# AF-ABLE in the Multi Agent Contest 2009

Howell Jordan, Jennifer Treanor, David Lillis, Mauro Dragone, Rem W. Collier and G. M. P. O'Hare

School of Computer Science and Informatics
University College Dublin
howell.jordan@lero.ie, {jennifer.treanor,david.lillis, mauro.dragone,rem.collier,gregory.ohare}ucd.ie

**Abstract.** This is the second 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 AFAPL agent programming language 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 2009 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 Java based Agent framework that is both modular and extensible. The overall architecture of the system, which is detailed in Section 3, was loosely based on the robotic control architecture SoSAA [2] in which system components are supervised by agents thereby providing goal-driven capabilities and ease of interoperability through agent communication.

Last year's entry offered the team a chance to build a base system that implemented the higher level searching and herding behaviours, within an overall architecture, consequently the contest aims remained fairly modest. While the team (named "Dublin Bogtrotters") did not feature at the top of the score board by the end of the contest, it did score within a handful of cows of some of the top contenders. This achievement, along with having a base system to build upon, was a strong motivator for a follow-up entry this year[2]. In addition, as evidenced from last year's success at herding through the infamous RazorEdge scenario, the team believed that with some foundation-level bug-fixes, the deliberative structure of the system was extremely promising. Since the system is based on SoSAA, which has a proven track record as a robust autonomous

[1] http://www.multiagentcontest.org/2009

[2] The agent code used by the team can be downloaded from http://www.agentfactory.com/multiagentcontest

and extensible framework, with some extensions and further implementation of SoSAA concepts would potentially make the entry a real contender. Another less competitive motivation was one of education and student integration. The contest and existing system structure lent itself well to introducing new students to agent-based coding using Agent Factory. The hope was that the team could extend the system to a workable solution that can be used as a programming challenge for final year students with the prospect of giving the most successful students the opportunity to integrate their solutions into future contest entries.

Even though the main aim was to improve significantly on last year's entry, more concrete short term goals were set initially. The first objective was to ensure the stability of the base system and improve system performance at the lowest level. The team was then free to explore the new contest requirements, namely new behaviours to tackle fence operation and to safeguard against new, more offensive opponent tactics. With a handle on these issues, members of the team then explored the issues of opponent interaction and higher-level task allocation strategies. Since the contest involves a range of different, unknown scenes the system parameters ideal for one scenario may not hold for another. Indeed, many areas of the system may depend in a non-trivial manner on the one parameter, making setting its value difficult. To this end, tunable parameters were introduced to the system.

Following the initial phase of stabilising the original system, an optimisation phase was introduced. This involved testing the new functionalities with the existing setup and using this testing session to assist in tuning parameters.

For the contest itself, the system ran on a single IBM server, as the processing load did not warrant the incorporation of additional nodes. However, the system was designed so as not to preclude the addition of further machines if necessary.

## 2 System Analysis and Specification

Our requirements process was loosely based on user stories [3], modified for a distributed development environment. This process was intended to built upon the specifications of the 2008 contest entry. User stories are 'lightweight' textual analysis artefacts that describe the high-level functional requirements of a system, while maximising conciseness and flexibility. Stories are ideally written by the end user and are therefore typically independent of technology, data, and algorithms. They are best suited to development contexts in which the full set of requirements is unknown or subject to frequent change.

User stories are not intended to capture non-functional requirements (such as performance and supportability), or the relationships between requirements. Consequently, they often correspond to tasks, but have no direct mappings to the agent-oriented concepts of roles, communication, and co-ordination. For example, consider the following user story:

"If 2 or more agents are assigned to a task that requires them to traverse a fence, the agent nearest to the fence should open it."

This story immediately suggests that 'open fence' should be a subtask of the existing 'explore' (move towards areas of the map that has yet to be visited by a herder) and 'herd' (collaborate with other agents in pushing one or more cows towards the corral) subtasks. However, it does not specify the communication between agents that will be necessary to determine which is closest; or the coordination required to ensure the fence is held open long enough for the other agent(s) to pass through.

We encountered several problems with this approach. Firstly, there was no central repository of stories, such as a project storyboard, to show development status at-a-glance. Thus we did not take full advantage of story-based requirements [4]. Secondly, many stories were expressed as modifications of the previous year's system, for which formal specifications were not created. Thirdly, testing was not considered; it is likely that some of the stories were not testable, and no special allowance was made for the acknowledged difficulties of unit testing multi-agent systems [5]. Finally, research and programming activities were not clearly separated; some stories took weeks or more to implement, thus negating many of the advantages of an agile approach.

