View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Liverpool Repository

Multi-robotic teamwork in AgentSpeak

Thesis submitted in accordance with the requirements of the

University of Liverpool for the degree of Master of Philosophy

by

ALI BOJARPOUR

July, 2014

Contents

1 Introduction and Background **4**

1.1 Robotics . . . 7

1.2 Urban Search and Rescue . . . 8

1.3 Multiagent Programming languages . . . 8

1.4 Motivation . . . 9

2 Programming primitives for robotic teamwork **11**

2.1 Plans and actions . . . 12

2.2 Requirements . . . 13

2.3 Seeking assistance . . . 13

2.4 Performing the plan . . . 15

2.5 BDI programming in AgentSpeak using Jason . . . 16

2.6 A framework for multi-robotic teamwork . . . 17

2.7 Recognition . . . 19

2.8 Team formation . . . 21

2.8.1 Plan execution cycle . . . 25

3 Case study **27**

3.1 Start . . . 29

3.2 Rescuer needs help . . . 31

3.3 Finding the help . . . 32

3.4 Team to the rescue . . . 33

4 Implementation **35**

4.1 Sensors . . . 35

4.2 Communication . . . 37

4.3 The robot . . . 40
4.4 AgentSpeak . . . 43
4.5 Rescuer agent . . . 46
4.6 Doctor agents . . . 52
4.7 Evaluation . . . 54

5 Conclusion **57**

Abstract

Within the multiagent community, agent programming languages and theories of agent teamwork are used together to create teams of cooperative agents to achieve goals in dynamic environments. Multi-robotic research studies robots working together in a team to perform a task and achieve a goal. In this thesis we propose a set of programming constraints for multi-robotic cooperation. We then use these programming constraints to build a multi-robotic cooperation framework. This framework is a bridge between multiagent programming languages and multi-robotic teamwork. The framework uses the natural and intuitive programming style that AgentSpeak provides for multiagent cooperation to create multi-robotic teams. We then use a group of LEGO® NXT 2.0 robots to test the proposed programming constraints, implemented in AgentSpeak, to operate a search and rescue scenario.

1 Introduction and Background

Teamwork has long been studied in the multiagent community. Intention plays the main role in multiagent teamwork. Agents that are working to complete a task individually have intention towards that task. For these agents to be considered as a cooperative team they also need to have a collective intention towards the task. For example when an agent who is not part of a team realises that his task is completed, he has nothing further to do. However an agent who is working cooperatively with others in a team is expected to inform his team members when he realises that the team task is completed. Each agent who is in a team has a responsibility towards other team members [12]. Cooperative agents are also committed to the course of action that they are performing. An agent is expected to be persistent once committed to a plan. This means that he should not drop this commitment unless the plan becomes redundant. A plan becomes redundant when it achieves the task, fails to achieve the task or the motivation for the task is not available any more [4]. In each case it is expected that the agent who realises this informs the team. Cooperative agents working as a team have individual commitments towards their part in the plan. They also have a joint commitment towards the overall course of action.

[12] introduces the concept of joint intention. A group of agents jointly intends to complete a task if they are jointly committed to completing that task, whilst mutually believing that they were doing it. Joint commitment is defined as a joint persistent goal. In a joint persistent goal, a group of agents wants to complete a task based on a motivation. The motivation is the reason that the group has the commitment. For example the task for a group of minesweepers is to clear a minefield. The motivation to do so is that the military needs to clear a

field that had been laid with land mines. This group of agents believes that it is possible to achieve the task but it has not yet been achieved. Each agent in the group has the team task as his own task. He retains this task until he believes that the termination condition holds: i.e. when the agent believes that the task is achieved or is impossible to achieve or the motivation to complete this task has been lost. Meanwhile once any agent realises that the termination condition has been reached then he has the task of making this known to other agents and keeps this task until it is mutually known.

[8] uses the above idea to present the concept of joint responsibility. Joint responsibility requires that all the agents in a group have a joint persistent goal to achieve the group's task. They also should remain committed and perform actions that they have agreed to undertake. This is unless an agent realises himself or is told by another agent in the group that the action is not necessary any more for one of the following reasons: (i) the outcome of the action is already available (ii) it will not result in the outcome (iii) it cannot be performed or (iv) has not been performed properly. Agents will communicate this information to each other.

[17] also expands the notion of joint intention and presents a framework called STEAM. The novelty of this framework is the concept of team operators. When agents select a team operator to perform they initiate a team's joint intention. A team of minesweepers simultaneously selects the team operator "clean minefield". In the service of this team operator they may choose a lower level team operator "find mines". Finally in the service of this team operator a minesweeper starts scanning for a mine. Using this system a team builds a complex hierarchical structure of joint intention, individual intentions and beliefs about the intentions of other team members.

Building on the above models, [18] presents a four stage formal model that covers the whole process of teamwork from beginning to end. The first stage is when an agent recognises the potential for cooperation. He may come to this recognition because he cannot complete the task alone or because he prefers a cooperative solution. In the second stage this agent solicits assistance from other agents and tries to form a team with them. In the third stage the agents negotiate a joint plan and in the final stage they perform the agreed joint plan.
Agent-oriented programming [14] proposes a new programming paradigm in which agents are directly programmed in terms of their beliefs, desires and intentions. Several multiagent programming languages have been developed based on this idea [2]. AgentSpeak is among the most successful of these languages. Originally proposed by Rao [13], the main idea of this language is to define the know-how of a program in the form of plans. Agents use plans to achieve their goals. Plans are also used to respond to events. This means that agents use the know-how provided in the form of the plans in order to achieve their tasks as well as to respond to the changing environment.
AgentSpeak and in particular its implementation Jason which we used in our framework is designed with cooperation in mind. Agents can communicate and coordinate with one another. This communication is in knowledge level which means that agents can exchange beliefs and delegate goals to each other. We discus AgentSpeak and Jason in more detail in section 2.5.
Multiagent programming languages such as AgentSpeak have been used to develop teams of cooperative agents that are working together to complete a task. Multiagent Programming Contest [6][1] is an arena where teams of agents participate in simulated cooperative scenarios. These teams have to solve a cooperative task in a dynamic environment. Some example of the scenarios imple-

mented in previous years are food gathering, herding and gold mining. Formal models of agents cooperation and multiagent programming languages have been used together to create teams of agents that solve these tasks. Multiagent programming languages and theories of agent teamwork have proven to be successful in developing teams of cooperative agents.

1.1 Robotics

Robotics has recently became part of the multiagent community and in recent years the interest in applying multiagent theories to robotics increased. International Conference on Autonomous Agents and Multiagent Systems (AAMAS), the most prominent venue for multiagent research, hosts a robotics workshop, Autonomous Robots and Multirobot Systems (ARMS),since 2011[1]. ARMS address the significant overlap between robotics research and multiagent systems and provides a forum for the researchers in these two fields to interact. Robots are agents which means that agents theories and practices can be applied to them. This provides us with a different insight into these theories than would have otherwise been possible [10]. Motors and sensors and their uncertainties and latencies are now to be considered. Communication is not flawless any more and robots are living in a three dimensional world. The environment now follows the laws of physics and needs to be well understood.
Agents that are operating in dynamic environments use their beliefs and goals to manage their planning process. These environments are often the setting for robots as well. Multiagent programming languages allow for representation of beliefs, goals and plans.

The robotic community has taken little notice of these theoretical frame-

[1] ARMS 2014: http://ii.tudelft.nl/arms2014/

works for agents' teamwork [9].

1.2 Urban Search and Rescue

Urban Search and Rescue (USAR) is a test course designed for measuring and evaluating the performance of robots [7]. In these environments injured peoples' location must be found quickly and their condition must be established as soon as possible. Delays can result in lose of life which means that robots need to sense and plan accordingly and haste. Search and rescue missions are time critical and they are performed in dynamic unstructured environments. There is often very little information available about the environment. The available information also may be obsolete due to the collapse of buildings. These conditions make USAR an attractive scenario for measuring the intelligence of robots. The complexity of scenarios demands sophisticated decision making skills from the robots.

USAR scenarios also provide an interesting environment for robotics' cooperation. A team of robots can start with an initial strategy. While performing their plans they communicate with each other and adapt further plans based on the information exchanged. Robots in the same team can have different capabilities. Therefore they are required to cooperate and coordinate to perform their plans.

1.3 Multiagent Programming languages

Multiagent programming was first introduced by Yoav Shoham in his paper titled Agent-oriented programming in 1993. In this paper, he presented the

concept of agent-oriented programming and a sample language, called AGENT0. This language embodied some of the ideas that he felt would be central to this new programming paradigm. Since those days, Shoham's ideas have been modified, refined and taken up in mutiagent community research. After that many multiagent programming languages were born. For example Jason [2] which is the interpreter for an extended version of AgentSpeak. It implements the operational semantics of AgentSpeak. Another example of multiagent programming languages is 2APL[3]. This language is intended for practical development of multiagent systems. This language provides programming constraints to specify a multi-agent system in terms of a set of individual agents and a set of environments in which they can perform actions. 2APL provides BDI architecture.This means an agent can be implemented using beliefs, plans, goals, actions, events and rules. The agent can use these rules to decide which action to perform. 2APL supports the implementation of both reactive and pro-active agents.

1.4 Motivation

In this thesis we present a pragmatic and comprehensive framework for robotic teamwork. This framework, which is built upon AgentSpeak, covers the entire process of robotic teamwork from recognition stage in which a robot realises that he cannot achieve a task alone, to finding other robots and forming a team with them and finally performing a joint plan and completing the task together. Chapter 2 discusses each stage of robotic teamwork. It looks at the programming primitives for each of these stages and presents then in AgentSpeak. It

[2] http://jason.sourceforge.net/

[3] http://apapl.sourceforge.net/

then uses these programming primitives to create a framework for multi-robotic teamwork.

