THE UNIVERSITY OF NEW SOUTH WALES

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Minimal Infrastructure Support for Teams

Minh Hoai Nguyen

Thesis submitted as a requirement for the degree of

Bachelor of Engineering (Software Engineering)

Submitted: November 1, 2005

Supervisor: Associate Professor Wayne Wobcke
Assessor: Associate Professor Arthur Ramer

Abstract

Building a group of agents which can work effectively as a team is more than just merely putting individual agents together. Agents working in a team require the abilities to plan, communicate and coordinate with one another. Furthermore, systems in which agents are simply equipped with precomputed coordination plans will fail to work in complex, dynamic environments. Thus it is extremely difficult to design and build a system of collaborative agents from the scratch.

This thesis describes MIST, a SharedPlans framework for team-based applications. MIST is based on SharedPlans but is further extended to deal with computational issues which are not directly addressed by SharedPlans. MIST is also striving towards a flexible, general framework which could reduce the implementation effort of developing practical applications.

Acknowledgements

First, I would like to thank my supervisor A/Prof Wayne Wobcke for his invaluable guidance, advice and encouragement throughout the project.

I would like to thank my family for their constant support and love.

I would also like to thank people of the class COMP4416 Intelligent Agents S2 2005. Their participation in class created many interesting discussions which helped me to shape my ideas.

I would like to thank Ms. Anh Nguyen for some general advice.

Last but not least, many thanks to Victor Phung for proofreading my thesis.

Contents

List of Figures

Chapter 1

Introduction

What are the most desirable skills that employers are looking for from their potential employees nowadays? Perhaps they are teamwork skills. Team activities are becoming increasingly important and popular as tasks are getting more and more complex. In team activities, the cooperative effort of team members enables them to achieve goals that no individual alone could possibly achieve. Also, collaborative activities have the potential of producing *synergy* [24] as "the whole is greater than the sum of its parts".

As in the case of human society, teamwork in the world of agents is also becoming more and more critical. Team activities can be seen in many types of multi-agent environments ranging from systems that employ teamwork to coordinate autonomous agents to systems that seek to understand interactions among human societies. Some examples of such systems are Robocup robotics and synthetic soccer [12], electronic commerce [11], defence simulation [14], and modelling team tactics in whole air missions [22, 23].

Unfortunately, it is often difficult to construct team-based applications. Creating teams of agents requires more than just grouping some individual agents together. As in human societies, specific teamwork skills are necessary for effective team activity. More precisely, to work effectively in a team, agents must be equipped with the abilities to plan, communicate, coordinate and cooperate with one another. Furthermore, the complexity, dynamicity and uncertainty of many environments are contributing factors that lead to failures in many systems. For those environments, highly flexible coordination and communication is essential as simply fitting precomputed coordination plans will not work [19].

An example in the military domain

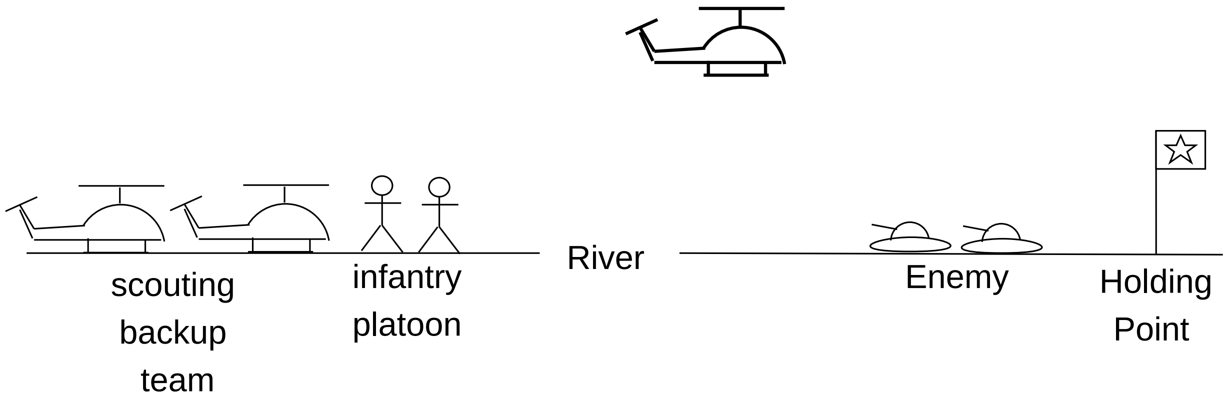

Figure 1.1: An example in the military domain

To see the difficulties in developing team-based application, consider the example in the military domain which is depicted in Figure 1.1. A team of several scouting helicopters and an infantry platoon receive a command from the top general to move to the battle field. The mission is to transport the majority of the infantry platoon to the holding point which is next to the battle field. The mission is still considered successful even if some helicopters or soldiers are shot down. However, since the scouting helicopters cannot carry many people, most soldiers will have to go on foot to the battle field. Furthermore, it is not safe for the infantry platoon to move to the battle field without having the path being scouted first. The infantry platoon is only armed with light weapons and will not be able to fight against the enemy if they encounter some strong enemy fire such as tanks and armoured vehicles. Thus several helicopters must fly forwards to the holding point, scout and eliminate all enemy troops on the way. There are two types of helicopters, light helicopters and heavy armed helicopters. The former type has the advantage of flying faster and thus can dodge bullets and rockets easier. However, they are only equipped with machine guns which are not enough to destroy enemy tanks. The heavy armed helicopters move slower but can destroy enemy tanks using rockets. When an area needs to be scouted, the light helicopters are normally used. The heavy armed helicopters are only used in situations when no more light helicopters are available or strong enemy forces are encountered. The rest of the team, while waiting for the scouting activity to be completed, need to construct a bridge to cross the river as the the old one was destroyed by the enemy aircraft. Thus, the infantry platoon can only move to the battle field once the bridge has been built and the path has been scouted successfully.

The above example exhibits several challenges. First, the domain environment is fairly dynamic with the involvement of many agents. Second, there is uncertainty in the environment such as the existence of enemy troops. Third, agents might be faced with incomplete or inconsistent information. No agent has complete knowledge about the environment and other agents. Thus, no single agent alone is capable of controlling and directing the other agents. Fourth, agents might be shot dead and cannot participate in team activities any more. Fifth, plans need to be repaired or changed if something unexpected occurs. Thus, it is extremely difficult to construct such an application from the scratch. One must consider many situations which could potentially lead to failures of systems. Several of such situations are (Tambe [19]):

- The scouting team fails to assign roles to its members. They do not know who should fly to the battle field first and who should stay home for the backup plan.

- The first scouting team reaches the destination. They think their mission is over and do not report at all or only report it to the chief general.

- The first scouting team reaches the holding point. However, they only report that information to the infantry platoon. The backup scouting team, receiving no information from the first scouting team, fly unnecessarily to the holding point as they assume the first scouting team are all dead.

- All members of the scouting team are shot down. No information is reported to the base. As a result, the infantry platoon wait forever.

- The first scouting team encounters some enemy troops, abort the mission and turn back. They report the information to everyone that the scouting task has failed. Upon receiving the message, the infantry platoon abort the mission since they do not know about the backup plan for the scouting action.

- The infantry platoon could not build a bridge because the river is too wide and the current is too strong. Hence, they must abort the action of building the bridge and cannot proceed to the battle field. However, the platoon did not notify the scouting team appropriately. As a consequence, some members of the scouting team continue sacrificing for nothing.

- The first scouting team scouts the path but do not see any enemy troops as they are masked under the bush. The scouting team reach the holding point and notify the infantry platoon. The infantry platoon finish building the bridge and start moving towards the battle field. However, they are all captured by the enemy troops. No information is propagated to the scouting team and they wait indefinitely at the holding point.

As can be seen from the above example, there are so many issues that a team-based application needs to address, such as, team formation, team planning, coordination, synchronisation. Communication and monitoring can be used to tackle some of those issues. However, monitoring is not always possible and communication is usually associated with a cost. Thus another question is what agents need to monitor, what and when to communicate. Consider the following examples to see how communication can supplement monitoring in team plan coordination.

Consider the case when you and your mum are cooking dinner together. This is an important dinner since your dad invites his boss over. So, it is really necessary to have the dinner ready on time. Your mum will be making some grilled chicken while you are baking a Hawaiian pizza. Both of these two dishes require the oven. However, the oven is so tinny that you and your mum decide that the chicken will be cooked first then the pizza. Now further suppose that while waiting for the chicken to be cooked, you stick around in the kitchen and help your mum. When the chicken is done, your mum takes it out and then you put your pizza in. In this situation no communication is made since you can monitor your mum's activity.

Now, suppose you are addicted to *The Simpsons* and you want to watch it while waiting for the chicken. Fortunately, the TV set is right in the living room which is next to the kitchen. You are very confident that you will smell grilled chicken when your mum takes the chicken out of the oven. So, you go to the living room and enjoy *The Simpsons*. However, when your mum takes the chicken out, she immediately puts it in a pan and covers it up to keep it away from a hungry cat which is wandering around. Therefore, you do not smell anything at all. Fortunately for you, since your mum think that you probably do not know that she has finished with the oven, she comes to the living room and grab you. This situation exhibits an interesting behaviour: although you did not ask your mum to call you, she does call you since she believes it would help you.

Next, consider the case where you want to watch *The Simpsons* in the living room but your family do not like the food smell entering the living room. That is why you have to keep the door between the kitchen and the living room shut at all cooking time. You believe that you will not be able to know when the oven is finished while watching TV with the door shut. Therefore, you explicitly ask you mum to signal you when she has finished with the oven. Unfortunately, your mum refuses to cooperate since she is still very angry with the result of your latest mathematics test. Even worse, she threatens to cut off your spending money if you cannot bake the pizza on time. But since you desperately want to watch *The Simpsons*, you turn to your grandma who is sitting right at the kitchen table sewing a new hat for your grandpa. You ask her to call you when your mother finishes cooking the chicken. Luckily for you, your grandma always nods for everything her sweetie asks for, so you go and enjoy *The Simpsons*. This situation shows that an agent can seek for assistance from its teammate as well a third person who is not directly involved in the team activity.

Now, suppose you are sitting comfortably in the sofa watching *The Simpsons* and expecting your grandma to call you when the oven can be used. Suddenly, you remember that your grandma is an absent minded person. Thus you come to believe there are chances that grandma will not call you on time. Because of that you occasionally yell out "Grandma, has mum finished with the oven yet". If she replies "Not yet" then you could continue enjoying *The Simpsons*. However, if the reply is "Oh sorry sweetie, she finished 10 minutes ago" then you have to rush into the kitchen, bake the pizza straight away with the hope that 10 minutes is not a long delay. This scenario shows that not only can an agent acquire information by registering his interests with others but also by querying them occasionally.

The above scenarios illustrate how reasoning about starting conditions of actions can lead to communication and arrangement between agents in case monitoring is not possible. Of course, there are other situations where communication is really necessary.

Over the years, several teamwork models, both theoretical and architectural, have been proposed. Those models, though different in approaches and categories, all try to address some issues of teamwork mentioned above. Examples of such theories are the Joint Intention theory of Cohen and Levesque [5, 6], the SharedPlans theory of Grosz and Kraus [9, 10], and the social structure theory of Tidhar [20, 21].

Among many models for collaborative activities, SharedPlans provides a theoretically sound framework for studying teamwork. SharedPlans aims to define the mental attitudes an agent must have to engage in collaborative activities. Several applications have been reported to be inspired by the SharedPlans theory, such as, an e-commerce system [11], a distance learning tool [15] and a discourse structure model [13]. Although SharedPlans have been shown to be useful, there is no framework which supports the rapid development of SharedPlans applications. The applications of SharedPlans mentioned above are just isolated solutions for specific problems. Thus there is an emerging need for building such a framework.

This thesis focuses on the development of a SharedPlans framework. The aim is to construct a flexible, general SharedPlans framework which could reduce the implementation effort of developing domain specific applications. Our work has resulted in MIST (Minimal

Infrastructure Support for Teams) which will be described in this thesis. MIST follows the SharedPlans theory fairly closely but also considers other computational issues which are not addressed in SharedPlans. In particular, MIST addresses the issues of dynamic team formation, plan establishment, communication, coordination, performance monitoring and failure recovery. MIST is built on JACK Intelligent Agents platform [1].

The rest of this thesis is structured as follows. Chapter 2 provides a summary of BDI agents and teamwork models. The implementation of MIST is described in Chapter 3. It will be followed by Chapter 4 where we show how a specific application can be developed from MIST. Comparisons of MIST with other systems are discussed in Chapter 5. Finally, Chapter 6 concludes the thesis and raises some questions for future study.

Chapter 2

Agents and teamwork models

This chapter discusses agents and agent teamwork models. It first revises some basic concepts such as agent, belief, mutual belief, desire and intention. Next, it explains the differences between recipes and plans. Section 2.2 will discuss JACK^{TM} Intelligent Agents which is the platform which our system, MIST, is developed on. The final sections describe some teamwork theories and frameworks which are closely related to our work. They are JACK^{TM} Teams, STEAM and SharedPlans theory.

2.1 BDI Agents

Wooldridge and Jennings [25] define agent as hardware or software-based system which exhibits the following characteristics:

- autonomy: "*agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state*";
- social ability: "*agents interact with other agents (and possibly humans) via some kind of agent-communication language*";
- reactivity: "*agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents,*

the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it ";

- pro-activeness: "*agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative*".

Some other specific definitions used in the literature are listed by Franklin and Graesser [7]. Those definitions view agents in slightly different ways. However, most of them agree that the key properties of agents are *autonomy, reactivity and pro-activeness. Social ability* is a mandatory requirement of agents in the definition of Wooldridge and Jennings but not in others. Social ability, the ability to communicate, is an extremely important characteristic in multi-agent systems, especially in systems where agents need to work closely together as a team, because communication is essential for coordination.

A subclass of "rational" agents are BDI agents (Belief-Desire-Intention agents). BDI agents have certain mental attitudes of *beliefs*, *desires*, and *intentions* which correspond respectively to the information, motivational, and deliberative states of the agents [17].

2.1.1 Belief

Belief refers to mental acceptance of truth or validity of something. Each BDI agent maintains a set of beliefs about the environment in which it inhabits. An agent situated in a dynamic environment constantly observe the environment and changes its beliefs to be consistent with its observations. An agent might believe in something which later turns out to be false. However, for rational agents, beliefs must be consistent at all times.

2.1.2 Desire

Desire is a mental attitude of an agent which provides motivation for the agent's activities. It is not unusual for agents to have more than one desire at a time. Furthermore, the desires of an agent could even conflict with its beliefs. For example, an agent can have both the desire of loosing weight and the desire of having more chocolate to eat while believing that the more chocolate he eats the rounder he becomes.

2.1.3 Intention

Intention is another mental attitude of an agent which is regarded as the commitment of the agent to some choice of action [5]. Bratman [3] further argues that intention is a primitive mental state which cannot be reduced to beliefs and desires. Unlike desires which could be conflicting with each other, intentions must be consistent among themselves and with the beliefs of the agents.

According to Bratman [3], intention has three functional roles. First, prior intentions frequently pose means-end problems for deliberation. For example, if an agent in Sydney intends to go to Melbourne tomorrow, he must gradually fill in his plan by figuring out how to get there. Second, prior intentions constrain the adoption of new intentions which conflict with the existing ones. For instance, an agent who intends to stay home to study for the exam cannot consistently adopt a new intention to go to the cinema with friends on the same night. Third, intentions control the conduct of agents; an agent eventually acts on his intentions.