No formal specifications for our multi-agent system were written; it was thought the existing Multi Agent Contest 2008 specifications would be adequate, with minor modifications to address the opening of fences and the possibility of 'stealing' an opponent's cows. With hindsight, it would have been worthwhile to specify at least the changes required to operate fences; the coordination issues involved in passing a team of herders through a fence are non-trivial, and would have been better resolved at the specification phase rather than during implementation.

Anticipating that very little specification work needed to be done, we did not adopt any particular multi-agent system methodology. User stories were assigned to the appropriate developers, and each developer modified the system as appropriate to the current story. Unfortunately, due to a lack of incidental communication between developers and the large amount of time taken to implement certain stories, the core system structure was often modified concurrently in incompatible ways, and much effort was wasted. The absence of unit tests contributed to this difficulty.

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 that represented the herders in the contest environment. 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 strategist; 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 mentioned in Section 2, the overall coordination strategy adopted in the AF-ABLE system is a centralised task allocation model. This approach was adopted over the previous approach [6], which employed an auction-based approach to task allocation because it was felt that the previous model had been over-complicated and difficult to debug.

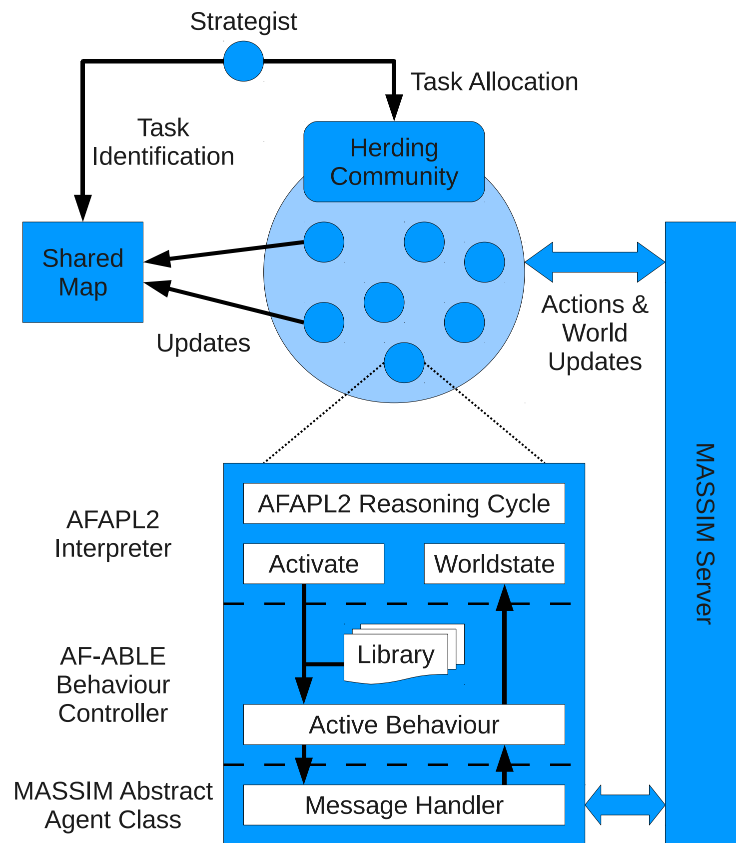


**Fig. 1.** Schematic of the Herder agent architecture

Figure 1 shows the architecture of the system. Central to the system is the group of herder agents that represent the herding entities in the contest environment. Each herder agent is an independent three-tier entity. The base tier is a thin wrapper around the AbstractAgent class that was provided by the contest organisers. This is used to connect to the contest server, parse the perception updates being sent from the server and reply with the agent's next move. Whenever percepts are sent from the server, an agent will record this information in a world model that shared amongst all of the agents. This way, each agent has access to information about the position and surroundings of its teammates.

Above this layer is the behaviour system. This consists of a number of Java classes representing a library of behaviours that are available to the agent. These available behaviours include such tasks as exploration, shortest path movement and obstacle avoidance. At any given point in time, each agent will have one currently active behaviour. This behaviour will calculate the agent's preferred next move, to be sent back to the server. The specific details of the implementation of the behaviour layer are presented below in Section 5.

The high-level layer of the agent is its deliberative layer. Based on the role that has been assigned to it, each agent is responsible for doing its own planning so as to carry out its task. Written in the AFAPL agent programming language, this deliberative layer activates and configures particular behaviours according to a goal-driven reasoning process. Thus the proactivity and autonomy of the individual agents is found in their own abilities to undertake planning so as to satisfy their goals. Although an agent's autonomy is limited in the sense that the role it must play (and consequently its goals) is dictated elsewhere, it is free to operate as it pleases within those bounds. A more detailed description of the AFAPL programming language and how it was used in the implementation of the deliberative layer of the herding agents can be found in Section 4. The upper AFAPL layer monitors the progress of the lower layer in parallel, responding to key events (which update its world state) raised by the behavioural layer that may require the selection and activation of new behaviour. For example, if an agent has been assigned a role whereby it is required to explore the map around a particular location, it would break its task into two simpler behaviours. Firstly, the agent moves along a shortest path towards the centre of the area it is to explore. Once it is close to this point, an event will be raised to indicate that the target is close, triggering the activation of a specific exploration behaviour.

