Supporting Group Plans in the BDI Architecture using Coordination Middleware*

(Extended Abstract)

Stephen Cranefield
Department of Information Science,
University of Otago, Dunedin, New Zealand
stephen.cranefield@otago.ac.nz

ABSTRACT

This paper investigates the use of group plans and goals as programming abstractions that encapsulate the communication needed to coordinate collaborative behaviour. It presents an extension of the BDI agent architecture to include explicit constructs for goals and plans that involve coordinated action by groups of agents. Formal operational semantics for group goals are outlined, and an implementation of group plans and goals for the Jason agent platform, based on integration with the ZooKeeper coordination middleware, is described.

Keywords

BDI agents; Group plans and goals; Coordination middleware

1. INTRODUCTION

Belief-Desire-Intention (BDI) agent programming is a powerful and popular model for developing software that is goal-oriented, adaptive to its circumstances, and equipped with pre-existing domain knowledge in the form of plans. However, when multiple agents need to coordinate their actions over sustained interactions, especially in interdependent activities such as working together towards a common team goal, manual implementation of the required communication actions can be complex and error prone. It can also compromise scalability and other desirable properties of distributed systems, such as robustness against network partitions.

Our aim in this work is to raise the level of abstraction when programming groups of BDI agents so that explicit coordination is no longer necessary, but rather is handled implicitly by robust industry-grade coordination middleware. This paper investigates the potential of group goals and plans to provide such a programming abstraction.

2. GROUP GOALS

We define a group goal as one requiring coordinated action by a group of agents. In this paper, we focus on conjunctive group goals, in which each agent has a specific *local* subgoal to achieve, and the group goal is satisfied only when *all* the subgoals are individually satisfied. We consider that the group goal becomes active when one or more agents begin working towards it, and can then eventually fail or succeed. However, to avoid agents waiting indefinitely for other agents to complete their subgoals, we allow a group goal to fail due to a "join timeout" or a "completion timeout". A join timeout occurs when some agents do not begin working on their subgoals within a certain interval after the earliest start of any of the other agents' subgoals. A completion timeout occurs when the result of some subgoals are still unknown within a certain interval since the last agent began work on its subgoal.

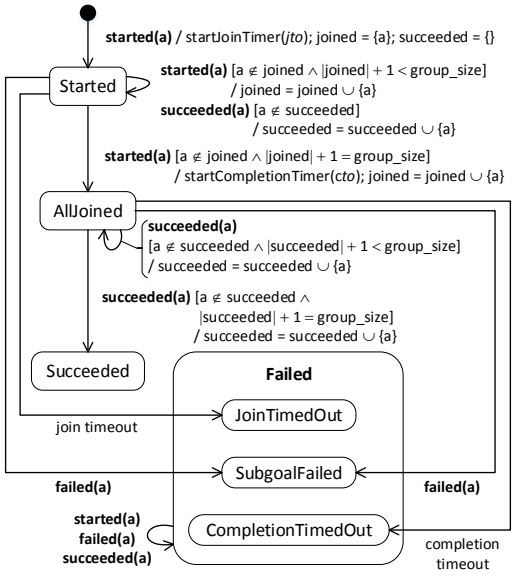

Figure 1: State diagram for a group goal

Figure 1 shows the lifecycle of a group goal as a UML state diagram. The variables $joined$ and $succeeded$ record the agents that have begun work and completed their subgoals.

3. GROUP PLANS

We conceptualise a group plan as one that defines a collective view of coordinated action amongst a group of agents, and rely on group goals to provide this coordination. We thus define a group plan as one in which all goals are group goals. A group plan is supported by a set of individual plans for each agent to achieve its local subgoals. These individual plans could be known by all agents, or they could be kept private, as the application requires.

In this paper we restrict our attention to group plans in which each group goal involves *all* agents in the group. However, any

*Full paper: `http://hdl.handle.net/10523/6232`

Appears in: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016), J. Thangarajah, K. Tuyls, C. Jonker, S. Marsella (eds.), May 9–13, 2016, Singapore.*

```
{ include("group_plans.asl") }
// Group plan
+!pincer(a1,a2,l1,l2,l3) <-
  !gg(surround, [loc(l1)[agent(a1)],
                 loc(l2)[agent(a2)]]);
  !gg(converge, [loc(l3)[agent(a1)],
                 loc(l3)[agent(a2)]]).

// Example local plan for a group goal
+!loc(L)[agent(a1), gg(surround)] <-
  move_to(L);
  ?loc(L).
```

Figure 2: An example group plan

agent's subgoal may be a trivial "do-nothing" goal (one having a plan that immediately succeeds), and BDI agents may have multiple goals (intentions) active at the same time. Thus it is possible for an agent to work towards its own local goals while waiting for the other agents to complete their work towards a group goal. We can also model sequential turn-taking behaviour by a sequence of group goals in which all agents except one have a do-nothing goal.