2.1.4 Mutual belief

Mutual belief is an important concept in team activities. A group of agents are said to have mutual belief about a proposition α if each agent believes in α, each agent believes the other agents believe in α, each agent believes that any other agent believes all other agents believe in α, etc. The argument covers the beliefs of each agent about any other agent's beliefs to any arbitrarily large depth.

2.1.5 Recipes and plans

Bratman [3] and Pollack [16] distinguish two kind of plans, plans which an agent knows and plans which an agent adopts to subsequently guide his actions. Pollack then names plans of the former type as “recipes”. A recipe for an action is just a set of sub-actions together with their constraints and orders of execution designed to achieve the action. Saying that an agent has a recipe for an action only means that the agent knows how to achieve that action. In contrast, an agent has a plan to do an action must hold some certain beliefs and intention about doing the action.

2.2 JACK^{TM} Intelligent Agents

JACK [1] is framework which encompasses the full Java syntax and extends it to provide agent-oriented features such as agents, plans and beliefs. Programs written in JACK can be preprocessed and converted to Java files which will then be compiled to Java virtual machine code using the standard Java compiler.

The structure of a typical JACK agent is depicted in Figure 2.1. Each agent has a static set of predefined plans. Each plan declares what events it can handle. The message dispenser receives external event messages from other agents or internal messages from some alive plans. Upon receiving a message, the message dispenser checks the type of the message and looks into the plan library to seek an appropriate plan which has the ability to handle messages of that kind. If such a plan is found, the message dispenser will create an instance of that plan and pass the responsibility of handling the message to it. The body of a plan contains a set of instructions (both the full power Java instructions and JACK's language extension) for handling the types of events which it is declared to handle. A plan can retrieve and update the agent's belief sets. It can also interact with other legacy systems (environment) by calling the Java interface functions of those systems. Furthermore, plans can be used in communication among agents; unlike the interaction between plans and the environment, communication between agents is done by exchanging messages (JACK messages). A plan can indirectly invoke other plans by posting internal messages to the message dispenser which, in turn, launches appropriate plans to handle the messages. Thus, it is possible for a plan to create some sub-plans, including concurrently executing sub-plans, to achieve a desired goal.

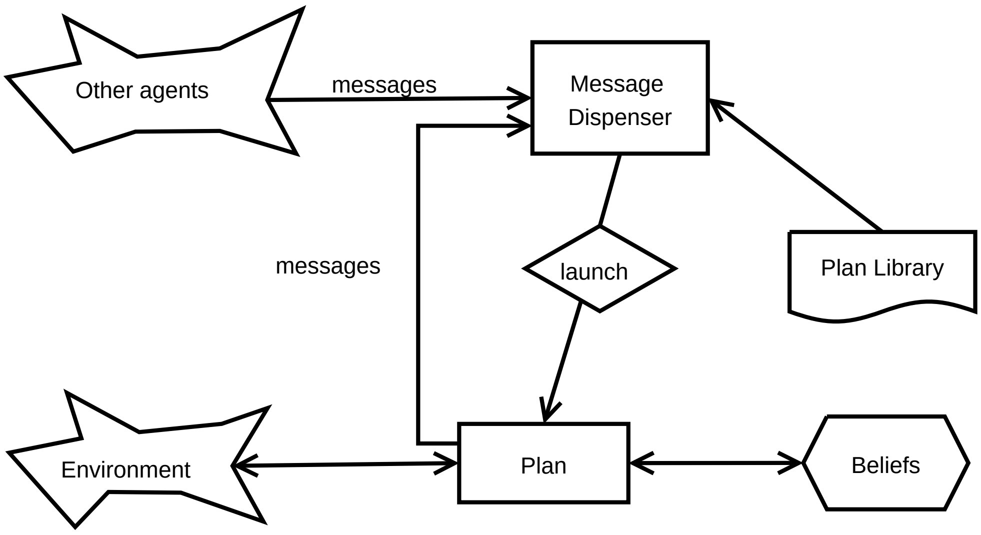

Figure 2.1: Structure of a typical JACK agent

There are some other features of JACK that make it appealing for agent development. First, it provides mechanism for parallel execution. Second, JACK includes a mechanism for handling plan failures. The message dispenser can invoke a new different plan to handle a particular message if the previously invoked plans failed. Furthermore, the default mechanism can be overridden by some special plans called meta-plans. Thus, it is possible to use JACK to build agents which exhibit plan-repairing abilities. The third interesting feature of JACK is the built-in types for belief sets. Compared with traditional Java classes, JACK's belief sets provide several additional functionalities such as unification, assertion, retraction and querying. In this manner, JACK is more or less similar to a Prolog system which supports a lightweight version of first order logic. This feature is very useful for creating agents with reasoning capabilities.

2.3 JACKTeams

JACKTeams [2] is an extension of JACK which supports team-oriented programming. This section discusses the team structure and coordination mechanism of teams developed by JACKTeams.

2.3.1 Team structure

The *team* is a basic entity in JACKTeams. Team can consist of sub-teams which, in turn, can have sub-sub-teams. A team knows its sub-teams but not normally the sub-teams of its sub-teams. It is possible for a team to participate in two different teams at the same time. Thus, it is possible to have a hierarchical team structure as in Figure 2.2. Each team has an abstract unique entity called *team agent*. Team agent is neither a sub-team nor a member of any sub-team; but it plays crucial role in team coordination (discussed in section 2.3.2). It is possible for teams to have no sub-teams. In that case, the team agent is identical to the team itself and the whole team can be regarded as a single individual agent.

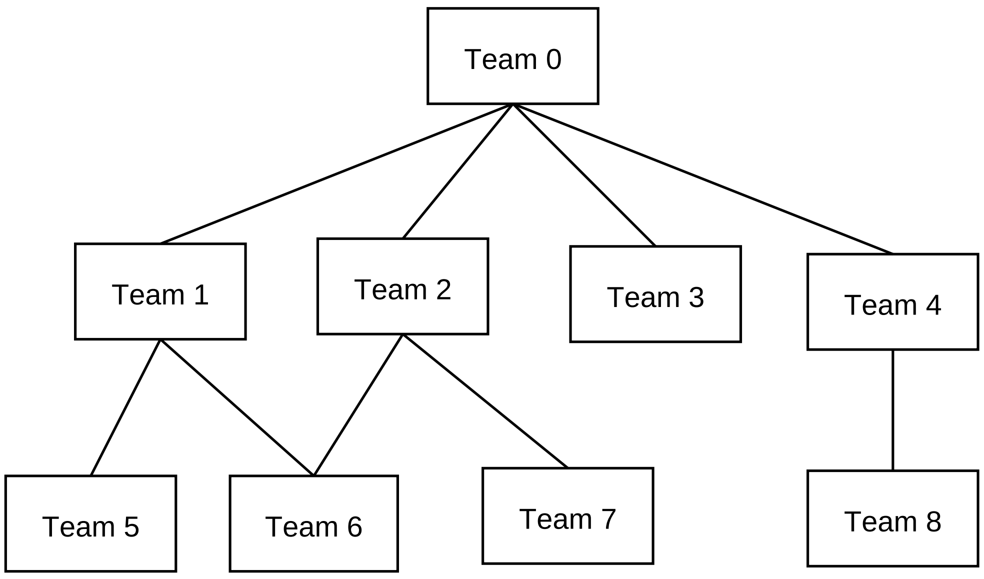

Figure 2.2: Hierarchical team structure

The team agent

The existence of the team agent is a distinctive feature of the JACKTeams approach. The team agent has its own beliefs, desires and intentions. Like individual agents, the team agent maintains a set of predefined reactive plans which would be used to handle some event messages when needed. We will refer to those plans as *teamplans*. The team agent declares what roles the team requires and what roles the team can perform as a sub-team in another team. The team agent is also responsible for finding sub-teams and assigning roles to them.

Relationship between team members

The relationships inside a team are many-to-one relationships. The team agent is the only one who establishes contacts and relationships with other members. It is possible for agents to involve in team activities without knowing their teammates. Thus, establishing mutual beliefs or joint intention in a team is unnecessary and irrelevant.

2.3.2 Team coordination

Team coordination of team members is handled completely by the team agent. The team agent is a reasoning entity of the team which acts like a commander who monitors and controls team activities. The team agent instructs team members what and when to do something. Teamplans are used by the team agent to serve that purpose.

Teamplans

The roles of teamplans to teams are analogous to those of individual plans to single agents. Each team, more precisely the team agent, maintains a set of predefined teamplans. The header of each teamplan declares what event messages it can handle. The body of a teamplan contains a set of instructions for how team members can coordinate and how the common goal can be achieved.

Consider the simplified code of an example of teamplan *FlyToTheMoon* in Figure 2.3. The beginning of the plan declares that it requires three roles, a technician, a main pilot and an assistant pilot. The plan does not abide roles to any specific agents or sub-teams in the plan definition stage. Instead, it uses the names *technician*, *mainPilot* and *assistantPilot* to refer to sub-teams which fill the roles. The plan also states that it can be used to handle the event *FlyToTheMoonEvent*.

```
teamplan FlyToTheMoon extends TeamPlan {
    #handles event FlyToTheMoonEvent;
    #uses role Technician technician;
    #uses role MainPilot mainPilot;
    #uses role AssistantPilot assistantPilot;
    body(){
        @team_achieve(technician, CHECK_ENGINE);
        @wait_for(elapsed(600));
        @team_achieve(technician, PUMP_FUEL);
        @parallel(SUCCEEDS_WHEN_ALL_SUCCEED){
            @team_achieve(mainPilot, FLY_SPACECRAFT);
            @team_achieve(assistantPilot, MONITOR_SPACECARFT);
        }
        @team_achieve(mainPilot, CALL_EARTH_CONTROL_CENTRE);
    }
}
```

Figure 2.3: A teamplan example

The body of the plan in Figure 2.3 basically expresses the sequence of execution of its sub-actions. Action *CHECK_ENGINE* must be executed first by the technician. *PUMP_FUEL* can be started 10 minutes (600 seconds) after the engine has been checked (Suppose, the engine gets hot after checking; that is why the technician must wait 10 minutes for it to cool down. Unlike car engines, the engine of this spacecraft might explode if the fuel is pumped in when hot). After the fuel tank has been filled, the spacecraft is ready for the mission to the moon. Actions *FLY_SPACECRAFT* and *MONITOR_SPACECRAFT* need to be done in parallel by *mainPilot* and *assistantPilot* respectively. Finally, when both actions *FLY_SPACECRAFT* and *MONITOR_SPACECRAFT* have been executed successfully (they arrive at the moon), the main pilot will need to inform the control centre by executing the action *CALL_EARTH_CONTROL_CENTRE*. The plan would fail and be aborted if any step of the above sequence fails and succeeds otherwise.

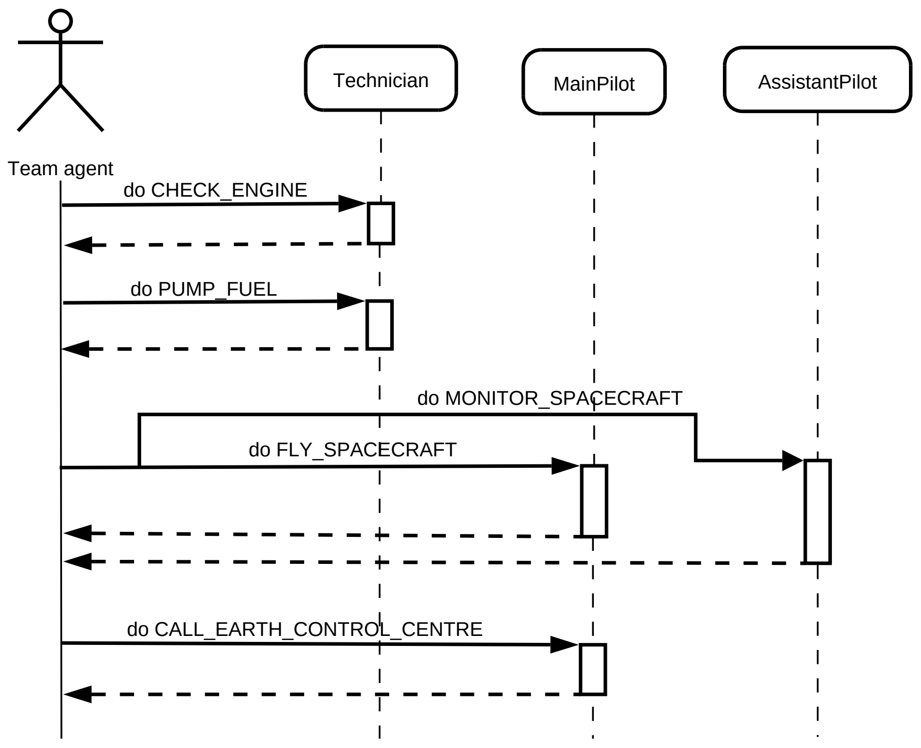

Figure 2.4: Team coordination in execution of FlyToTheMoon teamplan

Figure 2.4 shows the sequence of interactions between the team agent and other team members. First, the team agent tells the technician to check the engine. When the technician has done that, he reports it back to the team agent. The team agent looks at his watch and waits for 10 minutes. He then sends another instruction to tell the technician to fuel the spacecraft. After, the fuel tank has been filled, the technician reports back to the team agent. The team agent then instructs the pilots to fly to the moon by telling the main pilot to control the spacecraft while commanding the assistant pilot to monitor the spacecraft at the same time. Both the main pilot and the assistant pilot will individually notify the team agent when they finish their tasks. When the team agent knows both pilots have executed their actions successfully, it requests the main pilot to call the control centre. The main pilot does that and reports back to the team agent. At this time, the team agent knows that the teamplan has been executed successfully.

The execution of the *FlyToTheMoon* teamplan illustrates some important features of JACKTeams. Sub-teams do not know what they need to do until getting requests from the team agent. The team agent does not send several requests to the same sub-team at a time. Instead, it sends one request at a time and waits for the termination of the execution. Thus, it is not possible for the technician to continue to execute *PUMP_FUEL* after finishing executing *CHECK_ENGINE* without reporting back to the team agent.

2.4 STEAM

STEAM (A Shell for TEAMwork) is a framework for building team-based application. The construction of STEAM is aimed towards a flexible, general framework. This section outlines several key points of the STEAM framework which are described by Tambe in [18, 19]. In particular, this section describe the following aspects: theoretical foundations, team structure, team formation, the team state, team operator, plan establishment, monitoring and communication.

2.4.1 Theoretical foundation of STEAM

STEAM is based on the JointIntention theory of Cohen and Levesque [6] augmented by several concepts of SharedPlans theory. Cohen and Levesque focus on the definition of *joint intention*, the joint mental state of the team. A team jointly intend to do an action if they jointly committed to the action while mutually believing throughout that they were doing it. In turn, the joint commitment requires each agent to adopt a new goal of making the state of the joint action mutually believed by the group should the agent discovers the change in the state of the joint action. Thus, one could derive a communication obligation for agents from the JointIntention theory: an agent must notify his teammates once he discovers that the joint action has been achieved, will never be achieved or is no longer relevant. STEAM does enforce that communication obligation upon its agents. Unlike JointIntention, SharedPlans does not hold a key role in STEAM; it only lends the concept of hierarchical team structure and task decomposition to STEAM.