The behaviour 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) and so deliberation time is not constrained by the need to select an action in an allotted time frame.

Other than the herder agents, one other agent is present within the system. A strategist agent is responsible for assigning roles to agents, so as to maximise the success of the entire team. It bases its decisions on the shared world model that the herder agents maintain, to which it also has access. It communicates its decisions to the agents, causing them to take on alternative roles and corresponding goals. Further details of how the strategy of the system is decided upon

is discussed in detail in Section 5.2. Finally, the world model is also monitored by a visualisation tool, which allows developers to watch matches in real-time, seeing the same view of the world as the agents.

With regard to communication, two forms of communication were utilised in the architecture. The assignment of roles by the strategist agent takes the form of explicit communication by means of FIPA Agent Communication Language (ACL) messages. In addition, the shared world model represents a form of implicit communication between agents where each records its percepts about the state of its environment for others to use.

The coordination of teamwork amongst the herding agents takes two forms. Firstly, by using a centralised task allocation system, the strategist agent can ensure that agents multiple agents are not engaged in the same task. The exception to this is for tasks that require multiple agents for successful completion, for example herding. The herding task demonstrates the second form of teamwork. By monitoring the position of agents in its team (by way of the shared world model), each agent plans its own specific movements while taking its teammates into account. In this way, agents are aware of one another's movements without resorting to expensive, time-consuming ACL communication.

## 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 [7]. 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 Agent Factory Agent Programming Language (AFAPL) [8] Development Kit, which provides support for the fabrication of agents based on the AFAPL agent-oriented programming language. This kit consists of a purpose-built interpreter, a plugin for the NetBeans IDE and a custom set of views for the AF Debugger that allow the developer to inspect the internal state of AFAPL agents. A library of partial AFAPL programs is also provided to simplify the task of implementing an AFAPL agent.

In practice, AFAPL shares characteristics with Agent-0 [9], in that the key mental attitudes underpinning its mental state are *Beliefs* (an agent's view of the current state of its environment) and *Commitments* (the activities that the agent has chosen to perform).

These are accompanied by the notion of a *Commitment Rule*, which encodes situations in which the agent should commit to a specific activity; together with

a set of labelled plans and primitive actions that jointly represent the potential activities that an agent may commit to. Primitive actions are implemented as Java classes, known as *Actuators*. Preconditions and postconditions are specified as part of the declaration of each plan or action.

Information about the current state of the environment is gathered via a set of *Perceptors*: 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 AFAPL 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, AFAPL introduces the notion of a *Temporal Belief*, which is described elsewhere in [10]. 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.

One of the key motivations for the use of commitments rather than the more standard plans-as-intentions approach is that it allows for a notion of commitment strength (i.e. how committed the agent is to an activity) that is motivated by the notion of blind, single-minded, and open-minded commitment strategies [11]. AFAPL achieves this by including a maintenance condition as part of every commitment. For blind commitment, this condition is *BELIEF(true)*, while for single-mindedness, the condition would encode the situation(s) in which the agent should view the commitment as being unachievable. Support for open-mindedness has recently been realised through the introduction of explicit *goals*.

Explicit goals are supported in AFAPL through the extension of the mental state to include a goal attitude that models future states of the environment (i.e. future beliefs) that the agent should attempt to realise. That is, if the agent does not have a belief that corresponds to a goal that it should attempt to satisfy that goal by committing to an activity that it believes will bring about a belief that corresponds to the goal. Potential activities are identified by matching the postconditions of actions and plans (actions are preferred to plans). The agent then selects an action or plan based on the first one whose preconditions are satisfied and then commits itself to the selected action or plan.

Currently, AFAPL supports two types of goal. *Achievement* goals are goals that are dropped once they are either satisfied or are considered to be unachievable, whereas *Maintenance goals* are goals that are maintained even when they have been achieved. The agent attempts to achieve the goal whenever it believes that the goal is not satisfied.