We evaluate this framework in a case study in Chapter 3. This case study looks at the well known domain of search and rescue. We use our framework to create a team of LEGO® NXT 2.0 robots that save injured people affected by an earthquake. Chapter 4 explains the implementation of multi-robotic framework in AgentSpeak and on LEGO NXT robots. Chapter 5 provides a conclusion to the thesis and our plans for the future of this research.

2 Programming primitives for robotic teamwork

In this section we look at robots that cannot achieve tasks in isolation. We will see the steps they take from realising this to accomplishing the task in a team.

Robots are programmed to achieve *tasks*. They are the duties that are assigned to robots. For example a team of minesweeper robots has the task of clearing a minefield of land mines. This requires the team to detect, disarm and remove mines from the minefield. To achieve a task a robot normally needs to move, observe and make changes to the environment. To detect a mine, minesweeper robots move in the field whilst scanning for land mines. When they find one they attempt to disarm and remove the mine from the field. Minesweepers use actuators to move, sensors to scan and detect land mines and a combination of sensors and robotic arms to disarm and remove mines. By doing so they are moving toward achieving their task: to clear the minefield. Clearing the minefield is a complicated task. Robots often face tasks that are complicated and cannot be achieved alone. These tasks often require robots with many different capabilities. These robots can vary in shape, size and in the types of sensors and actuators they have. Each of these robots performs certain parts of the task based on his capabilities. To clear a minefield a team of minesweepers needs to work together. A robot in this team with the right sensor scans for mines. When he finds a mine he communicate this with the other robots. A second robot then disarms the mine. After the mine is disarmed a third robot removes it from the field. The robots use teamwork to clear the minefield from mines. When these robots work in a team they can achieve their task.

Heterogeneous robots with complementary skills work together in a team to accomplish a task. Some of these skills can be unique to one robot and other

skills may be common amongst the team. In the minesweeper example, the first robot has the skill of finding land mines, the second robot can disarm them and the third one removes them from the field. These three robots, when working together, have all the abilities needed to clear a minefield and achieve their task.

2.1 Plans and actions

A robot involved in the clearing of a minefield needs to carry out several activities. For example he uses a metal detector to scan for mines a few inches in front of him before any movement or lifting a disarmed mine from the minefield. A robot uses these actions to accomplish his task. An *action* is a basic operation that the robot performs in order to update his beliefs or to change the environment [3]. By scanning a few inches a head of him the robot updates his belief about the safety of the path. By lifting the mine the robot changes the environment and clears that area from mines.

A *plan* is a series of these actions that defines a way in which robots can act to accomplish the task [3]. A team of minesweepers may have a plan to: (i) scan a path every few inches to detect a mine; (ii) to stop when the sensor detects a mine; (iii) cut the right wire in order to defuse the mine, etc. Minesweepers that are performing the series of these actions, the plan, are moving towards accomplishing their task.

Normally more than one plan is associated with a task. This is because there are many ways to complete a task. If one of these plans fails the team can use another plan to complete the task. Having more than one plan to achieve a task gives the robotic team a second chance when the first plan fails.

2.2 Requirements

A robot normally needs some requirements to perform an action successfully. Requirements are the capabilities or resources that a robots who wishes to perform an action needs to have. For example a metal detector is a resource required to scan for mines. The robot who wants to find mines needs to have a metal detector. Another example is a robot who wants to remove mines. He needs to have the capability of removing mines. He needs to be able to use his arm to grab and pull the mine from the ground.
A robot can only perform an action when he has all the requirements for that action. Thus prior to performing an action the robot needs to check the requirements. If he does not have the requirements then he cannot perform the action. However if there are other robots who do have those requirements then they can perform that action instead. A minesweeper that does not have the capability of removing a mine can ask another robot with this capability to remove the mine for him. We can see that the lack of requirements motivates a robot to seek help from others.

2.3 Seeking assistance

A robot that requires assistance broadcasts a message to other robots and seeks help. This help request message contains the task he wants to accomplish, the plan that the team – if it forms – will perform to complete the task and the requirements for this plan. Robots who receive this message, check their beliefs to see if they know the plan and if they can provide some of the requirements. If so, they reply with their name, the plan they want to commit to and the requirements they can provide.

A minesweeper needs to disarm mines, among other things, in the process of cleaning a minefield. To disarm a mine a voltage sensor is needed to detect the right wire to cut. If he cannot disarm mines because he does not have the voltage sensor then he sends a message to other robots. In this help seeking message he states his task, which is to clean the minefield, the plan he has to achieve this task, and the requirements that are missing. The requirement he asks for is the voltage sensor. A robot nearby that receives this message wants to help. He starts by checking his beliefs to see if he knows the plan proposed for this task. He can only help if he is aware of plan that completes this task. He then checks to see if he has the voltage sensor. If so he replies with his name, the plan he will participate in, and that he is providing a voltage sensor. By this reply he volunteers to join the team that is about to be formed.

When the robot who was seeking help finds other robots that are willing to provide what he needs then he starts to form a team with them. This basically means that they will all be committed to performing the plan to complete the task. They will work together in a team to accomplish what has now become the team's task. The team also collectively have the resources which are required to perform the plan. A minesweeper team is a team that is committed to the plan for clearing the minefield and collectively has the resources required for this job (e.g. metal detection sensors to find mines, voltage sensors to disarm them, robotic arms to remove mines from the field, etc). Team members need to know each other and the requirements each member is providing for the plan. They also need to be able to communicate with each other.

2.4 Performing the plan

A team with a plan is expected to perform actions and move towards attaining its task. Each robot chooses an appropriate action and performs it. They choose actions based on the requirements that actions may have. Some actions can start only after another action is finished. The outcome of the finished action normally determines how the next action should be performed. A robot needs to consider this outcome when choosing an action. A mine can only be removed from a minefield after it has been disarmed. A robot that wants to remove a mine that has not yet been disarmed, needs to wait for another robot to disarm the mine first. A robot that chooses to remove a mine needs to consider the requirements which is disarming the mine. After he has been informed that the mine is now disarmed then he can start removing the mine.

Robots may need to perform an action many times. For example normally several mines are concealed in a minefield. The minesweeper team will cycle through detecting, disarming, and removing every mine. They repeat each action and the the whole cycle several times.

When a robot from this team realises that the entire field has been cleaned and all the mines are removed then he is expected to inform others that the task is achieved. A robot which realises that the task is finished is expected to inform other team members [5].

In summary a robot with a task that cannot be achieved in isolation follows the following steps [18]:

- *Recognition*: realising that he does not have sufficient capabilities to achieve the task in isolation

- *Cooperative solution*: finding others committed to his course of action toward the task

- *Team formation*: forming a functional team to achieve the task

- *Team plan execution*: performing actions accordingly in the team

2.5 BDI programming in AgentSpeak using Jason

AgentSpeak introduced in [13] is a multiagent programming language based on the Belief-Desire-Intention model. Jason [3] is an implementation of an extended version of AgentSpeak. Agents in Jason have three main components: Belief, Plan and Goal. Beliefs represent the information available to an agent. `colour(box1, blue)` means that the agent believes that the colour of box1 is blue. Goals represent states of affairs the agent wants to bring about. They are denoted by a "`!`" operator. `!find(box1)` means that the agent has the goal of achieving a certain state of affairs in which he believes that box1 is found. Addition of a belief or a goal is denoted by a "+" operator.
Agents use plans to react to events. Events happen as a consequence of changes in an agent's beliefs or goals. A Plan consists of three distinct parts: the triggering event, the context and the body. Each part is syntactically separated by "`:`" and "`<-`" as follows:

```
triggering_event: context <- body.
```

The triggering event denotes the events that the plan is meant to handle. The context represents the circumstances in which the plan can be used. Finally the body is the course of action to be used to handle the event if the context

is believed true at the time a plan is being chosen to handle the event. For example an agent can use the following plan to achieve the goal `!open(door)`:

```
+!open(X): close(X) <-  !move_to(X);
                        !push(X).
```

This plan means that for a triggering event that matches `!open(X)` for example `!open(door)`, if the condition `close(X)` holds meaning that the agent believes `close(door)` then perform the body of the plan. In this example the body itself consists of two sub-goals. They are `!move_to(X)` and `!push(X)`, each of which will trigger other plans.

2.6 A framework for multi-robotic teamwork

We start with a number of heterogeneous robots with complementary skills. To clear a minefield every mine needs to be detected, disarmed and removed. Some robots can detect mines while others can disarm and remove them. These robots have complementary skills for the task of cleaning a minefield. Those who can detect mines start finding them and informing others who disarm and remove them from the field. As a team they have all the skills required to clear the minefield.

The robot who has a task starts the process of robotic teamwork. Later when the team is formed this task will become the task of the team. Team members will work together to complete this task. In the minesweeper example, the robot who has the task of clearing the minefield stars the process. In his initial beliefs he has a set of plans associated with this task. These are a series of basic actions which define the way that the robot can act to accomplish the task. All of the plans can potentially complete the task. They are different ways to

approach the task. There are many ways to clean a minefield. Each of these ways is a plan for clearing the minefield. If the team fails while performing one of these plans, then they may be able to use another plan to achieve their task. For now the robot selects the first one of these plans for his task. He now has a mission; that is the task and a plan to accomplish it. He is now to perform the plan to complete the task. If the robot has the task t and the plan p then he believes:

```
mission(t,p).
```

This basically means that the robot has a mission to achieve task t for which he will be using plan p. A plan is a series of actions. A plan to clear a minefield consists of actions such as scanning five inches in front of the robot using a metal detector, checking for the voltage passing though a fuse and sending the coordinates of a mine to another robot. The robot has these actions in his initial beliefs. A robot with plan p which is a series of atomic actions from a_1 to a_n believes:

$$plan(p,[a_1,...,a_n]).$$

A robot needs to have the requirements for an action if he is to perform it successfully. For example the action of scanning the path for mines requires a metal detector sensor and the action of pulling a mine from the ground requires a robotic arm that can grab the mine. Actions need to be atomic which means that they are basic and simple enough to be performed by a single robot. Actions that are complex and may require more than a robot to perform should be divided to smaller and simpler actions that can be performed by a single robot. A robot with action a that needs requirements r_1 to r_m believes:

```
action(a,[r1,...,rm]).
```

In order to perform action a successfully, a robot or a team of robots needs to have requirements r_1 to r_m.

