Supporting group plans in the BDI architecture using coordination middleware

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 provided, and an implementation of group plans and goals for the Jason agent platform is described, based on integration with the Zookeeper coordination middleware.

1 Introduction

Belief-Desire-Intention (BDI) agent programming provides 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, and may negatively impact 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. Furthermore, group plans can serve as the external blueprint for the collective action of the agents, and provide all group members with common view of the intended group activity. Thus, we also aim to have a single plan used as the source code for all agents in the group, despite their differing local subgoals.

This paper makes the following contributions. We define a model for group goals and plans, with formal operational semantics that define how agents' local BDI computations are affected by the states of group goals. We then describe an implementation of our model using Jason, the Apache ZooKeeper middleware, and a set of domain-independent plans that serve as the interface between shared group plans and the group goal state in ZooKeeper.

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 can represent a group goal for a group of agents $\{a_1, \ldots, a_n\}$ by a term $gg(group\_goal\_name, [a_1{:}subgoal_1, \ldots, a_n{:}subgoal_n])$.

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 need to consider what happens if some agents do not succeed, fail, or even begin their own subgoal in a timely manner. We address this by allowing 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. The timeout values are specified as part of a group plan (see the next section).

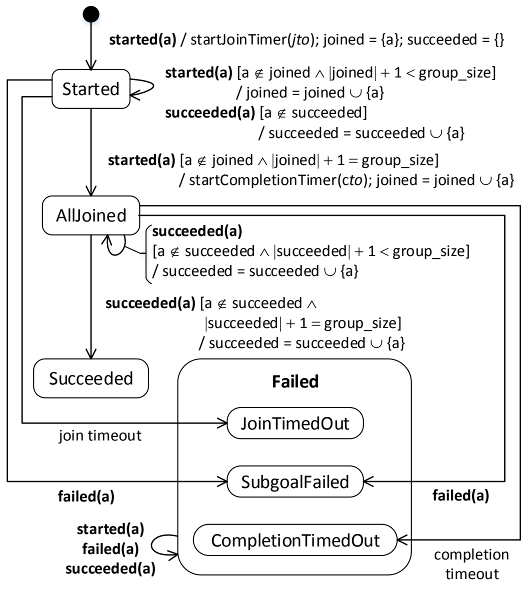
Figure 1 shows the lifecycle of a group goal as a UML state diagram. The variables $joined$ and $succeeded$ record the agents that have begun work and completed their subgoals.

3 Group plans

We conceptualise a group plan as one that defines a collective view of coordinated action amongst a group of agents, and which can be followed in a synchronised way by all agents. We rely on group goals to provide the coordination between agents, and 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 own subgoals. These individual plans could be known by all agents, or they could be kept private, as the application and group requires.

In this paper we restrict our attention to group plans in which each group goal gg involves *all* agents in the group. However, this is not as strong a restriction as it seems at first



started(a) / startJoinTimer(jto); joined = {a}; succeeded = {}
Started
started(a) [a ∉ joined ∧ |joined| + 1 < group_size] / joined = joined ∪ {a}
succeeded(a) [a ∉ succeeded] / succeeded = succeeded ∪ {a}
started(a) [a ∉ joined ∧ |joined| + 1 = group_size] / startCompletionTimer(cto); joined = joined ∪ {a}
AllJoined
succeeded(a) [a ∉ succeeded ∧ |succeeded| + 1 < group_size] / succeeded = succeeded ∪ {a}
succeeded(a) [a ∉ succeeded ∧ |succeeded| + 1 = group_size] / succeeded = succeeded ∪ {a}
Succeeded
Failed
JoinTimedOut
join timeout
SubgoalFailed
failed(a)
failed(a)
started(a)
failed(a)
succeeded(a)
CompletionTimedOut
completion timeout

