Bogtrotters in Space

Dominic Carr, Sean Russell, Balazs Pete, G.M.P. O'Hare, Rem W. Collier

University College Dublin

Abstract. This is the fourth year in which a team from University College Dublin has participated in the Multi-Agent Programming Contest[1]. This paper describes the system that was created to participate in the contest, along with observations of the team's experiences in the contest. The system itself was built using the AF-TeleoReactive and AF-AgentSpeak agent programming languages running on the Agent Factory platform. Unlike in previous years where a hybrid control architecture was used, this year the system was implemented using only agent code and associated actions, sensors, modules and platform services.

1 Introduction

This years entry to the 2011 Multi-Agent Programming Contest was designed and built using the Agent Factory framework, which provides support for the development and deployment of agent-based applications using a variety of Agent-Oriented Programming (AOP) languages [5]. As with the previous years entry, our approach was centered around the use of two specific AOP languages: *AF-AgentSpeak*, an AgentSpeak variant that is based on Jason [4] and *AF-TeleoReactive*, an implementation of Nilssons Teleo-Reactive programming language [8]. Both of these langages were implemented using the Common Language Framework [11], a set of reusable components that aims to simplify the prototyping of AOP languages.

This is the fourth year in which a team from University College Dublin has participated in the contest. Last year [1] the team performed well in the herding scenario coming in third in the contest, that entry building on the work of the two preceding years [7, 6]. This years team included four members from the previous year: Rem Collier (lecturer), Sean Russell (Ph.D. Student), Dominic Carr (Ph.D. Student) and Gregory O'Hare (professor). Rem and Gregory are active researchers in the area of Multi-Agent Systems and AOP languages, Sean and Dominic are working in the area of agent-enabled Wireless Sensor Networks. The final member of the team was Balazs Pete, a 2nd year undergraduate student who worked on the contest as part of a summer internship.

Our primary motivation to compete in the contest was to test and debug AgentFactory, and further refine and direct its development trajectory. As in in previous years we were strongly motivated to provide new researchers with practical exposure to AOP using Agent Factory. The contest fits this need, well providing an interesting problem to solve. We also wanted to drive language development within AgentFactory.

To this end, one of our goals was to remove our dependency on a behaviour-based architecture that had previously been used to implement core behaviours of the system. Instead, we aimed to increase our utilization of AOP languages, and to replace

[1] http://www.multiagentcontest.org/2011

the behavioural layer with Teleo-Reactive functions. Additionally, we sought to use the Environment Interface Standard (EIS)[3] integration provided with the contest server instead of building a custom solution. Further details of our approach can be found in section 3.

2 System Analysis and Design

The development model used was based on a team programming approach to system development that was adopted for the previous contest [1]. At any point in time, one team member was actively engaged with coding and the other team members provided strategy analysis, debugging assistance etc. Whenever the team identified a possible strategy, a "champion" was assigned to seperate from the main group and to flesh out the idea. Once finished, the "champion" then presented the idea to the rest of the group. If accepted by the group, the idea was prioritised and added to the to do list.

The overall approach adopted was decided upon at the start of our involvement in the project. In essence, our objective was to maintain the centralised task allocation model used in the previous architecture, but to replace the low level behaviours with teleo-reactive functions and EIS integration. All of our analysis and design work was targeted at solutions which were compatible with this general architecture. We did not consider any of the software engineering methodologies outlined in the literature. Where necessary, Agent UML Protocol Diagrams were used to illustrate interaction protocols and pseudo code, adapted to our planning language, was used to outline plans. For the AF-TeleoReactive programs, a function hierarchy diagram was used to outline designs.

In our architecture coordination of information is carried out through the use of a number of platform services, which represent shared platform wide resources for the agents. The primary service is the map service, in simplest terms this service could be viewed as a whiteboard where agents could post relevant information to be made available to other agents, secondly it also allowed some analysis of the data to be performed e.g. identifying the highest value zones or routing between two vertices in the graph. Team coordination was achieved through the use of another platform service, the group service, which was used by the leader agent to assign tasks to the individual agents.