2.4.2 Team structure and team formation

STEAM supports the building of hierarchical team structure just as JACKTeams does. Each team has a team leader and might consist of individual agents or sub-teams. The structure of each team, however, is predefined. The team leader is also predetermined. Furthermore, teams cannot be formed dynamically. Therefore, there is no team formation in STEAM.

2.4.3 The team state

The *team state* is a distinctive approach of STEAM to representing mutual beliefs. Each agent maintains its own copy (no shared memory) of the team state which is the agent's model of the team's mutual beliefs. Each agent maintains a copy of the team state for every team that it participates in. The copy of the team state of each agent is initialised with information about the team, and it can be updated. However, the consistency of team members' copies of the team state must be maintained at all time. STEAM does this by restricting the modification of copies of the team state. If the agent modifies its copy of the team state, it must notify others so that they can update their copies of the team state accordingly.

2.4.4 Team operators and plan establishment

The team operator is a key novelty in STEAM. Team operators are reactive plans which are analogous to Pollack's recipes or JACKTeams' teamplans. They describe the set of actions together with their constraints which need to be done to achieve a goal. In fact, each team operator consists of preconditions rules, applications rules and terminations rules. In addition, the relationship between a team operator and its contributions from individuals or sub-teams is explicitly represented.

When agents want to execute a team operator, they must establish it as a joint intention. The process of joint intention establishment involves several steps. The team leader first broadcasts a message to request the execution of the team operator. Receiving requests from the team leader, other agents broadcast their commitments. The joint intention is established when every agent receives confirmation from everyone else. At this point, the agents will create or update their copy of the team states accordingly.

2.4.5 Monitoring and re-planning

STEAM supports monitoring of team performance by exploiting the explicit representation of the relationship between team operators and their constituent actions (called roles in Tambe [19]). Thus, a team operator succeeds or fails depending on the execution of the boolean combination of roles which it depends on. In STEAM, each agent independently monitor the performance of the team operator. Agents only communicate when one of them discovers that the team operator has failed, succeeded or is no longer relevant. STEAM does not facilitate the tracking the performance of teammates as they are partly domain dependent.

If, based on the performance monitoring, an agent discovers that a team operator α

is unachievable, it would invoke another team operator β for repairing the plan (if there is such an operator). However, the agent does not need to act as a leader to facilitate communication to establish the joint intention for the repairing plan. The agent assumes that everyone will invoke the same team operator once they discover that α is unachievable. Thus, commitment to the repairing plan β is automatically achieved.

2.4.6 Selective communication

Tambe reasons that the large number of team operators might result in significant communication overhead as the agents need to communicate to establish and terminate team operators. Therefore, STEAM is integrated with decision-theoretic selectivity. This involves assigning costs and rewards to the possible actions: *communication* and *not communication*. Furthermore, one must determine the probability of mis-coordination due to communication or not communication. From this setup, agents reason about a choice which yields the largest possible expected outcome. Thus, agents will communicate only when the benefit of communicating is expected to be greater than that of not communicating.

2.5 SharedPlans

Grosz and Kraus, in their papers [9, 10], propose a formalisation of the notion of a group of agents having a SharedPlan. The formalisation specifies mental attitudes an agent must have to engage in collaborative activities. It also identifies responsibilities and commitments of agents in group activities. The formalisation uses first-order logic enhanced by some modal operators, meta-predicates and action expressions. Some axioms are also stated to govern commitments and rational behaviour of agents. This section outlines several key points of the formalisation. First, it describes what Intention To and Intention That are. They will be followed by the explanation of definitions of SharedPlans. Finally, we reveals several axioms of SharedPlans. We divide the axioms into two categories, *ratio-*

nality axioms and *group commitment axioms* (The names ‘rationality axioms’ and ‘group commitment axioms’ are terminologies given by us, not by the authors).

2.5.1 Intention To (IntTo) and Intention That (IntThat)

In SharedPlans, there are two kinds of intentions: Intention To and Intention That. SharedPlans theory enforces some constraints on agents’ beliefs and commitments if they intend to do some actions. If an agent intends to do a basic level action, the agent must believe he can do the action and commit himself to doing the action. If the agent intends to do a complex action, he must have a recipe for the action and intend to do all constitute sub-actions. The recipe might be partial. In this case, the agent must intend to elaborate the recipe.

IntThat is a novel intentional attitude [18]. IntThat is used to represent an agent’s expectation that some proposition holds. IntThat is similar to IntTo in the sense that it rules out the adoption of conflicting intentions and it constrains re-planning in case of failure. There is, however, a significant difference between IntThat and IntTo. IntTo commits agent to means-end reasoning and acting. In contrast, IntThat does not necessarily impose that.

2.5.2 Full SharedPlans vs partial SharedPlans

The SharedPlans theory distinguishes between partial and full SharedPlans. A full SharedPlan for an action is a complete plan in which all details for how to achieve the goal have been determined. Partial SharedPlans are basically the same as full SharedPlans but with some incomplete information. For example, an agent might just have a partial recipe for the action or he might not know how to execute some sub-actions in the recipe.

Figure 2.5 provides a informal definition for full SharedPlans. The clause (1a) basically requires that every agent commits to the performance of the action A_α. Furthermore, it requires that the commitment of every agent must be mutually believed. Clause (1b) ensures that the group have a common recipe and that recipe must be complete. Since

For a group GR to have a Full SharedPlan (FSP) to do a complex action A_α, it is required that:

1a. GR mutually believe that each member agent intends that the group do A_α,

1b. GR mutually believe that they have a full recipe for doing A_α,

1c. Each action in that recipe is **fully resolved**.

A single-agent action A_β is **fully resolved** if:

2a. Some agent G_β in GR have a full individual plan to do A_β,

2b. GR mutually believe that G_β have a full individual plan to do A_β, and

2c. GR mutually believe that each member agent intends that G_β be able to execute A_β.

Similarly, a multi-agent action A_κ is **fully resolved** if:

3a. Some subgroup G_κ in GR have a full SharedPlan to do A_κ,

3b. GR mutually believe that G_κ have a full SharedPlan to do A_κ and are able to do A_κ, and

3c. GR mutually believe that each member agent intends that G_κ be able to execute A_κ.

Figure 2.5: Informal definition of a Full SharedPlan (FSP). Source: Grosz et al [8, 9]

the recipe is known and complete, all sub-actions of the recipe are known. The definition of full SharedPlans requires that every action in the recipe is fully resolved.

The second part of Figure 2.5 explains what it means for a single-agent action to be fully resolved. Basically, there must be an agent who has an individual plan[1] to do the action and whose identity is known by everyone. Furthermore, the existence of such a plan is mutually believed by the group. Clause (2c) requires that everyone commits to the success of that agent in executing the action.

The conditions for a multi-agent action to be fully resolved are quite similar to those of single-agent action. There must be a sub-group G_κ, whose identities are known by everyone, which have full SharedPlan to execute the action. Furthermore, the whole group must mutually believe in the existence of a full SharedPlan for G_κ to execute the action; but the identity of the SharedPlan is not necessarily known to the group. Clause (3c) requires the commitment of everyone to the success of G_κ in executing the action.

Figure 2.6 presents the informal definition of partial SharedPlans. According to the clause (4a), the group must hold the mutual belief that all agents commit to the success of the action. Clause (4b) requires the group to mutually believe and agree on a recipe. If the recipe is partial, i.e. only some sub-actions that they need to be performed are identified, the group must have a full SharedPlan to complete the partial recipe. While it is possible for some necessary actions to remain unknown, every known action must be either at-least-partially-resolved or unresolved. That is what clause (4c) is about. For example, suppose a group want to get to Melbourne from Sydney. The group agree on a recipe to fly there. However, they do not know all the details of that recipe. All they know is they have to go to the airport and board a plane. They do not know whether they need tickets in advance; they do not know whether they need to check-in or not. Thus, there are some actions which have not been determined yet. However, the group have a plan to elaborate the partial recipe to the full one by calling up Sydney airport. For actions which

[1]SharedPlans theory also provides definitions for full and partial individual plans. An agent has individual plans must also hold some appropriate beliefs and intentions. Those definitions are not discussed here but they can be found in [9].

A group *GR* have a Partial SharedPlan (PSP) to do a complex action A_α if:

- **4a.** *GR* mutually believe that each member intends that the group do A_α,
- **4b.** *GR* mutually believe that they have a full recipe for doing A_α, or they mutually believe that they have a partial recipe that may be extended into a full recipe they can use to do A_α and they have a full plan to select such a recipe, and
- **4c.** Each action in the (possibly partial) recipe be either **at-least-partially-resolved** or **unresolved**.

A single-agent action A_β is **at-least-partially-resolved** if:

- **5a.** Some agent G_β in *GR* has an individual plan to do A_β,
- **5b.** *GR* mutually believe that G_β has an individual plan to do A_β, and
- **5c.** *GR* mutually believe that each member agent intends that G_β be able to execute A_β.

Similarly, a multi-agent action A_κ is **at-least-partially-resolved** if:

- **6a.** Some subgroup G_κ in GR have a plan to do A_κ,
- **6b.** *GR* mutually believe that G_κ have a SharedPlan to do A_κ, and
- **6c.** *GR* mutually believe that each member agent intends that G_κ be able to execute A_κ.

A single-agent action A_ϵ is **unresolved** if:

- **7a.** *GR* mutually believe that some member of *GR* could do A_ϵ, and
- **7b.** *GR* have a full SharedPlan to select such an agent.

Similarly, a multi-agent action A_μ is **unresolved** if:

- **8a.** *GR* mutually believe that some subgroup of *GR* could do A_μ, and
- **8b.** *GR* have a full SharedPlan to select such a subgroup.

Figure 2.6: Informal definition of a Partial SharedPlan (FSP). Source: Grosz et al [8, 9]

the group have already identified such as *go to the airport* and *board a plane*, they must form appropriate intentions and beliefs.

An individual-agent or a multi-agent action is considered *at-least-partially-resolved* if some agent G$_\beta$ or some subgroup G$_\kappa$ are planning to do it. The identity of that agent or sub-group must be known by everyone. Furthermore, the group must mutually believe that there exists a plan (either an individual plan or a SharedPlan) which could be executed by the agent/sub-group in order to achieve the action. Lastly, clauses (5c) and (6c) state that the commitment of each agent to the success of agent G$_\beta$ or sub-group G$_\kappa$ in performing the action is mutually believed.

The last parts of Figure 2.6 define what an *unresolved* action is. Clauses (7a) and (8a) requires the group to mutually believe in the existence of an agent or a sub-group which could do the job. The identity of that agent or group is not required to be known. However, the group must have a full SharedPlan to select such an agent or group. For example, suppose a team of several terrorists are planning a suicide attack on a US embassy. The plan involves an action in which one terrorist must drive a bomb-loaded truck to the embassy. The whole group knows that everyone of them can do the job and that is mutually believed by everyone. However, since no one really wants to die, no one steps up to take the mission. Thus, the action has not been resolved yet. The action is considered *unresolved* if the group has a full SharedPlan to select such a person. For example, the group might plan to toss a coin to choose one randomly or they might plan to sacrifice the oldest person.

2.5.3 Rationality axioms

The SharedPlans theory of Grosz and Kraus [9] also provides several rationality axioms which constrain the relationship between beliefs, intentions and other mental attitudes. This section lists only two of those axioms for illustrative purposes.

Axiom 1:

Bel$(G, CONF(\alpha, \beta, T_\alpha, T_\beta, constr(C_\alpha), constr(C_\beta)), T_i)) \Rightarrow$

$\{[Int.Tx(G, \alpha, T_i, T_\alpha, C_\alpha) \Rightarrow \neg(Int.Ty(G, \beta, T_i, T_\beta, C_\beta))] \wedge$
$[Int.Ty(G, \beta, T_i, T_\beta, C_\beta) \Rightarrow \neg(Int.Tx(G, \alpha, T_i, T_\alpha, C_\alpha))]\}.$

Here, $Int.Tx, Int.Ty \in \{Int.To, Int.That\}$ and α/β may be either an action (if Int.Tx/y = Int.To) or a proposition (if Int.Tx/y = Int.That)

In Axiom 1 above, the meta predicate *CONF* is used to express that two actions, or two propositions, or an action and a proposition are conflicting with each other. $CONF(\alpha, \beta, T_\alpha, T_\beta, constr(C_\alpha), constr(C_\beta))$ holds if one of the following situations occurs:

1. Both α and β are actions. The execution of α under the constraints of the context of α conflicts with the execution of β under constraints of β's context. This conflict may arise either because the execution of one action will lead to situation in which the agent is no longer able to perform the other action, or because there are conflicts between the constraints on the performance of the two actions.

2. α is an action while β is a proposition. α and β are conflicting if the performance of α will cause β not to hold, or conversely, α cannot be performed if β holds.

3. α and β are both propositions which cannot simultaneously hold.

Thus, Axiom 1 basically states that an agent cannot, at any time, intend to do two actions which are believed to be conflicting with each other. The same happens for propositions. An agent cannot intend that two conflicting propositions hold.

Axiom 2:
$(\forall \alpha, T_i, T_\alpha)[\text{basic.level}(\alpha) \wedge Bel(G, Int.To(G, \alpha, T_i, T_\alpha, C_\alpha), T_i)$
$\Rightarrow Int.To(G, \alpha, T_i, T_\alpha, C_\alpha)]$

Axiom 2 basically states that if an agent G believes it intends to do a basic-level action (an action which cannot be divided further) then it really intends to do the action.

2.5.4 Group commitment axioms

Apart from rationality axioms, SharedPlans theory also provides some axioms which regulate agents' behaviour once they commit to group activities or the success of other agents. Axiom 7 is given below:

Axiom 7:

$(\forall \eta, G_1, T_i, T_\alpha)\{$

$[single.agent(\eta) \wedge$

$(\exists \alpha, T_\alpha, R_\alpha, \beta, G_2)[multi.agent(\alpha) \wedge (\exists GR)[$

$(G_1 \in GR) \wedge SharedPlan(GR, \alpha, T_i, T_\alpha, C_\alpha) \qquad \wedge$

$[Int.Th(G_1, (\exists R_\beta) CBA(G_2, \beta, R_\beta, T_\beta, constr(C_{\beta/\alpha})), T_i, T_\beta, C_{cba/\beta/\alpha})] \qquad \wedge$

$[cost(G_1, Do(GR, \alpha, T_\alpha, constr(C_\alpha)), T_\alpha, C_\alpha, R_\alpha,$

$\neg Do(G_1, \eta, T_\eta, constr(C_{\eta/cba/\beta/\alpha})) \wedge Do(G_2, \beta, T_\beta, constr(C_{\beta/\alpha}))) \quad -$

$cost(G_1, Do(GR, \alpha, T_\alpha, constr(C_\alpha)), T_\alpha, C_\alpha, R_\alpha,$

$Do(G_1, \eta, T_\eta, constr(C_{\eta/cba/\beta/\alpha})) \wedge Do(G_2, \beta, T_\beta, constr(C_{\beta/\alpha}))) \quad >$

$econ(cost(G_1, Do(G_1, \eta, T_\eta, constr(C_{\eta/cba/\beta\alpha})), T_\eta, C_{\eta/cba/\beta/\alpha}, R_\eta))] \qquad \wedge$

$Bel(G_1, (\exists R_\eta) CBA(G_1, \eta, T_\eta, constr(C_{\eta/cba/\beta\alpha})), T_i)]]] \qquad \Rightarrow$

$Pot.Int.To(G_1, \eta, T_i, T_\gamma, C_{\eta/cba/\beta/cba})\}$

Axiom 7 is about helpful behaviours in the SharedPlans context. In Axiom 7, G_1 and G_2 are two members of a group *GR*. The group *GR* have a SharedPlan to execute the group action α. G_2 is assigned to execute action β and G_1 knows that. Furthermore, G_1 commits to the success of G_2 in performing β (one requirement of SharedPlans). If there is an action η (an action which is not part of the recipe for the group action α) such that G_1 believes he can perform it, and furthermore, doing it would lessen the burden or the cost of G_2 in bringing about β, then G_1 would consider doing η. To express that, the axiom uses auxiliary functions *cost* and *econ*. Thus, the agent is judging the benefit to the group against the effort that he might need to put in. The modal operator Pot.Int.To stands for

‘Potential Intend To’. Pot.Int.To is a variation of IntTo which represents the mental state of an agent when it is considering adopting the intention. Grosz and Kraus [9] refer to Bratman [4] that potential intentions provide motivation for an agent to weight different possible courses of actions or options.