Figure 1: State diagram for a group goal

```
{ include("group_plans.asl") }

join_timeout(_, 5000). // In milliseconds
completion_timeout(surround, 600000).
completion_timeout(converge, 300000).

+!pincer(A,B,L1,L2,L3) <-
  .union([A], [B], [a1,a2]); // Check group membership
  !gg(surround, [loc(L1)[agent(A)],
                 loc(L2)[agent(B)]]);
  !gg(converge, [loc(L3)[agent(A)],
                 loc(L3)[agent(B)]]).

+!loc(L)[agent(a1), gg(surround)] <-
  move_to(L);
  ?loc(L).

+!loc(L)[agent(a2), gg(surround)] <-
  move_to(L);
  ?loc(L).

+!loc(L)[agent(a1), gg(surround)] <-
  move_to(L);
  ?loc(L).

+!loc(L)[agent(a2), gg(converge)] <-
  move_to(L);
  ?loc(L).
```

Figure 2: Example group and local plans

glance. Any 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. To simplify the notation in cases like this, we adopt the do-nothing goal as the default for any agents that do not have a subgoal listed.

As group goals are only satisfied when all agents' subgoals are completed, the successful completion of each group goal is a point of synchronisation between agents. In other words, an agent that quickly completes its subgoal for one group goal cannot begin work on the next group goal until all other agents complete their subgoals. This is in line with the idea that a group plan is a mechanism for specifying the coordinated action of a group of agents. However, there may be cases when the programmer would like an agent with a subgoal that is achieved quickly to be able to begin working on the following group goal without delay. Given our current state machine for group goals, this early start to a group goal would start the join timeout timer goal before the previous goal is fully achieved. While it may be possible to predict an appropriate timeout period, adding this capability calls for a more sophisticated treatment of timeouts.

Figure 2 shows a group plan for two agents, A and B to perform a pincer attack on an enemy at location $L3$, with A moving to one side of the enemy (location $L1$), and B moving to the other side (location $L2$), before they both converge on location $L3$. A and B must be distinct members of the group $\{a1, a2\}$. Plans for the group members' local subgoals of the group goals are also shown. In this example, the two agents have the same local plans, but in general these could differ between agents and need not be shared with other agents. The plans are expressed in the Jason platform's version of AgentSpeak [4]. The `include` directive loads a set of plans for handling group goals, including one that handles `gg` goals. These plans coordinate the distributed maintenance of the state of each group goal, making use of the Apache ZooKeeper coordination middleware, and are described in Section 5.2. Jason does not support terms of the form *agent*:*subgoal*, within the `gg` goals, so we use Jason 'annotations' to identify the agent associated with each subgoal (e.g. *subgoal*`[agent(A)]`).

The body of the group plan is a sequence of two group goals, but group plans may be more complex and include any control structures provided by the BDI programming language. However, in this paper we do not address the use of context conditions. A context condition for a group plan would need to be evaluated by all agents, and so a mechanism for shared beliefs is needed to support this feature. Shared beliefs can be implemented using similar middleware techniques to those we use to implement group goals [16], but we leave that for future work. We also do not currently consider explicit communication between agents, although agents working in a team may need to share information.

4 Semantics of group goals

In this section we present a set of inference rules defining how the operation of each agents' 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, as shown in Figure 3. There can be multiple goal trees for each agent. For example, the goal tree shown in the left side of Figure 3 represents the goal-tree for a top-level goal g_0 and its stack of executing plans: one for goal g_0

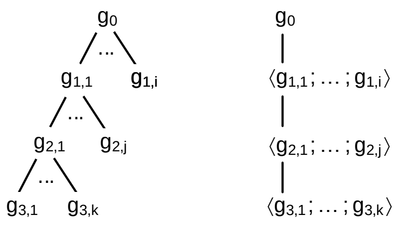


Figure 3: An example goal tree

with body $g_{1,1}; \dots; g_{1,i}$, another for subgoal $g_{1,1}$ with body $g_{2,1}; \dots; g_{2,j}$ and a third for subsubgoal $g_{2,1}$ with body $g_{3,1}; \dots; g_{3,k}$. We only consider plan bodies with sequential execution of goals, and we assume that successful goals are immediately removed from the tree. Thus, the tree is expanded in a depth-first manner by extending the bottom left node, and can therefore also be visualised as a stack of plan bodies (growing downwards), as shown on the right of the figure. Our syntax for goal trees turns this format on its side, and we express the tree in the figure as follows:

$$\langle g_0 \rangle \cdot \langle g_{1,1}; \dots; g_{1,i} \rangle \cdot \langle g_{2,1}; \dots; g_{2,j} \rangle \cdot \langle g_{3,1}; \dots; g_{3,k} \rangle$$

A goal tree can be seen as a projection of a goal-plan tree [20], in which the levels of the tree alternate between goal and plan nodes. In a goal-plan tree, the children of a goal node represent the applicable plans for that goal, and the children of a plan node represent the goals in the body of the plan. However, our semantics do not require the explicit representation of plans, and we can consider each goal node to be annotated with information about the applicable plans, and which plans have already been tried unsuccessfully. This information is needed by each agent's standard BDI reasoning, but not for the handling of group goals defined here. For the same reason, we do not model the agents' beliefs as these would be passed unchanged through each of our transitions.

Our operational semantics for group goals are presented in Figure 4. The rules define the valid transitions on a triple containing the goal state (the set of all agents' goal trees), a set of state machines for the group goals, and a timestamp. Transitions on these triples are denoted $\langle GS, S, t \rangle \longrightarrow \langle GS', S', t' \rangle$. Transitions of an agent a's set of goal trees in its local BDI engine are denoted $GS \xrightarrow{a} GS'$. We write g and gg for local and group goals, respectively, ρ for a (possibly empty) sequence of plans bodies, and π for the tail of a plan body. We decompose sets by using the disjoint union operator ($\uplus$). sm^{gg} represents the state machine for group goal gg, and we use a '.' notation to indicate sending an event to a state machine, e.g. $sm^{gg}.\,started(a)$, and to access the state machine's $joined$ and $succeeded$ variables. The creation of a new state machine for group goal gg, with its clock set to time t, is denoted $sm(gg, t)$, and $s.\,update\_clock(t)$ returns a copy of s with its clock set to t.

To define the interaction between our inference rules and the local BDI computations of each agent, we assume that a goal that has failed is first marked with a success or failure flag, written as $g^{✔}$ or (respectively) $g^{✘}$, before it is handled by the BDI engine. This allows our semantics to define how failure or success of an agent's local contribution to a group goal should be handled, and how group goal failure is indicated to a local agent's BDI engine.

The *Timer* rule indicates that the clocks of all state machines can be advanced.

The next three rules have a local agent computation in their premises (above the line) and a transformation on the combined state in the conclusion (below the line). These can be read procedurally from the bottom up. Given an initial combined state that matches the left hand side of the transition in the conclusion, check any constraints on that state appearing in the rule's premises. If these are satisfied, run a single step of the relevant agent's BDI engine, as shown in the premises. The conclusion then defines a transition that incorporates the updated local state change into the new combined state.

The *Local Computation* rule states that any single step transition in an agent's BDI engine can proceed as normal, provided that it does not involve a local subgoal of a group goal that the agent has not yet joined. These cases are handled by other rules.

The *Start Goal* rule states that the first agent to begin a group goal, by calling a local plan for that goal, causes a new state machine for that goal to be created and to receive the $started$ event for that agent.

The *Join Goal* rule states that subsequent agents beginning the group goal cause the corresponding $started$ event to be sent to the state machine for that goal.

The *Local Failure* and *Local Success* rules handle the cases when an agent's work on its own subgoal for a group goal gg results in failure or (respectively) success. In these cases, the state machine for gg must be notified of a's failure or (respectively) success.

Finally, the *Group Failure* and *Group Success* rules enable the use of an agent's local BDI failure and success handling mechanisms to cause transitions on a group goal once its state machine reaches the failed or succeeded states. Note that the Group Failure rule can be applied to an agent that has either already failed its local subgoal, or is still working on it.