The attributes of *autonomy*, *pro-activeness* and *reactiveness* were implemented by making use of two AOP languages and more specifically through the structure of the *AF-TeleoReactive* programs which can allow the agents to bypass their assigned task in favour one of their own goals.

Our system is a true multi-agent system with centralised coordination. The choice of centralised coordination was made in an effort to allow the rapid prototyping of different task allocation strategies during development while abstracting from communication issues. In order to facilitate this centralised coordination, a leader agent was specified to complement the in-situ ATPV agents. This strategist agent performs some rudimentary analysis of the graph to determine the cluster of vertices with the highest value, then based on the structure of the graph a number of vertices are selected on which to position agents, which are then associated with individual agents.

During the course of the contest or prior to this we did not discuss our strategies with any other teams, this was primarily in an effort to remain competitive as our experience from participation in last years contest highlighted the importance of effective strategies. To this end we only participated in a single test match which we used to ensure we could successfully connect to the server and perform actions in a reasonable time.

The implementation of the system took approximately 600 hours, the majority of this time can be attributed to an internship undertaken by Balazs Pete over a 12 week period from June until the end of August. The remainder of the time can be accounted for when the rest of the team picked up the development two weeks before the contest.

3 Software Architecture

As in previous years, our system utilised Agent Factory (AF) [5] as our underlying agent technology. AF is an open-source Java-based framework that provides support for the development and deployment of agent-oriented applications. Specifically, it provides a generic Run-Time Environment (RTE) for deploying agent-based systems that is based on the FIPA standards [9] together with a set of *development kits* that facilitate the implementation of diverse agent types, ranging from custom agent architectures to agent programming languages.

The RTE consists of a set of configurable agent platforms that contain the machinery necessary to deploy these agent types together with support for the deployment of platform-level resources, known as platform services, that are shared amongst the agents residing on the platform. Other support includes a range of monitoring and inspection tools that aid the developer in debugging their implementations.

The development kits provide the core agent interpreter/architecture together with appropriate customisations for the AF tool support. This will typically include a set of plugins for the AF Debugging Tool and an Eclipse plugin for the AF IDE (which is a set of plugins for Eclipse). For the purposes of this competition, we made use of two of the AOP language development kits packaged with AF, which are introduced next.

3.1 AF-TeleoReactive

AF-TeleoReactive is based on Nils Nilsson's Teleo-Reactive agent paradigm [8] which was designed to react to a changing environment (hence reactive) whilst still performing actions which take it to it's goal (hence teleo, meaning goal oriented). The functional components of AF-TR agents are represented by an ordered list of production rules.

An example of a production rule would be $K \rightarrow A$, where the element K represent conditions on the input from the sensors or the model of the environment, and the element A represent an action on the environment. When a sequence is being interpreted it is scanned from the top until it comes across a rule whose condition is satisfied. The corresponding action is then performed and the interpreter is then restarted from the top of the list.

Information about the current state of the environment is gathered via a set of *Sensors*: Java classes that convert raw sensor data into beliefs that are added to the

agents belief set. To handle the potentially dynamic nature of the environment that the agent is sensing, beliefs stored in the AF-TR belief base do not persist by default. Instead they are wiped at the start of each iteration of the agent interpreter. To cater for beliefs that should persist, consideration must be given to this when creating the sensor, which allows the programmer to define which types of beliefs should persist. Whether a belief should persist or not depends on the nature of the item being observed. For instance, in the context of the agent contest, it would safe to adopt a temporal belief regarding the position of a edge within the map (which by its very nature cannot move) whereas a belief about the location of a enemy agent will change over time.

AF-TeleoReactive was developed based on the notion of blind commitment, is so far as the agent will continue performing an action until its actions have modified the environment sufficiently to cause another condition to fire. As such it is assumed that the continuous execution of an action will cause such a change in the environment.

3.2 AF-AgentSpeak

AF-AgentSpeak is based on Jason [4], a purpose-built agent-oriented programming language that implements an extended and improved version of Rao's AgentSpeak(L) language [10]. The language consists of a set of plan rules, examples of which are shown in Fig. 1. Each plan rule consists of a triggering event, a context and a plan containing a number of actions that should be adopted if the plan rule is selected.