2.5.5 Summary

In short, the SharedPlans theory specifies mental attitudes an agent must have to engage in collaborative activities. Furthermore, using axioms, the theory constrains rational behaviours and commitments of agents in group activities. The theory, however, does not address some computational issues such as team coordination or communication. This will be discussed in the next chapter where we explain how SharedPlans is implemented.

Chapter 3

MIST - Minimal Infrastructure Support for Teams

This chapter describes MIST, a framework for team activities. MIST is based on SharedPlans which specifies what agents need to believe to engage in group activities. SharedPlans theory is used to guide the process of establishing team plans and decomposing complex actions. Meanwhile, the coordination mechanism requires team agents to reason about actions' starting and termination conditions and make necessary communications. This chapter first describes the language and platform that MIST is built on. Then, it discusses the team structure, team formation of MIST teams. They will be followed by the descriptions of recipe library and capability library. Finally, the chapter leads the reader through some key processes in plan establishment and plan coordination.

3.1 SharedPlans theory in MIST

One of the original motivations of MIST is to implement faithfully SharedPlans. This section describes aspects of SharedPlans which are and which are not incorporated in MIST.

SharedPlans is such a very complex theory that it is impossible to have an implementation in which all notations are explicitly represented and all axioms are strictly enforced. SharedPlans uses meta-predicates, modal operators, recursive definitions which are not computationally tractable. The authors also admit that SharedPlans is not intended to be directly implemented by approaches such as theorem proving systems. They suggest that it is rather be used as specification for agent design. MIST employs the definitions of full SharedPlans and partial SharedPlans to decide whether a team has a plan (SharedPlan) or not. Whether or not a SharedPlan is established is extremely important for an agent as he must form appropriate intentions towards constitute sub-actions. MIST, however, does not directly implement all the axioms of SharedPlans as they cannot be efficiently enforced in an automatic manners.

SharedPlans does not address computational issues directly. For example, the issues of what and when to communicate, what and how to monitor group performance, when and how to repair plans are not addressed. Furthermore, although the SharedPlans theory provides definitions for full SharedPlans and partial SharedPlans, it does not indicate how they can be achieved. In particular, how mutual beliefs could be attained is not actually specified. Thus, an implementation of SharedPlans needs to address issues which are absent in the SharedPlans formalism. How MIST handles those issues will be discussed in later sections of this chapter.

3.2 Language and Platform

MIST is developed on top of JACK; none of the teamwork features of JACKTeams are used. More precisely, the components used are agent, plan, event, belief set, capability. Components provided in JACKTeams such as team, teamplan, team data, role are not used at all. The only extension of JACK employed in MIST is the *@parallel* statement and *ParallelMonitor* class. Both *@parallel* statements and *ParallelMonitor* are also used in JACKTeams. However, the usage of parallel mechanism in JACKTeams are quite different from our usage. In JACKTeams, *@parallel* statements are used in team plans to assign several subtasks to some team members to execute them in parallel. In MIST, each individual agent might use *@parallel* statements to tell itself to handle several tasks in parallel. Some parts of MIST are written in Java such as the recipe and capability parsers. Both Java components and other components are compiled using the JACK compiler.

3.3 Team structure and team formation

MIST supports the building of nested hierarchical teams. Each team can consist of several individual agents or sub-teams or both individual agents and sub-teams. Each agent can participate in different teams and perform various roles at the same time. There is no restriction on the structure of teams.

Each team has a team leader which bears several extra duties in team activities. The team leader is responsible for initiating the group efforts in establishing mutual beliefs and commitments. He is also responsible for receiving, synthesising suggestions and making group decisions. The team leader is also in charge of monitoring the success or failure of team activities. The team leader has the obligation to notify other team members once he believes the joint activity has come to an end (either success or failure). In addition, the team leader acts as a representative of the team to the rest of the world. He will report the result of his team's joint activity to whoever needs the information in the outside world. Similarly, he will notify his team members if he detects an event affecting the plan in the outside world.

The team leader can be determined in one of the following two ways. In the first situation, if an agent sees the need for a group of agents, which includes himself, to participate in a joint activity, he will initiate the activity and volunteer himself as the team leader. In the second way, if an existing team leader sees the need of forming a sub-team of agents to execute a sub-action, he will randomly assign the team leader role to one of the agents in the new sub-team. The team leader of the parent action might not necessarily be the leader of the new sub-team even when he is part of the new sub-team.

Team formation is done dynamically. When an agent sees the need of executing a joint action, he chooses a group of agents which are potentially capable of executing the action.

His decision about which group to choose is based on his own knowledge of teammates' capabilities. He will sends messages to those agents requesting them to join group activity. The team is formed once everyone confirms their commitment.

Team organisation can be done in a top down fashion. This means that a team for a super-action is formed before sub-teams for some sub-actions are established. In seeking for a solution for the super-action, team members might discover the need of forming sub-teams to execute some of the sub-actions. Only in those situations will new sub-teams be formed.

3.4 Recipe library

The recipe library is a collection of recipes. In MIST, recipes are referenced by their unique names. Each agent has his own recipe library, and the recipe libraries of different agents are not necessarily identical. For instance, recipe library of an agent might contain some recipes which are not included in those of others. However, recipes referred by the same name in different recipe libraries must be exactly the same. This is to ensure that recipes can be identified by their names across all agents. In other words, it is enough for agents to communicate the names of the recipes only.

In MIST, the recipe library is initiated from a predefined database, more specifically, from a text file. The recipe library tends to be static and cannot be updated at run time. However, it should not be used as a characteristic of our system. With some more effort, the recipe library could be made dynamic. That means recipe library would be able to be updated and it would reflect the learning capability of agents.

3.4.1 Recipe

Each recipe stores enough information for how to achieve some actions. Every recipe declares what actions they support. In addition, each recipe states its sub-actions and the interdependencies between them. The interdependencies here could be constraints and precedences of actions. The interdependencies of sub-actions will be discussed in more details shortly. Each recipe also contains information about roles. Using roles is just a convenient way of grouping actions which must be performed by the same agent or sub-team. Finally, each recipe defines its success or failure conditions.

3.4.2 Recipe format

In order to understand the capability of the language used to define recipes, it is necessary to explain the format of recipes. The recipe library is initialised by reading a file containing a list of recipes. Each line of the recipe description is one of the followings.

$$\text{recipeName} => \mathit{supportAction}_1, \mathit{supportAction}_2, ..., \mathit{supportAction}_n \qquad (1)$$

$$\text{SUBACTIONS} : \mathit{subaction}_1, \mathit{subaction}_2, ..., \mathit{subaction}_m \qquad (2)$$

$$\text{SUCCEED\_WHEN} : \text{DependencyExpression} \qquad (3)$$

$$\text{roleName} :: \mathit{subaction}_{k_1}, \mathit{subaction}_{k_2}, ..., \mathit{subaction}_{k_l} \qquad (4)$$

$$\text{subaction}_h := \text{DependencyExpression} \qquad (5)$$

The minimum description of a recipe will contains at least the first three lines above. The first line of each recipe (1) tells us what actions the recipe supports. The next line (2) lists all the sub-actions contained in the recipe. The third line (3) expresses the success condition of the recipe. In other words, the third line tells us when the execution of the recipe would be considered successful. This would depend on the success or failure of some of its sub-actions. The format and capability of dependency expressions will be described in more detail below.

The first three lines are followed by any number of lines of the format in (4) or (5). A line in the format (4) describes a role with the role name followed by some sub-actions in the role. Using roles is just a convenient way to constrain that some sub-actions must be performed together by one agent or a sub-team of agents. A line written in the format (5) describes the starting condition for a sub action. The formula tells us that the sub-action on the left hand side should start when the conditions about some events in the right hand side are met.

Grammar of Dependency Expression

The dependency expressions are generated by the following grammar.

```
DependencyExpression -> "(" DependencyExpression ")"
DependencyExpression -> DependencyExpression "+" DependencyExpression
DependencyExpression -> DependencyExpression "*" DependencyExpression
DependencyExpression -> Time ":" Event | Time | Event
Event -> EventName "@" EventType | EventName
Time -> NUMBER
EventName -> STRING
EventType -> "SUSSCESS" | "FAILURE" | "STOP"
```

Semantics of Dependency Expression

- *Expr1 + Expr2* is satisfied iff either *Expr1* or *Expr2* is satisfied.
- *Expr1 * Expr2* is satisfied iff both *Expr1* and *Expr2* are satisfied.
- *Time : Event* is satisfied iff the event *Event* has occurred and since then the amount of time *Time* has elapsed.
- *Time* is just a special case of *Time:Event* when *Event* is empty.
- *Event* is another special case of *Time:Event* when *Time* is 0.
- *EventName @ SUCCESS* is satisfied if the the event with the name *EventName* has finished successfully. Similarly, *EventName @ FAILURE* is true when the event *EventName* has finished in failure. *EventName @ STOP* is true when either *EventName @ FAILURE* or *EventName @ SUCCESS* is true.
- *EventName* is a short form of *EventName @ SUCCESS.*

Example of a Recipe

Consider the following simple recipe example:

ScoutMoveRecipe => MoveToBattleField (6)

SUBACTIONS: Scouting,Scouting2,Moving,BuildingBridge,PumpingFuel (7)

SUCCEED_WHEN: Moving @ SUCCESS (8)

MainTroopRole :: Moving, BuildingBridge (9)

ScoutingRole :: PumpingFuel, Scouting (10)

Moving := 84600:BuildingBridge * (Scouting + Scouting2) (11)

Scouting := PumpingFuel * SunRise (12)

Scouting2 := Scouting@FAILURE (13)

Figure 3.1 is a graph representation of the recipe ScoutMoveRecipe described above. However, the figure only provides partly the information contained in the recipe. It depicts the interdependencies of the sub-actions.

Despite being simple, the above example illustrates several interesting points of recipes. The first line (6) tells us that the recipe name is *ScoutMoveRecipe* and it supports the action *MoveToBattleField.* The second line reveals all the sub-actions of the recipe. The sub-actions are *Scouting, Scouting2, Moving, BuildingBridge*, and *PumpingFuel.* It is worth mentioning that sub-actions in a recipe might not be basic actions. For example, the sub-action *BuildingBridge* might be a complex action. There might be several recipes for the action *BuildingBridge* alone.

The third line (8) expresses that the recipe is considered successfully executed if the sub-action *Moving* is executed successfully. Notice that the success condition is expressed solely in terms of the success of the sub-action *Moving*. This is because the main aim of the action *MoveToBattleField* is to move the majority troops to the battle field. The action would still be regarded successful even though some scouting helicopters are shot down. However, it is not quite true to say that the success of the recipe is purely dependent on the *Moving* action because *Moving*, in turn, relies on other events.

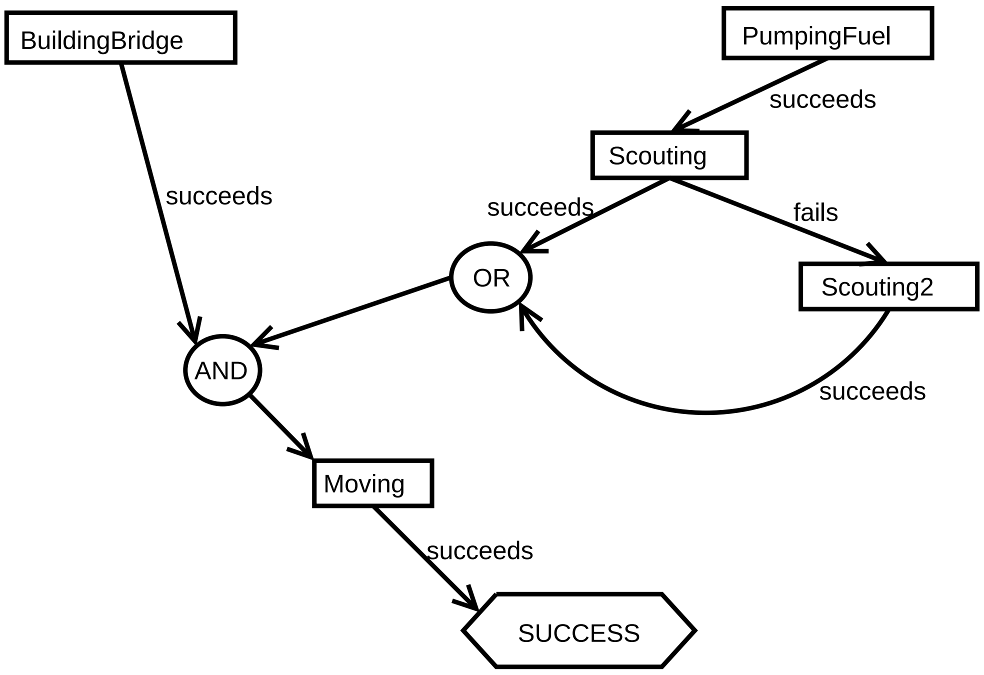

Figure 3.1: Sub-action interdependencies in ScoutMoveRecipe recipe

The next two lines (9) and (10) group some sub-actions together to form roles. Basically, line (9) states that *Moving* and *BuildingBridge* must be done by the same agent or sub-team. Similarly, line (10) forces whoever does the *Scouting* sub-action to do the sub-action *PumpingFuel* as well.

Line (11) expresses the starting condition of the action *Moving*. *Moving* should only be executed when at least one of the two scouting actions is successful and at least one day after the construction of the bridge is done. Obviously, it is not safe to move all the troops from the base to the battle field without having successfully scouted the moving path. Therefore, the *Moving* sub-action should wait for the scouting activity. Moreover, a bridge must be built to help the troops cross the river. That is why *Moving* cannot be done before the bridge is built. In addition, it is wise to let the team rest for as least one day after building the bridge.

Line (12) put constraints on the starting condition of *Scouting*. *Scouting* must be done after enough fuel has been filled and after *SunRise*. Note that *SunRise* is not one of the sub-actions, it is actually an external event.

