From Bogtrotting to Herding: A UCD Perspective

Sean E. Russell, Dominic Carr, Mauro Dragone,
Rem W. Collier and G. M. P. O'Hare

School of Computer Science and Informatics
University College Dublin
{sean.russell, dominic.carr, mauro.dragone,
rem.collier, gregory.ohare}@ucd.ie

Abstract. This is the third year in which a team from University College Dublin has participated in the Multi Agent 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. A hybrid control architecture inspired by the SoSAA strategy aided in the separation of concerns between low-level behaviours (such as movement and obstacle evasion) and higher-level planning and strategy.

1 Introduction

This year's entry to the 2010 Multi Agent Contest was designed and built using the multi agent framework Agent Factory [1]. Agent Factory, which is described in further detail in Section 4, is a modular and extensible Java based Agent framework. Specifically two languages constructed using Agent Factory's common language framework, AF-TeleoReactive and AF-AgentSpeak, were used in the system. The overall architecture of the system, which was described in detail in [2], is loosely based on the robotic control architecture SoSAA [3] in which system components are supervised by agents thereby providing goal-driven capabilities and ease of interoperability through agent communication.

Building on the last two year's entries to the contest (named "AF-ABLE" [2] and "Dublin Bogtrotters" [4]) allowed the team to focus on integrating some new technologies developed within Agent Factory, namely the languages AF-TeleoReactive and AF-AgentSpeak, and also to focus on optimisation of the lower level behaviours. Whilst we did not feature highly in the score boards in the previous two years (placing 6th and 7th respectively), the performance of the system was gradually increasing and as such we were confident in building upon the our existing system. This was our main motivator for entering the competition this year[2] as it was our belief that optimisation of the base system along

[1] http://www.multiagentcontest.org/2010

[2] The agent code used by the team can be downloaded from http://sourceforge.net/projects/agentfactory/

with a more refined strategy would result in a competitive system. An ongoing motivation of our involvement in the competition is that of education and student integration, as the contest provides an interesting problem with which to introduce students to agent-based programming within Agent Factory. This goal was realised through the inclusion of the agent-oriented programming language AF-TeleoReactive developed as a final year project by an undergraduate within the UCD School of Computer Science and Informatics.

Even though the main aim remained to improve significantly on last year's entry, this was subdivided into several objectives. The first of which was to ensure the stability of the base system and improve the performance of the low level components of the system. The team was then free to optimise the performance of the high level behaviours such as fence operation and herding whilst simultaneously developing new strategies geared toward maximising performance. Additionally quite an amount of time was spent analysing higher level task allocation and incorporating the newly developed strategies into the allocation model.

The hardware which was used to run the system for the competition was contingent on who was monitoring the simulation. As such the machines used were a Dell XPS running Windows 7, a MacBook running OS X 10.5 and a HP tablet pc running Ubuntu 10.04. The ability to run the system on heterogeneous hardware and operating systems stems from the fact that Agent Factory is Java based. As the processing power of the laptops was sufficient for the needs of the contest, only one was used at a time even though the system was designed so as not to preclude the addition of further machines if necessary.

2 System Analysis and Specification

In previous years the requirements analysis process was based on user stories [5], in which tasks, behaviours etc are defined in a high level manner denoting expected functionality. These tasks were then disseminated to and implemented by team members.

For this year's entry, in an effort to overcome the numerous problems encountered previously, a team programming approach was adopted. In which one team member actively engaged with coding supported by other team members providing strategy analysis, debugging assistance etc. This alleviated the problem of inconsistent modifications to the code base encountered last year. Despite a lack of provision for parallel development, a significantly rapid development pace was achieved. This approach allowed for a free, almost constant, flow of ideas which resulted in the development of new strategies. Similarly this approach allowed for easy management of a centralised repository of plans detailing strategies to overcome bugs or implement new functionality.

At the outset of team programming sessions an informal requirements analysis and update was conducted, from this goals were broken down into logical sub-components for implementation.

There was no formal specification generated, it was felt that the relatively small degree of change to the underlying system negated the need for formal specification.

Anticipating that very little specification work needed to be done, we did not adopt any particular multi-agent system methodology. System specifications were built upon during the team programming sessions. The application of unit testing greatly improved the efficiency of programming efforts.

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 strategist agent was specified to complement the herding agents. This strategist agent operates by monitoring a shared world model, which is used by the agents to record and share their percepts about their environment and surroundings. A list of possible tasks is generated by the world model; the utility of each task, and its cost to each agent, is evaluated; then agents are centrally assigned to tasks. These tasks may include the opening of a fence, exploration or the herding of a particular group of cows. The mechanism by which tasks may be achieved is left to each individual herding agent's own proactivity and autonomy and is done without further central input. Agents carry out their tasks until either the task is complete, or the agent is reassigned. Further details of task selection and allocation is contained in the following sections.

3 System Design and Architecture

As discussed in Section 2, the overall coordination strategy adopted in the UCD Bogtrotters system is a centralised task allocation model, this approach remains consistent with last year's entry.

The composition of the system is shown in Fig. 1. At the core of the system is a group of herder agents, implemented in the AF-TeleoReactive agent oriented programming language, which are representative of the herding entities present in the contest.

Herder Agents

Each herder is a three-tier entity. A low level tier serves as wrapper for the *AbstractAgent* class, provided the contest organisers. In this layer a connection to the server is established and maintained. Perceptions are received from the server and details of the agent's movements are passed to the server. Received perceptions are recorded in a world model which is shared by all agents, this provides each agent with information regarding his team-mates position and surroundings.