The deliberation cycle of AF-AgentSpeak is an adaptation of the algorithm used in Jason that is compliant with the AF common language framework.

1. An event is selected from the set of internal and external events.
2. All plan rules triggered by this event are then selected.
3. The list of rules is reduced to those whose context evaluates to true.
4. From this list a single plan rule is selected and added to a new or existing intention stack depending on whether it is a sub plan or new plan respectively.
5. The next step from each of the agents current intentions is executed in parallel by the agent.

As with Jason, AF-AgentSpeak offers an extended suite of functionality that is not available in the original version AgentSpeak(L). This includes support for inheritance, partial plans, abstract plans, and plan overriding as described in [2]; and an extended set of plan operators, including: *foreach* (plan expansion), *while* (loops), *if* (selection), *wait* and *when* (delayed execution), and = (assignment). This extended planning language is provided as part of the Common Language Framework [11].

Figure 1 contains part of the code for the Leader agent. As can be seen in this figure, the Leader agent accesses the Group and Map services through two purpose-built modules (these same modules are used by the AF-TeleoReactive agents). On creation, the leader sets up 3 groups: a *home* group, which is responsible for building and holding the zone; an *away* group, containing the Saboteurs, and which is charged with the task of attacking the enemy; and a *support* group which contains a single Repairer and is responsible for supporting the away group. Once the groups are setup, the *@setup*

```
#agent Leader

module groups -> com.agentfactory.mapc.GroupModule;
module map -> com.agentfactory.mapc.MapModule;

@initialization
+initialized : true  <-
    groups.setup(groups, [
        team(home, [Repairer, Inspector, Sentinel, Sentinel, Explorer, Explorer, Inspector]),
        team(support, [Repairer]),
        team(away, [Saboteur, Saboteur])]),
    @setup;

+sim(end) : true <-
    groups.reset,
    map.resetService,
    .println("resetting"),
    @setup;

#partial @setup <-
    .abolishAll(strategy(?t, ?x)),
    groups.setShared([cash(0), stepNo(0)]);
...
```

Fig. 1. Example AF-AgentSpeak code from the Leader agent

partial plan is invoked, which does additional configuration steps. A partial plan is used here because the code is common to both the initialization of the agent and the resetting of the agent when the current simulation ends.

3.3 Core Architecture

The core architecture of the system is shown in Figure 3, the agents communicate through the use of the two platform services *MapService* and *GroupService*. No agent communicates directly with another agent wether it represents an ATPV or the leader which has no embodiement in the simmulation. Receiving shared data is done without any requirment to request or subscribe to particular forms of data. All agents automatically pull all relevant information during the perception phase of their execution, allowing the agent to use the shared beliefs as if they were it's own percepts. An example of this is shown in the first rule in Figure 2, this is taken from the repair function, which is activated when a dissabled agent is on the same vertex as a Repairer. The first percept is one shared through the MapService and the second and third are shared through the GroupService. The second line of the example is taken from the *doAction* function where the *?action* variable is the first parameter of the function (in this case repair) and the *?param* variable is the second (the name of the agent to be repaired).

Agent communication with the server is handled through the Agent Factory EIS layer and the eismassim jar, this is shown by the second rule in Figure 2 whereby the action and parameters are passed to the service in the form of a predicate.

```
agentInfo(?name,?team,?ver,Saboteur,disabled) & memberOf(home) & groupOrder(away, ?name, primary)
          -> doAction(repair,?name, .nil)

step(?step) & (?action == repair) & ~doneActionForStep(?step, ?act) -> eis.perform(repair(?param))
```

Fig. 2. Example of role based action selection

The agents were not distributed across multiple machines, implementation of this would have required a modification to the platform service. this was not done due to time constraints.