The last line (13) illustrates an interesting point. The sub-action *Scouting2* should only be performed if the action *Scouting* cannot be successfully executed. This shows that the language used to describe recipes is rich enough to express the repair strategy for a recipe. This feature is very useful as it makes the writing of complex recipes possible.

3.5 Capability beliefs

Each agent maintains his own beliefs about the capabilities of himself and other agents. Like other belief sets, the capability beliefs of each individual can be incorrect but must not be inconsistent. For example, agent Bob might believe that Alice can cook roast chicken while Alice, in fact, cannot. Bob cannot both believe Alice can cook roast chicken and Alice cannot cook roast chicken at the same time. Furthermore, the capability belief set must be consistent with other belief sets. For instance, Bob cannot believe that Alice can cook roast chicken at 5pm and, at the same time, believes Alice can cook pizza at 5pm while believing that these two actions cannot be done in parallel by any single person.

In MIST, each agent is created with some initial beliefs about himself and others. Those initial beliefs are read in from text files. Those inborn beliefs can certainly be different for different agents. Unlike recipe libraries which tend to stay static, capability beliefs can be updated as agents have more and more experience. MIST employs a simple learning mechanism here. If the agent sees someone or a group of agents that can successfully execute an action, he would add that fact into his capability belief set. Conversely, if he notices some agents are struggling with a particular action, he would remove his belief about those agents' capability in doing the action if he did believe in them.

3.6 Establishment of SharedPlans

This section describes the coordination mechanism among team members to establish SharedPlans. To work effectively in a team, agents need to commit themselves to the team activity and supporting others. Communication is a means to share information, establish mutual beliefs, and reach group decisions. This section assumes all team members have been identified and they all have received requests to join the team. Team formation has been described in Section 3.3.

During the plan establishment phase, the team must establish mutual belief that all intending to join the group do the requested action. Then they must determine a common recipe that the group can execute. After that, the team must figure out what actions need to be done and who is going to do what. Finally, the group must establish mutual beliefs that all sub-actions are either *at-least-partially-resolved* or *unresolved*. Figure 3.2 displays the interaction diagram between the team leader and another team member. The following sub-sections will step more carefully through these processes. But first, we discuss how mutual belief can be achieved.

3.6.1 Attaining mutual belief

Although the concept of mutual belief is widely used in SharedPlans, no indication for how mutual beliefs can be achieved in practice is provided. The definition of mutual belief in section 2.1.4 is not an approach for obtaining mutual beliefs. In reality, reasoning agents (including humans) obtain mutual belief based on making additional assumptions about the environment.

In our system, we make some assumptions relating to communication and trust. First, the communication channel is reliable. In other words, an agent Alice sending a message to another agent Bob can assume that Bob receives the same message within a reasonable amount of time after the message is sent. On the contrary, if Bob does not receive any message from Alice, Bob can assume that Alice did not send him any message. Another assumption of our system is that an agent can trust and does trust other agents' words about their beliefs. In other words, Alice would trust what Bob says about Bob's beliefs and Bob believes that Alice trusts what he tells her about his beliefs. The above assumption does not require total trust of one agent in another agents' abilities. For instance, the fact that Bob sends a message "I can do α" to Alice does not entail that Alice believes Bob can

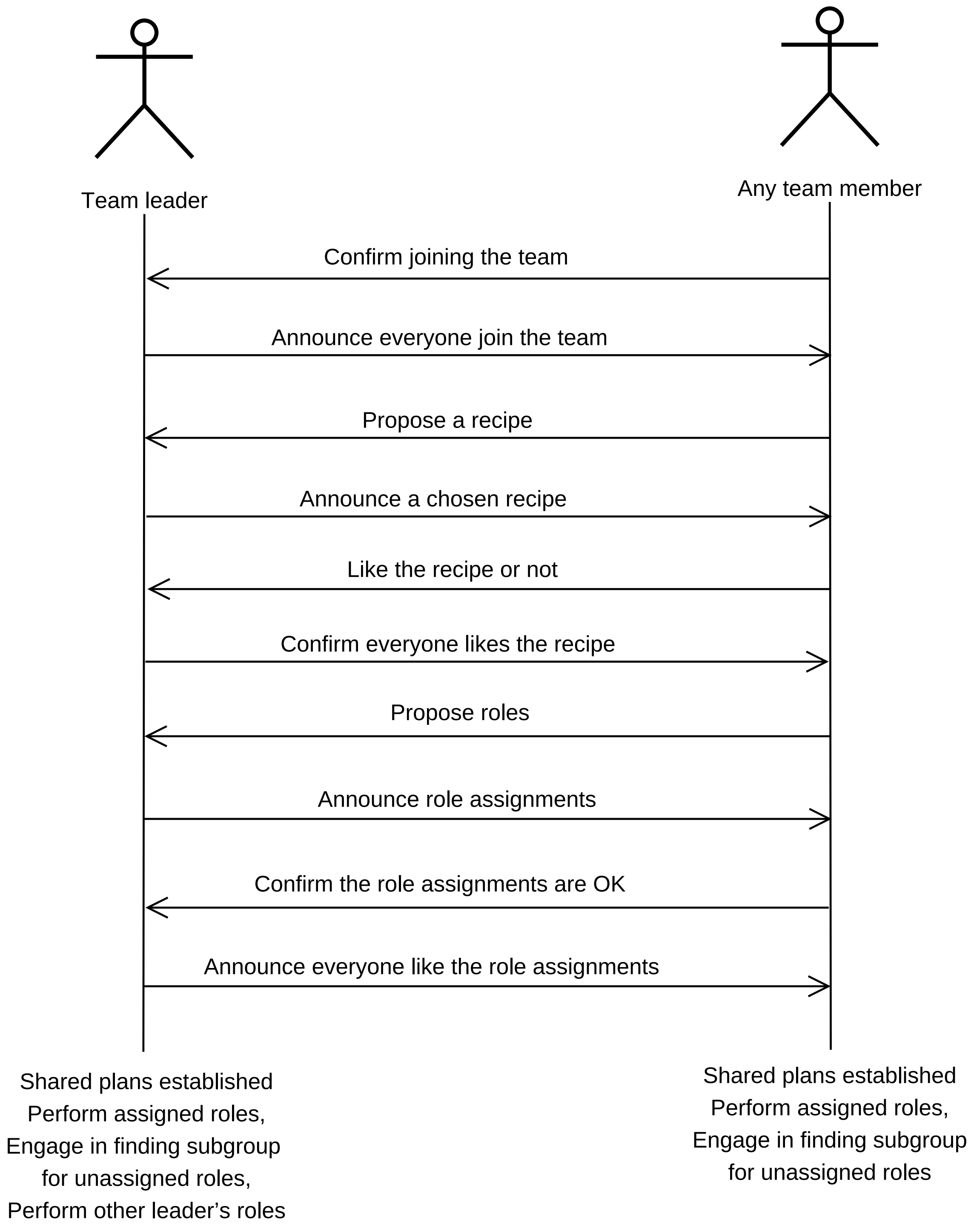

Figure 3.2: Message exchanged in plan establishment

do α. What the above assumption is saying is that Alice would believe that Bob believes Bob can do α.

With the assistance of the above assumptions, from the point of view of an individual agent, mutual belief of a group about a proposition α can be achieved if:

1. He, himself, believes in α.

2. He tells everyone that he believes in α. Moreover, in every message he sends, he includes a list of recipients to whom he sends the same message, and the list includes everyone in the group.

3. He is told by each other member of the group that they believe in α. Moreover, the list of recipients of each message which he receives contains everyone in the group.

The conditions for obtaining mutual belief can further be simplified with the exploitation of the team leader. Thus, from the point of view of an individual agent, mutual belief about a proposition α can be achieved if:

1. He, himself, believes in α.

2. He tells the team leader that he believes in α.

3. He is told by the team leader that everyone believes in α.

3.6.2 Committing to group activity

Upon receiving a request to join in a team activity, each agent needs to confirm if he wants to join it or not. He needs to send a confirmation or a rejection message to the team leader. Meanwhile, the team leader waits for messages from everyone. There is a time limit for this activity. In other words, the team leader will not wait longer than a certain timeout period. Within the set timeout, if everyone confirms their commitment, the leader will send messages to the others to announce that the team commitment has been established. In contrast, if anyone refuses to participate in the team activity or fails to respond within the timeout period, the leader will broadcast termination messages instead.

This activity is to establish the mutual belief that everyone intends that the group will do the the joint action (a requirement of Partial SharedPlans). The team leader acts as a facilitator to establish the mutual belief. From the point of view of an individual agent, mutual beliefs about the group commitment of executing the joint action are considered attained when he/she receives the confirmation message from the team leader.

3.6.3 Deciding on a common recipe

Having received the confirmation message from the leader about the team commitment, each agent now has the option of suggesting a recipe to be used by the whole team. As in establishing the team commitment, each agent needs to send a message containing the name of his proposed recipe to the leader agent. Again, the leader only accepts recipes proposed within a timeout period. The SharedPlan would fail if no recipe is proposed during that time. In this case, the team leader will send a termination message to everyone in the group. However, if more than one recipe is proposed, the team leader must decide which one will be used by the whole group. In MIST, the team leader is programmed to pick a random recipe among all proposed ones.

Having decided on a common recipe, the team leader will communicate his decision to every other team member. Upon hearing the decision from the leader, other agents must reply to say if they have any problem with the proposed recipe. Once more, there is a time limit for this activity. The shared activity will only be continued if positive replies from all other team members reach the leader on time. Otherwise, the plan fails and a termination message will be broadcast. In the case that the leader believes everyone agrees to the proposed recipe, he will announce this information and the mutual belief that everyone commits to the recipe is established. Any agent who receives the confirmation message from the team leader will believe that the team has a mutual belief that the proposed recipe has been selected‘. This process is to satisfy the second condition in the definition of Partial SharedPlans.

3.6.4 Resolving sub-actions

Once mutual belief in the applicability of the proposed recipe has been established, agents can now propose what they can do and what they would like to do. They register their capabilities and interests to the team leader. The team leader continues to accept interest registration until either all team members have registered or until the timeout period has elapsed. After the registration process has finished, the team leader examines the interests of agents for each role. Some roles might not receive any interest at all while some other might be volunteered by more than one agent. The team leader will have the honours and responsibility to decide one agent for any role which receives more than one interest. Once all necessary decisions have been made, the team leader sends a message to everyone detailing the role-agent allocations. Obviously, some roles might not be filled yet since no one volunteers.

Upon receiving the message about the role allocation from the team leader, each agent will examine the leader's decision to see if the proposed scheme is appropriate and feasible. Not only does each agent need to consider the roles which are assigned to him, but also the capabilities of others in executing their allocated actions. Each agent then either needs to confirm or reject the proposal. If every agent confirms their agreements on the proposed scheme within a certain timeout period, the leader agent will broadcast that fact to everyone. The role allocation proposal from the team leader might not contain all sub-actions of the recipe because some roles, and therefore sub-actions, are not volunteered by any agent at all. For each unassigned role or sub-action, each agent will need to engage in one team activity to find a sub-team for that role or action. Having done all of those, each agent would believe that each action in the recipe is either *at-least-partially resolved* or *unresolved*. Thus, the third condition of the Partial SharedPlans is satisfied.

From the point of view of an agent, a SharedPlan is formed when he believes that the three conditions of the partial SharedPlan definition have been met. He then forms appropriate intentions to execute or even execute his allocated sub-actions if he needs to do them straight away. The agent might believe the plan is established even though some sub-actions or roles have not been allocated to anyone yet. In this case, the plan is still partial and agents need to invoke some processes to elaborate the unresolved sub-actions. Agents do not wait until the plan is fully elaborated, in fact, elaboration and execution can be done in parallel.

3.6.5 Finding sub-teams for unfilled roles

The process of finding a sub-team for an unresolved sub-action is quite similar to the process of finding a common recipe for the joint goal. Each agent starts the finding sub-team process by suggesting a sub-group for the requested action. His suggestion would be based on his own knowledge and beliefs about the capabilities of other agents. If he thinks a sub-group of agents could get together to perform the requested action, he would suggest them to do so by sending a suggestion message to the team leader. The team leader, once again, accepts suggestions in a certain period of time. When all agents have made their suggestions or the timeout has passed, the leader agent will consider all the suggestions and decide a group to take responsibility for executing the sub-action. Not only does the leader have final say about the chosen sub-group, he also assigns the leader role to one of the members in the sub-group. After finalising his decisions, the team leader informs the other agents about his decisions and suggest members of the chosen sub-group to form a team. In MIST, the team leader would pick a sub-group randomly among all proposed sub-groups and assign the leader role to a random agent in the sub-group.

By announcing the members of the chosen sub-group, the team leader passes the responsibility of executing the sub-action to them. Members of the chosen sub-group need to get together to form a team to perform the sub-action. Agents who are not part of the chosen sub-group are not required to engage in resolving the sub-action any more. In fact, those agents will not care how the sub-action is executed. All they need to believe is the commitment and capability of the newly formed sub-team to handle their assignment. Though not very interested in the detailed execution of the sub-team, agents not in the sub-team occasionally need to exchange information with the sub-team. They do so by communicating with the sub-team leader.

There are situations in which the team leader does not receive any suggestion about a capable sub-group. In those circumstances, the team leader would request the team to further elaborate the sub-action. In other words, team members start proposing recipes for the sub-action. This brings them back to the process of deciding on a common recipe and allocating roles again.

3.7 Execution of SharedPlans

After a SharedPlan has been established (either full or partial), agents must get ready to execute their allocated roles. However, there might be interdependencies between sub-actions and dependencies of the joint action as the whole to some other external factors. Agents usually need to monitor those dependency in order to execute their sub-actions at the right time. This section describes the processes which correspond to the intentions of MIST agents and how the agents coordinate their activity in the execution of SharedPlans.

3.7.1 The intention processes

After an agent believes that a SharedPlan has been established, he will form appropriate intentions towards his assigned sub-actions. To handle and keep track of intentions, the agent creates an *intention process* for every intention he has. The use of intention processes in MIST exhibits the commitment of agents towards their intended actions. The pseudo-code of an intention process which corresponds to IntTo(α) (intention to execute α) is given in Figure 3.3. Action α will be executed when all the pre-conditions of α have been satisfied unless the intention is revoked earlier. In that case, the intention process which corresponds to α will be killed and α will not be executed.

The following sections will describe the other statements in the intention processes in more details.

```
WAIT_FOR(command_whistle_of_super_action_of_$\alpha$);
WAIT_FOR(right_moment_to_execute_$\alpha$);
EXECUTE($\alpha$);
NOTIFY(people_who_need_info_about_$\alpha$);
```

Figure 3.3: Pseudo-code of an intention process which corresponds to IntTo(α)

3.7.2 The command whistle

Involving in a team activity, each agent is usually assigned several sub-actions. However, agents can only execute their assigned sub-actions if the starting conditions of the super-action are satisfied. Therefore, each agent needs to wait for the moment when the super-action can be started. However, in MIST, instead of having each individual agent explicitly monitor for the starting condition of the super-action, each agent relies on the team leader. The team leader has the responsibility to monitor that starting condition and "blow" a "command whistle" when it believes the action can be started.