As can be seen in the sample snippet of Herder agents AFAPL code shown in Figure 2, the AF-ABLE system uses only achievement goals. Specifically, as can be seen in the commitment rule (lines 26-32), the agent adopts an achievement goal in response to receiving a task assignment request from the Leader agent (actually, we assume naively that the agent sending the message is the Leader agent). In the event that the agent already has a goal to achieve a previously

```
01 // Import core herder code. This includes actions and perceptors for interacting with the
02 // behaviour controller & the module that interfaces the agent and the behaviour controller.
03 IMPORT agentcontest.core.BasicHerder;
04
05 // Import the core agent program (this provides support for FIPA-ACL based communication)
06 IMPORT com.agentfactory.afapl2.core.agent.BasicAgent;
07
08 // Declare a perceptor that gathers beliefs about the current state of the world (based on
09 // the shared world model)
10 PERCEPTOR worldinfo {
11     USES promas;
12     CLASS agentcontest.cac.perceptor.WorldInfo;
13 }
14
15 // Install some basic behaviours...
16 COMMIT( ?self, ?now, BELIEF(true),
17     PAR(
18         addBehaviour(massim.af.behaviours.BehaviourStop),
19         addBehaviour(massim.af.behaviours.BehaviourExplore),
20         addBehaviour(massim.af.behaviours.BehaviourMoveToViaShortestPath),
21         addBehaviour(massim.af.behaviours.BehaviourMoveTowardHerdingPosition),
22         addBehaviour(massim.af.behaviours.BehaviourOpenFence)
23     )
24 );
25 // If you get assigned a new task, drop the existing goal task and adopt the new one
26 BELIEF(message(request, ?sender, doTask(?task, ?params))) =>
27 COMMIT(?self, ?now, BELIEF(true),
28     SEQ(FOREACH(GOAL(completed(?oldTask, ?oldParams)),
29             RETRACT(GOAL(completed(?oldTask, ?oldParams)))),
30         ADOPT(GOAL(completed(?task, ?params)))
31     )
32 );
33
34 PLAN exploreArea(?x, ?y) {
35     PRECONDITION BELIEF(true);
36     POSTCONDITION BELIEF(completed(Explore, params(?x, ?y)));
37
38     BODY
39         PAR(activateBehaviour(MoveToViaShortestPath(x, ?x, y, ?y, tolerance, 5)),
40             DO_WHEN(BELIEF(target(close)),
41                 activateBehaviour(Explore)
42             )
43         );
44 }
45
46 PLAN singleHerd(?x, ?y) {
47     PRECONDITION BELIEF(true);
48     POSTCONDITION BELIEF(completed(Herd, params(?x, ?y)));
49
50     BODY
51         activateBehaviour(MoveTowardHerdingPosition(herd_x, ?x, herd_y, ?y));
52 }
53
54 PLAN stop {
55     PRECONDITION BELIEF(true);
56     POSTCONDITION BELIEF(completed(Stop, ?params));
57
58     BODY activateBehaviour(Stop);
59 }
60
61 PLAN openFence(?x, ?y) {
62     PRECONDITION BELIEF(true);
63     POSTCONDITION BELIEF(completed(OpenFence, params(?x, ?y)));
64
65     BODY
66         PAR(activateBehaviour(OpenFence(x, ?x, y, ?y)),
67              AWAIT(BELIEF(competition(finished)))
68         );
69 }
```

**Fig. 2.** AFAPL Code for the Herder agent

assigned task, lines 28-29 of the plan cause the agent to drop the outstanding goal prior to the adoption of the new goal on line 30.

The goal itself is satisfied by the agent committing itself to one of the four plans that are specified below:

- *exploreArea*: the agent explores the region of space around the given area
- *singleHerd*: the agent starts herding cows at a given set of coordinates
- *stop*: the agent stops moving
- *openFence*: the agent opens the fence at the given coordinates

The plan adopted by the agent depends of the specific task that the agent has been assigned to perform.

For further details on the current incarnation of Agent Factory and the non-goal based version of the AFAPL language, the reader is directed to [10]. Also, a discussion on the evolution of Agent Factory since its inception in the early 1990s can be found in [12].

## 5 Agent Team Strategy

### 5.1 The Behavioural Sub-System

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+) [13, 14]. 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) [15], with each primitive expressing the cost they associate with each manoeuvre when voting. The manoeuvre with minimum global cost is then selected as the final control command.

Cooperative behaviour-coordination mechanisms in general are a way of defining different global behaviours that concurrently satisfy multiple objectives. This satisfies the extensibility requirement for our behaviour system as new behaviours can be synthesised by acting either on the weights of pre-existing primitives, or by adding and/or removing behaviour primitives.

However, VFH+ can be better described in terms of a mixed (competitive/cooperative) behaviour-coordination mechanism, in particular one where the first stage simplifies the successive selection by removing competitive dynamics with the obstacle avoidance, as this has already acted by filtering some of the available commands. Such a mechanism is fundamental for avoiding the stuck-in-the-middle situations so common with obstacle avoidance algorithms based on the potential field method [16].