We have modelled these rules using Maude [5], together with six rules defining standard (individual agent) semantics for transforming goal trees when subgoals fail or succeed[1]. We then modelled three additional rules defining the transition systems for agents with a single common group goal and one local subgoal. Using Maude's LTL model checker, we have verified that in the two-agent case, individual subgoal successes (when timeouts do not occur) and failures necessarily lead to group goal successes and failures.

5 Design and Implementation

The semantics in Figure 4 model agents that interact sequentially with a shared set of group goal state machines to produce a sequence of transitions. This is not a realistic model in a distributed setting. In this section we show how the use of coordination middleware allows us to implement group goals for a group of distributed agents. In particular, we use the Apache ZooKeeper coordination middleware [13] to provide Jason agents, potentially running on different hosts, with an eventually consistent view of the state of their group goals.

[1]To simplicity failure handling, we currently assume there is a single subgoal for each goal.

$$\frac{\mathit{update\_time}(t,t')}{\langle G,S,t\rangle \longrightarrow \langle G,\{s.\,\mathit{update\_clock}(t) : s \in S\},t'\rangle}\ \text{Timer}$$

$$\frac{\nexists \rho_2, gg, \pi' : \rho_1 = \rho_2 \cdot \langle gg;\pi'\rangle \text{ unless } sm^{gg} \in S \wedge a \in sm^{gg}.\,\mathit{joined} \quad \{\rho_1 \cdot \langle g;\pi\rangle\} \xrightarrow{a} \{\rho'\} \quad \mathit{update\_time}(t,t')}{\langle\{a : \rho_1 \cdot \langle g;\pi\rangle\} \uplus GS, S, t\rangle \longrightarrow \langle\{a : \rho'\} \cup GS, S, t'\rangle}\ \text{Local Computation}$$

$$\frac{\nexists\, sm^{gg} \in S \quad \{\rho \cdot \langle gg;\pi\rangle\} \xrightarrow{a} \{\rho \cdot \langle gg;\pi\rangle \cdot \langle g;\pi'\rangle\} \quad \mathit{update\_time}(t,t')}{\langle\{a : \rho \cdot \langle gg;\pi\rangle\} \uplus GS, S, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g;\pi'\rangle\} \cup GS, S \cup sm(gg,t').\,\mathit{started}(a), t'\rangle}\ \text{Start Goal}$$

$$\frac{sm^{gg}.\mathit{state} \neq \mathit{failed} \quad a \notin sm^{gg}.\,\mathit{joined} \quad \{\rho \cdot \langle gg;\pi\rangle\} \xrightarrow{a} \{\rho \cdot \langle gg;\pi\rangle \cdot \langle g;\pi'\rangle\} \quad \mathit{update\_time}(t,t')}{\langle\{a : \rho \cdot \langle gg;\pi\rangle\} \uplus GS, S \uplus sm^{gg}, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g;\pi'\rangle\} \cup GS, S \cup sm^{gg}.\,\mathit{started}(a), t'\rangle}\ \text{Join Goal}$$

$$\frac{sm^{gg}.\mathit{state} \neq \mathit{failed}}{\langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g^{✗};\pi'\rangle\} \uplus GS, S \uplus sm^{gg}, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g^{✗};\pi'\rangle\} \cup GS, S \cup sm^{gg}.\,\mathit{failed}(a), t\rangle}\ \text{Local Failure}$$

$$\frac{a \notin sm^{gg}.\,\mathit{succeeded}}{\langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g^{✓};\pi'\rangle\} \uplus GS, S \uplus sm^{gg}, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \langle g^{✓};\pi'\rangle\} \cup GS, S \cup sm^{gg}.\,\mathit{succeeded}(a), t\rangle}\ \text{Local Success}$$