The behavioural layer is composed of a set of Java classes providing a library of available behaviours for use by the agents. Included therein are behaviours for tasks such as obstacle avoidance or exploration. At any time the agent will

be engaged in one active behaviour. The agent's next preferred move will be calculated by their currently executing behaviour. Specific detail regarding the implementation of the behavioural layer is described in [2].

The deliberative layer is the agent's top most layer. Dependent upon their assigned role the agent is responsible for conducting its own planning so as to carry out the task. This layer is written in AF-TeleoReactive, it serves to activate and configure particular behaviours according to it's own reasoning process.

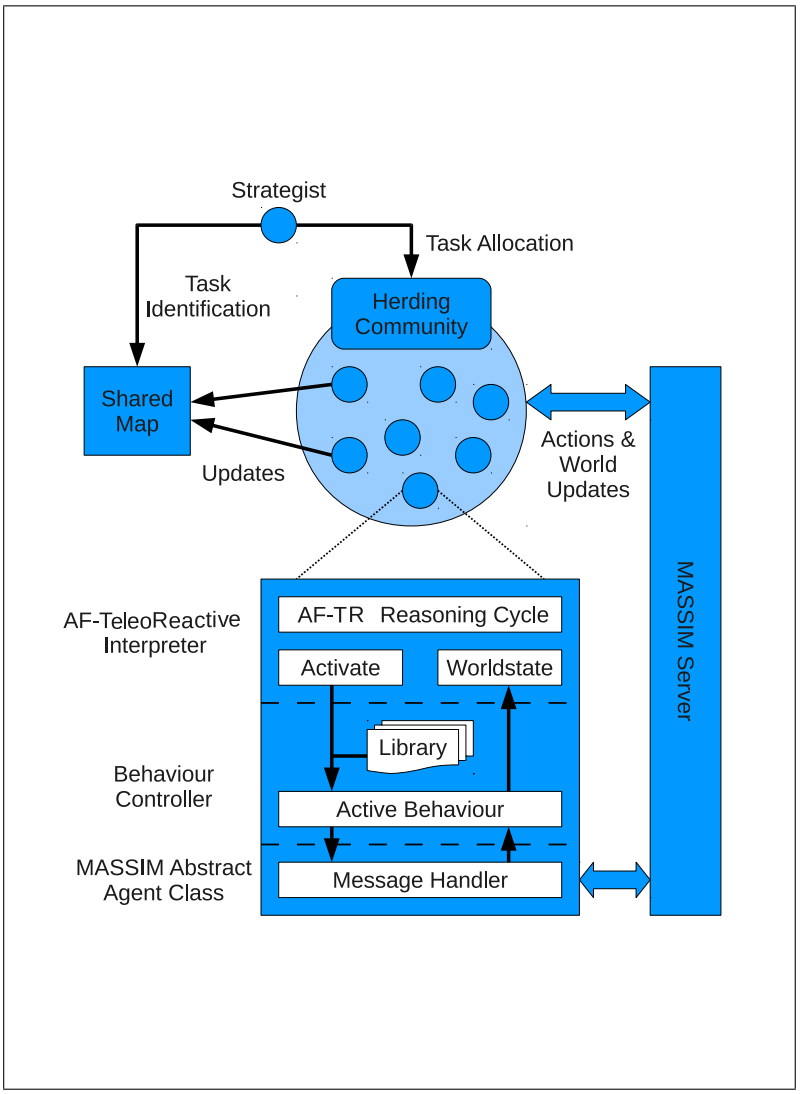

Fig. 1. Schematic of the Herder agent architecture

Agent autonomy is constrained by the task they have been assigned, and the goals associated with that task, within a given task an agent is free to operate as it pleases. A detailed description of the AF-TeleoReactive language and its use in the implementation in the deliberative layer is provided in Section 4. The AF-TeleoReactive layer monitors the behavioural layer and responds to key events, those which update the agents world state, issued in the behavioural layer. These key events may necessitate the selection and activation of new behaviour. For example, when an agent is assigned exploration of a particular area of the map, the task could be decomposed into two simpler behaviours. Firstly the agent moves via the shortest path towards the centre point of the area, when the agent nears the location, an event will be raised to indicate this which in turns triggers a specific exploration behaviour.

The behavioural and deliberative layers operate in their own execution contexts (the lower layer in its own thread while the upper layer applies whatever scheduling policy has been installed on the agent platform) as such deliberation time is not constrained by the need to select an action in an allotted time frame.

Strategist Agent

Aside from the Herder agents, the system encompasses one other vital agent, the Strategist agent. The Strategist agent is responsible role assignment, this is conducted so as to maximize the success of the whole team. It's decisions are based upon the shared world model, maintained by the precepts of the herder agents. Its decisions are communicated to each agent, this results in them taking on new roles associated goals. The world model may also be accessed through a visualization tool, giving developers the ability to watch matches in real-time.

Communication: Two differing communication models are utilized within the system architecture. Role assignment, as dictated by the strategist agent, is disseminated through explicit communication, conducted via FIPA Agent Communication Language (ACL) messages. The shared world model is representative of implicit agent communication, as each agent records its precepts of the environment state for use by all other agents.

Coordination: The coordination of teamwork amongst the herding agents takes two forms. Firstly, through use of a centralised task allocation system, the Strategist agent can ensure multiple agents are not assigned to the same task. Exceptions are made for tasks for which successful completion requires multiple agents e.g. defensive herding. The herding task demonstrates our second form of teamwork. By way of the shared world model an agent monitors the position of its team-mates. This allows the agent to plan its own movements while factoring in the movements of other agents on its team. This method avoids the necessity of expensive, time-consuming ACL communication while still ensuring that accurate, relevant information is available.