The team leader does not tell each individual agent when to do what. The team leader only notifies its team members about the starting condition of the super-action. It does not mean that the team members need to execute their allocated sub-actions straight away after receiving the starting signal from the leader. Each sub-action usually depends on some other sub-actions and events. Individual agents are solely responsible for monitoring those events.

To illustrate this point, consider the following example. A team of three soldiers Allan, Bob and Craig receive the command to destroy the enemy base. Bob is the leader of the team. They establish a SharedPlan with a recipe that the three of them will attack three different sides of the enemy base. First, Allan will blast the North side. Five minutes later, the East side needs to be blown up by Bob. Finally, Craig will bomb the South side 5 minutes after Bob has performed his duty. Thus, they have established a SharedPlan and now everyone knows what they need to do and they have formed appropriate intentions to perform their assigned tasks. However, none of them can start executing their actions until they receive the "command whistle" from their team leader, Bob in this case, that the whole joint action can be started. As the leader of the team, Bob knows that the joint action can only be started after midnight and after the guards in the watch towers have fallen asleep. So, Bob would monitor this condition and notify his teammates as soon as it is satisfied. Upon receiving the "command whistle", Allan will proceed and blow the North side of the enemy base. Meanwhile, Craig, receiving the same message, sits idle and waits for his turn. Thus the "command whistle" only means that the joint action can be stared; it does not entail that the sub-actions must be executed straight away.

3.7.3 Waiting for the right moment

The starting conditions of sub-actions sometimes depend on some other events. Agents who are assigned such actions must wait for their starting conditions to be true before they can start executing them. Agents know the starting conditions of their sub-actions by looking at the containing recipe, since the containing recipe contains information about any starting condition of those sub-actions. If, however, the starting conditions of a sub-action are not described in the recipe, the only thing the agent needs to wait for is the team leader's command whistle, as discussed above. In general, agents need to wait for both the command whistle and the starting conditions of their allocated sub-actions.

The starting conditions of sub-actions are expressed in term of dependency expressions. From the dependency expressions we can build a dependency tree. There are three types of node in the dependency tree, namely *AND*, *OR* and *LEAF* nodes. An *AND* node succeeds when all of its branches succeeds and fails as soon as one of them fails. An *OR* node succeeds as soon as one of them succeeds and fails if all fail. A *LEAF* node succeeds when the event represented by that node occurs. The trees are not necessary binary trees. The *AND* and *OR* nodes can have as many branches as they need.

Consider the recipe *ScoutMoveRec* from Section 3.4.2 again. Whoever does the sub-action *Moving* must wait for the success of the execution of *BuildingBridge* and one of the scouting actions. However, that agent will not wait for the termination of the sub-action *Scouting2* if she knows the sub-action *Scouting* is executed successfully. If the sub-action *BuildingBridge* finishes in failure, the agent will drop its intention to do *Moving* and stop waiting for the execution of both scouting actions. The *Moving* action will never be executed and will be reported as a failure. The dependency tree of *Moving* is depicted in Figure 3.4.

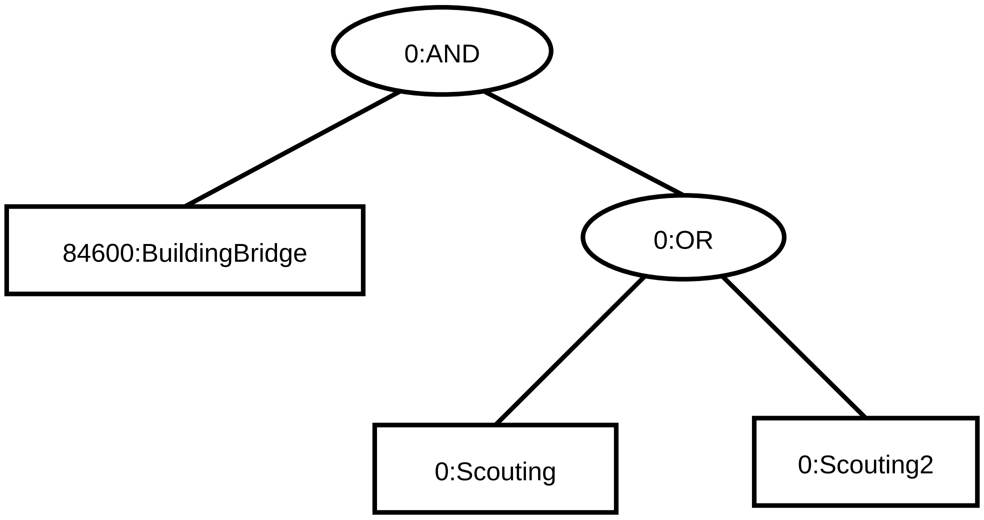

Figure 3.4: Dependency Tree of *Moving*

3.7.4 Reporting execution results

After executing an action, the agent will report the result to whoever needs the information. The list of those people can be obtained by examining the interdependencies between sub-actions in the containing recipe. For example, consider the recipe *ScoutMoveRec* from Section 3.4.2 again. The actor of *Scouting* would know that other agents who intend to do *Moving* or *Scouting2* require information about the execution of *Scouting*.

An agent only reports an action's execution result to actors of actions which depend directly on that action. An action's directly dependent actions are ones which immediately follows that action in the dependency diagram. The dependency diagram of recipe *ScoutMoveRec* is given in Figure 3.1. As illustrated in the diagram, the *Moving* action is not directly dependent on *PumpingFuel*. Therefore, actor of *Moving* will not be notified when *PumpingFuel* is finished.

An agent only reports the execution result of an action to responsible agents of dependent actions. The responsible agent of a group level action is the leader of the team which the action is assigned to. The responsible agent of an individual level action is the assigned actor of that action. For instance, the actor of action *Scouting* only needs to report to the leaders of *Moving* an *Scouting2* teams once he finishes executing *Scouting*.

There is a special case in which one of dependent actions of an action is the success of the super-action. In this case, the agent who will be reported to is the team leader of the super-action. In the above example, the team leader of the action *MoveToBattleField* will get notified when the action *Moving* is finished. It is worth mentioning that an action is considered terminated if one of the following situations occurs:

1. The action is executed successfully.

2. The action is executed but the desired goal is not achieved.

3. The intention to execute the action is dropped because the starting condition of that action will never be satisfied.

3.7.5 Monitoring

Communication is not the only way to obtain information about the execution of other actions. It is not necessary to have an agent wait indefinitely long for the communication messages from its peer fellows. Agents can be programmed to encapsulate monitoring and other reasoning capabilities. Consider the sub-team of sub-action *Scouting2* in the above recipe again, they might have other ways to reason about the execution of *Scouting*. For example, they might know that Scouting team cannot report back to them if all members in that team are shot down. Therefore, they might believes that action *Scouting* fails after a long waiting period. Having believed that, the Scouting2 team would fly off the base and perform the backup plan.

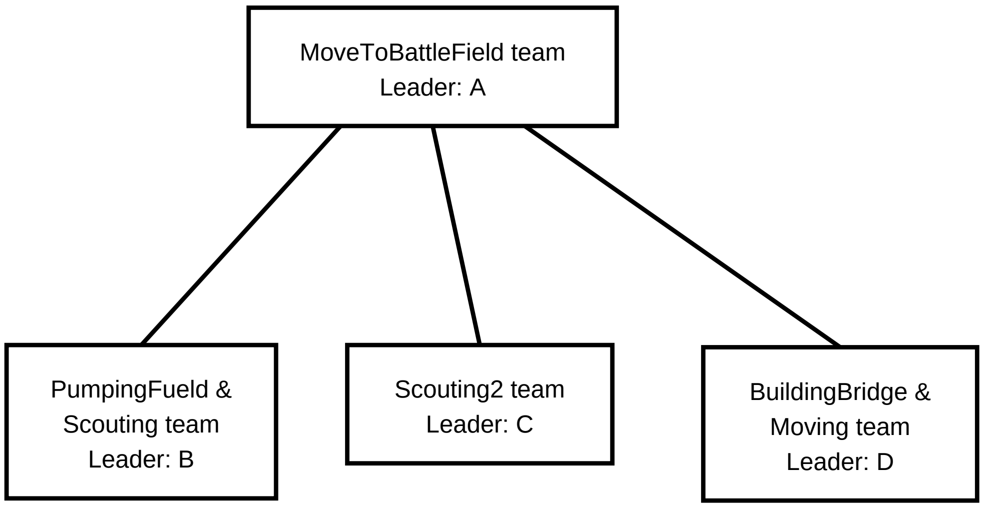

Figure 3.5: Team structure for action *MoveToBattleField* using recipe *ScoutMoveRec*

Furthermore, there are events which are external to the control of the whole team. Agents whose actions depend on such kind of events need to do the monitoring themselves. In the above example, the *Scouting* team would need to keep track of when *SunRise* occurs.

A monitoring capability is essential for every agent. However, what agents should monitor and what agents can monitor are really domain dependent. As a consequence, monitoring activities are not included in the core part of MIST. However, using the JACK plan language, monitoring activities can be added to agents developed using MIST.

3.8 Responsibilities of team leaders

In order to discuss the roles of team leaders, let us consider the recipe *ScoutMoveRec* for the action *MoveToBattleField* again, and suppose after the process of role allocation, the structure of the team is as shown in Figure 3.5. The team consists of three sub-teams with the leaders B, C and D. The leader of the super-action *MoveToBattleField* is A.

The responsibility of the leader agent to other agents in its team is to inform them when the status of the joint action changes. It includes when the joint action can be started (command whistle), when it has terminated and when it is no longer relevant. For example, suppose agent A believes that the action *MoveToBattleField* is unachievable due to some problems originated in building the bridge, agent A would notify all members in the *MoveToBattleField* team about that fact. Receiving that information, other agents who are executing or intending to execute some sub-actions would abort their missions or their intentions to pursue them. For example, if the *Scouting* team receives that information while they are on the way to the battle field, they would turn back to the base since *Scouting* is no longer necessary.

The leader of a sub-team is the representative of the team in communication with the upper level team. For example, when the action *Scouting* is executed successfully, agent B is the only person who needs to report that back to the agents in the upper-level team, the team which is in charge of executing *MoveToBattleField* action. In this situation, B would report the success of *Scouting* to both C and D which are the leaders of *Scouting2* team and *Moving* team respectively. These two leaders, in turn, will notify members in their groups accordingly.

3.9 Execution of individual actions

When it comes to individual activities, actions are handled by JACK plans. Agents engage in group activity to resolve complex group actions by dividing them to simpler individual ones. The establishment of SharedPlans and the coordination of group members are processed automatically in the core part of MIST. The execution of individual actions, however, are domain dependent. As a result of that, individual actions cannot be handled in an uniform way. In fact, domain dependent JACK plans must be written to deal with individual actions.

3.10 Summary

In summary, MIST is a JACK implementation of SharedPlans which also addresses other computational issues such as team formation and team coordination. This chapter has described the components of MIST and processes which MIST agents undergo in establishing and executing SharedPlans. In the next chapter, we will discuss how MIST can be used in developing team-based applications.

Chapter 4

Development of a specific application

This chapter illustrates the use of MIST in developing domain specific applications. First, it describes a simple simulation in the military domain. The second section explains all the steps in the development process. The third section discusses teamwork in action.

4.1 A Simulation

A simple simulation in a military domain is constructed. In the simulation, there are two forces the allies and their enemies. The allies have four members: *Agent001*, *Agent002*, *Agent003* and *Agent004*. *Agent001* is a helicopter agent which is equipped with a machine gun. *Agent003* and *Agent004* are two other helicopter agents but they are equipped with much heavier weapons such as anti-tank rockets. Any helicopter can perform scouting activity. However, only *Agent003* and *Agent004* are capable of destroying enemy troops provided there are not many of them. *Agent002* is a single agent but it represent a infantry platoon of soldiers which are capable of building bridges and moving between point and point on the ground. There might be some enemy troops and their capabilities and quantity are unknown for the allies. Helicopters agents have vision sensors which provide the number of enemy troops in the limited region surrounding them. They can also detect whether the they reach the holding point not not. The simulation is depicted in Figure 4.1. The allies can communicate with each other by radio signals.

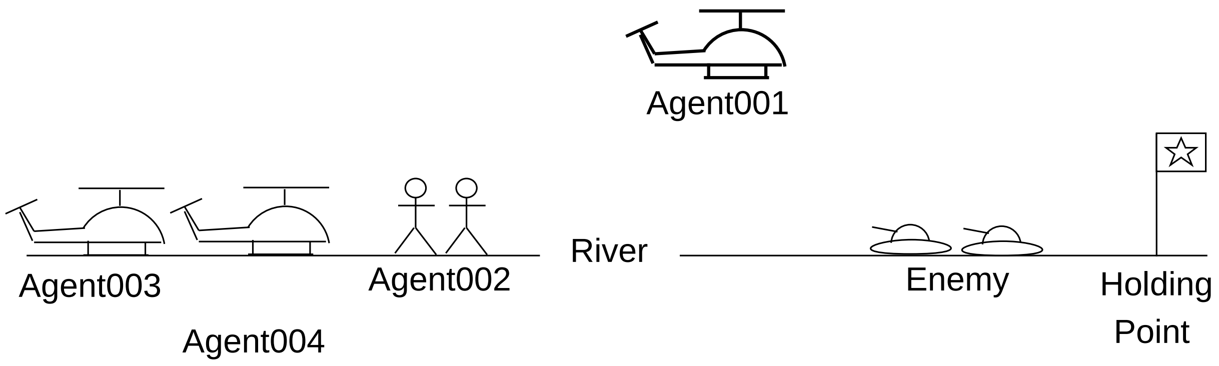

Figure 4.1: A simulation in military domain

4.2 Development

Our task is to develop a team of four agents which corresponds to *Agent001*, *Agent002*, *Agent003* and *Agent004*. The agents must be able to coordinate smoothly together and deliver their mission. This section describes the steps involving in developing such a team.

4.2.1 The mission

The mission of the allies is to get the infantry platoon, *Agent002*, to the holding point which is next to battle field. It is acceptable for some agents to sacrifice for this mission. That means the mission is considered successful when *Agent002* get to the holding point even though some other agents are shot down. *Agent002* cannot move to the holding point straight away as it might be dangerous if they encounter some strong enemy force on the way. Thus, it is always wise to have the path scouted in advance.

4.2.2 Creating agents

Creating agents is the first step where some agent classes in JACK must be defined. Each agent must have teamwork skills called *TeamPlayer* capability. *TeamPlayer* is the minimum desired skills which each agent needs for team coordination in MIST. *TeamPlayer* capability is a predefined component of MIST's core so agent classes only need to include it in their declarations if they want their agents to have that capability. Apart from *TeamPlayer* capability, agent classes are almost empty. We need to fill them with domain skills in the later steps of the development. Then, four agents *Agent001*, *Agent002*, *Agent003* and *Agent004* must be instantiated as instances of their agent classes.

4.2.3 Creating knowledge about capabilities of agents

```
4
Agent001   Scouting PumpingFuel
Agent002   Moving BuildingBridge
Agent003   Heli2 Heli1 RightWingFighting PumpingFuel
Agent004   Heli1 LeftWingFighting PackingAmmo
3
MovingToBattleField   Agent001:Agent002:Agent003:Agent004
Scouting2   Agent003:Agent004
DestroyingEnemy   Agent003:Agent004
```

Figure 4.2: Content of CapInit.txt file

The second development step is to initialise the agents' knowledge about others' capabilities. As discussed previously, this can be done by creating a text file for each agent detailing what beliefs the agent hold towards other agents. For simplicity reason, we create only one text file CapInit.txt which will be shared by all agents. The content of the file is given in Figure 4.2. The CapInit.txt file has two parts. The first part states the capabilities of agents. The second part specifies the capabilities of groups of agents.

