Interaction Patterns in a Multi-Agent Organisation to Support Shared Tasks

Moser Silva Fagundes, Felipe Meneguzzi, Renata Vieira, and Rafael H. Bordini

Postgraduate Programme in Computer Science – School of Informatics (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS) – Porto Alegre – RS, Brazil
{moser.fagundes,felipe.meneguzzi,renata.vieira,rafael.bordini}@pucrs.br

Abstract. We aim to help the coordination of the activities of groups of users who share certain tasks. In particular, we are working towards automatically predicting the context of each user, in particular which task each user is trying to accomplish. We also intend to predict how probable it is that users will be able to successfully accomplish theirs tasks. In case a failure is likely, we help the users in negotiating task reallocation among group members. This paper presents the interaction patterns we use for information exchange among agents in order to determine the context needed to make those predictions.

1 Introduction

In daily life, groups of people cooperate to successfully complete tasks that are in the interest of their respective members. The members of such groups can be geographically distributed and subject to setbacks which unexpectedly interrupt the progress of their activities. For example, consider a delivery service whose employees pick up and deliver packets in geographically distributed locations. Such activities can be interrupted by traffic jams, cancellations of service orders, mechanical problems in the vehicles, etc. When the activities unfold as planned, coordination requires minimal attention to detail. However, when the plans deviate from the expected courses of action, it is expected that someone estimates how likely it is that the plan will fail. If a plan is expected to fail, then a reallocation of tasks could be made. For example, if an employee gets stuck in a traffic jam and realizes that he will not arrive in time to make a programmed pickup, he can call another employee to perform this task.

In such dynamic and unpredictable environments, the Multi-Agent Systems (MAS) paradigm [3, 5, 4, 6] can be employed to develop complex applications that integrate multiple autonomous entities, both human and computational. Such applications can be designed to facilitate the interaction of users operating several types of devices (e.g., smartphones, tablets, laptops), predicting which tasks each group member is trying to achieve and suggesting possible alternative courses of action that might increase the chances of success in finishing all the group's tasks. However, to make possible the development of predictive multi-agent applications like these, we need to enable autonomous agents to provide and request relevant information to each other so as to reconstruct the *context* of the users.

In this paper, we put forward a multi-agent organisation to support activities shared among members of a group, focusing on the description of the agent roles and interaction patterns between the roles so as to exchange information in order to gather the context of all the users. We show how these agent interaction patterns can be implemented in the *Jason* agent programming language [1] and we illustrate our approach by means of a scenario related to a delivery service company.

The remainder of this paper is organised as follows. Section 2 describes our proposal for a multi-agent organisation. Section 3 introduces a scenario to illustrate our approach. Finally, Section 4 draws some conclusions and points towards future work directions.

2 A Multi-Agent Organisation to Support Group Activities

A multi-agent organisation consists of a collection of roles and relationships which govern the behaviour of the agents [2]. In such organisations, roles can be employed to determine suitable interaction partners by providing additional information about the individuals. In this section, we employ the notion of organisation to describe a multi-agent system to support the activities of a group of users.

2.1 Roles

There are two roles in our MAS organisation: Interface Agent and Planner Agent. Interface agents operate in devices of the human users (e.g., smartphones, tablets, laptops). These agents encapsulate the methods needed to run properly in particular devices, taking into account their hardware and operating system configurations. An agent playing this role collects information about the human user from different sources (e.g., social networks, calendar, GPS) and provides information to the planner agent (this role is detailed below). An interface agent can deliver information to a planner agent in two ways: (i) *proactively*, when the interface agent believes that the information is relevant to the tasks carried out by the planner agent; and (ii) *reactively*, when the planner agent requests a particular information. Within our organisation, there is one interface agent per device, and this agent interacts with one user and his respective planner agent.

