Going for Gold with 2APL

L. Astefanoaei, C.P. Mol, M.P. Sindlar, and N.A.M. Tinnemeier

Utrecht University
Department of Information and Computing Sciences
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
{astefano,christian,michal,nick}@cs.uu.nl

Abstract. This paper describes our approach to the Multi-Agent Programming Contest in coordination with ProMAS and AAMAS 2007. The object of the contest is to mine as much gold as possible in competition with other teams in a multi-agent goldrush scenario. Our agents are implemented in 2APL, a BDI-based agent-oriented programming language. As required by the contest, we designed and specified our approach using a multi-agent methodology. Several methodologies were evaluated, and eventually we chose a combination of $\mathcal{M}$oise$^+$and Tropos.

1 System Analysis and Design

Before implementing the multi-agent system, we analyzed and designed our case of study by specifying it with the help of existing methodologies. For this purpose, we chose Tropos [1] combined with $\mathcal{M}$oise$^+$[2], rather than using Gaia [3] or Prometheus [4], as this is more suitable for our application. Gaia was deemed to be focused too much on the specification of organizational structure, without giving any guidelines for the implementation. Furthermore, it only vaguely defines notions such as goals and plans. Also, the lack of a specific notation was considered a significant drawback. Prometheus details the implementation phase at a finer grain, and provides a precisely specified notational technique. Where it comes to identifying and describing the system's functionalities, Prometheus was found to be quite helpful. However, the small-scaleness of the contest scenario, and the fact that specifics which would normally have been decided through using the methodology (such as the number of agents) are known from the start makes it superfluous.

In the following we present our specification, combining Tropos and $\mathcal{M}$oise$^+$. Designing our system in Tropos consists of four phases. The first one, Early Requirements Analysis (Figure 1), describes the main scenario. Each `player` has its own private instance of the map. `Scouts` that explore the map inform the `leader` about what they have seen. As soon as the `leader` has gathered enough information about the map, he sends a map update to all players who require it. The `leader` also keeps track of locations where gold has been spotted and the positions of the `players` that roam the map. It is the task of the `miners`, that harvest the gold, to inform the leader about locations of gold. They only send these locations in case they are not able to carry it themselves. The `leader` then assigns the gold to the nearest available `player`. In this phase we decide the dependence relations: `scouts` and `miners` rely on the `leader` for map updates, and he depends on the `miners` and `scouts` for information resources.

Figure 1 can be separated into two distinct diagrams (Figures 2 and 3) by means of $\mathcal{M}$oise$^+$concepts.

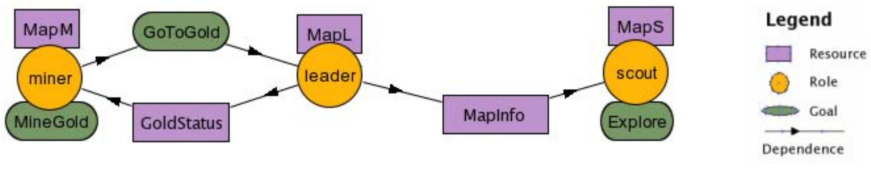

Fig. 1. Tropos: Early Requirement Analysis

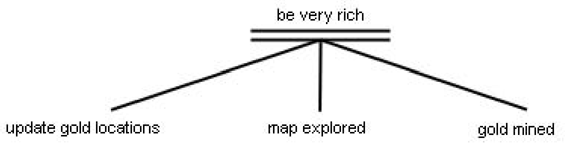

Fig. 2. $\mathcal{M}$oise$^+$: Goal Decomposition Tree

As follows from the Goal Decomposition Tree in Figure 2, the ultimate goal of our "greedy" team is to be the richest. In order to achieve this goal, we decompose it into three subgoals: updating the gold locations, exploring the map, and mining the gold. Furthermore, goals can be either performative goals (update gold location), or achievement goals (exploring region, collecting gold).

Roles, and the relation between them, are specified in $\mathcal{M}$oise$^+$at the structural level. An agent, in our model, can play three different roles: `leader`, `scout`, and `miner`. From Figure 3 it follows that our team can have at most one `leader`, and zero to six `players`, who can have the role of either `scout` or `miner`. The `leader` communicates both with the `scouts` and the `miners`. A large part of the coordination is thus inherently within the `leader`. Coordination takes place in terms of task ordering: first the `scouts` explore the wilderness, then the `miners` can gold-enrich the team.

At the deontic level, we couple roles and goals in terms of commitments and responsibilities. The `leader` is responsible of assigning to-be-mined regions to `players`. When such a player is in the role of `scout`, he switches to the role of `miner`. The `leader` can thus influence when a `player` should change its role.

Having a more accurate vision on roles and goals, we can return to the second specification phase in Tropos. This is the Late Requirement Analysis, which takes each actor as a system and describes its functions. For example, in our case, this means that the `leader` has to consult the map before sending a `miner` on mission. Every `scout` can have its own strategies when exploring an unknown

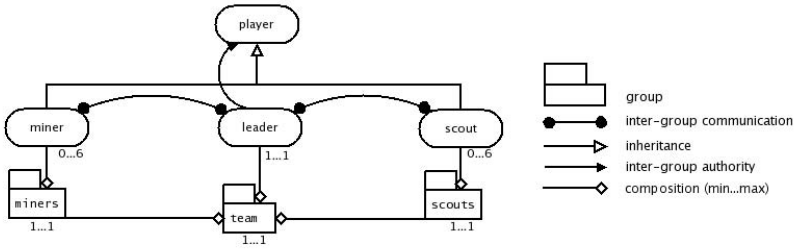

Fig. 3. $\mathcal{M}$oise$^+$: Role Diagram

territory. As for `miners`, their task is to mine the gold. Our agents depend on the `leader`, but can also perform their tasks themselves, thus preserving their autonomy.