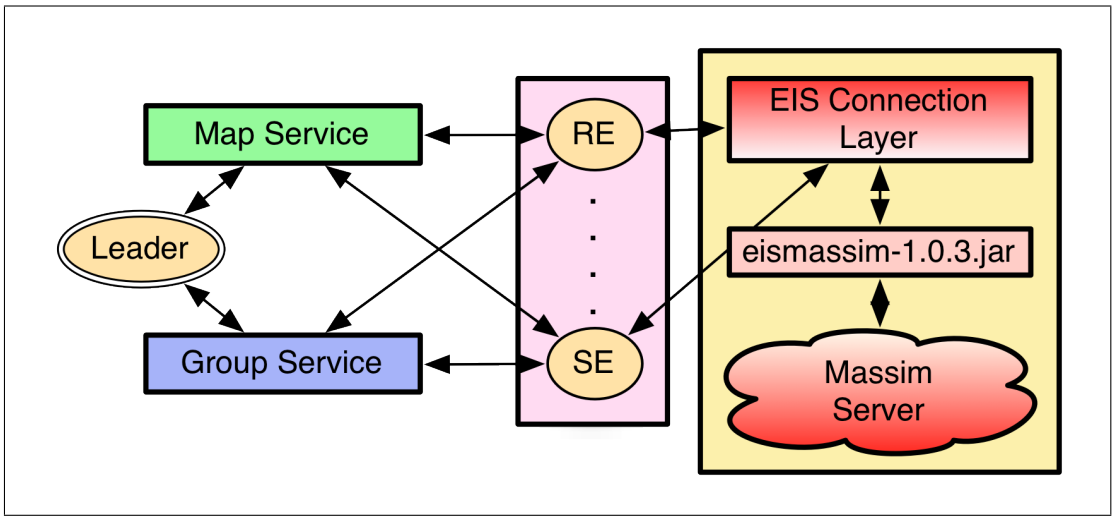


Fig. 3. Core Architecture

4 Strategies, Details and Statistics

4.1 Strategy

The overall team strategy is the combination of a number of role dependent strategies and the over all zone creation strategy. As discussed in Section 3.3 agent coordination and information sharing is done through a number of whiteboard type services. Based on information shared through these services, the agents will perform different actions based on their role or assigned tasks.

4.2 Agent Mental State

In general the prevailing goals which drive the agents are to hold the positions assigned by the leader, but not all agents were assigned positions. Secondary to this goal some agents may override this goal with their own periodically. An example of this would be that *Inspectors* periodically decide to refresh the teams knowledge of the attributes of the enemy and as such move to inspect each one. Agents can change their behaviour based on a number of factors such as the conditions shown in Figure 4, or the exceptions is given in Table 1.

Saboteurs are not assigned tasks, they have a priority based attack system where the priority targets are Saboteurs then Repairers and then everyone else, when en-route to a target a Saboteur will attack any enemy agent the happens to be on the same vertex. Saboteurs also use a targeting separate targeting system when two friendly Saboteurs are on the same vertex ensuring that they both selected different targets.

Repairers are also not assigned tasks directly, rather they are associated with a Saboteur which they are charged to support. To mitigate the risk of repairers being disabled when traveling to repair a teammate, we elected to have the disabled agent travel to the Repairer. This decision also allowed us to keep the two repairers in relatively close proximity to their assigned Saboteur to minimise the risk of one of them being disabled.

Table 1. Agent Autonomic Actions

Role	**Condition**	**Action**
Inspector	Enemy with data older than 100 steps	Inspect the enemy
All agents	Become disabled	Move to closest Repairer
Explorers	Map becomes dominated (all vertices owned)	Adopt goal to probe every vertex
All Agents (Except Saboteurers)	Enemy Saboteur on vertex	Parry presumed attack
Repairer	Disabled teammate on vertex	Repair teammate

The agents contain an explicit mental state composed of the percepts received from the server as well as all internally generated beliefs and those received from platform services.

There are both implicit and explicit hierarchies within the structure of the agents. Informally the Leader agent is responsible for assigning tasks to a sub group of the ATPV agents. Additionally the Repairers were tasked in a support role following the Saboteurs but did not answer to them explicitly.

Agent path planning was done using a form of breadth first search which can ignore edges above a certain weight, this algorithm was chosen as it seemed that the number of steps taken to travel the a distance was more important than the amount of energy expended.