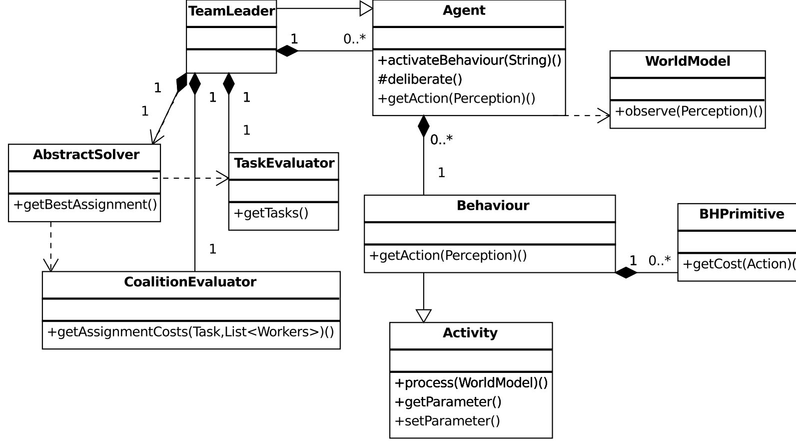


**Fig. 3.** Class Diagram of the AF-ABLE Behaviour System

To serve as template for the skills of multiple agents, our implementation (see class diagram in Figure 3) breaks down the VFH+ algorithm with a number of components, respectively: (i) a number of *BHPrimitive* activity classes encapsulating each behaviour primitive, (ii) a *BehaviourVFH* class that is responsible for the coordination of the BHPrimitive activities and (iii) a number of specialisations of BehaviourVFH, each defining and configuring the set of primitives employed in a different agent's skill.

Additionally, our implementation generalizes the VFH+ algorithm by allowing any primitive to eliminate (veto) some manoeuvres from the final selection. This is in contrast to VFH+ itself, where only obstacle avoidance may veto manoeuvres. In this manner, obstacle avoidance becomes just one of the primitives, while other primitives may take care of other control's properties by removing further compromises when they judge a particular manoeuvre to be contrary to the constraints or objectives they represent.

Each primitive must state its estimated cost for each of the directions allowed by the agent in its current position, or associate an infinite cost ($\infty$) to manoeuvres to be excluded (vetoed) from the final selection. The resulting action-selection function takes the following form:

$$min_j \sum_{i=1,\ldots,N} c^i(a_j, S^i(t))w^i) \text{ where } c^i < \infty$$

Here, $N$ is the total number of primitives and $\{a_j = (a_{jx}, a_{jy}),\ a_{jx}, a_{jy} \in \{-1, 0, +1\}, j = 1, \ldots, M\}$ is the set of actions available to the agent at any given time, such as that an agent in position $p(t) = (x, y)$ will be in $p(t+1|a_j) = (x + a_{jx}, y + a_{jy})$ after executing action $a_j$ (i.e. $a_1$ = North = (0, 1), $a_2$ = North East = (1, 1)). $S^i(t)$ represents the information (the combination of state and sensor data at time $t$ (as stored in the WorldModel class) used by the $i$-th behaviour primitive (i.e. the position of the known obstacles, enemies, allies,

```
public class BehaviourMoveToViaShortestPath extends BehaviourVFH {

    BHPrimitiveFollowFloodGradient primitiveGradient = null;
    BHPrimitiveRandom primitiveRandom = null;

    public BehaviourMoveToViaShortestPath() {
        installPrimitive(new BHPrimitiveRandom(0.05));

        // declare all the parameters of this behaviour primitive activity
        // with associated default values
        addParameter("minDistanceFromCows", "minimum distance allowed from cows", "3");
        addParameter("minDistanceFromAlly", "minimum distance allowed from ally", "3");
        addParameter("minDistanceFromEnemy", "minimum distance allowed from enemy", "2");
        addParameter("x", "target x", "0");
        addParameter("y", "target y", "0");
    }

    public void start() {
        installPrimitive(new BHPrimitiveFollowFloodGradient(50.0,
                Integer.parseInt(getParameterValue("x")),
                Integer.parseInt(getParameterValue("y")));

        installPrimitive(new BHPrimitiveAvoidVicinity("AvoidVicinityCow", 40,
                    Observation.TYPE_COW,
                    Integer.parseInt(getParameterValue("minDistanceFromCows"))));

        installPrimitive(new BHPrimitiveExcludeVicinity(Observation.TYPE_ALLY, 0));

        super.start();
    }
```

**Fig. 4.** Part of the Java code for the BehaviourMoveToViaShortestPath behaviour

corrals, etc.), and $w^i$ is the weight assigned to the $i$-th primitive. In addition to $\infty$, the cost $c^i$ takes a value in the range $[0, \ldots, 1]$, where 0 means strong approval of the manoeuvre and 1 means strong disapproval.