$$\frac{sm^{gg}.\mathit{state} = \mathit{failed}}{\langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \rho_2\} \uplus GS, S \uplus sm^{gg}, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg^{✗};\pi\rangle\} \cup GS, S \cup sm^{gg}, t\rangle\}}\ \text{Group Failure}$$

$$\frac{sm^{gg}.\mathit{state} = \mathit{succeeded}}{\langle\{a : \rho \cdot \langle gg;\pi\rangle \cdot \rho_2\} \uplus GS, S \uplus sm^{gg}, t\rangle \longrightarrow \langle\{a : \rho \cdot \langle gg^{✓};\pi\rangle\} \cup GS, S \cup sm^{gg}, t\rangle\}}\ \text{Group Success}$$

Figure 4: Operational semantics of group goals

Agents (running the Jason BDI interpreter) may be distributed across multiple hosts, and use a distributed fault-tolerant set of ZooKeeper servers to maintain the state of group goals. On each host, the agents run within a container provided by the camel-agent middleware [6]. This provides the ability for agent action invocations to be delivered as messages to the Apache Camel message routing and mediation engine [14], and for percepts to be delivered back to the agents. Due to its domain-specifc language for message filtering and transformation, and its existing support for communicating with ZooKeeper, Camel provides a convenient mechanism for connecting agents to ZooKeeper 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. Below, we discuss the representation of the state of group goals in ZooKeeper, and the Camel routes.

5.1 Storing group goal states in ZooKeeper

ZooKeeper [13] 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, rather than large amounts of operational data. Znodes are identified by a path from the root node and may contain data, which is an array of bytes, and metadata such as the time of creation, time of last modification, and number of child znodes. ZooKeeper provides several guarantees about the view of the znodes across multiple clients [13]: "*Linearizable writes:* all requests that update the state of ZooKeeper are serializable and respect precedence; [and] *FIFO client order:* all requests from a given client are executed in the order that they were sent by the client."

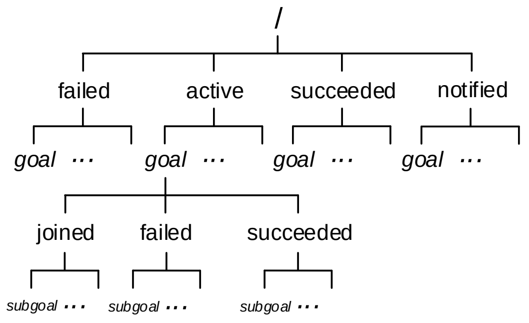


Figure 5: The ZooKeeper data model for group goal state

Figure 5 shows the znode structure we use to maintain the group goal state. When an agent first begins a local subgoal, this is recorded by creating the node /active/$goal$/joined/$subgoal$, where $goal$ is the term representing the group goal. The node is automatically created if it does not already exist. The node data records the number of subgoals for the goal, the join and completion timeout periods, and whether the goal is already known to have failed or succeeded (both initially false). If an agent's subgoal for that group goal fails, the failed subgoal is recorded by creating the node /active/$goal$/failed/$subgoal$, and successful completion of a subgoal is recorded in a similar way using the succeeded subtree of the goal's node. The metadata for /active/$goal$/joined/ and /active/$goal$/succeeded/ record how many children these nodes have, and these counts are used to detect when a join timeout has occurred (in conjunction with the goal node's creation date), when a goal has failed (if there

Start goal
Trigger: Agent action start_goal($goal$, $subgoal$, jto, cto)
Response:

- Create or overwrite znode /active/$goal$ with the data:
  - number of subgoals (computed from $goal$ term)
  - join timeout (jto)n
  - completion timeout (cto)
  - -1 for the "all-joined" time (meaning no agents have started their subgoal)
  - whether the goal has already failed (initially false)
  - whether it has already succeeded (initially false)
- Create joined, failed, and succeeded nodes beneath this new node, if they do not exist.
- Create node /active/$goal$/joined/$subgoal$