Often a single robot does not have all the requirements he needs to perform all the actions. As said above the group of robots we are talking about have complementary skills. A single robot may not have all the requirements but together they have all that is needed to perform all the actions successfully. For example a robot may have a metal detector sensor to find mines but not a robotic arm that can grab and pull them from the ground. The robot that has the requirements r_1, r_2, etc, believes:

```
available_requirements([r1, r2, ...]).
```

Later we will see that when a robot want to perform an action he uses the above beliefs to infer whether or not he has the requirements needed to successfully perform the action. We have seen that a robot starts with a set of initial beliefs. They are: (i) the task he may have, (ii) a set of plans and the actions that form these plans and (iii) the requirements that are available to him for these actions.

2.7 Recognition

The process of robotic teamwork starts with a robot having a task and realising that he cannot achieve it in isolation. He needs to perform actions to complete his task. If he does not have all the requirements for the actions then he will not be able to achieve the task alone. If a minesweeper does not have a metal detector he cannot complete the task of clearing the minefield. He cannot perform the action of scanning for mines because he does not have the requirement which is the metal detector. However if there exists a group of robots with

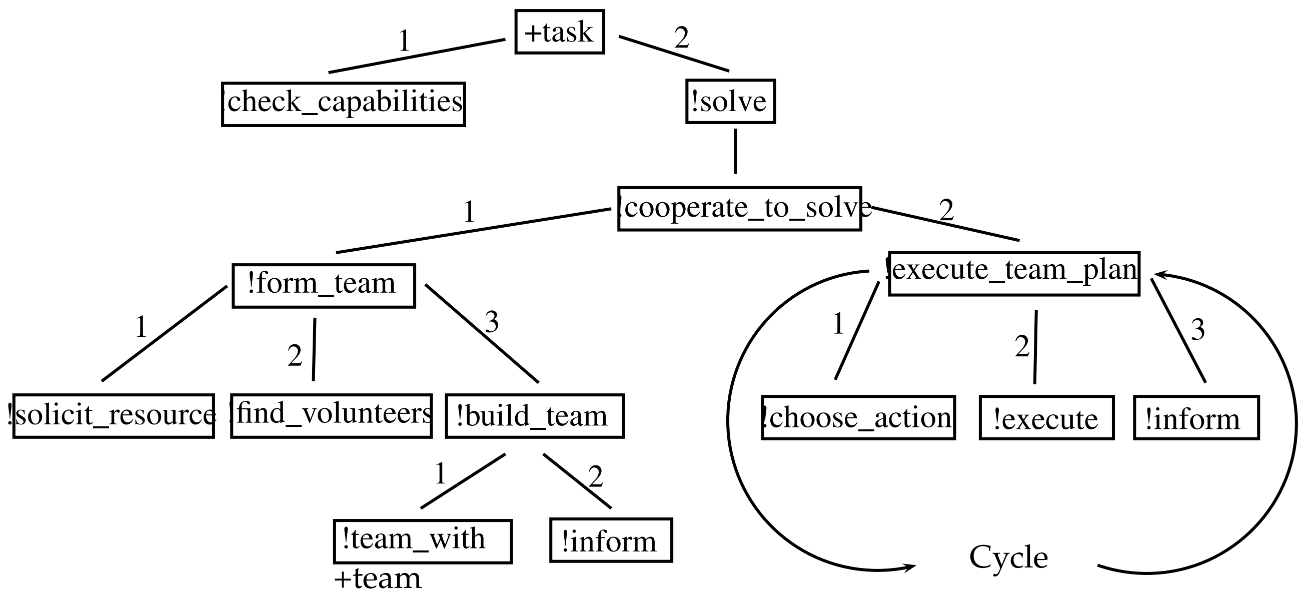

Figure 1: Robotic teamwork process

whom he can team, then they can perform the actions together and achieve the task. If there are other robots who can perform the actions he cannot, for example detect the mine for him, then they can work in a team and clear the minefield.

This means that the robot first needs to check the requirements which are available to him to know if he has all that is needed for his plan. He starts by checking the available requirement against the requirements of the plan.

Pseudocode 1: Mission

```
+mission(Task,Plan) <-                                      1
     +check_requirements(Task,Plan,Requirements,            2
        Missing_requirements);
     !cooperate_to_solve(Task,Plan,Requirements,            3
```

```
        Missing_requirements).
```

When the robot believes that he has a `mission`, that is when he chooses a plan for his task, then he needs to know the requirements for this `Plan`. When he find out the requirements then he can check them against his `available_requirements` to know which are available to him and which are missing. As we said above some of the requirements for this plan are available to him and some are not. `!check_requirements` unifies the requirements for the `Plan` with `Requirements` and the requirements he is missing for this plan with `Missing_requirements`. It is important to node that pseudocode 1, as the name suggested is a Jason style pseudocode and not a Jason application code. It is intended to capture and display the idea we are discussing in a similar style of code that AgentSpeak used. The robot is now ready to solve the task. He has a plan for this task and knows what the requirements for this plan are. He also knows what requirements are available to him and what he should seek from other robots. If he can find other robots who have these missing requirements then he can form a team with them to perform the joint plan. This is when the robot recognises the potential for a cooperative solution. Cooperative solution requires a team of robots committed to a plan. The robot who seeks a cooperative solution is expected to start forming a team.

2.8 Team formation

The robot that chooses a cooperative solution starts forming a team. A cooperative solution is when a group of robots work together to achieve a task. This means that the first step is to find other robots with whom to form a team. If the robot with the task manages to form a team successfully then they can continue

to perform the plan together.

Pseudocode 2: Cooperative solution

```
+!cooperate_to_solve(Task,Plan,Requirements,                1
   Missing_requirements) <-
      !form_team(Task,Plan,Missing_requirements,Team);      2
      !execute_team_plan(Plan,Team).                        3
```

When the robot chooses !cooperate_to_solve he knows what requirements are missing for his plan, i.e. Missing_requirements. He uses this to form a team, to find others who can provide him with these missing requirements and are also committed to his plan.
Therefore the first step to form a team is to solicit for the missing requirements.

Pseudocode 3: Team formation

```
+!form_team(Task,Plan,Missing_requirements,Team) <-         1
      !solicit_requirements(Plan,Missing_requirements)      2
         ;
      !find_volunteers(Plan,Missing_requirements,           3
         Volunteers);
      !build_team(Plan,Volunteers,Team).                    4
```

The !solicit_requirements uses a modification of the Contract Net protocol [16] to find volunteers for the team. The robot announces his Plan and the requirements he needs, Missing_requirements. Other robots who know this Plan and can provide some of the Missing_requirements reply and become Volunteers for the team. Volunteers are robots that are willing to participate in this plan by providing the requirements needed. When a possible volunteer

receives a request for requirements, he checks if he knows the plan mentioned in this request and has the requirements that are asked for. If he knows the plan and has any of those requirements then he will reply to the robot who is asking for assistance. The reply consist of his name, the plan he is committing to and the requirements he is providing.

After a deadline the robot who is seeking for help finds those who became volunteers (`!find_volunteers`) and continues to build a team with them (`!build_team`). The deadline is certain amount of time that agent waits for others to reply. Based on the environment that the agents or robots are implemented and the communication media they use, this deadline time can be adjusted. When finding volunteers the robot checks the requirements they are providing. He is looking for the missing requirements for his plan.

Volunteers share the same plan. This means that they are all committed to the same course of action that is to be carried out to achieve the task. The team is built upon a plan to which the team members are committed. When a robot sends a request for help using `!solicit_resource` he informs other robots of the plan he has chosen for this task. When the volunteers reply they show their commitment to this plan. Later when the team is formed this will be the plan they will perform together.

One might expect a team of robots to negotiate a plan after they have agreed to join the team. This however is not feasible in practice. When working with a small group of robots (about 3 to 6 robots) it is necessary to have all the robots engaged in the performing of the plan. We would like all the robots to participate in the teamwork. In a virtual agent environment it is easy and straightforward to create and remove agents, where as in a robotic environment, robots are scarce and valuable to the mission. The operator does not want to add and

remove robots manually. The early commitment to the plan, whilst the team is being formed, is a pragmatic approach that eliminates the need for later negotiation. This removes the chance of failure in negotiation which may result in some robots not participating in the plan. When the team is formed the members already know the plan to perform and therefore they continue to perform this plan.

A team consists of team members — formerly volunteers — A, B,..., Q, and the requirements L_A, L_B,..., L_Q they are providing, where, for each robot R, L_R is a list of requirements r_1, r_2,..., r_R. It also includes the task t they are trying to complete and the plan p they are using for this. Pseudocode 4 gives our representation of a team.

Pseudocode 4: Team

```
+team(t,p,[A, L_A],[B, L_B],...,[Q, L_Q]).    1
```

This team is working towards accomplishing task t using plan p. Task t which originally belonged to the robot who initiated cooperative solution is now the task of the team. Team members are working together to accomplish this task. The robot who initiated the teamwork process uses !build_team to build a team with volunteers. He will then inform team members of this new team. The team knows the task they are trying to complete and the plan they are using for this. They know who else is in the team and what resources they are providing. The volunteers are called team members from now on.