4.2.4 Creating recipe library

In this step, one must create a text file called *RecipeInit.txt* for each agent listing all the recipes which should be contained in the recipe library of the agent. For simplicity reason, only one RecipeInit.txt file is created and it is shared by all agents. The content of that file is described in Figure 4.3. There are 3 available recipes for three group actions. The syntax and semantics of recipes have been discussed in section 3.4.2.

```
ScoutMoveRec => MovingToBattleField
SUBACTIONS: Scouting, Scouting2, Moving, BuildingBridge, PumpingFuel
SUCCEED_WHEN: Moving @ SUCCESS
MainTroopRole :: Moving, BuildingBridge
ScoutingRole :: PumpingFuel, Scouting
Moving := 5000:BuildingBridge * (Scouting + Scouting2)
Scouting:= PumpingFuel * SunRise
Scouting2 := Scouting@FAILURE
===========

TwoFightingHeliRec =>  DestroyingEnemy
SUBACTIONS : LeftWingFighting, RightWingFighting
SUCCEED_WHEN: LeftWingFighting * RightWingFighting
===========

TwoScoutingHeliRec => Scouting2
SUBACTIONS: PackingAmmo, Heli1, Heli2
SUCCEED_WHEN: Heli1 + Heli2
Heli2 := PackingAmmo
Heli1 := PackingAmmo
```

Figure 4.3: Content of RecipeInit.txt file

4.2.5 Writing code for individual domain-specific actions

Finally, one must provide instructions for domain-specific actions such as Scouting, PumpingFuel and PackingAmmo. The domain-specific instructions are written as JACK plans. Figure 4.4 shows the JACK plan for *Scouting* action. Basically, whoever uses plan *ScoutingPlan* to execute action *Scouting* will keep moving forward to the holding point until it runs out of fuel or if it sees some enemy troops. The plan is considered successful if the while loop exits normally. The plan is considered failed if the statement *false* is encountered. The agent will abort the scouting action and return to the home base if he sees some enemy troops. As can be seen from the diagram, no explicit instructions are required to tell what the agent needs to do when the *Scouting* action is terminated. This is automatically taken care by MIST. MIST will report the information to whoever it needs to report to. Thus, MIST reduces the task of developers in keeping track of agents' responsibilities in team activities.

4.3 Teamwork in action

This section describes how agents coordinate when the system is in action.

4.3.1 Establishing SharedPlans

At the start, the top chief general (the user) commands *Agent002* to lead a group consisting of *Agent002* and three other agents *Agent001*, *Agent003*, and *Agent004* to perform the action *MovingToBattleField*. *Agent002* will then pass the command to all other agents.

Upon receiving the requests, agents start elaborating a plan for their joint action. They confirm their engagement in the activity, propose a recipe, and propose roles they want to perform. After that, a partial SharedPlan is formed for *MovingToBattleField* and *Agent002* is going to execute *BuildingBridge* and *Moving* while *Agent001* takes responsibility of *PumpingFuel* and *Scouting*. The action *Scouting2* is unresolved as no single agent is capable of doing it. The whole team then engage in finding a sub-team for *Scouting2*

```
plan ScoutingPlan extends Plan{
   #handles event ScoutingEv scoutingEv;
   #uses interface HeliFighter self;
   #posts event Returning2BaseEv r2bEv;
   body(){
      String agentName = self.getAgentName();
      int fuel = 1000;
      while (true){
         //Out of fuel, cannot fly further
         if (fuel < 0) false;
         //Reach holding point, finish scouting
         if (BattleField.reachMeetingPoint(agentName))
            break;
         //See some enemy troops, cannot continue
         if (BattleField.getNoEnemyTroops(agentName) > 0){
            @post(r2bEv.return2base("Now");
            false;
         }
         BattleField.moveForward(agentName);
         fuel--;
      }
   }
}
```

Figure 4.4: Domain-specific instruction for Scouting

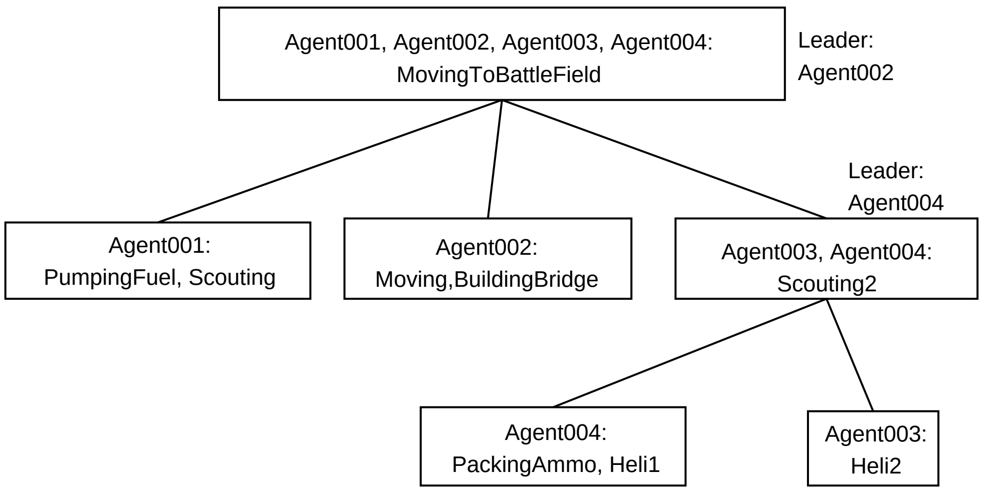

Figure 4.5: Team structure and role allocation

and they find *Agent003* and *Agent004*. From that time onwards, *Scouting2* is the responsibility of *Agent003* and *Agent004*, they will need to find a common recipe and form a SharedPlan to execute it. Figure 4.5 shows the team structure and role allocation after they have a full SharedPlan.

4.3.2 Execution of SharedPlans

The execution of individual actions start right after the collaborative plan is formed. *Agent002* starts executing *BuildingBridge* while *Agent001* starts pumping fuel to his helicopter and flying towards the holding point. After a while, *Agent002* finishes building bridge, they rest and wait for the path being scouted before moving to the holding point. However, *Scouting* fails as *Agent001* encounters some enemy troops on the way. He aborts the mission and notifies that to *Agent002* as well as the leader of *Scouting2* group which is *Agent004*. *Agent004*, in turn, notifies *Agent003* that the backup plan can be started. *Agent004* then goes packing some ammo. After loading enough heavy ammo and anti-tank rockets, both helicopters fly to the holding point to perform the action *Scouting2*. However, as they are flying toward the holding point, they discover some enemy tanks and see the need of executing *DestroyingEnemy*. Thus, they suspend their currently executing actions *Heli1* and *Heli2* and form a SharedPlan to execute *DestroyingEnemy* instead. After the enemy force has been destroyed, they resume their scouting activity and fly towards the holding point again. When they reach the holding point, they report to *Agent002*. *Agent002* then match to the holding point. Upon reaching the holding point, *Agent002* send messages to other agents to conclude that the joint activity *MovingToBattleField* is executed successfully.

Chapter 5

Comparison with other frameworks

This chapter compares MIST with other related computational frameworks. First, it discusses the differences between MIST and JACKTeams. That is followed by the comparison between MIST and STEAM.

5.1 MIST versus JACKTeams

The are many differences between MIST and JACKTeams. This section will compare them side by side in the following aspects: team structure, team formation, plan establishment, plan execution, failure recovery, application development and system performance. We also discuss the expressiveness of teamplan language of JACKTeams and recipe language of MIST.

5.1.1 Team structure and agents' relationships

Although both MIST and JACKTeams supports the creation of hierarchical team structure, there are some differences between the two. First, the use of the team agent is a distinctive approach of JACKTeams. The team agent has excessive power and duty but he is not among the people who do the actual work. Meanwhile, the team leader in MIST is one of the team members. The team leader has some extra duties but not as many as the team agent of JACKTeams. Teams in MIST are more decentralised than those of JACK-Teams. In JACKTeams, agents only need to know the team agent and the relationships between them are many-to-one relationships. In JACKTeams, teams have team beliefs and team intentions. There are no such things in MIST. In MIST, there are only mutual beliefs which are beliefs about beliefs.

5.1.2 Team formation and plan establishment

The are many dissimilarities in the ways teams and plans are formed between the two systems. In JACKTeams, the team formation and plan establishment are completely handled by the team agent. First the team agent picks a teamplan and and try to locate agents who can performs teamplan's roles. A team with a plan is considered established once the team agent identifies actors for the roles in the teamplan. Thus there is no need for communication during the establishment of plans. In contrast, establishing SharedPlans in MIST is much more complicated. Agents have to exchange messages to request and confirm their involvement in the group activities as well as to establish certain mutual beliefs. Also, MIST agents are given opportunities to propose group recipes and roles which they would like to perform; yet, the final decisions are in the hands of the team leader. MIST SharedPlans could be partial while there is no equivalent concept for teamplans of JACKTeams.

5.1.3 Plan execution

In the execution of plans, the way MIST addresses monitoring and coordination is different from the way JACKTeams does. JACKTeams solves those issues using a centralised approach in which the team agent handle everything. In JACKTeams, every agent acts and only acts when receiving requests from the team agent. There is absolutely no need for agents to monitor and reason about teammates and team activities. Conversely, MIST agents have more responsibilities, they need to monitor events related to their own actions. MIST agents have a cooperative attitude and they reason about the other agents. This attitude helps agents to coordinate their activities by communicating when necessary.

5.1.4 Plan failure recovery

There is an interesting distinction between failure recovery mechanisms of the two frameworks. JACKTeams teams do not form and reason about recovery plans until failure occurs. Unlike JACKTeams, MIST teams adopt appropriate intentions to the backup plans even when nothing has gone wrong. Consider the situation when two agents Alice and Bob want to go to Melbourne together. Since Melbourne is far away, they prefer to fly rather than to go by train. If Alice and Bob are JACKTeams agents, they will not consider the plan of catching a train unless they have already tried and failed to go there by air. If, however, Alice and Bob are MIST agents, they would form appropriate intentions and SharedPlans to catch a train as well as to catch a flight. That means they need to reason about both the plans at the same time. Of course, they will not catch a train if they have successfully flown to Melbourne.

5.1.5 Application development

There are not many differences in the ways systems are built from JACKTeams and MIST. In MIST, one must create agents and provide them with predefined recipes and initial knowledge about beliefs. Similarly, JACKTeams requires the creation of team agents with hand-coded teamplans. Capabilities of teams and agents in JACKTeams are also public knowledge. Thus, the development processes based on JACKTeams and MIST are principally equivalent.

5.1.6 Performance measure in term of communication

The number of messages exchanged during the execution of a MIST's plan α is much less than that number in execution of an equivalent JACKTeams's teamplan β. Suppose the total number of actions in each of α and β is **N**. There are two communication messages for every action in β. Therefore, the total number of exchanging messages in β is **2N**.

```
FlyToTheMoon => FlyToTheMoonEvent
SUBACTIONS : CHECK_ENGINE, PUMP_FUEL, FLY_SPACECRAFT,
    MONITOR_SPACECRAFT, CALL_EARTH_CONTROL_CENTRE
SUCCEED_WHEN: CALL_EARTH_CONTROL_CENTRE @ SUCCESS
Technician :: CHECK_ENGINE, PUMP_FUEL
MainPilot :: FLY_SPACECRAFT, CALL_EARTH_CONTROL_CENTRE
AssistantPilot :: MONITOR_SPACECRAFT
PUMP_FUEL := 600 : CHECK_ENGINE
FLY_SPACECRAFT := PUMP_FUEL
MONITOR_SPACECRAFT := PUMP_FUEL
CALL_EARTH_CONTROL_CENTRE :=
    FLY_SPACECRAFT * MONITOR_SPACECRAFT
```

Figure 5.1: An equivalent recipe of the teamplan in page 15

Meanwhile, the execution of plan α requires at most **N + 1** messages since agents only report the status of their assigned actions to agents who really need the information. Thus the number of exchanging messages during the execution of α in the worst case is just half of that number in the execution of β. For instance, consider the teamplan given in Figure 2.3 and its equivalent MIST recipe given in Figure 5.1. The total number of messages exchanges are 10 and 4 respectively. If the messages communicated during the initialisation (the command whistles) and termination period of MIST plans are also counted, the total number of messages would be **2K + N + 1**, where **K** refers to the number of agents in the group. That figure is smaller than that of β when **N** is much bigger than **K**.

5.1.7 Expressiveness of plan languages

JACKTeams uses *teamplans* to express plans for group activities. Meanwhile, MIST uses group recipes to specify how joint actions could be achieved. It is interesting that the recipe language of MIST is more expressive than that of JACKTeams. This section shows that there is a MIST's recipe for every JACKTeams' teamplan but not vise versa. In this section we reserve the term *teamplan* for JACKTeams' teamplan, and the term *recipe* to refer to MIST's recipe.

Equivalence of *recipe* and *teamplan*

A recipe α of MIST is considered equivalent with a teamplan β of JACKTeams if the following conditions hold:

1. The set of sub-actions of α is the same with the set of sub-actions of β.
2. There is one to one correspondence between roles in α and those in β.
3. The starting condition of each sub-action in α is the same for that of corresponding sub-action in β.
4. The success condition of α is the same with that of β.

Legal statements of *teamplans*' language

Strictly speaking, *teamplans* can be used to write any program because it is capable of incorporating full level Java code. By doing so, team agent could indirectly communicate with other agents by flagging some global variables or doing something similar. However, we do not consider this type of *teamplans*. In fact, only *@team_achieve*, *@wait_for*, and *@parallel* statements are regarded as legal statements in the bodies of *teamplans*.

Expressiveness of MIST's *recipes*

There is a corresponding *recipe* for each *teamplan*. To see that, consider the following informal proof. Suppose we have a teamplan which consists of some sub-actions. First of all, it is not hard to create a recipe which contains the same number of sub-actions and the same number of roles (with the same names). Furthermore, since the *dependency expression* of MIST is capable of expressing any boolean combination of events and time,

```
XX => YY
SUBACTIONS : A, B, C, D, E
SUCCEED_WHEN: E * D
B := A
C := A
D := C
E := B * C
```

Figure 5.2: A recipe which teamplans' language cannot express

one can make the starting condition of a sub-action in the recipe equivalent to the starting condition of the corresponding sub-action in the teamplan. The success condition of the recipe can be constructed in the same way. Thus, it is possible to create an equivalent recipe for every teamplan. For example, consider the teamplan example given in Figure 2.3 on page 15. One equivalent recipe is given in figure 5.1.

Expressiveness of JACKTeams' teamplans

The language of teamplans in JACKTeams is not as expressive as the language of MIST's recipes. Consider the counter example given in Figure 5.2. The recipe consists of five basic individual sub-actions *A, B, C, D* and *E* which can not be divided further. The action *A* does not have any constraint so it will be executed first. Both *B* and *C* will be executed in parallel after *A* succeeds. Action *D* is performed right after *C* while *E* has to wait for *B* and *C*. The recipe is accomplished when both *D* and *E* succeed. We will show that there is no teamplan which could include *A, B, C, D, E* and preserve the starting conditions of those actions.

