this information to all agents so that they can update their world model accordingly (unlike in [3], we did *not* use shared memory for obstacle information in this implementation).

4 Discussion

The allocation protocol we used to assign new pieces of gold to agents is quite simple but efficient. All agents know all pieces of gold found by the team, who is committed to which gold, and the distance of the other agents to the gold locations. They can therefore calculate which is the best gold to fetch considering the others' options. Any novelty in the scenario may trigger the choice of a new (better) gold. Although this protocol requires a lot of information exchange, we did not note performance problems during the competition since the numbers of agents and golds are relatively small.

```
+!goto_depot
   <- ... plan to achieve
         the goal ...
-!goto_depot
  : <condition to repair>
  <-  <actions to repair>;
      !goto_depot.
-!goto_depot
  <- !!choose_goal.
```

Fig. 5. Failure Handling.

In this version of the team, we have emphasised the modelling and programming of the team by means of goals. This allows us to maintain a high abstraction level and a use good style in coding with the chosen programming language, as can be seen in the code shown in Figure 4. Regarding the set of goals, during the competition we noted that the preference order we have established is not ideal in all types of scenarios. Since the depot might be far from the agents, sometimes it is better to continue searching for gold instead of going to the depot (during this trip to the depot, the opponent team can discover more golds mines). We should evaluate this issue more carefully, taking the fatigue of the agent carrying the gold also into consideration.

The goal-based modelling we used also allows us to take advantage of the ***Jason*** features for handling plan failure. For instance, if the goal to go to depot fails for same reason, the agent may try to identify the problem and then chose another goal to achieve. Figure 5 contains a common pattern of code used to handle failures. Plans of the form $-!g$ in the figure are plans to handle a failure in achieving goal g.

5 Conclusion

The AgentSpeak code for the team of gold miners is, in our opinion, quite an elegant solution, being declarative, goal-based (based on the BDI architecture), and also adequately 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, the level for which AgentSpeak is an appropriate language).

On the other hand, using a new programming paradigm [1] is never easy, and ***Jason*** being a relatively new platform, some features had never been thoroughly tested before. The development of the ***Jason*** team was a good opportunity for experimenting with multi-agent programming and the improvements of the ***Jason*** platform that ensued.

References

1. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Springer-Verlag, 2005.
2. R. H. Bordini, J. F. Hübner, and M. Wooldrige. *Programming Multi-Agent Systems in AgentSpeak using* **Jason**. Wiley, 2007.
3. R. H. Bordini, J. F. Hübner, and D. M. Tralamazza. Using ***Jason*** to implement a team of gold miners. In *Proc. of CLIMA VII*, LNAI 4371, pp. 304–313. Springer-Verlag, 2006.
4. 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, pp. 3–37.
5. L. Padgham and M. Winikoff. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, 2004.
6. A. S. Rao. AgentSpeak(L): BDI agents speak out in a logical computable language. In *Proc. of MAAMAW'96*, LNAI 1038, pp. 42–55, London, 1996. Springer-Verlag.
7. A. S. Rao and M. P. Georgeff. BDI agents: From theory to practice. In *Proc. of ICMAS'95*. AAAI Press / MIT Press, 1995.