In summary the robot who has chosen a cooperative solution starts forming a team by soliciting the missing requirements from other robots. Those who know the plan and can provide requirements will reply and become volunteers which will then form a team. The members will be informed, by the robot who

originally had the task, of this new team after it is formed. Robots are now ready to start performing team actions and move toward accomplishing the task.

2.8.1 Plan execution cycle

We now have a team of robots committed to a joint plan. The next stage is to start performing this plan. A team plan is a series of actions that will be performed by team members based on the requirements available to them. The plan execution cycle starts by a robot choosing an action. Before choosing an action the robot needs to consider certain elements. First and foremost are the requirements for that action. Only the robot who has the necessary requirements can perform an action successfully.

Different actions can be performed simultaneously by different robots. For example disarming a mine and searching for another mine can be performed simultaneously by two different robots. While the first robot is disarming a mine the second robot can search to find another mine. Other actions, however, may have prerequisites. This means that they can only be performed when another action has been completed. Prerequisites of an action are other actions that need to be complete before this particular action can start. Normally this is because to perform this action the robot needs to use the outcome of another action. After a minesweeper completes actions that disarm a mine, he will communicate the outcome to the team members. In this case he announces that he managed to disarm a particular mine successfully. Another robot who wants to perform the action of grabbing and pulling a mine from the ground, now knows that this particular mine is safe to be removed. Therefore to perform the action of grabbing and pulling a mine from the ground the prerequisites, which are actions that disarm the mine, need to be completed. A

robot needs to make sure that the prerequisites of an action are complete before he starts performing that action.
A robot selects an action, performs it and communicates the outcome to the team. He then chooses the next action and the plan execution cycle continues.

Pseudocode 5: Execution cycle

```
+!execute_team_plan(Plan,Team)<-                          1
        !choose_action(Plan,Action,Requirements);         2
        !execute(Action,Outcome).                         3
        !inform(Team,Action,Outcome).                     4
```

The robotic team cycles through the `!execute_team_plan` to complete the `Plan`. They repeatedly choose an action to perform, execute the action and inform the outcome to other team members. This cycle starts by `!choose_action`. A robot, based on the requirements, selects an action. He then uses `!execute` to perform the action. Actions normally have outcomes. The robot uses `!inform` to communicate this outcome to the team.
Robots may need to repeat actions and the plan execution cycle multiple times to achieve the task.
When all of the actions are performed and the task is achieved then the plan execution cycle ends. The robot that realises this is expected to inform the team. The team will then believe that the team task is accomplished. A robot may also realise that the task cannot be achieved or may not be relevant any more. In both cases the plan execution cycle ends. The robot needs to inform the team if the task cannot be completed. A team of robots that cannot achieve their task may need to choose a different plan. In consequence that may need a new team to be formed. We are planning to study these situations in more

detail in future.

Robots start the plan execution phase by selecting suitable actions, executing them and informing the team. This cycle continues until the task is achieved, found irrelevant or impossible to complete.

3 Case study

Section 2 described a framework for multi-robotic teamwork. This sections explain how this framework is used in a multi-robotic search and rescue scenario. For this example we use LEGO® NXT 2.0 robots running LeJOS[4] Java virtual machine. These robots are programmed using NXJ which is a modified version of Java for LeJOS. The robots use Bluetooth to communicate with a PC that is running Jason. Locomotion and sensory data collection is programmed in NXJ. For example moving in a random path, avoiding obstacles including other robots and detecting the change in colour of the surface of the table is programmed in NXJ. Higher level operations such as choosing the appropriate plan to perform are programmed in Jason.

Search and rescue has long been studied in multi-robotic teamwork [11]. A search and rescue team consists of a group of robots that are working in an area dangerous or inaccessible to humans. For example in an area affected by an earthquake, robots find and save injured people. Robots in these cases are often heterogeneous. Some may specialise in finding injured people and others in providing basic medical care. Later the injured person will be moved to a hospital for further medical attention. In this case study we are looking at two types of robots; rescuers who finds injured people and doctors who provide

[4]http://lejos.sourceforge.net/

Dcotor1 has informed Rescuer that he is free which means that when Rescuer finds an injured person he will inform Doctor1 first.

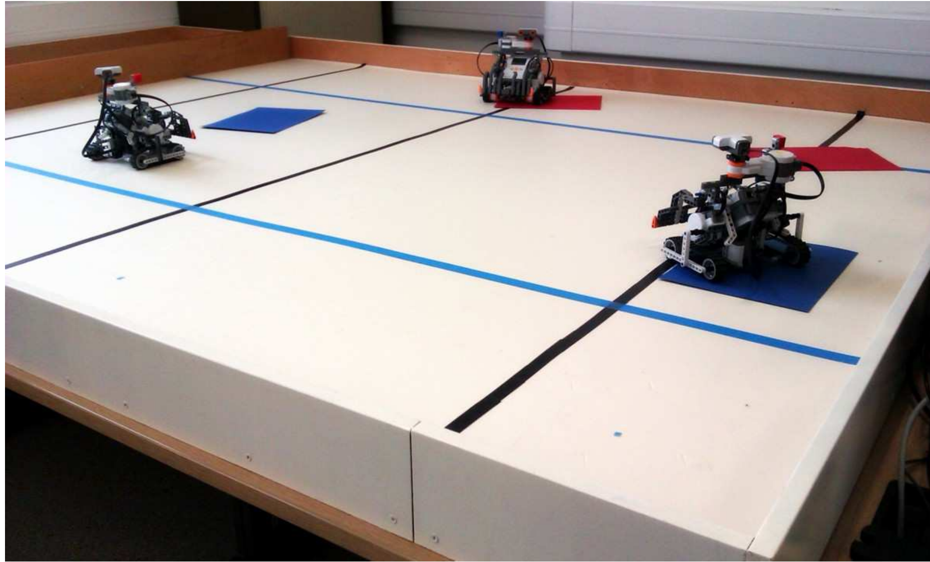

Figure 2: Multi-robot search and rescue

medical care.

Scenario 1. *Three robots, a Rescuer and two doctors, are placed in an area affected by an earthquake. Four people are injured in this area, some with more severe injuries than others. Rescuer has the task of saving the injured people. Rescuer finds them and the doctors provide medical care. Rescuer and the doctors need to work in a team to save people injured by the earthquake.*

These three robots are placed in a 240 by 200 centimetres (94.4 by 78.7 inches) table as shown in figure 2. The injured people are represented by sheets of A4 coloured paper. The colour indicates the severity of their injury. A red sheet represents a person with a more severe injury than a navy sheet. The robots are equipped with a colour sensor, an ultrasonic sensor and a touch sensor.

The colour sensor is facing down toward the surface of the table and is used to identify the coloured paper. The ultrasonic and touch sensors are used to avoid collision with other robots and the wall of the table. Robots use dead reckoning to keep track of their coordinates. A robot uses these coordinates to inform others when he finds an injured person. Robots are also aware of their starting position[5].

3.1 Start

Rescuer has the task of saving injured people. To save an injured person, he first needs to locate them. Rescuer needs to move in a random path and use the colour sensor to find a sheet of coloured paper. After an injured person is found he needs to receive basic medical care. The doctor should move to the location of the injured person and spend a certain amount of time with him. The doctor provides basic medical care to the injured person whilst he is there. The doctor robot moves over the sheet of coloured paper and waits in that location for ten seconds. After the doctor has provided the care, the injured person is ready to be sent to a hospital for further medical attention. The rescuer sends a message to the operator who sends an ambulance to collect the injured person. In our example the rescuer robot sends a message to the PC and the operator removes the sheet of coloured paper from the table.
The plan to save injured people consists of three basic actions:

- a_1: Scan for sheets of coloured paper
- a_2: Spend ten seconds over a sheet of coloured paper
- a_3: Send the location of the sheet to the PC

[5] A video of robots performing this scenario can be see here: http://vimeo.com/42409674

The rescuer has the task of saving injured people. He has a plan for this task which is the series of the above actions. The rescuer believes `mission(t,p)` where `t` is saving four injured people and `p` is the above series of actions. The rescuer also believes `plan(p,[a1,a2,a3])` which means that plan `p` is a series of actions from a_1 to a_3.

Each of these actions has some requirements. The first action, a_1, requires a robot with a colour sensor to find sheets of coloured paper. It also needs a robot that moves randomly on the table and scans for coloured papers. The third action, a_3, needs a robot that keeps track of all the injured people that are found. This robot should also send a message to the operator PC when an injured person has been attended by the doctor and is ready to be taken to hospital.

We give the name r_1 to the collection of all the requirements needed for actions a_1 and a_3. This means that both a_1 and a_3 have the same requirements which is r_1[6]. Therefore a robot that wants to perform a_1 and a_3 needed to have r_1. Action a_2 requires a robot that can receive coordinates of an injured person locations and move there. This robot then needs to provide medical care to the injured person. In our example he spends ten seconds over the sheet of coloured paper. We give the name r_2 to the requirements that are needed to perform a_2. Therefore a robot that wants to perform a_2 needs to have r_2.

The robots believe:

```
action(a1,r1).
```

[6]We need to clarify that although a_1 and a_3 are two different actions but they they both have the same set of requirements which we are calling r_1. If an agent or robot has this requirement, r_1, then he can perform both actions: a_1 and a_3. This choice is made to display that Rescuer that has the requirement r_1 can perform both actions a_1 and a_3

```
action(a_2,r_2).
action(a_3,r_1).
```

The robots also know the requirements that are available to them. The rescuer believes `available_requirements(r_1)` and the doctor believes `available_requirements(r_2)`. The rescuer has the requirements for actions a_1 and a_3 and the doctors have the requirements for action a_2. To perform the plan, they need to work together in a team.