Record local failure
Trigger: Agent action notify_my_part_failed($goal$, $subgoal$)
Response: Create znode /active/$goal$/failed/$subgoal$

Record local success
Trigger: Agent action notify_my_part_succeeded($goal$, $subgoal$)
Response: Create znode /active/$goal$/succeeded/$subgoal$

Notify group failure
Trigger: A change to the children of /failed, in the form of an updated list of child nodes
Response: For each child node /failed/$goal$, check whether /notified/$goal$ exists. If not, add failed($goal$) to each agents' set of pending percepts (i.e. those waiting for its next perception)

Notify group success
As for Notify group failure, but replace failed with succeeded

Master route
Route policy: ZooKeeper route policy. This setting encapsulates a leadership election using Camel. It ensures that only one instance of Camel will run this route. If the leader fails, another will take over.
Trigger: A timer tick, set at a configurable frequency
Response: Get the list of children for znode /active. For each znode /active/$goal$:

- Read the data and metadata for /active/$goal$
- Read the data and metadata for /active/$goal$/joined
- Read the data and metadata for /active/$goal$/failed
- Read the data and metadata for /active/$goal$/succeeded
- Compute the state of the group goal from this information, as outlined above in the description of the znode data mode used
- If /active/$goal$ has neither failed=true nor succeeded=true:
  - If state is Failed:
    - ∗ Update /active/$goal$ with failed=true
    - ∗ Create node /failed/$goal$
  - If state is Succeeded:
    - ∗ Update /active/$goal$ with succeeded=true
    - ∗ Create node /succeeded/$goal$
  - If state is AllJoined:
    - ∗ If /active/$goal$ has allJoinedTime $= -1$
    - ∗ Update /active/$goal$ with allJoinedTime set to the last modified time for node /active/$goal$

Figure 6: Description of Camel routes

are any sub-sub-nodes for failed agents), and when a goal has succeeded (when the number of agents with successful subgoals equals the group size). When a goal is known to have failed or succeeded, a new node for it is created as a child of /failed/ or /successful/, and this node creation can be detected by other clients using ZooKeeper's "watch" mechanism. The changes to the znode structure described above are made by "routes" defined in Camel, as shown in Figure 6.

5.2 Generic plans for group goals

The Camel routes described in Figure 6 are not sufficient to implement our semantics for group goals on their own. In addition to the user-supplied group plans, additional agent code is needed to trigger the "Start goal" Camel route when an agent begins working on its subgoal for a group goal, and to handle `failed` and `succeeded` percepts from the "Notify group failure" and "Notify group success" routes. This logic is provided by a set of generic plans for handling group goals that are included into the user's group plan code using a preprocessor directive (see line 1 of Figure 2). These plans define how to achieve the `gg` goal for beginning a new group goal, as used in Figure 2 (lines 8–11). The plan for `gg` creates a new top-level goal `achieve_my_part` to begin execution of the agent's local subgoal. This triggers a plan that performs the `start_goal` action (to update the ZooKeeper state, via a Camel route), calls the agent's subgoal (passed as a parameter of `achieve_my_part`), and then (if successful) notifies ZooKeeper (via Camel) of this success. Two other plans handle percepts from Camel indicating the success or failure of the group goal. These wake up the suspended `gg` plan with a success or failure message. In addition, the plan handling the failure percept aborts the `achieve_my_part` goal. One more plan handles the event indicating that the agent's local subgoal failed. This performs an action to update ZooKeeper (via Camel) about this failure.

There is a direct correspondence between these plans and the semantic rules in Figure 4. The use of a separate intention for the local subgoal implements the constraints on the Local Computation rule—it is necessary to avoid the success or failure of the subgoal directly affecting the group goal based on local BDI computation. The plan for `achieve_my_part` implements the Start Goal and Join Goal rules (which are only distinguishable from the state machine's point of view), as well as the Local Success rule. The plan handing failure of the `achieve_my_part` goal implements the Local Failure rule. The last two plans implement the Group Failure and Group Success rules.