4.3 Implementation Details

The topology of the map was not analysed in any great detail, basic clustering was performed to determine the best position to use as the centre of the captured zone, this is modelled on breadth first search where the cumulative value of the vertices is summed. When the value is not known it is assumed to be 1.

The agents use the eismassim package in order to communicate with the server, this is achieved through an existing integration of EIS with AgentFactory.

The role of an agent is captured during the initial percept and stored for reference within the code, in this way during the execution of the agent. An example of this is given in figure 4, the first line represents the goal that we should know the role of each of the enemy agents and the second rule states that we should refresh the information we hold on that agent if we haven't inspected them in the last 100 steps.

```
role(Inspector) & agentInfo(?name,?team,?vertex,?role,?s) & (?role == none) -> inspectAll
role(Inspector) & lastInspected(?n,?s)  & step(?step) & (?step > ?s+100) -> inspectEntity(?n)
```

Fig. 4. Example of role based action selection

4.4 Zones

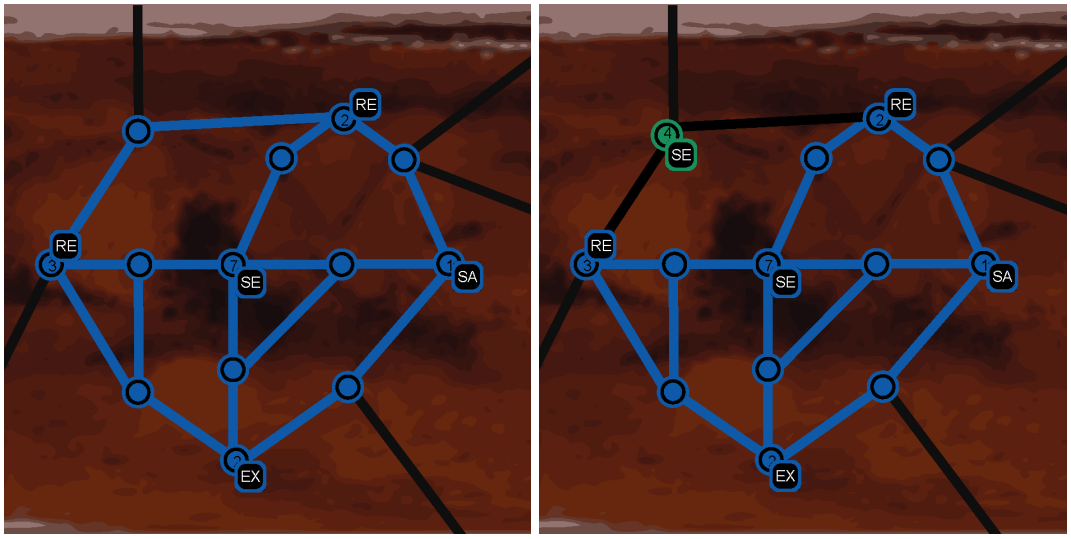
Zone selection was very simplistic and was not concerned with the actual value we would attain if we we holding certain positions, instead we opted to use a simple clustering algorithm based on breadth first search to identify a good position to use as the centre of our zone.

Zone defense was designed around a well known concept in ship design whereby it is divided into a number compartments, the basic principle is that when agents are positioned two steps away from other agents, and sometimes the centre of the zone, this naturally creates a pattern where subzones exist within the zones. An example of this is shown in Figure 5(a), when an enemy attempts to break the frontier of the zone the compartment collapses but the rest of the zone remains, this is shown in Figure 5(b).

We did not specifically implement a team behavior for attacking enemy zones but this occasionally emerges out of aggressive Saboteur behavior, or frontiers may be broken by an Inspector approaching to inspect enemy agents.

4.5 Buying Algorithm

We used a complex recursive buying function designed to maximise the effectiveness of the our Saboteur agents, it was designed to react to the attributes of the enemy agents always ensuring that based on our current knowledge we could disable any of the enemies



(a) Zone configuration (b) Zone configuration under attack

Fig. 5. Subgraphs showing zone configuration

agents with one hit and survive one hit from the enemy saboteurs. The function was designed recursively such that when the cash was available a number of agents could purchase simultaneously based on a hard coded priority.