3.2 Rescuer needs help

The rescuer needs to perform plan `p` to achieve his task. This plan consists of action a_1 to a_3. The rescuer does not have the requirements that are needed for these actions, namely r_2. He needs to find other robots who can provide this missing requirement, form a team with them and together they perform the plan.
Initially the rescuer does not know what actions he can or cannot perform. He needs to check his available requirements to find this out. The rescuer uses pseudocode 1 to find this out. The rescuer has a mission consisting of task `t`, finding four injured people and plan `p` which is actions a_1 to a_3. He now needs to check for the requirements that are needed for his actions. He uses `+check_requirements` where `Task` and `Plan` are `t` and `p`. `+check_requirements` unifies `Requirements` with r_1, r_2, r_3. These are the requirements needed to perform plan `p`. `Missing_requirements` also unifies with r_2. The rescuer has r_1 and r_3 but he is missing r_2. This is when the rescuer realises the potential for cooperation. He now continues with a cooperative solution. The rescuer uses `!cooperate_to_solve` to find other robots that can provide him with r_2 and so

perform plan p together to achieve the task.

3.3 Finding the help

In the last section we have seen that the rescuer chooses a cooperative solution to complete his task. The first step in a cooperative solution is to form a team. The team will then work together to complete the task. In pseudocode 2 the cooperative solution starts with !form_team. A team is formed based on task t and plan p. Missing_requirements is r_2. If forming a team is successful then it will be unified with Team. After the team is formed they will use !execute_team_plan to jointly perform plan p.

The rescuer now uses !form_team from pseudocode 3 to form a team. He needs to find robots that can provide r_2. He broadcasts a message using !solicit_requirements to find robots who can provide him with Missing_requirements which is r_2 and are willing to be committed to plan p. The doctors who receive this message, first check their initial beliefs to see if they know the plan p. They also need to check their available_requirements for r_2. The doctors will now reply with their names, doctor1 and doctor2, the plan they commit to perform, p, and the requirement that they are providing, r_2. By doing this they are becoming volunteers for the team that is about to be formed. The rescuer, after a deadline, uses !find_volunteers to find the volunteers which will be unified with [doctor1,doctor2]. The rescuer now knows that there are two robots with whom he can build a team with. The rescuer uses !build_team to team with the doctors. Former volunteers, [doctor1,doctor2], are called team members from now on. The team, that !build_team unifies with Team, is:

+team[t,p,[rescuer,[r_1,r_3]],[doctor1,[r_2]],[doctor2,[r_2]]]

The rescuer now needs to communicate this to all team members. He sends a message to the team members and informs them of this newly build `team`. Each member now knows that he is part of a team committed to plan `p` to complete task `t`. He also knows who else is in the team and which requirements they are providing. Note that task `t` is now the team's task. The team is formed and the members are ready to perform the plan.

3.4 Team to the rescue

The rescuer and the doctors are now ready to save injured people. They are going to perform the joint plan and complete the task. They will use pseudocode 5 to cycle through the actions until the task is attained (or the team fails to complete the task). Each robot uses `!execute_team_plan` to choose an action, perform it and communicate the outcome to the team.

The rescuer uses `!choose_action` to select a_1. The requirement for a_1 is $[r_1]$ which he has. He then uses `!execute` to perform a_1. The rescuer moves on a random path until, using his colour sensor, he finds a sheet of coloured paper. At this moment performing a_1 is finished. The outcome of this action is the the location of the sheet of paper and the colour of the sheet. The colour indicates the severity of the injure. This will be unified with `Outcome`. The rescuer now needs to inform the doctors. The rescuer uses `!inform` to send the outcomes of this action to the team. He informs them of a_1 which is the action he has performed and the outcomes of this action. They are the coordinates and the severity of the injury. Both doctors have been waiting to perform a_2 and provide medical care to the injured person. However they first need to know the coordinates of an injured person. This is a prerequisite of a_2. Now that this has been provided by the rescuer they can perform a_2. The first doc-

tor that receives the outcome message from the rescuer takes the job and starts performing a_2. He informs the team and hence the second doctor waits for another injured person to be found. Assuming that `doctor1` received the message first then he chooses a_2 and moves towards the injured person. He provides medical care to the injured person by spending ten second on that location. After the time is over the injured person is ready to be discharged to the hospital. This is when a_2 ends. `doctor1` communicates the outcome of this to the team. He informs the team that this injured person is ready to be removed to a hospital for further medical care. Meanwhile the rescuer is cycling though the `!execute_team_plan` and searching for other injured people. When he finds one he sends the coordinates to the team.

When a rescuer receives discharge message from a doctor, he chooses a_3. That is to discharge the injured person to the hospital. In our example the rescuer sends a message to the operator at the PC to remove that particular sheet of paper from the table. The doctor moves to the next injured person after he becomes free. If while a doctor is attending an injured person, he receives the location of another person with a more severe injury then he abandons the person he is currently attending and moves towards the person with the more severe injury. He communicates this with the team. The rescuer sends the location of the abandoned injured person to the next doctor that becomes free. When all four injured people have been attended by doctors and discharged by the rescuer then the task is completed. When the rescuer discharges the last injured person he realises that the task is accomplished. He send a message to the team and informs them that the team's task has been accomplished.

4 Implementation

This chapter explains how the multi-robotic programming constraints that we discussed above are implemented. We start by looking at how a multi-robotic framework based on the constraints explained above is built. This framework consists of two parts, one that runs on robots, codded in Java and running on the lejos virtual machine and a second part codded in AgentSpeak which run on the PC. As mentioned before they communicate using Bluetooth.

4.1 Sensors

We make use of three different sensors for each robot. They are the Ultrasonic, Colour and Touch sensors. As mentioned in chapter 3, Ultrasonic and Touch sensors are used to avoid collision with other robots and the wall of the table. Colour sensor is used to detect sheets of coloured paper. These sensors are managed by dedicated classes. These classes share a common design pattern. Each sensor needs to run on a separate thread. This is because the robot needs to have access to sensory data at all time. To make use of multithreading, the classes implement the `java.lang Runnable` interface. Using the `Runnable` interface is common multithreading practice in Java.
We will look at the ultrasonic class as an example. The color and touch sensor classes follow the same pattern.

Code 6: Snippet from the ultrasonic sensor class (Ultrasonic.java)

```
int distance;                                   1
public void run(){                              2
  while(true){                                  3
    distance=us.getDistance();                  4
```

```
    if (distance>=100) Thread.sleep(4000);
    else if (distance>=50) Thread.sleep(2000);
    else Thread.sleep(100);
  }
}
public int getDistance(){
  return distance;
}
```

The ultrasonic class - `Ultrasoic.java` - accesses the Ultrasonic sensor using the the `lejos.nxt UltrasonicSensor` class. Sensor classes are deigned to have an internal process that keeps updating a local variable using the sensory reading at a certain time interval. The local process running on a dedicated thread uses an infinite while loop to (i) read the sensory data, (ii)update a local variable and (iii) place the thread on hold for a certain amount of time. This time may differ based on the data read by the sensor and, of course, the type of sensor used. The Ultrasonic sensor class also has a public method that returns the local value holding the sensory reading.

Code 6 displays a snippet of the Ultrasonic sensor class. `distance` at line 1 is a private variable to store the value of the Ultrasonic sensory reading data. The `run()` is called in a separately executing thread and contains the infinite `while` loop. Line 4 reads the distance from the Ultrasonic sensor and updates the `distance` variable. Lines 5,6 and 7 place the `Thread` on `sleep` based on the `distance`. If it is further than 100 cm the next reading will be done after 4 seconds. For distances between 100 cm to 50 cm there will be a 2 seconds wait. And for less than 50 cm it will be a short 100 milliseconds wait. This is to

provide a chance to other threads to perform as well [7]. `getDistance()` is the public getter for `distance`. It is used by other classes to read the distance.
An alternative method of implementing a sensor class is to read the sensory data on request and directly from the sensor. That is rather than having a separate class to provide the sensory data, the application reads the sensory data when needed directly from the sensor. In this method the sensory data, when it becomes available to the application, is the most recent reading of the sensor. However that comes with cost of a slow application. Accessing the sensors and reading data from them is a considerably slower compared to performing operations in the processor and memory. If an application, while running on the main thread, needs to access a sensor, the main thread would have to wait until a reply from the sensor arrives. This means that the whole robot operation will come to a halt until the main thread resumes, which, of course, is not desirable. The method discussed above and implemented in this project is a reliable approach that keeps the main thread unblocked and thus the robot responsive at all times. `Thread.sleep()` can be adjusted when higher accuracy of sensory data is needed. In code 6, the seconds that the thread is set to sleep was adjusted to give us an accurate reading from the Ultrasonic sensor.

4.2 Communication

Robots and the PC communicate through Bluetooth. They send and receive messages to and from the PC which often happens simultaneously. This means that whilst the robot is sending a message he may also be receiving one from the PC. Similar to how the sensory data was managed by different threads, sending

[7] LEGO NXT robot possessor has a single core and is only able to execute a single thread at a time. LeJOS virtual machine manages the thread scheduling and switches between threads

and receiving massages also need to take advantage of multithreading. An interface needs to be provided to the robot for sending messages to the PC. We provide this interface though the `BTSend.java` class.

Code 7: Snippet from the Bluetooth massage sender class (BTSend.java)

```
NXTConnection conn = Bluetooth.waitForConnection();   1
DataInputStream in = conn.openDataInputStream();      2
Queue<String> q = new Queue<String>();                 3
public void run(){                                     4
  while(true){                                         5
    while(q.empty()) Thread.yield();                   6
    out.writeUTF((String)q.pop());                     7
  }                                                    8
}                                                      9
public void write(String message){                     10
  q.push(message);                                     11
}                                                      12
```

Code 7 is a snippet from the `BTSend.java`. Lines 1 uses `lejos.nxt.comm NXTConnection` to create a Bluetooth connection and line 2 uses this Bluetooth connection to create a stream. This is the standard mechanism of Bluetooth communication in LeJOS. In line 3 we use a `java.util Queue<E>` class to create a queue collection for storing messages. A Queue is a first-in-first-out data structure. Methods `push, pull and empty` are used to add a message to the back of the queue, extract the message from the front of the queue and check for an empty queue respectively. We use a queue to store messages before being sent because sending a message to the PC can be slow. It is also possible that a message is lost.