4 Programming Language and Execution Platform

The underlying agent technology utilised by our team is Agent Factory (AF) [1], an open-source Java-based development framework that provides support for the development and deployment of agent-oriented applications.

Agent Factory provides a generic run-time environment for deploying agent-based systems that is based on the FIPA standards [6]. Central to this environment is a configurable agent platform that supports the concurrent deployment of heterogeneous agent types employing a range of agent architectures and interpreters. AF also supports the deployment of platform-level resources in the form of platform services that are shared amongst agents, along with monitoring and inspection tools that aid the developer in debugging their implementations.

Support for the implementation of specific types of agents is realised via the notion of a *development kit*, an example of which is the AF-TeleoReactive (AF-TR) [7] Development Kit, which provides support for the fabrication of agents based on the AF-TR agent-oriented programming language. This kit consists of a purpose-built interpreter, a plugin for the Eclipse IDE and a custom set of views for the AF Debugger that allow the developer to inspect the internal state of AF-TR agents.

For the competition we utilised two types of agents, AF-TeleoReactive developed as the final year thesis of one of our undergraduate students [7], and AF-AgentSpeak, developed as a demonstration of Agent Factory's constructs for rapidly prototyping agent-oriented programming languages. AF-TeleoReactive agents were used to control the herders and an AF-AgentSpeak agent was used as the Strategist agent. In the following sections we shall describe both agent programming languages.

4.1 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.

$$
\begin{array}{c}
K_1 \rightarrow A_1 \\
K_2 \rightarrow A_2 \\
\cdots \\
K_i \rightarrow A_i \\
\cdots \\
K_n \rightarrow A_n
\end{array}
$$

Fig. 2. Example of AF-TeleoReactive production rules

In the example given in Fig. 2, the element K_i represent conditions on the input from the sensors or the model of the environment, and the element A_i 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 wall within an arena (which by its very nature cannot move) whereas a belief about the location of a cow 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.

As can be seen in the Herder agent code shown in Figure 3, goals are defined implicitly through the *ActiveTask* belief. When the agent receives a message detailing it's task from the leader agent (actually, we assume naively that the agent sending the message is the Strategist agent), the corresponding action is to define this as the active task (line 4). As there can be only one active task at any time one of the following lines (05 - 11) would activate a particular function. Goals in this system can be viewed as both *Maintenance* and *Achievement*, where an Achievement goal is dropped once the agent believes is is complete and Maintenance goals are maintained even when they have been achieved. As an example, line 03 defines the goal that the agent be connected to the server, in this way until the agent believes that it is connected it will continue to execute the connect function. Once the agent believes it is connected the function is no longer executed (*Achievement*). Should the agent no longer believes that is is connected the function will be activated again until it believes it is connected (*Maintenance*).

The active task of the agent defines which of the functions specified below is activated

- *explore*: the agent moves towards a given set of unexplored coordinates
- *annoyEnemy*: the agent moves into the enemy corral to remove the enemy's cows
- *defendCorral*: the agent positions itself outside the corral pushing cows back in when the enemy removes them

```
00 #agent BasicHerder;
01 module bcm -> agentcontest.core.module.BehaviourControllerModule;
02 function main{
03   ~connected -> connect
04   message(REQUEST, ?agentID, doTask(?task, ?params)) -> bcm.setActiveTask(?task, ?params)
05   activeTask(Explore, params(?x, ?y)) -> explore(?x, ?y)
06   activeTask(Herd, params(?x, ?y, ?h, ?p)) -> herd(?x, ?y, ?h, ?p)
07   activeTask(AnnoyEnemy, params(?x, ?y)) -> annoyEnemy(?x, ?y)
08   activeTask(DefendCorral, params(?x, ?y)) -> defendCorral(?x, ?y)
09   activeTask(OpenFence, params(?x, ?y)) -> openFence(?x, ?y)
10   activeTask(Stop, params(?x, ?y)) -> stop
11   ~activeTask(?task, ?params) -> stop
12 };
13
14 function connect{
15   mapService(?id) & ~service(?id) -> .bind(?id)
16   ~activity(ActivityReadWorldModel) -> bcm.addActivity(massim.af.behaviours.ActivityReadWorldModel)
17   ~behaviour(MoveTo) -> bcm.addBehaviour(massim.af.behaviours.BehaviourMoveTo)
18   ~behaviour(MoveToViaShortestPath) -> bcm.addBehaviour(massim.af.behaviours.BehaviourMoveToViaShortestPath)
19   ~behaviour(AnnoyEnemy) -> bcm.addBehaviour(massim.af.behaviours.BehaviourAnnoyEnemy)
20   ~behaviour(DefendCorral) -> bcm.addBehaviour(massim.af.behaviours.BehaviourDefendCorral)
21   ~behaviour(OpenFence) -> bcm.addBehaviour(massim.af.behaviours.BehaviourOpenFence)
22   ~behaviour(SingleHerd) -> bcm.addBehaviour(massim.af.behaviours.BehaviourSingleHerd)
23   ~behaviour(Stop) -> bcm.addBehaviour(massim.af.behaviours.BehaviourStop)
24   serverDetails(?host,?port) & credentials(?id,?pw) & mapService(?sid) &id(?aid)&~connected ->
                 bcm.connect(?host, ?port, ?id, ?pw, ?sid,?aid)
25 };
26
27 function explore(?x, ?y) {
28   true -> bcm.activateBehaviour(MoveToViaShortestPath(x, ?x, y, ?y, tolerance, 5))
29 };
30
31 function openFence(?x, ?y) {
32   true -> bcm.activateBehaviour(OpenFence(x, ?x, y, ?y))
33 };
34
35 function annoyEnemy(?x, ?y) {
36   true -> bcm.activateBehaviour(AnnoyEnemy)
37 };
38
39 function defendCorral(?x, ?y) {
40   true -> bcm.activateBehaviour(DefendCorral)
41 };
42
43 function herd(?x, ?y, ?h, ?p) {
44   true -> bcm.activateBehaviour(SingleHerd(herd_x, ?x, herd_y, ?y, herders, ?h, position, ?p))
45 };
46
47 function stop{
48   true ->bcm.activateBehaviour(Stop)
49 };
```