Figure 2 shows a group plan for two agents, a_1 and a_2, to perform a pincer attack on an enemy at location l_3, with a_1 moving to one side of the enemy (location l_1), and a_2 moving to the other side (location l_2), before they both converge on location l_3. This is expressed in the Jason platform's version of AgentSpeak [1]. The `include` directive loads a set of plans for maintaining the state of group goals via an integration between Jason and the Apache ZooKeeper coordination middleware. The group plan above comprises a sequence of two group goals, but group plans may include any control structures provided by the BDI programming language. However, we do not currently address the use of context conditions.

4. SEMANTICS OF GROUP GOALS

We have defined a set of inference rules stating how the operation of each agent's underlying BDI platform is extended to take account of the shared state of group goals. We model the combined goal states of the agents as a set comprising *goal trees* labelled with agent names. Each goal tree records a top-level goal for an agent as the root of a tree of subgoals, generated by a stack of currently executing plan bodies. There can be multiple goal trees for each agent. The inference rules then define the valid transitions on a triple comprising this set of labelled goal trees, a set of group goal state machines, and a time stamp. A *local computation* rule allows local agent goal-tree transitions to proceed as usual, if they do not involve an immediate local subgoal of a group goal that the agent has not yet joined. A *join goal* rule allows an agent to begin a subgoal of a non-failed group goal, and updates the state machine for that group goal as defined in Figure 1. *Local failure* and *local success* rules ensure that failure or success of an agent's local subgoal causes the state machine for the parent group goal to be updated. Finally, *group failure* and *group success* rules react to a group goal's state machine reaching the failure or success state by marking each agent's local subgoal for that group goal as having failed or succeeded. This allows the agents' local computation to handle this failure or success.

Using LTL model checking in Maude [2], we verified for the two-agent case the desired relationships between individual subgoal and group goal failures and successes.

5. IMPLEMENTATION

We have implemented our semantics in a distributed setting by using Apache ZooKeeper [4] to provide Jason agents, potentially running on different hosts, with an eventually consistent view of the state of their group goals. ZooKeeper is a high performance coordination middleware system that maintains data objects replicated across a set of servers. The data objects, known as *znodes* are organised into a hierachical namespace like a file system, and are designed to support the storage of coordination metadata in distributed applications. In our implementation, distributed Jason BDI agents run within containers provided by the camel-agent middleware [3]. This provides the ability for agent action invocations to be delivered as messages to the Apache Camel message routing and mediation engine [5], and for percepts to be delivered back to the agents. Message processing "routes" written in Camel's domain-specific language interpret certain agent actions as ZooKeeper client requests to update the state of group goals, while other routes deliver updates about their state to agents in the form of percepts. The Camel routes collectively manage the representation of the state machines for group goals as a collection of ZooKeeper znodes. In particular, one master route, run repeatedly at a set frequency, checks znodes representing the lists of agents who have begun and completed their subgoals of a group goal, and updates the znode encoding of the state machine's state.

The semantics defined by our inference rules (outlined in Section 4), are implemented by a standard set of plans that can be included in the code defining a group plan (see line 1 in Figure 2). These plans define (i) how to achieve the `gg` goal for beginning a new group goal, (ii) how to handle percepts generated by Camel routes (via camel-agent) to indicate the success or failure of a group goal, and (iii) how to handle the Jason event signalling the failure of an agent's local subgoal for a group goal—this plan performs an action to update ZooKeeper (via Camel) about this failure.

6. CONCLUSION

This paper has presented a model for coordinating the activities of a group of BDI agents by providing them with shared group plans containing group goals. These concepts provide an abstraction layer that removes the need for the programmer to provide explicit communication actions to share information about the agents' failures and successes in pursuit of a larger goal. We defined the lifecycle of group goals as a state machine and described formal operational semantics for the interaction between the individual agents' BDI execution engines and a set of shared state machines for the group goals. We also described an implementation of this model using Jason and the industrial-strength coordination middleware, Apache ZooKeeper.

REFERENCES

[1] R. H. Bordini, J. F. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak using Jason*. Wiley, 2007.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In *Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer, 2003.

[3] S. Cranefield and S. Ranathunga. Embedding agents in business processes using enterprise integration patterns. In *Engineering Multi-Agent Systems*, volume 8245 of *Lecture Notes in Computer Science*, pages 97–116. Springer, 2013.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2010.

[5] C. Ibsen and J. Anstey. *Camel in Action*. Manning, 2010.