Proof:

Suppose there is a teamplan β which can include all action *A*, *B*, *C*, *D*, *E* and preserve the starting conditions of those actions. From the teamplan β, construct a graph in the following way. First eliminate all statements in the plan which are different from

```
body(){
    @team_achieve(_, a1);
    @parallel(SUCCEEDS_WHEN_ONE_SUCCEEDS){
        {
            @team_achieve(_, a2);
            @parallel(SUCCEEDS_WHEN_ALL_SUCCEEDS){
                @team_achieve(_, a4);
                @team_achieve(_, a5);
                @team_achieve(_, a6);
            }
            @team_achieve(_, a8);
        };
        {
            @team_achieve(_, a3);
            @team_achieve(_, a7);
            @team_achieve(_, a9);
        };
    };
    @team_achieve(_, a10);
    @parallel(SUCCEEDS_WHEN_BLAH_BLAH){
        @team_achieve(_, a11);
        @team_achieve(_, a12);
    };
    @team_achieve(_, a13);
}
```

Figure 5.3: An example of teamplan with nested statements

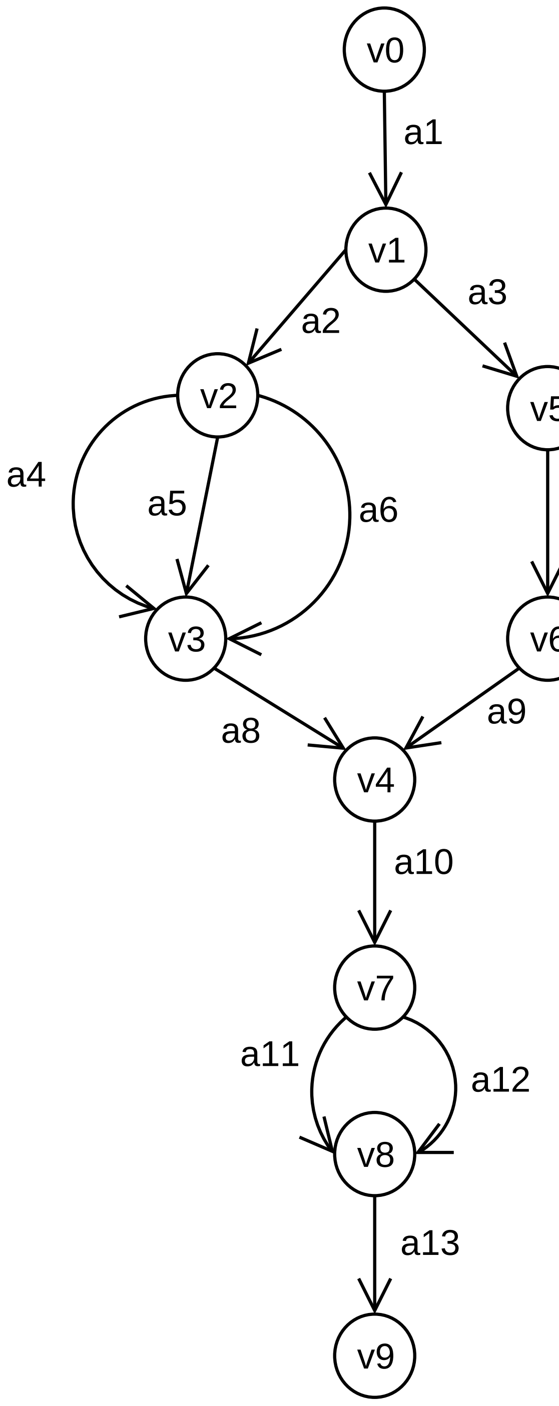

Figure 5.4: Graph for teamplan in Figure5.3

@*team_achieve* or @*parallel*. Each vertex is added for every place in the plan which lies immediately between the end of a statement and the beginning of another statement at the first level (statements inside a @*parallel* statement is considered secondary statements). Another two special vertexes are added to mark the start and the end of the plan. Each directed edge for every first level @*team_achieve* statement. Each edge joins two vertexes which represent the places before and after @*team_achieve* statement which it corresponds to. The vertexes correspond to the start and the end of a @*parallel* statement are joined by edges in which each edge matches one branch of the @*parallel* statement. Now repeat the process with every secondary statement inside @*parallel* statements. Thus, we would obtain a graph which is similar to the one in Figure 5.4. That figure is constructed from the teamplan given in figure 5.3 (You might be wondering we do not show the graph for example given in Figure 5.2. Well, we are trying to prove that no such graph exists). Since roles are not important

Though not depicted in the graph, each vertex stores hidden information about the starting condition of the edges originating from it. That starting condition depends on the parameter of the @*parallel* statement and the incoming edges. However, there is only one starting condition at each vertex and it is uniquely determined. Therefore, any two outgoing edges from any vertex must share the same starting condition.

Now consider the two actions D and E. Since the starting conditions of D and E are different, their corresponding edges cannot have the same origins. Suppose the corresponding edges of D and E start from v_k and v_l respectively ($v_k \neq v_l$). Since both D and E depend on action C, v_k and v_l must have the edge which represents C as incoming edges. However, that is impossible since each action is represented by exactly one edge and no edge can end at two different vertexes. Hence, neither does the above type of graph nor the language of teamplan could express the recipe in the counter example above. That concludes our proof.

5.2 MIST versus STEAM

STEAM is similar to MIST as both are intended to be general flexible frameworks. However, there are also many distinctions. This section discusses the similarities and differences between the two.

5.2.1 Platform

For a general system, which platform it is built on does not really matter. However, for frameworks such as MIST and STEAM, they are worth comparing. As mentioned in previous chapters, STEAM is developed on Soar while MIST is developed using JACK. Hence, developing applications from STEAM would require the provision of Soar rules. Meanwhile, one must create JACK agents and equip them with JACK plans if he is using MIST to create his systems. The JACK language is very close to Java and is possible of encapsulate full level Java code. Soar, on the other hand, is closer to a LISP language. Our assertion is that it is easier to develop agent-based systems on JACK, and programs written in JACK are probably more comprehensible and manageable than ones written in Soar. Hence, MIST has an advantage over STEAM.

5.2.2 Theoretical foundations

JointIntention and SharedPlans are theoretical foundations of STEAM and MIST respectively. In STEAM, to execute a team operator, a team must establish joint intention towards that operator. The joint intention requires agents to jointly commit to the team operator. That obligates agents to notify their teammates if their copies of the team state change. Meanwhile, MIST agents establish SharedPlans to achieve a goal based on the SharedPlans theory. MIST agents will form appropriate intentions once they think they have SharedPlans. However, MIST agents are not forced to notify their teammates even when they believe their SharedPlans are over.

5.2.3 Representations of mutual beliefs

The way STEAM represents mutual beliefs are significantly different from ours. First, STEAM assumes that every team has an abstract unique team state. Since there is no shared memory, each agent maintains its own copy of the team state. STEAM ensures the consistency between different copies of the team state by restricting the type of modification operators. Furthermore, communication is required to synchronise the update of the copies of the team state. MIST agents do maintain information about teams and their plans (SharedPlans). However, MIST does not explicitly assume the existence of an unique state of the team. Thus, there is no need for consistency maintenance. Different team members might have different information about the team and the state of their plans.

5.2.4 Team structure and team formation

Similar to MIST, STEAM facilitates the building of hierarchical team structure as STEAM borrows the concept of hierarchical team and goal decomposition from SharedPlans. Yet there are some differences in team formation. In STEAM, all teams are predefined and their leaders are predetermined. MIST supports dynamic team formation in a top-down fashion. Thus, the super-team might be formed before its sub-teams. For a newly formed team, the team leader is normally chosen at random.

5.2.5 Plan establishment

The processes of establishing plans have several differences. To execute a team operators, STEAM agents need to establish a joint intention. The team leader initiates the process by broadcasting a request message. The joint intention is considered formed when everyone has broadcast their commitments. Since, MIST teams are not predefined, the establishment process of SharedPlans is much more complicated. They have to establish mutual beliefs about group commitment. They have to decide on a common recipe as well as resolve any sub-action. They also have to engage in activities to address the role-allocation issues. Another difference between MIST and STEAM is the use of team leader. In STEAM, the team leader only initiates the process of establishing joint intention. From that point onwards, the team leader has no other privilege or duty. On the other hand, team leaders in MIST are employed to facilitate the establishment of mutual beliefs. Everyone in the team communicate with the leader instead of broadcasting their ideas.

5.2.6 Monitoring

There are some differences in the way STEAM and MIST agents monitors their joint activities. In STEAM all agents monitor the team operator and they do so independently. They only notify each other when they think the team operator has come to an end. In a MIST team, the team leader is responsible for monitoring the success of their SharedPlan. Other team members are required to help the leader. Consider the following situation. A team of three agents A, B, and C, with the leader is A, have a collaborative plan to achieve both X and Y. Thus the success condition of the plan would be: $X \wedge Y$. This type of information is available in both team operator of STEAM and MIST recipe. If A, B, C are STEAM agents, they will monitor $X \wedge Y$ independently. Hence, there is situation in which the group has achieved the joint goal $X \wedge Y$ but the agents fail to anticipate that. It can happen when each agent just has partial information about the joint goal $X \wedge Y$ such as in the case that only B discovers X has been achieved and only C knows that Y has been achieved. This type of failure would not occur in MIST. If A, B, C are MIST agents, they will monitor $X \wedge Y$ differently. Any agent discovers either X or Y has been achieved would notify the team leader. The leader synthesises all information received and notifies other members once he believes $X \wedge Y$ is satisfied.

5.2.7 Plan failure recovery

There is a clear distinction between the approaches of MIST and STEAM in handling failure. In the extent of handling failure, STEAM is similar to JACKTeams. They do not plan for failure. In contrast, MIST agents anticipate and form appropriate intentions towards failure scenarios. Please refer to 5.1.4 for an example which illustrates this point.

5.2.8 Communication

The way STEAM agents reason about communication is interesting but not always feasible. STEAM agents are integrated with decision-theoretic communication selectivity. Agents consider communication costs, benefits in order to decide if they need to communicate. This is an interesting approach to reduce communication overhead. However, this method requires knowing and assigning numerical valued costs and rewards to actions and events. This types of information might not always available and estimating them are usually very difficult and ad-hoc.

Chapter 6

Conclusion

This thesis has described MIST, a framework for team-based applications. MIST is based on SharedPlans theory which specifies the mental attitudes of agents to engage in team activities. The SharedPlans thoery, however, does not address computational issues such as coordination or communication. Aiming towards a flexible, general framework for practical applications, MIST is equipped with additional components to address some computational issues. MIST facilitates the establishment of SharedPlans and dynamic team formation. Domain independent team coordination is handled by having agents to reason about action dependencies. MIST tackles the issues of performance monitoring and failure recovery by exploiting the explicit representation of success conditions in group recipes. MIST is among several systems which strive towards flexible, general frameworks. In this thesis, MIST has been compared with JACKTeams and STEAM. There are many distinctions between MIST and these frameworks handle teamwork issues. Although there are things that MIST handles better than STEAM or JACKTeams, we are not claiming that MIST is superior to any of these systems. MIST should rather be regarded as an alternative approach for team-based applications.

There are several issues that MIST currently does not address which could be the subject for future work. First, MIST does not facilitate synchronisation of actions (neither do STEAM and JACKTeams). Solutions for the synchronisation problem are dependent on domains and just cannot be supported by a general framework. Second, MIST agents do not have the ability to negotiate. Conflicts and disagreements between team members are resolved by casting decisions from the team leader. Third, there is no concept of partial recipe in MIST whilst it is possible to have partial recipes in SharedPlans. At the moments all recipes are predefined; they cannot be created dynamically. It would great for agents to have partial recipes and abilities to combine several partial recipes to a complete one. For our future study, we would investigate possibilities of incorporating MIST agents with abilities to tackle the above issues.

Bibliography

[1] The Agent Oriented Software Group. *JACK^{TM} Intelligent Agents Agent Manual*, 2005. Version 5.0.

[2] The Agent Oriented Software Group. *JACK^{TM} Intelligent Agents Teams Manual*, 2005. Version 5.0.

[3] M.E Bratman. What is intention? In P.R. Cohen, J. Morgan, and M.E. Pollack, editors, *Intentions in Communications*. MIT Press, Cambridge, MA, 1990.

[4] M.E. Bratman, D.J. Israel, and M.E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

[5] P.R. Cohen and H.J. Levesque. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261, 1990.

[6] P.R. Cohen and H.J Levesque. Teamwork. *Nous*, 25:487–512, 1991.

[7] S. Franklin and A. Graesser. Is it an agent or just a program?: A taxonomy for autonomous agents. In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, *Intelligent Agents III*. Springer-Verlag, Berlin, 1997.

[8] B.J. Grosz, L. Hunsberger, and S. Kraus. Planning and acting together. *AI Magazine*, 20(4):23–34, 1999.

[9] B.J. Grosz and S. Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1995.

[10] B.J. Grosz and S. Kraus. The evolution of SharedPlans. In A.S. Rao and M. Wooldridge, editors, *Foundation and Theories of Rational Agency*, pages 227–326. 1999.

[11] M. Hadad and S. Kraus. SharedPlans in electronic commerce. In M. Klusch, editor, *Intelligent Information Agents*, pages 204–230. Springer-Verlag, Berlin, 1999.

[12] H. Kitano et al. The RoboCup synthetic agent challenge 97. In *International Joint Conference on Artificial Intelligence (IJCAI97)*, 1997.

[13] K.E. Lochbaum. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4):525–572, 1998.

[14] A. Lucas and S. Goss. The potential for intelligent software agents in defence simulation. Technical Note 2, The Agent Oriented Software Group, Oct, 1999.

[15] C.L. Ortiz and B.J. Grosz. Interpreting information requests in context: A collaborative web interface for distance learning. *Autonomous Agents and Multi-Agent Systems*, 5(4):429–465, 2002.

[16] M.E Pollack. Plans as complex mental attitudes. In P.R. Cohen, J. Morgan, and M.E. Pollack, editors, *Intentions in Communications*. MIT Press, Cambridge, MA, 1990.

[17] A.S. Rao and M.P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)*, pages 312–319, 1995.

[18] M. Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 22–28, 1997.

[19] M. Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7:83–124, 1997.

[20] G. Tidhar. Team-oriented programming: preliminary report. Technical Note 41, Australian Artificial Intelligence Institute, Apr, 1993.

[21] G. Tidhar. Team-oriented programming: social structures. Technical Note 47, Australian Artificial Intelligence Institute, Sep, 1993.

[22] G. Tidhar, C. Heinze, and M. Selvestrel. Flying together: Modelling air mission teams. *Applied Intelligence*, 8(3):195–218, 1998.

[23] G. Tidhar, M. Selvestrel, and C. Heinze. Modeling teams and team tactics in whole air mission modeling. In *Proceedings of the 8th international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 373–381, 1995.

[24] E. Turban, J.E Aronson, and T.P. Liang. *Decision Support Systems and Intelligent Systems.* Prentice-Hall, Upper Saddle River, NJ, Seventh edition, 2005.

[25] M. Wooldridge and N.R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 11:205–244, 1995.