This simple voting mechanism is performed by the BehaviourVFH class of the behaviour currently active in the behaviour controller of each agent. This is done every time the server requests an action from it. In order to illustrate this process in more detail, Figure 4 shows part of the Java code of the *BehaviourMoveToViaShortestPath*, which is the specialisation of BehaviourVFH used to drive an agent toward a target position by loosely following the shortest path.

The constructor of BehaviourMoveToViaShortestPath installs only the *BHPrimitiveRandom* in the generalized VFH algorithm. This primitive's only responsibility is to add noise to the voting mechanism by associating random costs to each manoeuvre. This serves the purpose of avoiding situations where agents are stuck in unforeseen situations, e.g. trying to move to a position occupied by an enemy, without having to explicitly account for such a situation.

All other primitives are instantiated once the behaviour is activated (by calling its *start* method). Firstly, the *BehaviourMoveToViaShortestPath* behaviour class uses the *BHPrimitiveFollowFloodGradient* primitive to favour manoeuvres leading to the intended target. Secondly, two primitives, namely *BHPrimitiveAv-*

*oidVicinity* and *BHPrimitiveExcludeVicinity*, are used to avoid disturbing cows encountered along the way by respectively: (i) avoiding getting too close and (ii) vetoing positions currently occupied by cows that are close to the agent.

Figure 5 shows a snapshot from the AF-ABLE visualisation tool, which we implemented to monitor the operations of our agents. Figure 6 shows the same scenario after the map has been flooded with the Dijkstra algorithm from the centre of the corral (bottom left).

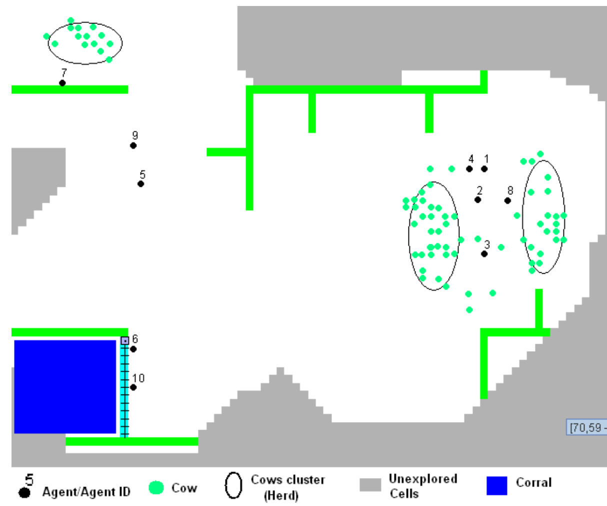


**Fig. 5.** Snapshot from the AF-ABLE visualisation tool

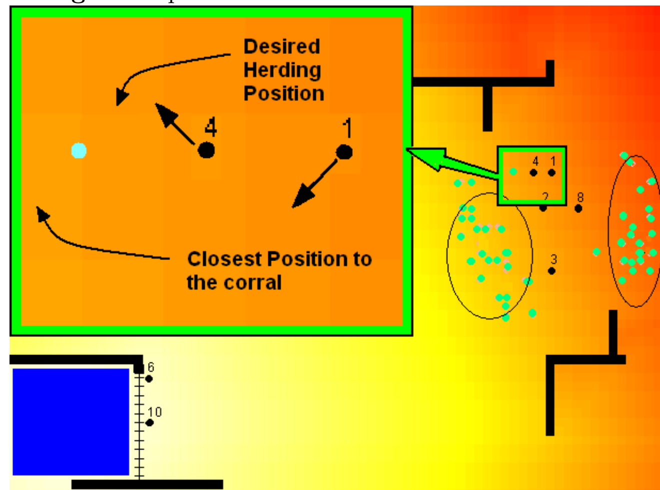


**Fig. 6.** The field flooded from the corral and details of the action selection process of two agents (top left frame)

This step is performed by the BehaviourMoveToViaShortestPath class, and repeated at each step to reflect new agent's perceptions. Cells in Figure 6 are coloured with darker shades as they get further away from the target.

The top-left frame in the same figure magnifies the situation of two agents: agent 1 is moving toward the corral while agent 4 is herding the cow on his left by trying to position itself behind the cow (in blue) in the opposite direction of the map's gradient, as indicated by the black arrows reporting the winner of the voting process used to select the next action of the agent. In this case, for each actions $a_j$, the *BHPrimitiveFollowFloodGradient*'s associates the following vote:

$$c^{flood}(a_j) = flood(p(t+1|a_j)) - min_j(flood(p(t+1|a_j)))$$

where $flood(p)$ is the shortest distance from $p$ to the target. The BHPrimitiveAvoidVicinity's vote is computed as:

$$c^{\text{avoidvicinity}} = \begin{cases} 0 & \text{if min } D < Threshold; \\ 1 - \frac{(minD - D(a_j))}{D(a_j)} & \text{else.} \end{cases}$$

where $D(a_j)$ is the distance from the closest cow to the next expected agent's position $p(t+1|a_j)$, $minD = minD(a_j)$, and $Threshold$ is the distance above which cows are not considered irrelevant by the primitive.