Fig. 3. AF-TeleoReactive Code for the Herder agent

– *herd*: the agent starts herding cows at a given set of coordinates
– *openFence*: the agent opens the fence at the given coordinates
– *stop*: the agent stops moving
– *connect*: maintains the agents connection to the server

The *BehaviourControllerModule*, which is an encapsulation of the behavioural sub-system described in section 5, is introduced on line 01 of the Herder agent (Fig. 3) and provides the user with a set of *Sensors* and *Actions*. Actions are Java classes which enable the agent to interact with the environment. An example of such an action being used is shown on line 28.

When programming the herding agents we had intended to place a higher level of logic in the task functions (lines 23-49), which is why we structured them as individual functions rather than simply activating the behaviours in the main function. Unfortunately as the contest date approached we prioritised the optimisation and improvement of the underlying behaviours over this. As an example the explore function (lines 24-27) simply sends an agent to a specific position, we had intended that once the agent reached this point it start iteratively expanding into unexplored areas.

4.2 AF-AgentSpeak

AF-AgentSpeak is based on Jason [9], 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, an example of which is shown in Fig. 4, which are defined by a triggering event, a context and a plan containing a number of actions or events.

```
+triggeringEvent : context <-
   step1,
   step2,
   ...,
   stepn;
```

Fig. 4. Example AF-AgentSpeak plan rule

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.

```
01 module bcm -> agentcontest.core.module.BehaviourControllerModule;
02 module ta -> agentcontest.cac.module.TestAllocatorModule;
03
04 +initialized : mapService(?sid) <-
05    .bind(?sid),
06    .println("configuring..."),
07    ta.setWorldModel(?sid),
08    foreach(herder(?name, ?addr)) {
09        ta.addWorker(?name)
10    };
11
12 +assignment(?agt, ?task, ?params) : herder(?agt, ?addr) <-
13    .send(request, agentID(?agt, ?addr), doTask(?task, ?params));
```

Fig. 5. AF-AgentSpeak Code for the Strategist agent

The *TestAllocatorModule*, which is an encapsulation of the strategy subsystem described in section 5, is introduced on line 02 of the Strategist agent (Fig. 5) and provides the user with a set of *Sensors* and *Actions*. Sensors are Java classes that convert raw sensor data into beliefs and events that are added to the agents belief set and event store respectively. Actions are Java classes which enable the agent to interact with the environment. An example of such an action being used is shown on line 07.

In addition to the *Sensors* and *Actions* provided by the *TestAllocatorModule* a number of in built *Actions* and *Sensors* are provided to all agents which provide basic functionality such as printing, ACL communication and the ability to bind to platform services.

The plan rules of the Strategist agent are quite simple, the entire code of the agent is given in Fig. 5. The first rule (lines 04-10) fires upon initialisation and is responsible for the configuration of the *TestAllocatorModule*, setting up the world map and adding herder agents as workers in the system. The second rule (lines 12 & 13) fires upon the generation of an assignment and utilises the ACL communication system to inform the relevant Herder of it's new task.

5 Agent Team Strategy

The behaviour system used in our hybrid control architecture provides the functional basis upon which the basic capabilities of agents can be implemented and extended. The principal inspiration behind its design is the Vector Field Histogram family of navigation algorithms for mobile robots (VFH/VFH+) [11, 12]. Based on the discrete encoding of behaviour response, VFH+ starts with examining a number of manoeuvres available in the robotic platform. This set of available manoeuvres is then filtered by excluding those leading to collision, based on the information the robot has about its surrounding obstacles. The final control command is then decided upon by availing of a DAMN-like voting coordination mechanism (VFH) [13], with each primitive expressing the cost

they associate with each manoeuvre when voting. The behavioural system itself remains unchanged from that of our previous entry to the competition, for further details of the behavioural subsystem see [2].

As discussed in Section 3, our system is based on a single agent in charge of deliberating the goals for the whole team. The multi agent system relies on a simple master-slave protocol within which the strategist agent distributes, via FIPA ACL, a list of tasks to all the herder agents, which then perform the corresponding action.

The shared world model utilised by the system provides a navigational mechanism through the application of the Dijkstra algorithm [14] which floods the map from the centre of the corral at the start of each simulation step.Using this method the system maintains the shortest path from any point to the center of the corral.

The functions of the strategy sub-system run periodically, where execution is controlled by the shared world model. This execution model was chosen as there were synchronisation problems when it was controlled by the agent, resulting in tasks being allocated based on a partially updated world model. This allowed the configuration of the frequency at which task assessment and allocation occurred, a higher frequency resulting in a more reactive system and a lower frequency providing much more stability in terms of task allocation.

The main types of tasks considered within the strategy sub-system are the herding, exploring, defence, offence and fence opening tasks. Each task is labelled with an identifier, and it is described by:

1. An (x, y) target representing the coordinates in the world map associated with the task
2. The minimum and maximum number of agents required for its successful execution
3. A (benefit, cost) pair used to evaluate the task in the context of the global assignment process.

For further information on the strategy subsystem the user is directed to last years paper in which this was discussed in detail [2].