Reasons for this are loss of Bluetooth connection and the need to reconnect, or delay in receiving a message from the PC due to the PC processing a message from another robot. When the robot want to send a message it will use the public method `write(String message)` at line 10 and 11 to add the message to the queue. When the class is initiated on a separate thread, the `run()` at line 4 is called which in order runs the infinite `while` loop at line 5. Line 6 places the Thread on hold using `Thread.yield()` whilst the queue is empty. When a message is added to the queue by the robot, line 7 extracts it from the queue - `q.pop()` - and writes it to the stream.

Additionally when a massage arrives from the PC, there needs to be a receiver to collect and parse the massage and provide the information to the robot. The message receiver needs two main components: (i) a continuous listener for arriving messages from the PC and (ii) a message parser. This job is done by `BTReceive.java`.

Code 8: Snippet from the bluetooth massage receiver class (BTReceive.java)

```
NXTConnection conn = Bluetooth.waitForConnection();   1
DataInputStream in = conn.openDataInputStream();      2
public void run(){                                    3
  while(true) parse(in.readUTF());                    4
}                                                     5
private parse(String message){                        6
  //parse the massage and pass it to the robot        7
}                                                     8
```

Code 8 is a snippet used in the `BTReceive.java` class. Lines 1 and 2 create a Bluetooth connection and stream. `run()` at line 3 is called once the thread starts. The method will start an infinite `while` loop at line 4 in which the reading stream, `in.readUTF()`, waits for a message to arrive. Here we do not need to use `Thread.yield` because this is already implemented in the `java.io DataInputStream` class. When a message is received it will be passed to the `parse(String message)` method at line 6. This method will extract and parse information from the message passes it to the robot [8].

4.3 The robot

In this section we discuss the main class that controls the behaviour of the robot. This class makes use of three main components. They are locomotion, sensors and communication. We discussed in the above sections how the sensors and communication are implemented. In this section we will see how the robot uses this implementation to function. We will also see how the robot uses the navigation and localisation mechanism provided by the LeJOS platform for locomotion.

The robot class makes use of a scanner class. The scanner is used to inform the robot of any obstacles that will be on its path. The scanner makes use of the Ultrasonic and Touch sensor classes, discussed above, to scan and detect obstacles. To improve the functionality of the ultrasonic sensor, it is mounted over a rotating motor placed at top of the robot. The motor performs a 25 degrees rotation to left and right and covers a wide angle in front of the robot

[8] Detail of this process can be seen the source code

for the ultrasonic sensor. Lego NXT Ultrasonic sensors have a range of 30 degrees. This means that a static ultrasonic sensor can detect any object within a 30 degree angle in front of it. In addition to this 30 degree angle our 50 degree rotating angle makes a wide area in front of the robot detectable by the ultrasonic sensor. The speed which the motor rotates needs to be adjusted based on the environment in which the robot is operating.

The Scanner class - `Scaner.java` - has internal thread that operated the motor.

Code 9: Scanner motor rotation system (Scanner.java)

```
private void startRadar() throws Exception{
  final int ANGLE = 25;
  Motor.C.setSpeed(750);
  while(true){
    Motor.C.rotateTo(ANGLE);
    Motor.C.rotateTo(-ANGLE);
  }
}
```

Code 9 shows how the scanner motor operates. Lines 2 and 3 set the rotation angle and the speed of the motor and the infinite while loop at line 4 rotates the motor back and forth. This methed runs on a separated background thread. The while loop terminates when the main thread reaches the end (i.e. when the application ends).

The Scanner was added to hide the details of obstacle avoidance from the robot. The robot delegates the process of obstacle avoidance to the scanner. This results in organised and easy to manage code structure in the robot class.

NXJ API provide a variety of classes for locomotion. We use `DifferentialPilot`, `OdometryPoseProvider` and `Navigator` in this project. `DifferentialPilot` provides methods for control of robots' movements: travelling forward and backward in a straight line or moving in a circular path and rotating to a new directions. This class automatically updates the `OdometryPoseProvider` with the robots new location and heading. A robot's wheel diameter and track width will be passed to the `DifferentialPilot` upon initialisation of this class. The unit of measure for travel distance, acceleration and speed is the same as the one used for wheel and track diameter and width [9]. Motors used for tracks movement are also passed to the `DifferentialPilot` on initialisation.

Code 10: Robot's locomotion control

```
DifferentialPilot pilot = new
  DifferentialPilot(3.25f,19.8f,Motor.B,Motor.A);
OdometryPoseProvider poseProvider = new
  OdometryPoseProvider(pilot);
poseProvider.setPose(new Pose(20,100,0));
Navigator robot = new
  Navigator(pilot,poseProvider);
```

Lines 1 and 2 in code 10 create and initialise a new `DifferentialPilot`. `3.25f` and `19.8f` are wheel diameter and track width and `Motor.B` and `Motor.A` are the motors used.

`OdometryPoseProvider` keeps track of robots location and heading. We mentioned in chapter 3 that robots use dead reckoning to keep track of their coordinates. `OdometryPoseProvider` uses the odometry data for dead reckoning.

[9]Centimeters in our case

Lines 3 and 4 create and initialise a new `OdometryPoseProvider` by passing the `DifferentialPilot` created in the lines before. Line 5 sets the starting location and heading of the robot [10].
`Navigator` class uses the `DifferentialPilot` to traverse a path while updating the `OdometryPoseProvider`. Lines 6 and 7 create and initialise the `Navigator`. From the point onward the robot class uses the `Navigator` for locomotion.

4.4 AgentSpeak

Jason provides a basic set of functionalities. More advanced features such as bluetooth communication can be added to Jason by means of user-defined internal actions. These internal actions are programmed in Java. Jason is also distributed with a set of standard internal actions. For example `.send` for agents to send messages to each other or `.print` to output a message to the Jason's console.
In our scenario we created two user-defined internal actions for bluetooth communication and message parsing. Before we start looking at Jason code for each agent, let us explain how these internal actions perform.

Code 11: Agents communication (Communication.java)

```
LinkedBlockingQueue<String> q =                    1
    new LinkedBlockingQueue<String>();             2
NXTConnector conn =                                3
    new NXTConnector();                            4
```

[10]In our scenario we use three robots each located 100 cm away from each other and facing the Y axis

```
boolean connected =                                   5
    conn.connectTo(nxtName,nxtBTAddress,2);           6
DataInputStream dis = new DataInputStream(conn.       7
   getInputStream());
DataOutputStream dos = new DataOutputStream(conn.     8
   getOutputStream());
```

Communication.java is a user-defined internal action which sends and receives message to and from the robot using bluetooth. This internal action has similar structure to the robots' communication method discussed in the above section. Messages are stored in a BlockingQueue and an input and output streams - DataInputStream and DataOutputStream - are used to read and write from the bluetooth channel. Line 1 of code 11 creates the BlockingQueue to store messages. Line 3 creates the communication channel and Line 5 connects to this channel. Upon initialisation of the class, robots' bluetooth address and names, i.e. Rescuer, Doctor1 and Doctor2, are sent to the PC for identification. Lines 5 connects to the robot with the name of nxtName and bluetooth address of nxtBTAddress. Lines 7 and 8 use the channel to create an input and output stream. These streams are used from this point onward to send and receive message between the robot and the PC where Jason agents are located.
A newly arrived message from the robot will be places in the BlockingQueue.

Code 12: Agents communication - reading from a stream

```
public void run(){                                    1
  while(true){                                        2
    q.put(dis.readUTF());                             3
    Thread.sleep(100);                                4
```

```
  }
}
public String read(){
     while(q.size()==0) Thread.yield();
     return q.poll();
}
```

Method run at line 1 of code 12 which runs on a separate thread uses an infinite loop to continuously listen to and read messages arriving from the robot. Line 3 adds the message to the BlockingQueue. Method read is used for reading messages from the BlockingQueue. At line 8 while the BlockingQueue is empty the thread is paused. Line 9 returns the earliest message from the BlockingQueue.

Code 13: Message parsing

```
String colour=
  inj.toString().split(";")[3]
  .replace('"', ' ').trim();
if (colour.equals("Red")) severityLevel=3;
else if (colour.equals("Navy")) severityLevel=2;
else if (colour.equals("Green")) severityLevel=1;
NumberTerm result =
  new NumberTermImpl(severityLevel);
```

We explained in the previous section how a message to the robot is constructed. When the message arrives at the robot it will be parsed by the Severity.java class. This class maps the three colours of green, navy and red to the three levels of 1, 2 and 3 of severity where level 1 is the highest severity level and 3 is the lowest. Injured people with the highest severity level are the highest priority for

the doctors. We will see below how the doctor agents use these severity levels to decide in which order to attend for the injured people. The doctor agent then passes these values to the robot agent in Jason. Code 13 uses the arrived message from the robot to indicate the severity of the injury. Lines 4 to 6 set the level of severity and lines 7 and 8 pass the value to the agent. When all the injured people are found, Rescuer robot sends a `end` message which the agent uses to end the task.

4.5 Rescuer agent

In this section we look at the Jason code for Rescuer agent. All agents start with the initial beliefs of `free`. The two doctors `send` a message to Rescuer to `tell` him that they are `free`. This essentially adds `free[source(Doc)]`, where `Doc` is unified with the Doctor that sent, the message to Rescuer's beliefs.