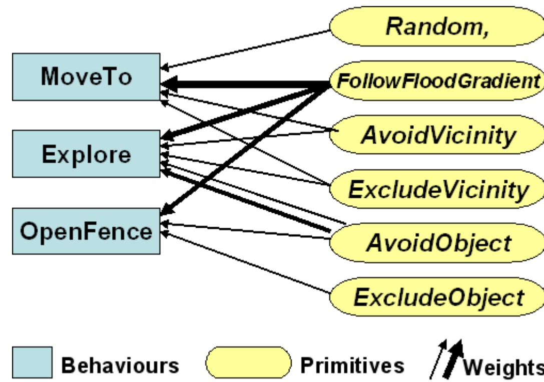


**Fig. 7.** Some of the behaviours implemented for the contest showing their weight relationship with their constituent behaviour primitives

### 5.2 The Strategy Sub-System

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 carry out their own plans to achieve the goals corresponding to the given tasks.

The activity of the strategist agent runs asynchronously with the sensing-acting cycles of the herder agents. On the contrary, its actual rate can be configured in order to avoid having to repeat costly computation every time a message is received from the server and also to lend a sense of stability to the behaviour of the collective by avoiding changing the tasks assigned to agents too often.

All the activities of the strategist agent are organised in a strategy sub-system. This is designed in a modular manner with the aim of supporting the implementation and the testing of different team strategies. Figure 3 depicts the organisation of the strategy sub-system, in terms of both its abstract (application-independent) classes and the specific implementations used during the multi agent contest. The main class, the *StrategySolver* class, is a concrete specification of the *AbstractSolver* shown in the diagram. It collaborates with two auxiliary classes: the *TaskEvaluator*, and the *CoalitionEvaluator* classes. The TaskEvaluator has the responsibility of examining the WorldModel describing the global situation of the system and producing a list of tasks with an associated (benefit, cost) pair. Benefits and costs are considered from a global perspective, by ignoring which agent will actually contribute to the process of completing the tasks. Additionally, the tasks are initially generated without considering system's constraints, i.e. in terms of number of agents available to the system.

The responsibility of considering the situation of individual agents is given to the CoalitionEvaluator, which can be queried to estimate how suitable a given group of agents (coalition) is to achieve a given task. Finally, the role of the StrategySolver is to use the TaskEvaluator and the CoalitionEvaluator to search the task/coalition space in order to find a suitable task assignment for each agent in the collective. In doing so, the StrategySolver needs to consider both resource constraints and eventual causal dependency between tasks.

The main types of tasks considered within the strategy sub-systems are the herding, exploring, and fence opening tasks. Each task is identified by an identifier, and it is described by: (i) an $(x, y)$ target representing the coordinates in the world map associated with the task, (ii) the minimum and maximum numbers of agents required for its successful execution, and (iii) a (benefit, cost) pair used to evaluate the task in the context of the global assignment process.

Herding tasks are found by the TaskEvaluator through a simple and fast online clustering algorithm, with which the cows recorded in the WorldModel are cumulatively assigned into groups (herds). Specifically, each cow is assigned to the closest existing herd if its centre of gravity falls within a given Euclidean distance from the cow ($< 8$ in the configuration used during the contest). Alternatively, a new cluster is created with the cow being examined by the algorithm. Such an algorithm is performed at every sensing-acting cycle to return the list of herds grouping all the known cows.

Exploration tasks are found by simply splitting the game field in a given number of rows and column ($5 \times 5$ in the contest's configuration), and by assuming the centre of the cells in the resulting grid as the exploration target.

Finally, the TaskEvaluator associates a fence opening task to every known fence. The coordinates of these tasks are set to the coordinates of the switch controlling the fence, or the mid-point of the fence, if the switch has not been explored yet by the agent collective. If dispatched to a fence's mid-point, an agent will seek the switch itself once it has arrived.

For all types of task, the TaskEvaluator class uses simple heuristics to estimate the benefit and the shared cost associated to any given task. In this phase, only shared costs are considered, that is, costs measuring the effort required to the collective for executing the task independently from the specific agents contributing to its execution. In order to simplify the combination of different heuristics across different types of tasks, benefits are always expressed in terms of game score (points), while costs are always expressed in terms of numbers of iterations required to complete the task.