Apart from several bug fixes and the addition of new behaviours the system as a whole remained mostly unchanged from our previous entry. As such we attribute our improved performance mostly to the inclusion of domain specific knowledge within the behaviours. These took the form of our three main strategic components;

1. Herding behaviours
2. Offensive behaviours
3. Defensive behaviour

5.1 Herding

As this is the most important behaviour we prioritised it's development over all others. Initially we developed a single herding behaviour *Curved herding*, which

attempted to encircle the cows and force them in the direction of our corral, however this algorithm proved less effective on maps with enclosed spaces. As such we developed a second behaviour *Straight line herding*, which utilised the obstructions in the map to funnel the cows towards our corral.

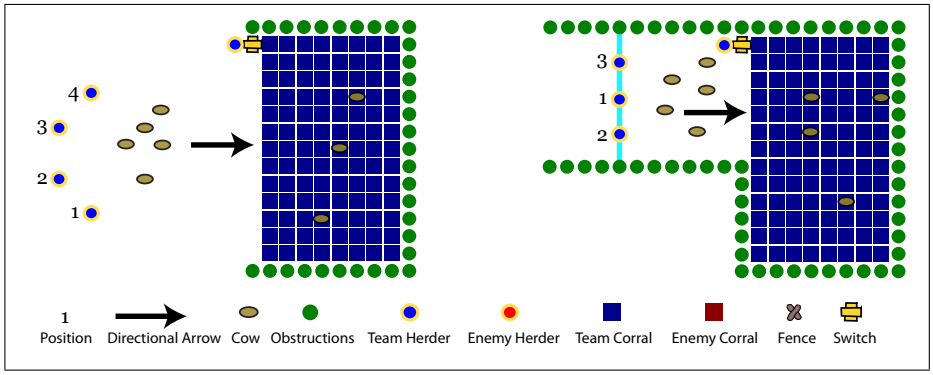

Fig. 6. Example of the herding behaviours

Both herding behaviours shown in Fig. 6 function by passing each of the agents four parameters;

1. X coordinate of the centre of the herd,
2. Y coordinate of the centre of the herd,
3. The number of agents herding, n,
4. The agents position in the team, x, where $\{1 \leq x \leq n\}$.

Curved Herding: Within this behaviour (shown on the left of Fig. 6) each agent uses these parameters to calculate the X and Y coordinates of it's desired position by starting at the centre of the herd and then moving back and to the side based on the agent's position within the team. The positions are encoded in a clockwise manner and the positions are designed to be symmetrical with respect to the directionality of the corral. These positions are taken to force the cows towards the corral whilst keeping them clustered together.

Straight Line Herding: Within this behaviour (shown on the right of Fig. 6) each agent calculates the point on the opposite side of the herd from the corral (taking obstructions etc. into account), then based on the agents position within the team it selects a point on the line perpendicular to the path of the herd to the corral (shown in light blue). The positions in the behaviour are symmetrical with respect to the herds path to the corral, such that for odd numbers the agent in position 1 will be in the centre and the following agents will be alternated from side to side at increasing intervals of distance.

5.2 Offence

Due to the success of offensive behaviours in last years contest we elected to develop an effective offensive strategy to supplement our herding behaviours. With the changes in the contest rules, offensive strategies such as removing all the cows from the enemy corral would no longer be effective. The new scoring system as well as the increase in the numbers of agents implied that a designated team for disruption of the enemy would be the most effective approach.

As the scoring system favoured securing cows into the corral early, it was imperative that the enemy corral was located and attacked as early in the simulation as possible. To facilitate this we utilised our offensive team as explorers initially, exploiting symmetry we located areas on the map likely to be the location the enemy corral and dispatched our agents there. Once the enemy corral was found all the offensive agents converged on upon it.

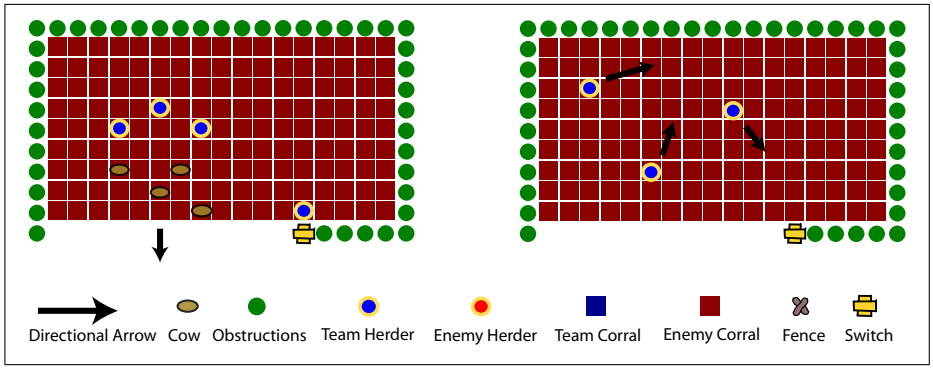

Fig. 7. Example of the offensive behaviours

Once the agents were within the confines of the enemy corral, our team engages in two types of behaviours:

1. Random movement
2. Offensive herding

Random Movement: This strategy (shown on the right of Fig. 7) was employed when the enemy had no cows in the corral, the purpose of the random movement was to discourage cows from entering the corral. A more effective strategy could have been lining the agents across the fence of the corral, however this would not have scaled well to corrals with more than one fence. For this reason we chose to move the agents randomly such that it would provide influence the cows over the whole of the corral.