How the team interacts with the environment is specified in the Architectural Design phase. The environment, by definition, provides information for our team, and furthermore the `players` act upon the environment. We can specify how the data is communicated at the Detailed Design level. In the next section, we describe how the specification connects to the implementation phase.

2 Software Architecture

Our agents are implemented in the BDI-based programming language 2APL [5]. The following gives an overview of the language in general, and, more specifically, to our approach for this contest.

2.1 2APL

2APL (`http://www.cs.uu.nl/2apl`), the successor to 3APL, is an agent-oriented programming language that facilitates the implementation of multi-agent systems. At the multi-agent level, it provides programming constructs to specify a multi-agent system in terms of a set of individual agents, and a set of environments in which they can perform actions. Multiple agents can run together in a single instance of the 2APL platform, each with its own thread of control. The platform also allows communication among agents, and can run on several machines connected in a network.

At the individual agent level, 2APL provides programming constructs to implement cognitive agents based on the BDI architecture. In particular, one can implement an agent's beliefs, goals, plans, actions (such as belief updates, external actions, or communication actions), events, and a set of rules through which the agent can decide which actions to perform. 2APL supports the implementation of both reactive and proactive agents. The next subsection sketches how those concepts can be used to implement the design as discussed in section 1.

2.2 An implementation in 2APL

In the analysis phase we identified a `leader` and a `player`. Each `player` performs either the `miner` or `scout` role, but not both at the same time. Three 2APL files define the player: `player.2apl`, which specifies behavior common to all players, and `scout.2apl` and `miner.2apl`, for role-specific behavior. The role of `leader` is implemented by a file `agent007.2apl`.

Each `leader` and `player` has *beliefs* and *goals* which may change during the agent's execution. A `scout`, for instance, has beliefs about the cells it has explored, and `miners` have a belief about the maximum amount of gold they can carry. Updating beliefs in 2APL is realized by performing belief updates like the ones specified below:

```
BeliefUpdates:
  { role(X) }  Enact(Y)     { not role(X), role(Y) }
  { true }     Seen(X,Y)    { seen(X,Y) }
```

For example, when an agents executes `Enact(Y)`, then the precondition is that he currently believes `role(X)`, and the postcondition specifies that after the belief update he will be believe to be enacting `role(Y)`, and not `role(X)`. Notice that, in this rule, `X` and Y can denote the same role.

The goals of a 2APL agent consist of a list of ground conjunctions each of which denotes a situation the agent wants to realize (not necessary all at once). Goals are the highest-level construct for governing agents' behavior. In our implementation, `scouts` have a performative goal of exploring the map, and an achievement goal of having explored specific cells. The `miners` have an overall goal of having mined gold, which is achieved when they carry no gold, and are not aware of any gold locations to mine. At this point, they switch back to the `scout` role in order to find more gold.

In addition to actions to manipulate the belief base, the `players` and `leader` use communication actions to communicate with other agents, external actions to act upon the environment (moving, picking up gold, etc.), actions to test their belief and goal bases, and actions to add and drop goals. All of these are provided by 2APL.

In 2APL, so-called `PG`-rules can be used to specify that an agent should generate a plan if it has certain goals and beliefs:

```
PG-rules:
  explored(REGION) <- role(scout) | { ...  }
```

The body of the rule, which is omitted, would state that the agent should first go to the region to be explored, and then specifies a particular heuristic for exploring the region. For general path planning we use the A*-algorithm. Having a clear distinction between `miner` and `scout` roles enabled us to implement a role-dependent A* in which `scouts`, whose primary goal is to explore new regions, prefer to travel over cells that are marked as unexplored, whereas `miners`, whose perception deteriorates as they carry more gold, prefer already explored cells over unexplored ones to travel over.

During their execution, `players` receive messages from the `leader` requesting them to mine a certain gold locations, or informing them about updates to the map. In 2APL, so-called `PC`-rules generate plans as a response to such messages and events. For example, we use the following `PC`-rule to deal with a request from the `leader` to mine a certain gold location.

```
PC-rules:
  message( agent007,request,mine(GX,GY) ) <- role(R) | {...}
```

2APL is built on JADE [6] framework, which allows for running instances of the platform in a distributed fashion on multiple machines. We exploited this feature by running our agents on different machines, in order to ensure maximum stability. If any of the agents crashed for some reason, and came back online during a match, it received the necessary information, such as the most recent copy of the map, from the `leader`.

2APL has a well-defined API for creating custom Java environments. The following piece of code shows the update of a specific cell in the contest environment map with the information that it contains an obstacle.

```
@goldworld( updateCell(X, Y, obstacle), L )
```

3 Conclusion

We have presented our approach for programming a multi-agent system for the gold-mining contest in 2APL. It involved two phases; first we specified our system using Tropos and $\mathcal{M}$oise^{+}, and then we implemented it. We found that having a formal description can simplify design and implementation. In this sense, the possibility of mapping a specification automatically to 2APL code would be an interesting and useful object of study, and is indeed part of our future work.

References

1. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology (2004)
2. Hübner, J.F., Sichman, J.S., Boissier, O.: Moise^{+}: Towards a structural, functional, and deontic model for MAS organization. In: Proc. of AAMAS 2002, pp. 501–502 (2002)
3. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)
4. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems. John Wiley & Sons Ltd, Chichester (2004)
5. Dastani, M., Hobo, D., Meyer, J.J.C.: Practical extensions in agent programming languages. In: Proc. of AAMAS 2007 (2007)
6. Bellifemine, F., Bergenti, F., Caire, G., Poggi, A.: JADE - A Java Agent Development Framework. In: Multi-Agent Programming, pp. 125–147 (2005)