Code 14: Free agents are added to the team

```
free[source(rescuer)].                                    1
                                                          2
+free[source(Sender)]: true <- +team(Sender);             3
    .broadcast(tell, team(Sender)).                       4
```

In section 2.8 we discussed how teams are formed. Lines 2 and 3 of code 14 are used to form a team. They add agents to `team`. Line 2 of this code informs other agents of the new member who is added to the team. It does so by using a `broadcast` message which adds `team(Sender)` to other agents beliefs. To keep the code simple and pragmatic we had to omit a few options. We assumed that all agents will eventually become a member of the team. Therefore every agent that sends a `free` message is automatically added to the team and others are

informed of this.

Rescuer agent starts by searching for injured people while initially the two doctors are free. Rescuer, thus, starts by the initial achievement goal of `!rescue` and doctors start with initial beliefs of `free`.

Code 15: Rescuer agent initialisation

```
!rescue.                                                    1
+!rescue <- .wait(500);                                     2
    rescuer.a1(Inj);                                        3
    !check_end(Inj).                                        4
+!check_end(Inj): Inj =="end" <- !check_injureds;           5
    .wait(1000);                                            6
    !check_end(Inj).                                        7
+!check_end(Inj): Inj \=="end" <- .wait(500);               8
    .print("Found␣", Inj);                                  9
    +injured(Inj);                                          10
    !check_injureds(Inj);                                   11
    !!rescue.                                               12
```

Line 1 of the code 15 shows the initial belief of Rescuer robot. The plan to achieve this goal - `+!rescue` - at line 2 starts by a 500 milliseconds of `.wait` for bluetooth handshake to complete. It will then wait for a message from the robot. The robot sends a message upon finding an injured person. Line 3 uses the internal action `a1` for Rescuer robot to achieve this. The user-defined internal action `a1` uses the `Communication.java` to connect Rescuer agent and Rescuer robot though bluetooth. It will also uses `Severity.java` to parse messages. Information about found injured people are unified with `Inj` at line 3.

The agent will then pursue the goal of `!check_injureds(Inj)` at line 4.

If the `end` message is not communicated, the plan at line 8 is performed. The agent prints out the found injured person information in console and adds him to his beliefs. At line 10 of code 15 Rescuer agent uses `+injured(Inj)` to add the injured person to his beliefs. The next step is to pursue the goal of `!check_injureds(Inj)` followed by `!!rescue` at lines 11 and 12. `!check_injureds(Inj)` starting with a single `!` means that it will be pursued as a separate intention while `!!rescue` with two `!` means that the agent will pursue another separate intention to achieve `!!rescue`. `!!` are often used at the end of recursive plans [11].

Code 16: Rescuer agent - checking for injured people

```
+!check_injureds:injured(Inj)                              1
    <- !check_injureds(Inj).                               2
+!check_injureds:not injured(Inj)                          3
    <- .broadcast(tell, end).                              4
+!check_injureds(Inj): free[source(Doc)]                   5
    <- .print("Sending ", Inj, " to ", Doc);               6
    -free[source(Doc)];                                    7
    .send(Doc,achieve,go_to(Inj));                         8
    +attending(Inj)[source(Doc)];                          9
    -injured(Inj).                                         10
+!check_injureds(Inj): not free[source(Doc)]               11
    <- .print("Broadcasting");                             12
    .broadcast(tell,check(Inj)).                           13
```

[11] http://jason.sourceforge.net/faq/faq.html#SECTION00095000000000000000

So far we have seen how Rescuer agent checks for the incoming messages and identifies the injured person or the end message. Code 16 are the plans Rescuer has when an injured person is found which in essence is to inform the doctors. `!check_injureds` are called by line 5 of code 15. If an injured person exists in the agents belief base - `injured(Inj)` - then this information is retrieved and passed to `!check_injureds(Inj)` at lines 5 or 11 of code 16. We will see later that doctor agents inform Rescuer when they are free, including at the initialisation of the program. When a doctor agent is free it sends a message adding `free[source(Doc)]` to Rescuer agents belief base where `Doc` is unified with the name of the doctor agent i.e. doctor1 and doctor2. We discussed in chapters 2 and 3 that when teamwork is over, team members will be informed. Line 2 of code 16 sends a `broadcast` message to all the members to inform them that task is over.

If a doctor is free Rescuer agent performs the plan from line 5. It starts from line 6 by printing a message to the console followed by line 7, removing the belief that the doctor to whom it is about to send a message to is free. At line 8 it will send a message to the doctor. This is an `achieve` message, `go_to(Inj)`, that informs the doctor of information about the injured person unified with `Inj`. This adds the `!go_to(Inj)` to doctor's goals. Now that the doctor is in charge of the injured person Rescuer adds `+attending(Inj)[source(Doc)]` to its beliefs. This is to it keep track of doctors busy with injured people at any given time. Finally Rescuer agent removes the information about the injured person from its belief using `-injured(Inj)` at line 10.

Code 17: Doctor agent - check for severe injuries

```
+check(Inj)[source(Res)]: free                    1
  <- !go_to(Inj)[source(Res)].                    2
```

```
+check(Inj)[source(Res)]: severity_check(Inj)          3
  <- -+new_inj(Inj);                                    4
  .print("Severe␣injured␣to␣be␣attended:␣", Inj);       5
  -check(Inj)[source(Res)].                             6
+check(Inj)[source(Res)]: not severity_check(Inj)      7
  <- .print("Ignored␣", Inj);                           8
  -check(Inj)[source(Res)].                             9
```

If there are no free doctors, Rescuer chooses the plan at line 11 of code 16. It broadcasts a message to all doctors using `.broadcast(tell,check(Inj))` at line 13. This broadcast message informs the currently busy doctors about the injured person, `Inj`.

Code 17 are doctor plans for `check(Inj)`. When a doctor agent becomes free it will pursue the `!go_to(Inj)[source(Res)]` goal at line 1 and 2 of code 17. The doctor agent will also check for the severity of the person's injury.

We explained above that doctors attend people with more severe injuries with a higher priority. If they are engaged with an injured person with a less severe injury they will leave the current injured person and attend the one with more severe injury. Lines 3 and 7 display that the doctor agent checks for level of severity of the person's injury. `severity_check(Inj)` checks how severe is the injury. If the injury is more severe, line 4 replaces the current injured person with the more severely injured one. It does so by removing and adding the `new_inj(Inj)` belief. If the injury is not more severe then line 8 prints an ignored message to the console.

Code 18: Rescuer agent - discharge injured people

```
+discharge(Inj)[source(Doc)]                            1
```

```
  <- -injured(Inj);                                   2
  -attending(Inj)[source(Doc)];                       3
  -discharge(Inj)[source(Doc)].                       4
+abandoned(Inj)[source(Doc)]                          5
  <- -attending(Inj)[source(Doc)];                    6
  +injured(Inj);                                      7
  -abandoned(Inj)[source(Doc)].                       8
```

An injured person will be ready for discharge after having been attended by a doctor. However it is Rescuer that discharges the injured person. Doctors inform Rescuer that the injured person they attended is now ready for discharge. They do so by sending a message that adds `discharge(Inj)` to Rescuers belief. Line 1 to 4 of code 18 display Rescuer agent's plan for this belief addition. At line 2 Rescuer removes the injured person using `-injured(Inj)` from its beliefs followed by removing the belief about the doctor who was attending this injured person, `-attending(Inj)[source(Doc)]` at line 3.

A doctor informs Rescuer robot when leaving an injured person to attend to one a with more severe injury. The doctor does so by sending a message adding `abandoned(Inj)[source(Doc)]` to Rescuer's belief. Rescuer's response to this is displayed at lines 5 to 8. Rescuer removes the belief that the doctor `Doc` is attending an injured person `Inj` using `-attending(Inj)[source(Doc)]`. He will then uses `+injured(Inj)` to add `Inj` to its beliefs as an injured. Finally he cleans up its belief base by removing `-abandoned(Inj)[source(Doc)]` at line 8.

4.6 Doctor agents

Initially the doctors do not have any injured person to attend and they are free. A doctor starts with a `free` belief.

Code 19: Doctor agent - initialisation

```
free.                                                        1
+free: .send(rescuer, tell, free).                           2
severity_check(Inj) :- robot.severity(Inj,S) &               3
  injured(Inj_now,S_now) &                                   4
  S > S_now.                                                 5
!inform.                                                     6
+!inform: free <- .send(rescuer,tell,free).                  7
```

`severity_check(Inj)` at line 2 of code 19 uses the user-defines internal action `Severity.java`, code 13, to inform the doctor if the current injured person he is attending has a more severe injury than the one broadcast by Rescuer. The doctor agent's initial goal is to inform Rescuer that he is `free`. Line 6 sends a message to inform Rescuer.

Code 20: Doctor agent - attending an injured person

```
+!injured_check(Inj) <- robot.severity(Inj,S);               1
  -+injured(Inj,S).                                          2
+!go_to(Inj)[source(Res)]                                    3
  <- -free;                                                  4
  !injured_check(Inj);                                       5
  .print("Attending␣", Inj);                                 6
  doctor.a2(Inj, Done);                                      7
  .wait(1000);                                               8
```

```
!has_attended(Inj).                                    9
```

We have seen at line 8 of code 16 that Rescuer sends an achieved message go_to(Inj) to the doctor. Code 20 is the plan a doctor uses to for this goal. It starts by removing the belief that he is free. He will then pursue the goal of !injured_check(Inj) which in consequence uses lines 1 and 2 of code 20 to check for severity and add the injured person to his belief. Line 7 is a user-defined internal action. It is to communicate with the robot to move to the location of the injured person and attend to the injuries. Basically that is for a doctor robot to spend a few seconds at that location which may vary based on the severity of the injury. Finally at line 9 it pursues !has_attended(Inj).