The achievement points were very important in our strategy as a number of our victories were based on the fact that our Saboteurs we able to dominate the enemy Saboteurs. Secondarily we relied on excess achievement points being spent on a very high visibility for one of the teams Sentinels in order to keep as much information on the enemy movements as current as possible.

As an aside during the final game when it was clear that we could not increase our standing in the league table we attempted a strategy whereby the team did not spend any of the achievement points. By the end of the simulation we had amassed 54 points resulting in a much more competitive and exciting simulation.

5 Conclusion

From our participation in the contest were were enable to evaluate and test the functionality of modifications made to the languages we were using. The contest proves every year to be an impetus for development of new features and thorough testing within the languages created using the Common Language Framework. It has further affirmed the stability of the chosen languages for their roles within the architecture of the system, In that Af-Agentspeak is suited for the organisational role of the leader and AF-TeleoReactive is suited for the more reactive and time dependent control of the ATPV units in the simulation.

The team overall performed well, the individual roles of the agents sometimes combining to work very well together, at an individual level the strategy of the agents worked quite well but on some occasions the team as a whole became uncoordinated. This resulted from a error in our approach to development, focusing too much on the low level individual strategies than the higher level coordination strategies.

As in our conclusion last year we believe that the results of the competition reflect the effectiveness of the team strategies. In both of the games with the top two teams we were outclassed in terms of strategy and as a result we were comprehensively beaten. Our closest match was with third placed TUB, despite losing all the simulations, it was much closer and much more exciting than any of the games with the lower ranked teams.

For next years contest some changes we would consider beneficial would be to introduce more static assignment of roles to agents, we believe that this could allow a greater diversity in capabilities and characteristics of the ATPVs.

References

1. From bogtrotting to herding: a ucd perspective. *Annals of Mathematics and Artificial Intelligence*, pages 1–20.
2. Reuse by inheritance in agent programming languages. *Intelligent Distributed Computing V*, pages 279–289, 2011.
3. Tristan Behrens, Koen Hindriks, and Jürgen Dix. Towards an environment interface standard for agent platforms. *Annals of Mathematics and Artificial Intelligence*, 61:3–38, 2011.
4. R.H. Bordini, J.F. Hübner, and M. Wooldridge. *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. Wiley-Interscience, 2008.
5. Rem W. Collier. *Agent Factory: A Framework for the Engineering of Agent-Oriented Applications.* PhD thesis, School of Computer Science and Informatics, 2002.
6. M. Dragone, D. Lillis, C. Muldoon, R. Tynan, R. Collier, and G.M.P. O'Hare. Dublin bogtrotters: Agent herders. *Programming Multi-Agent Systems*, pages 243–247, 2009.
7. H. Jordan, J. Treanor, D. Lillis, M. Dragone, R.W. Collier, and G.M.P. O'Hare. AF-ABLE in the multi agent contest 2009. *Annals of Mathematics and Artificial Intelligence*, pages 1–21.
8. N. Nilsson. Teleo-reactive programs for agent control. *Arxiv preprint cs/9401101*, 1994.
9. S. Poslad, P. Buckle, and R. Hadingham. The FIPA-OS agent platform: Open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, pages 355–368, 2000.
10. A. Rao. AgentSpeak (L): BDI agents speak out in a logical computable language. *Agents Breaking Away*, pages 42–55, 1996.
11. Sean Russell, Howell Jordan, Gregory M P O'Hare, and Rem W Collier. Agent factory : A framework for prototyping logic-based aop languages. In *In Proceedings of the Ninth German Conference on MultiAgent System Technologies MATES 2011*, 2011.

6 Short Answers

6.1 Introduction

1. To involve new researchers with agent-oriented development, and to drive development of Agent Factory.
2. This is the Fourth year in which a UCD team has entered the contest. The team has had different configurations over the years, with Dr. Collier as our constant.
3. UCD Bogtrotters
4. The team had 4 developers, all of who were involved in the system design. We have 2 PhD candidates, one lecturer, and one undergraduate student.
5. Sean Russell and Dominic Carr are involved in WSN research, Balazs Pete is still undertaking undergraduate education, Dr. Collier is an active researcher in the Agents field.