6 Discussion

The key requirements of an implementation of group plans are: (1) that all agents see the same outcome for each group goal; (2) that the successful completion of a group goal acts as a synchronisation point for the agents, and (3) that timeouts are handled in an appropriate way. We argue that requirement 1 is met due to the guarantees provided by ZooKeeper (see Section 5.1) and the correspondence between our semantic rules and the generic plans for handling group goals.

For requirement 2 we must weigh up the application's required temporal precision for synchronisation against the potential need for scalability and partition tolerance. While most applications may not require group plans for large groups, there may be a need to support many teams running different group plans at once. Thus, we apply an eventual consistency approach to updating agents about the group goal state. This means that there will be some variation in the times at which agents receive a percept informing them about a group goal's failure. For example, a subgoal failure causes a Camel route to create a new znode recording the failure, and this is then noticed by the Master route. The Master route then creates a znode representing the overall goal failure, and the "Notify group failure" route detects this and creates `failed` percepts for the agents to perceive. We have not yet evaluated the performance of this mechanism. ZooKeeper guarantees that a "clients view of the system is guaranteed to be up-to-date within a certain time bound … on the order of tens of seconds)" [2], which is hopefully a worst case guarantee. On the other hand, coordination via standard inter-agent messaging may be no better, and is more prone to programmer error.

In a distributed setting, handling timeouts (requirement 3) is not a straightforward matter due to potential differences between agents' clocks. Therefore, we take a forgiving approach to timeouts. The Master route is the arbiter, but it cannot be guaranteed to check for a timeout at the precise moment that it would be due to occur. Therefore when it next polls the group goal state, it checks for state changes that would avoid a timeout before checking whether a timeout has occurred. Thus, the specified timeout intervals are really lower bounds on the time at which timeouts will be checked.

The Master route must use its own local time when checking for timeouts, as the only temporal information available from ZooKeeper is the creation and last modification times of znodes. This means the Master route uses its own current time against ZooKeeper's last modification time for the relevant znode. This is a shortcoming that could be resolved by implementing a route to synchronise the Master route's clock with ZooKeeper's, by repeatedly creating nodes and reading their creation time.

The Master route currently operates by polling. It would be more efficient to use ZooKeeper watches to receive updates to the znodes of interest. However, this is not straightforward in Camel as it requires dynamically creating routes at run-time whenever a group goal is started, which is not standard practice in Camel. Another approach would be to avoid the need for a master process by changing to some other middleware that directly supports replicated state machines in a peer-to-peer architecture. One option is the Cassandra distributed database, which supports the Paxos protocol [17] via "lightweight transactions" [8]. It is also supported by Camel.

7 Related work

There is a large body of research on many aspects of teamwork, cooperation and collaboration in multi-agent systems [7; 19; 18; 11; 21; 9]. Here we focus on the work related to BDI agent reasoning with a pre-designed plan library.

Kinny et al. [15] present an expressive formalism for *joint plans* structured as graphs comprising actions, goal, belief queries, and 'and' and 'or' nodes. Possible worlds semantics for the model are presented as well as operational semantics in terms of graph transformations. Coordination is based on the transformation of graphs for joint plans into role-specific graphs that have added communication acts to broadcast the result of each edge traversal in the plan graph. In contrast, we focus on the more limited but well known AgentSpeak model of BDI execution, and investigate the use of mainstream highly scalable coordination middleware to support the execution of group plans.

Griffiths et al. [10] investigate a model of cooperation between BDI agents arising from *cooperative plans*, which contain actions that must be performed by other agents. They present a formalism for their model in Z, and propose an approach for selecting trusted partners to help them execute certain actions. However, no model of plan execution is given.