Planner agents operate in "the Cloud" and they interact with interface agents in order to get information about the users. In our multi-agent system, there is one planner agent per user, and this agent can interoperate with the interface agents running on various devices of this user (often a user interacts with several devices such as smartphones, tablets, etc. for the same task, depending on their current context). This way, the planner agents can infer the context of the users on the basis of information from multiple devices. This is fundamental to the type of system that we envision, given that all complex tasks performed by the planner agents (recognition of intentions, negotiation, and task reallocation) are based on the users' context. The agents playing this role are designed to run the reasoning processes that require high computational performance, hence alleviating the burden of the interface agents running on portable devices that have limited computational resources.

In summary, for each user there is one planner agent and a set of interface agents (at least one such agent). Figure 1 illustrates a scenario in which there are three users

(`userA`, `userB`, and `userC`) and three planner agents (`pa`, `pb`, and `pc`). The arrows between these agents indicate that they are capable of interacting with each other (in this case, they interact to exchange information about the users). Each planner agent communicates with one or more interface agents of the same user. For instance, `pb` communicates with three interface agents, namely `ib1`, `ib2`, and `ib3`, running on a laptop, a tablet, and a smartphone, respectively. The arrows between `pb` and these interface agents indicate that they are capable of interacting. Figure 1 also shows that a single user can have multiple devices (for example, `userA` has a smartphone and a tablet).

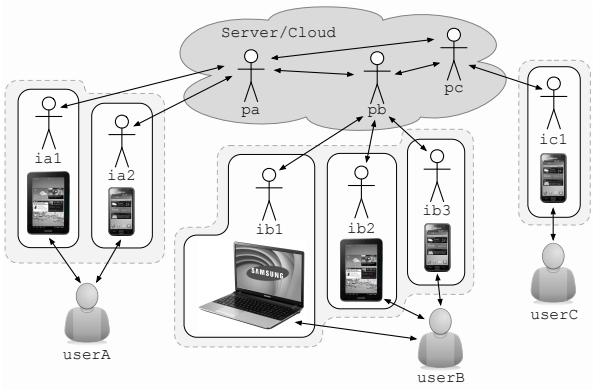

Fig. 1. Scenario with three users, three planner agents, and six interface agents.

2.2 Agent Interactions

Within the multi-agent organisation, the following interactions are allowed:

- **Interface Agent– Planner Agent** .
  - **Interaction #1:** An interface agent is capable of proactively sending information to its planner agent. This behaviour is triggered by the arrival of new information that the interface agents believe to be relevant to the construction of the user context. Figure 2(a) shows the protocol for this interaction.
  - **Interaction #2:** A planner agent can tell its respective interface agents about which information it considers to be relevant. Figure 2(b) shows the protocol for this second type of interaction.
  - **Interaction #3:** A planner agent is capable of asking the related interface agents for information in order to construct and update the context of a user. In this interaction, the planner agent asks for specific information, and the interface agent returns such information or tells that it is not available. The protocol for this interaction is specified in Figure 2(c).
- **Planner Agent– Planner Agent** .

Interaction #4: planner agents are capable of asking other planner agents for information in order to create or update a representation of the users' context. This interaction follows the same protocol as Interaction #3, except that it happens between two planner agents.

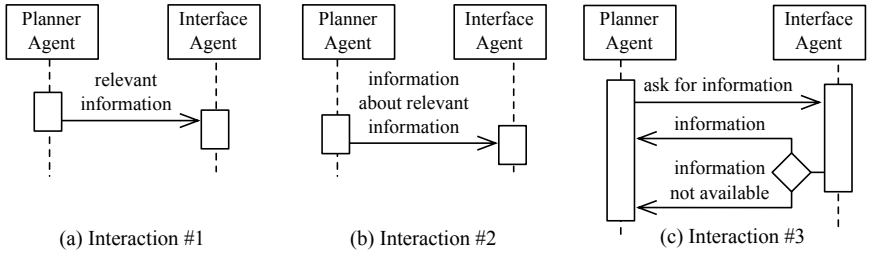

Fig. 2. Interaction protocols.