6.2 System Analysis and Design

1. Due to time constraints, our system was not specified or designed using any particular multi-agent system methodology.
2. No, all information is disseminated to all agents.
3. Communication is performed through platform services providing simple whiteboard type functionality.
4. The attributes of *autonomy*, *pro-activeness* and *reactiveness* were realised in the use of Agent-Oriented Programming languages and through the structure of the agent programs which allow bypassing of the assigned task in favour one of their own goals.
5. Our system is a true multi-agent system with centralised coordination.
6. 600 man hours.
7. We did not discuss our strategies with other developers. We participated in one test match to test the connection and to assess if actions were completed in a timely manner, more testing was not carried out due to time constraints.

6.3 Software Architecture

1. The Java programming lanquage was used as it forms the basis for AgentFactory.
2. The system was implemented using the AF-TeleoReactive and AF- AgentSpeak multi-agent programming languages running on the Agent Factory platform.
3.
4. Our development platform was the Eclipse IDE with the inclusion of the AgentFactory plugin. All members were familiar with Eclipse and, new members spent 1-2 hours learning how to best use the plugin.
5. AgentFactory.
6. During initial development a number of features were identified as missing from AF-TeleoReactive such as parallel action execution and executing actions for all available bindings. These were implemented prior to the start of the competition.

7. Parallel action execution and executing actions for all available bindings simplified the development task greatly.
8. A modified version of breadth first search designed to be aware of the weight of edges and maximum energy of the agent.
9. The agents were not distributed across multiple machines, implementation of this would have required a modification to the platform service. this was not done due to time constraints.
10. The reasoning cycle of the agent was asynchronous with respect to the receive-percepts/send-action cycle, as such we had to prevent the agent from performing more than one action in a single simulation step.
11. The most difficult part of the development process was the synchronisation of individual agent behaviours to have the intended combined effect.
12. Java Code: 3863, Agent Code: 407, Total: 4270

6.4 Strategies, Details and Statistics

1. The overall team strategy is the combination of a number of role dependent strategies and the over all zone creation strategy.
2. The leader agent assigns task to the other agents, information is shared though platform services.
3. The results of map analysis are exploited to find the best position to use as the centre of the captured zone.
4. The agents use the eismassim package to communicate with the server.
5. All agents have the same source code, an agents role was stored as a predicate with the agents beliefs and used to determine the actions the agent would consider/perform.
6. We employed a simple clustering algorithm to identify a position to use as the centre of our zone.
7. Zone defence was based on subdivision of zones. When agents are positioned two steps away from other agents, this creates sub-zones within a zone, which can persist if the frontier is breached. We did not implement an aggressive behaviour, but this occasionally emerges out of the Saboteurs or Inspectors behaviour.
8. Agents can change their behaviour based on a number of factors such as the presence of enemy agents or being disabled.
9. Breadth first search.
10. We exploited the buying mechanism to maximise the effectiveness of the our Saboteur agents, the algorithm was reactive to the current knowledge of enemy capabilities.
11. The buying mechanism was used to attempt to gain supremacy in individual battles between saboteurs.
12. Yes, they have a mental state made up of internal beliefs and shared information from the platform service.
13. Agents communicate through a shared a platform service. The leader assigns tasks through this service.
14. There are both implicit and explicit hierarchies within the structure of the agents. Informally the Leader agent is responsible for assigning tasks to a sub group of the

ATPV agents. Additionally the Repairers were tasked in a support role following the Saboteurs but did not answer to them explicitly.
15. Individually the agents performed as expected, but at a team level emergent behaviour was observed.
16. The leader agents perform advance planning as regards placement of ATPV agents in the future steps.

6.5 Conclusion

1. The suitability of different AOP languages to different positions within a multi agent system.
2. Strong: Individual agent strategy, Weak: overall team strategy.
3. The chosen programming languages were very suited to the roles they were applied to.
4. More Diversity between agent roles.
5. Strategy was the primary factor in the final tournament rankings.
6. The contest may be of interest to researchers in the broad AI field who are not directly working with agents.
7. Increasing the size of the maps and the number of agents would create a more challenging contest.