Code 21: Doctor agent - discharge or abandoned an injured person

```
+!has_attended(Inj): new_inj(New_inj)                          1
  <- .print(Inj, " abandoned for ", New_inj);                  2
  .send(rescuer,tell,abandoned(Inj));                          3
  -new_inj(New_inj);                                           4
  !!go_to(New_inj).                                            5
+!has_attended(Inj): not new_inj(New_inj)                      6
  <- .print(Inj, " ready for discharge.");                     7
  .send(rescuer,tell,discharge(Inj));                          8
  .send(rescuer,tell,free);                                    9
  -injured(Inj,S);                                             10
  +free.                                                       11
```

When a doctor robot finishes attending an injured person he removes that injured person from his belief base and informs Rescuer. He sends a message that the injured person is ready for discharge. Line 8 of code 21 displays this. If the

doctor agent needs to move to an injured person with a more severe injury he uses line 3 to inform Rescuer that he `abandoned` the injured person.

4.7 Evaluation

The above implementation can be viewed in action at `http://vimeo.com/42409674` and the source code is available at `https://github.com/alibojar/RoboticCPS`. The video displays Rescuer robot and the two doctor robots. The Jason program running on the PC initiates a Bluetooth connection to the robots. Line 2 of Code 15 is a 500 milliseconds wait for this Bluetooth connection to be established. After experimenting with different delay times we found that 500 milliseconds is optimum time needed for a successful Bluetooth connection. It allows for sufficient time for the PC to connect to the robot but not too long to cause a significant delay in the performance of robots. After the Bluethooth connection is established other parts of the robot which are sensors and motors will be initiated.

Setting up a Bluetooth connection is not always successful. In the same way as other Bluetooth devices such as earphone, mice and keyboards sometimes do not connect to mobile phones or computers as expected, NXT robots occasionally fail to connect to PC via Bluetooth. In these cases we have to restart the process on the PC and robots.

The first robot to connect to the PC is Rescuer followed by Doctor1 and Doctor2. Rescuer starts moving first because the goal he has to achieve - `!rescue` - requires him to move around and scan for injured people. At this point the two doctors remain still waiting for Rescuer to find an injured person.

When Rescuer finds an injured person he will assign the person to the first free doctor available. At the start of the scenario both of the two doctors are free and

will inform Rescuer. The /texttfree message send from Doctor1 often arrives at Rescuer first. This is due to the configuration of agents in Jason where agents will start running based on the order they are created. However this does not make any difference in the process of their teamwork. The next injured person can be assigned to any of the two doctors based on how severe the injury is and which of the doctors is free at that time. For example if Doctor1 is attending an injured person with a less severe injury and another person with a more severe injury is found, by Rescuer, then Doctor1 leaves the current injured person to attend the newly found injured person with a more severe injury.

Automatic robotic actions such as moving to a certain coordinate or avoiding an obstacle based on sensory scans are written using LeJOS platform. This basically means that they run on the robot rather than Jason. Scanning for sheets of coloured paper using a colour sensor or avoiding obstacles such as other robots or the walls of the environment, are programmed using API that LeJOS platform provides. Motors are also programmed using LeJOS. Information is exchanged using the communication layer between the robot and its mirroring agent in Jason.
The Jason code, on the other hand, concentrates on communication between agents, high-level decision making and reasoning. The team is formed between agents in Jason and they communicate using the mechanism provided in Jason. Decisions about injured people, who to attend them and when to discharge them, are also made in Jason. The Jason code concentrates on agent teamwork aspects of this scenario. AgentSpeak and Jason are made for multiagent programming and are suitable for agent and team related aspects of this scenario. LeJOS, on the other hand, is a basic robotic platform for controlling motors and

sensors.
AgentSpeak and Jason as intended for multi-agents programming while LeJOS is a robotic programming platform. In this project any aspects of robots related to agents and their communication and teamwork is programmed in Jason. Other parts such as robot locomotion and managing sensors are codded in LeJOS.

AgentSpeak is a perfect choice for robotic programming however it comes with its limitations. Jason is written in JAVA and make a heavy use of multi-threading. This makes Jason slow at times. While operating robots, we occasionally came across situations that where due to slow replies from Jason application robots did not stop at the right positions. We also experienced cases where the messages send from doctors to the Rescuer did not get processed on time. Although the message was send by a doctor and arrived at the rescuer, the rescuer did not perform the corresponding plan. We believe this also due to Jason's slow performance. Json's slow performance in general was noticeable during execution of our scenarios. We used intentional suspense at parts of our code to minimize this slow performance. These delayed, for example line 2 of code 15 allows the other parts of the program to catch up with the rest.
Another difficulty we faced during the implementation of this project was unreliable Bluetooth connection between robots and PC. We occasionally faced loose of Bluetooth connection which meant that we had to restart the process of search and rescue which resulted in lose of time.

5 Conclusion

Multiagent programming languages have advanced rapidly in the last few years. They have been used to solve many different cooperative agent tasks where agents are placed in dynamic environments and can only achieve their task though cooperation. Robot is a specific type of an agent. The multi-robotic community can greatly benefit from progress in multiagent programming languages and agents teamwork in general. To demonstrate this we used AgentSpeak to develop a comprehensive multi-robotic framework that covers the whole process of teamwork from start to end. We tested and evaluated our framework using a well known robotic scenario. This framework made the development of a robots team for search and rescue intuitive and natural.

An advanced and complete framework for robotic teamwork needs to address the important issue of recovery from failure. A team of robots is much more likely to fail while performing plans than a team of artificial agents. This is because robots are located in a physical world. It is thus important to consider the failure-prone nature of robots. Some of this has been addressed and resolved in a small scale on the development of our framework. For example early commitment to the joint plan removes the need for future negotiations. This reduces the likelihood of plan failure.

However a robust framework requires a system to re-plan and recover when a plan fails. A robotic team should be able to reorganise itself and choose a different course of action when need be. For example if a robot is removed from a team, the team needs to re-evaluate their plan to understand whether they can continue with the original course of action. If they cannot, then they many need to choose a new plan or even start to form a new team.

In this framework a scenario starts with roles that are clearly defined. A future area of study is role allocation in the team of robots upon the start of a scenario. Another path of future development for this framework is the area of group obligation. That is which robot is responsible and what should do if the group did not fulfil its responsibility. In the scenario above we have seen that should a doctor fails to continue attending injured people, the remainder of the group - the rescuer and the other doctor - are still capable of working together together as a team. However if the rescuer cannot perform in the team any more, the remainder - two doctors - cannot achieve the goal. A rescuer is always needed in the team.
This project can be continued by looking at situations where a team realises that the current course of action will not accomplish the task. This could be because of a change happening to the team (e.g. a member has left the team) or a change in the environment. For future work we also suggest modifying our framework to handle situations where things do not go as expected.

References

[1] Behrens, T., Dix, J., Hubner, J., and Koster M. *Annals of Mathematics and Artificial Intelligence [Special Issue]: The Multi-Agent Programming Contest: Environment Interface and Contestants in 2010.* 61(4):257–383, Springer, 2011.

[2] Bordini, R. H., Dastani, M., Dix, J. and Seghrouchni, A. E. *Multi-Agent Programming; Languages, Platforms and Applications.* Springer, 2005.

[3] Bordini, R. H., Hübner, J. F. and Wooldridge, M. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley and Sons Ltd, 2007.

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

[5] Cohen, P. R. and Levesque, H. J. *Teamwork.* Nous, 25(4):487–512, 1991.

[6] Dix, J., Behrens, T., Dastani, T., Koester, M. and Novak, P. *Annals of Mathematics and Artificial Intelligence [Special Issue]: The Multi-Agent Programming Contest: History and Contestants in 2009.* 59(3–4):275–437, Springer, 2010.

[7] Jacoff, A., Messina, E., and Evans. J. *A standard test course for urban search and rescue robots.* In Proceedings of the Workshop on Performance Metrics for Intelligent Systems (PerMIS), 2000.

[8] Jennings, N. R. *Controlling cooperative problem solving in industrial multi-agent systems using joint intention.* Artificial Intelligence, 75(2):195-240, 1995.

[9] Kaminka, G. A. *Autonomous Agents Research in Robotics: A Report from the Trenches*. AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI, 2012.

[10] Kaminka, G. A. *I have a robot, and I'm not afraid to use it!*. AI Magazine, 33(3):66–78, 2012.

[11] Kitano, H., Tadokoro, S., Noda, I., Matsubara, H., Takahashi, T., Shinjou, A., Shimada, S. *RoboCup Rescue: search and rescue in large-scale disasters as a domain for autonomous agents research*. International Conference on Systems, Man, and Cybernetics Proceedings (IEEE SMC '99), 6:739–743, 1999.

[12] Levesque, H. J., Cohen, P. R. and Nunes, J. H. T. *On acting together*. Proceeding of the Eighth National Conference on Artificial Intelligence (AAAI-90), Boston, MA, 94–99, 1990.

[13] Rao, A.S. *AgentSpeak(L): BDI agents speak out in a logical computable language*. Proceedings of Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW-96), 1996.

[14] Shoham, Y. *Agent-oriented programming*. Artificial Intelligence, 60(1):51–92, 1993.

[15] Sklar, E., Ozgelen, A. T., Schneider, E., Costantino, M., Munoz, J. P., Epstein, S. L., and Parsons, S. *On Transfer from Multiagent to Multi-Robot Systems*, Proceedings of the Workshop on Autonomous Robots and Multirobot Systems, Valencia, Spain, 2012.

[16] Smith, R. G. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*. IEEE Transactions on Computers, C-29(12), 1980.

[17] Tambe, M. *Towards flexible teamwork*. Journal of AI Research, 28:203-242, 1997.

[18] Wooldridge, M. and Jennings, N. R. *The Cooperative Problem Solving Process*. Journal of Logic and Computation, 9(4)563–592, 1999.