2.3 Jason Implementation

This section shows how to implement in *Jason*[1] the interactions described in the previous subsection. Before describing the implementation of the interactions, we describe the agents' beliefs about the organisation structure. Agents playing the `interface` role know their respective `user` and `planner`. For example, `ia1` in Figure 1 believes that:

```
user(userA).
planner(pa,userA).
```

Agents playing the role `planner` have beliefs about their respective user and interface agents, and about other planners. For example, `pb` in Figure 1 believes that:

```
user(userB).
planner(pa,userA).
planner(pc,userC).
interface([ib1,ib2,ib3]).
```

An interface agent implements **Interaction #1** using the following plan template:

```
@tellXxx[interaction(1)]
+!tellXxx(Y) : user(User) <-
     ?xxx(Y);
     ?planner(Agent,User);
     .send(Agent,tell,xxx(Y)).
```

in which `xxx(Y)` is the belief (information) to be told; the "context part" of the plan instantiates `User`, which is used in the body of the plan to select the name of the planner.

In **Interaction #2**, a planner agent tells its interface agents about which information it considers relevant. We propose an implementation of such interactions using the

[1] For the sake of space, this paper assumes that the reader is familiar with the *Jason* programming language. For details about the *Jason* platform, see Bordini et al. [1].

`tellHow` performative, by means of which the planner informs the interface agents of plans that should be executed when they get to know something that is relevant to the planner. Our implementation uses the following plan template:

```
@tellXxx[interaction(2)]
+!tellXxx : user(User) <-
     ?interface(InterfaceAgents);
     .my_name(Agent);
     .concat("@tellXxx [interaction(2)] ",
             "+xxx(Y) : true <- .send(", Agent, ",tell,xxx(Y)).", Plan);
     .send(InterfaceAgents,tellHow,Plan).
```

in which `xxx(Y)` is the belief (relevant information) that interface agents are asked to provide to the planner. The plan body consists of retrieving the names of interface agents, and specifying and sending the plan to be executed by them.

Interaction #3 takes place when a planner agents fails to retrieve some information and asks its adjacent interface agents for this information. This third type of interaction is implemented in *Jason* by means of plans triggered by test goals of the planner, which are specified according to the following plan template:

```
@determineXxx[interaction(3)]
+?xxx(Y) : user(User) <-
     ?interface(InterfaceAgents);
     .selectIA(InterfaceAgents,Agent);
     .send(Agent,askOne,xxx(Y),xxx(Y));
     +xxx(Y)[source(Agent)].
```

This plan retrieves the list of interface agents and selects one of them with `selectIA`. The name of the selected agent is stored in variable `Agent`. Then, the planner agent sends an `askOne` message to `Agent`. If `Agent` successfully unifies `xxx(Y)` with its beliefs, then the planner agent adds this predicate to its belief base, which will include an annotation that indicates the source of this information.

Interaction #4 is similar to Interaction #3, except that the planner asks another planner instead of an interface agent. These interactions use the following template:

```
@determineXxx[interaction(4)]
+?xxx(Y) : not user(User) <-
     ?planner(Agent,User);
     .send(Agent,askOne,xxx(Y),xxx(Y));
     +xxx(Y)[source(Agent)].
```

in which `xxx(Y)` is the belief (information) to be acquired. Note that the context part of this plan is "`not user(User)`", which indicates that this plan is applicable only when the agent needs to know something about a user that is not its own.

3 Delivery Service Scenario

This section describes a delivery service scenario to illustrate our approach. There are three employees participating in this scenario, and each employee drives a vehicle to make programmed pickups and deliveries. The structure of the organisation is the same used in Figure 1, so we can say that `userA` corresponds to `employeeA`, `userB` corresponds to `employeeB`, and `userC` corresponds to `employeeC`.

Consider a situation in which `pa` aims to determine the location of `employeeA` and `employeeC`. In the code of `pa`, specified in Listing 1, `@someGoal` tries to unify

`?location(employeeA,LocEA)` and fails because it has no beliefs about the location of `employeeA`. As an attempt to determine this location, agent `pa` triggers `@determineLocation[interaction(3)]`, which sends an `askOne` message to one of the other related interface agents `[ia1,ia2]`.