Offensive Herding: This strategy (shown on the left of Fig. 7) was utilised to remove the enemy cows from the corral by herding them to the area immediately outside the corral. It was decided that having the agents remain within the enemy corral was more effective than having the agents attempt to herd the cows to our corral, as they would have to would have to overcome any possible enemy defences and herd the cows over a large distance. The alternate behaviour however presented itself when the herd was the closest to our corral, whereby another team of agents would be assigned to herd the cows. In the case where the enemy exhibits a strong defensive behaviour, remaining within the corral offsets its effectiveness.

5.3 Defence

The requirement for a defensive strategy was an obvious inference from the success of the offensive strategies in last years contest. However it was through experimentation with an early version of our offensive behaviour that the defensive strategy was conceived, in which a default behaviour of agents assigned to offensive tasks was to position themselves a certain distance from our corral. Through experimentation the unintended elegance of this strategy became apparent.

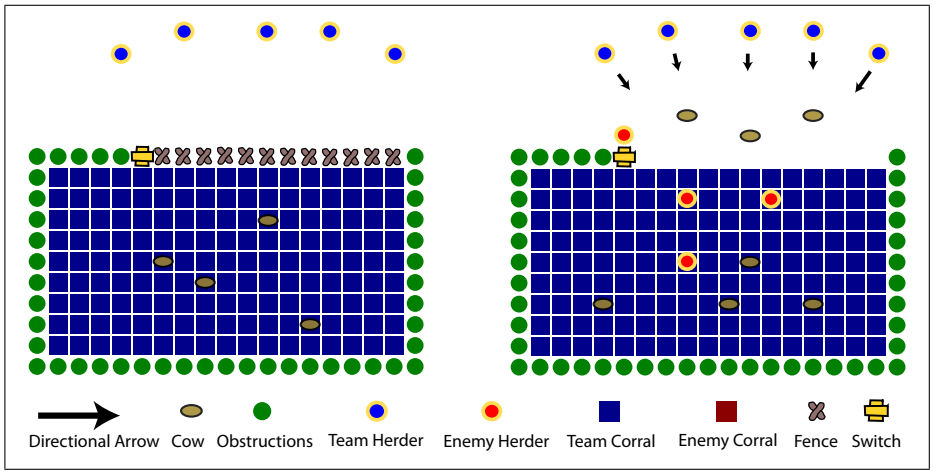

Fig. 8. Example of the defensive behaviours

The default positioning of the agents is shown in the left of Fig. 8 and evolved from a combination of three of the VHF primitives with which they were instructed to:

1. Follow the flood gradient of the shared map.
2. Avoid the vicinity of allied agents.

3. Avoid the vicinity of the corral.

The first primitive brings the agents close to the fence of the corral, the second keeps them spread out and the third stops them from entering the corral and becoming a destructive force.

Defensive Herding: When any cows either leave the corral, through their own default behaviour, or are pushed out by the enemy the defensive team immediately becomes focused on herding the cows back in. This is shown on the right of Fig. 8. This behaviour provided a mechanism to facilitate the retention of successfully herded cows.

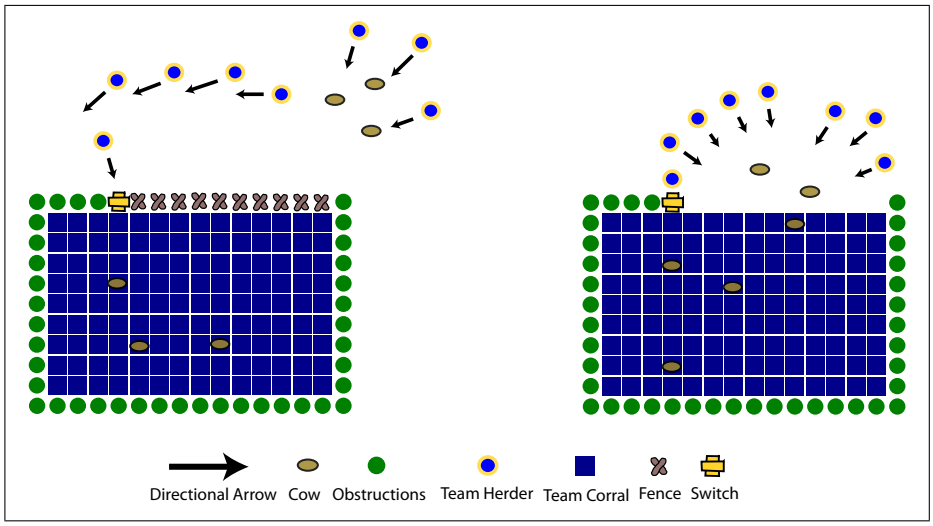

Fig. 9. Example of the assisted behaviours

Herding Assistance: An unanticipated but advantageous consequence of the defensive strategy was an emergent behaviour which we define as *Herding Assistance*. An example of this is shown in Fig. 9, this behaviour is a result of the combination of the ally avoidance primitive and the defensive herding behaviour. Whereby the ally avoidance primitive causes the defensive team to avoid the path of the incoming herding team, when the herd is close to the corral the defensive herding behaviour activates, the defensive team moves to assist the herding team force the cows into the corral.

6 Technical Details

The execution of the agents within the system was performed asynchronously with respect to the simulation, in this way a Herder agent on receipt of a new

task would not have to wait for the next simulation step to instantiate the new behaviour. Thus the allocation of tasks to agents and their instantiation of behaviours could be viewed as background processing. However with respect to the behavioural sub-system no background processing was conducted. This concurrency strategy has the advantage that, except when a task has just been completed, herder agents always have a task assignment and do not waste any deliberative cycles.