The JACK agent platform includes support for "team-oriented programming" [1]. Teams are defined in terms of roles with constraints on how many members must play each role. Teams have their own beliefs, desires and intentions, which can be propagated up or down the team hierarchy. Team plans can post goals for subteams (including individuals) and these may be executed in parallel. The programmer has control over how success or failure of a team is derived from success or failure of the subteams, and whether to wait for all subteams to finish their goals before execution in the parent team moves on. This is a powerful but procedural model with no semantics defined. Our work aims to provide similar features in the context of a more standard declarative model of BDI agents.

JaCaMo [3] enhances the Jason BDI platform with the ability to use *coordination artifacts* that implement the MOISE organisation model [12] for multi-agent systems. However, while MOISE has group plans (goals decomposed into missions), JaCaMo does not support these as a first-class construct. In JaCaMo, each agent's plans must explicitly coordinate with the group via a *GroupBoard* artifact and/or explicit requests to other agents to achieve goals [3, Fig. 6]. Our approach removes this need. We also provide formal semantics specifying how state changes for group goals come about from local BDI computation steps, and how changes to group goal states affect the local BDI computation. MOISE and JaCaMo do not have formal semantics.

8 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 presented 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.

There are many avenues for extending this work, such as supporting disjunctive goals and context conditions defined over shared beliefs, providing more flexible options for the use of timeouts, supporting explicit communication between agents (for purposes other than group goal coordination), investigating the use of Cassandra and other middleware, and evaluating the performance of the approach.

References

[1] Agent Oriented Software Pty Ltd. Jack intelligent agents teams manual, release 5.5. `http://www.aosgrp.com/documentation/jack/JACK_Teams_Manual.pdf`, 2005.

[2] Apache Software Foundation. ZooKeeper programmer's guide, ZooKeeper 3.1 documentation. `https://zookeeper.apache.org/doc/r3.1.2/zookeeperProgrammers.html#sc_zkDataModel_znodes`, December 2010.

[3] O. Boissier, R. H. Bordini, J. F. Hbner, A. Ricci, and A. Santi. Multi-agent oriented programming with JaCaMo. *Science of Computer Programming*, 78(6):747–761, 2013.

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

[5] 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 (RTA 2003)*, number 2706 in Lecture Notes in Computer Science, pages 76–87. Springer, 2003.

[6] 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.

[7] K. Decker. TAEMS: A Framework for Environment Centered Analysis & Design of Coordination Mechanisms. In G. O. Hare and N. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 429–448. Wiley, 1996.

[8] J. Ellis. Lightweight transactions in Cassandra 2.0. DataStax Developer Blog post, `http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0`, 2013.

[9] M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. L. Arcos. AMELI: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 236–243. IEEE Computer Society, 2004.

[10] N. Griffiths, M. Luck, and M. d'Inverno. Annotating cooperative plans with trusted agents. In *Trust, Reputation, and Security: Theories and Practice*, volume 2631 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2003.

[11] B. J. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

[12] J. F. Hubner, J. S. Sichman, and O. Boissier. Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. *International Journal of Agent-Oriented Software Engineering*, 1(3-4):370–395, 2007.

[13] 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.

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

[15] D. Kinny, M. Ljungberg, A. S. Rao, L. Sonenberg, G. Tidhar, and E. Werner. Planned team activity. In *Artificial Social Systems*, volume 830 of *Lecture Notes in Computer Science*, pages 227–256. Springer, 1994.

[16] E. S. L. Lam, I. Cervesato, and N. Fatima. Comingle: Distributed logic programming for decentralized mobile ensembles. In *Coordination Models and Languages – 17th IFIP WG 6.1 International Conference, COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2015.

[17] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[18] H. J. Levesque, P. R. Cohen, and J. H. T. Nunes. On acting together. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 94–99. AAAI Press / The MIT Press, 1990.

[19] D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423, 2002.

[20] J. Thangarajah, L. Padgham, and M. Winikoff. Detecting & avoiding interference between goals in intelligent agents. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 721–726. Morgan Kaufmann, 2003.

[21] M. Wooldridge and N. R. Jennings. The cooperative problem-solving process. *Journal of Logic and Compututation*, 9(4):563–592, 1999.