Listing 1 - pa

```
user(employeeA).
interface([ia1,ia2]).
planner(pb,employeeB).
planner(pc,employeeC).

@someGoal
+!someGoal : true <-
     ?location(employeeA,LocEA);
     ?location(employeeC,LocEC); ...

@determineLocation[interaction(3)]
+?location(User,Location) : user(User) <-
     ?interface(InterfaceAgents);
     .selectIA(InterfaceAgents,Agent);
     .send(Agent,askOne,location(User,Location),location(User,Location));
     +location(User,Location)[source(Agent)].

@determineLocation[interaction(4)]
+?location(User,Location) : not user(User) <-
     ?planner(Agent,User);
     .send(Agent,askOne,location(User,Location),location(User,Location));
     +location(User,Location)[source(Agent)].
```

Assume that planner agent `pa` selects the interface agent `ia1` (Listing 2) as the receiver of the message. When the agent `ia1` receives the message, it attempts to unify `location(employeeA,LocEA)` and fails to do so. This failure triggers the addition of a test goal, which is handled by plan `@determineLocation`. This plan reads the current location from the tablet's GPS of `employeeA` (`tablet.getLocation`). When `location(employeeA,Location)` is added to the belief base of `ia1`, it replies to `pa`, which resumes the execution of `pa`'s plan.

Listing 2 - ia1

```
user(employeeA).
planner(pa,employeeA).

@determineLocation
+?location(User,Location) : user(User) <-
     tablet.getLocation(Location);
     +location(User,Location).
```

Further, in the body of plan `@someGoal`, the agent `pa` tries to unify `?location(employeeC,LocEC)` and fails. This failure triggers the plan `@determineLocation[interaction(4)]`, which sends an `askOne` message to the respective planner agent (in this case, `pc`), starting an instance of Interaction #4. Sometimes, the planner agents do not have the information requested by other agents. In this case, they have to interact with their interface agent in order to get the requested information. For instance, consider a situation in which `pc` does not know the location of `employeeC`. So, when `pa` asks `pc` about the location of `employeeC`, `pc` asks its interface agents about it (this is Interaction #3 taking place within Interaction #4). This can be seen in Listing 3 and Listing 4, the code for `pc` and `ic1`, respectively.

Listing 3 - pc

```
user(employeeC).
interface([ic1]).
planner(pa,employeeA).
planner(pb,employeeB).

@determineLocation[interaction(3)]
+?location(User,Location) : user(User) <-
     ?interface(InterfaceAgents);
     .selectIA(InterfaceAgents,Agent);
     .send(Agent,askOne,location(User,Location),location(User,Location));
     +location(User,Location).
```

Listing 4 - ic1

```
user(employeeC).
planner(pc,employeeC).

@determineLocation
+?location(User,Location) : user(User) <-
     smartphone.getLocation(Location);
     +location(User,Location).
```

4 Conclusion

This paper presented an organisation of agents to support group activities, focusing on the specification and implementation of interaction patterns between the agent roles as an infrastructure to enable the exchange of information about users and their context.

There are three main directions for future work. The investigation and development of plan recognition and negotiation techniques using contextual information for proactive multiuser assistance, as well as the use of ontologies to support the generation of plans for the *Jason* platform using the templates introduced in this paper.

Acknowledgements

Part of the results presented by this paper were obtained through the project named Semantic and Multi-Agent Technologies for Group Interaction, sponsored by Samsung Eletrônica da Amazônia Ltda., under the terms of Law number 8.248/91.

References

1. Bordini, R., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. Wiley Series in Agent Technology, John Wiley & Sons (2007)
2. Horling, B., Lesser, V.R.: A survey of multi-agent organizational paradigms. Knowledge Eng. Review 19(4), 281–316 (2004)
3. Jennings, N.R.: On agent-based software engineering. Artif. Intell. 117(2), 277–296 (2000)
4. Jennings, N.R.: An agent-based approach for building complex software systems. Commun. ACM 44(4), 35–41 (2001)
5. Jennings, N.R., Sycara, K.P., Wooldridge, M.: A roadmap of agent research and development. Autonomous Agents and Multi-Agent Systems 1(1), 7–38 (1998)
6. Wooldridge, M.: Agent-based software engineering. IEE Proceedings - Software 144(1), 26–37 (1997)