The detection of system crashes was handled manually. On the rare occasion of a system crash, our system was restarted on an alternate back up system, this was made possible as the system was deployed on several heterogeneous platforms, see Section 1. This approach ensured a minimal period of disconnection from the contest server. So as to avoid the loss of environmental information, and to avoid incurring the associated cost of re-exploration, the strategist agent periodically saved the static elements of the shared world model (obstacles, fences etc) to disk. If a restart were necessary while a match was in progress, the maps static elements are reloaded to the shared world model. A series of bugs were detected using the Agent Factory [15] debugger and subsequently removed.

Supplementary to the Agent Factory debugger, the team utilised a stripped down version of the contest server, "FastSim", which was optimised for speed in order to rapidly test and debug behaviours. Using this system we developed a number of unit tests to assess the performance of specific behaviours and combinations of behaviours. FastSim was created by combining the behavioural sub-system with the contest server using a set of custom classes to encapsulate the functionality of the agents, the *FastAgent* class, and teams, the *FastTeam* class. As the code was all running on a single machine this negated the need for the costly networking overhead and allowed the simulation to step as fast as the behavioural layer could compute.

Running a simulation of two teams of agents over 1500 steps took approximately 10-20 min depending on the speed of the computer running the simulation. This provided us with a great advantage as simulations run through the contest server could take up to 3 hours to complete.

Our overall system stability was improved over last years entry which incurred frequent crashes. This stability was improved incrementally over the course of the contest. In previous years we noted problems associated with the compartmentalization of development team knowledge as a preceptor of numerous system bugs and behavioural inconsistencies, much of this was overcome by way of our team development approach. System stability was further improved through use of automated unit testing. Additionally it is felt maintaining a high level of system documentation would result in stability and improved developer productivity. By the competitions end we had a stable system, we do not envisage issues of stability being relevant in future.

7 Discussion and Conclusion

The approach taken by the team in the development of the system was that of agile development augmented by team programming sessions. This approach suited the team well and allowed us to make rapid progress, however the speed at which development progressed also caused problems, bugs and undesired behaviour emerged at roughly the same pace. The impact of this was diminished through the application of unit testing, despite this our testing was not comprehensive enough to catch all the bugs and as a consequence system performance, whilst improving over the entire course of the contest often degraded over the short term. An example of this can be seen in Table 1 where the average score per simulation reduced from Day 2 to Day 3, however it can be seen that the average score rose as evidenced by the dramatic difference between the Day 1 and Day 4 results.

Table 1. UCDBogtrotters results

Simulation	Day 1	Day 2	Day 3	Day 4
Circles	1.690	2.199	1.285	3.587
Spirals	1.285	0.000	0.854	3.148
Prairie	0.814	8.258	0.000	7.651
Circles	-	1.284	3.147	14.463
Spirals	-	0.480	0.315	7.058
Prairie	-	4.836	5.939	10.723
Average	1.263	2.843	1.923	7.772

Through participation in the multi agent contest we gained insights into the suitability of languages to different tasks, as an example AF-TeleoReactive was suited to the control of the behavioural layer but was not ideal as a Strategist agent. For this reason we chose to also utilise AF-AgentSpeak as the Strategist agent as it contains more complex structures such as for loops which allowed the iterative application of functions to workers and in our initial design the periodic control of the task allocation functions. Implementing this functionality in AF-TeleoReactive whilst possible would have been counter intuitive.

The only problem faced whilst choosing our approach was in selecting the appropriate languages for control of the agents in the system, which was negated to an extent by the diversity of compatible languages created using the Agent Factory common language framework, detailed in section 4. As the underlying system was well established another problem faced was the introduction of new members to such a complex code base, this problem was an attributing factor in our selection of a team programming approach as it allowed new team members to become immersed in the development process.

Last year's welcome introduction of fences to the herding scenario added the necessity for a new level of inter-agent coordination. This year use of a higher number of fences within the scenario provided a challenging scenario. This was

particularly evident in the challenging "Prairie" scenario. The "Spirals"scenario proved a challenge for our herders, the position of the fences near the boundary and the problem posed by the diverging paths at that point made effective herding difficult. This resulted in a large number of cows moving in the wrong direction and splitting from the herd.

7.1 Extensions

We feel the scoring system used in this years contest is the most fair system thus far and allows more choice of strategy in terms of speed vs size herding strategies. In the implemented model important considerations can be made about the trade-off over spending significant periods herding a large number of cows, or spending minimal time herding several small sets of cows.

This years contest saw the increased effectiveness of offensive strategies, whilst we believe this to be a welcome challenge to be overcome, others felt that it was overly aggressive. We suggest a possible solution to the problem.

- A small number of enemy agents be allowed in the corral (2-3).
- Agents in excess of this threshold could be transported to an undesirable location on the map.
- Alternatively agents could be transported to their own corral.

The alternate solution of agents being transported to their own corral could be used in a positive manner, such as to swiftly return from a large distance to help defend the corral.

Further to this we believe an interesting extension to the aforementioned scenario would be the introduction of tunnels or wormholes which instantly transport agents and cows from one point on the map to another. This could add a new dimension to the scenario by removing the linearity from the environment map. It is envisioned that on the map there would exist a number of coupled way points through which agents and herded cows could travel instantly to the other associated way point. Herders would have to be careful not to push cows through this points if it was not desirable, the converse also being true. For our specific implementation this would have some interesting effects on our world model. This would add an addition element to discovery and shortest path route construction.

References

1. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications. PhD thesis, School of Computer Science and Informatics (2002)
2. Jordan, H., Treanor, J., Lillis, D., Dragone, M., Collier, R., O'Hare, G.: AF-ABLE in the multi agent contest 2009. Annals of Mathematics and Artificial Intelligence 1–21
3. Dragone, M., Lillis, D., Collier, R.W., O'Hare, G.: SoSAA: A Framework for Integrating Components and Agents. SAC '09 (2009)

4. Dragone, M., Lillis, D., Muldoon, C., Tynan, R., Collier, R., O'Hare, G.: Dublin bogtrotters: Agent herders. Programming Multi-Agent Systems (2009) 243–247
5. Cohn, M.: User stories applied: For agile software development. Addison-Wesley Professional (2004)
6. Poslad, S., Buckle, P., Hadingham, R.: 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. (2000) 355–368
7. Russell, S.E.: Teleo-reactive agents. Final Year Thesis, School of Computer Science and Informatics, University College Dublin
8. Nilsson, N.: Teleo-reactive programs for agent control. Arxiv preprint cs/9401101 (1994)
9. Bordini, R., Hubner, J., Vieira, R.: Jason and the Golden Fleece of agent-oriented programming. Multiagent systems artificial societies and simulated organizations **15** (2005) 3
10. Rao, A.: AgentSpeak (L): BDI agents speak out in a logical computable language. Agents Breaking Away (1996) 42–55
11. Borenstein, J., Koren, Y.: The vector field histogram fast obstacle avoidance for mobile robots. IEEE Transactions on Robotics and Automation, 7(3):278.288 (1991)
12. Ulrich, I., Borenstein, J.: Vfh+: Reliable obstacle avoidance for fast mobile robots. In International Conference on Robotics and Automation, pages 15721577, Leuven, Belgium (1998)
13. Rosenblatt, J.K.: Damn: A distributed architecture for mobile navigation. Journal of Experimental and Theoretical Artificial Intelligence 9(2-3): 339360 (1997)
14. Dijkstra, E.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1) (1959) 269–271
15. Collier, R.: Debugging Agents in Agent Factory. Lecture Notes in Computer Science **4411** (2007) 229

A Summary

1.1 *This entry was developed using the multi-agent framework of Agent Factory.*
1.2 *Our primary motivations were building upon our previous entries and involving new researchers in Agent-Orientated development.*
1.3 *Our agent platform was run on several heterogeneous hardware configurations dependent upon the team member monitoring the simulation.*
2.1 *At the outset of team programming sessions an informal requirements analysis and update was conducted, from this goals were broken down into logical sub components for implementation.*
2.2 *There was no formal specification generated, it was felt that the relatively small degree of change in the underlying system negated the need for formal specification.*
2.3 *Our system was not specified or designed using any particular multi-agent system methodology.*
2.4 *Autonomy, role, proactiveness, communication, team-working, and coordination were not explicitly specified.*
2.5 *Our system is a true multi-agent system with centralised coordination.*
3.1 *The system is built on a hybrid control architecture, consisting of a high-level deliberative layer based on the AF-TeleoReactive programming language and a lower level responsible for executing simple behaviours.*
3.2 *No particular methodology was used. The architecture of the system is described in detail in Sections 3-5*
3.3 *Roles are assigned to herder agents by a central strategist agent. Once assigned roles, herder agents are free to take whatever steps they deem necessary to satisfy the goals of their roles.*
4.1 *A hybrid agent architecture, based on the SoSAA robot control architecture was used. The basic abilities of the agents were implemented as behaviours that were deployed in the lower reactive layer, while coordination and behaviour selection was realised through the use of two languages built using the Agent Factory common language framework, AF-TeleoReactive and AF-AgentSpeak.*
4.2 *AF-TeleoReactive is an agent oriented programming language based on Nils Nilsson's Teleo-Reactive agent paradigm, the agents are designed to perform reactively in a dynamic environment whilst still moving towards a goal. AF-AgentSpeak is based on Jason, a purpose-built agent-oriented programming language that implements an extended and improved version of Rao's AgentSpeak(L) language.*
4.3 *answer*
4.4 *answer*
4.5 *answer*
5.1 *Our navigational system is inspired by the Vector Field Histogram family of navigation algorithms for mobile robots (VFH/VFH+). Agents perform shortest path computations to move toward intended locations and push cows to the corral. Herds are found via a simple online clustering of all the known cows.*

5.2 *The team coordination strategy is based on the use of a shared world model and centralised task allocation. Whereby the assignments are distributed by FIPA ACL.*

5.3 *During task allocation agents are assigned tasks based on the distance between them and the objective, allowing the minimisation of distance travelled by agents.*

5.4 *All perception data is shared by every agent through the use of a shared world model, and the Strategist agent disseminates tasks to agents using FIPA ACL.*

5.5 *The communication complexity is very low in our system with each herder agent sending only one message per simulation step and the strategist sending 20 messages when task allocation had occurred. In practice this occurred every 5 - 10 steps.*

5.6 *We believe that the overall strategy could be improved by breaking the agents into several groups, each with it's own strategist and primary function.*

6.1 *In practice no explicit background processing occurred, however the deliberation of the agents in the system was asynchronous to the execution of the simulation*

6.2 *Crash recovery was performed manually. As regards network disconnection, the agent would attempt reconnection autonomously.*

6.3 *By the end of the contest the stability of the system solidified and was quite reliable. The stability could be slightly improved through minor bug fixes in some of the individual behaviours.*

7.1 *Overall the development process was suited to the needs of the team, however it would have benefited from more comprehensive unit testing.*

7.2 *Interesting insights we gained regarding the suitability of different agent oriented programming languages to different tasks.*

7.3 *The only problem faced whilst choosing our approach was in selecting the appropriate languages for control of the agents in the system, which was negated to an extent by the diversity of compatible languages created using the Agent Factory common language framework, detailed in section 4*

7.4 *A mechanism could be put in place to reprimand aggressive corral invasive behaviours, we propose a system by which invasive agents are transported back to their own corral.*