All the information collected in the shared WorldModel is harnessed to compute these heuristics. For instance, the estimated benefit of a herding task is proportional to its size while to estimate its shared cost, the TaskEvaluator class considers the shortest route from the centre of gravity of the herd to the centre of the corral. Noticeably, in doing so, the TaskEvaluator floods the map of the field from the centre of the corral by considering free from obstacles any unknown (previously unexplored) cell. However, in order to account for the risk of discovering unforeseen obstacles on route, the cost of traversing these unexplored cells is augmented (by 50% in the configuration used during the contest).

The same method is used to estimate the costs associated with an exploration task, while the estimation of the associated benefits are computed indirectly, by estimating the number of cows likely to be discovered in all the unexplored cells that belong to the area centred at the coordinates of the exploration task. To this end, the TaskEvaluator considers the number of all the cows ever observed in the WorldModel together with the ones known to be already inside the corral.

For obvious similarities with the contest scenario, we looked at methods developed in the robotics community for the algorithm used by the StrategySolver to decide which task to assign to each of the available agents in the team.

In distributed robotics, solutions to the multi agent dynamic task assignment problems [17] consider an environment populated with both robots and tasks/objectives. These are usually associated with a location that a group of robots should explore, or a box that a group of robots should lift. In all cases, the common objective that defines the task requires collaboration of the group of agents that are assigned to it.

The key to solving this type of problem is to account for changes in the environment caused by varying resources and objectives, and to support time extended scheduling, by respecting domain constraints in terms of the number of tasks a single agent can perform at any given time, and the type or number of agents that need to collaborate to specific type of tasks. Robotics solutions usually aim to enhance the system's performance, typically by reducing the overall execution time in order to minimize a cost associated with the total distance travelled by the agents [17–19], and by continuously adjusting the robot

task assignment depending on changes in the task environment or group performance [20–22]. However, the vast majority of the experiments conducted using these algorithms also used relatively simple problems, and often very few agents. In most cases, this valuable research was applied with specialised algorithms that are not suited to solving the entire space of problems without some modification. Finally, none of the work surveyed confronts the dynamic task assignment in competitive scenarios similar to the one presented in the multi agent contest.

The centralised architecture of the strategy sub-system adopted in this year's entry was motivated by the need to ease the testing of different assignment strategies, independently of inter-agent communication issues. The actual strategy used during the contest was inspired by the family of distributed market-based coordination algorithms surveyed in [23], similar to what we used in our previous entry to the contest.

Team: 1

| Agent | Task | Ass. No. | Cost | Value | X | Y |
|---|---|---|---|---|---|---|
| 1 | Herd [49,34] | 51 | 352.8 | 8600.0 | 49 | 34 |
| 2 | Herd [69,28] | 51 | 415.2 | 7100.0 | 69 | 28 |
| 3 | Herd [66,41] | 51 | 304.2 | 9900.0 | 66 | 41 |
| 4 | Herd [66,41] | 51 | 304.2 | 9900.0 | 66 | 41 |
| 5 | Herd [69,28] | 51 | 415.2 | 7100.0 | 69 | 28 |
| 6 | Open Fence [... | 51 | 0.0 | 1.797693134... | 14 | 51 |
| 7 | Herd [69,28] | 51 | 415.2 | 7100.0 | 69 | 28 |
| 8 | Herd [49,34] | 51 | 352.8 | 8600.0 | 49 | 34 |
| 9 | Herd [66,41] | 51 | 304.2 | 9900.0 | 66 | 41 |
| 10 | Explore [100,... | 51 | 401.0 | 395.5 | 100 | 33 |

| Task | Cost | Value | X | Y |
|---|---|---|---|---|
| Open Fence [14,51] | 0.0 | 1.7976931348623... | 14 | 51 |
| Herd [66,41] | 304.2 | 9900.0 | 66 | 41 |
| Herd [49,34] | 352.8 | 8600.0 | 49 | 34 |
| Herd [69,28] | 415.2 | 7100.0 | 69 | 28 |
| Herd [55,14] | 594.0 | 3900.0 | 55 | 14 |
| Explore [100,33] | 401.0 | 395.5 | 100 | 33 |
| Explore [60,55] | 361.0 | 310.3 | 60 | 55 |
| Explore [100,55] | 401.0 | 275.2 | 100 | 55 |
| Explore [100,11] | 569.0 | 235.9 | 100 | 11 |

**Fig. 8.** Task and assignment tables in the AF-ABLE visualisation tool

In the distributed, multi agent version of our market-based algorithm, each task is offered in auction by an agent auctioneer. For each task, each agent responds with a bid that reflects its own estimation of the costs it would incur if chosen to perform the task. The winner(s) of the auction is (are) determined by the auctioneer through a winner-selection strategy, before being assigned to the task. Finally, the process is repeated until all the tasks have been assigned or there are no more available agents. In this year's centralised version, the StrategySolver performs the role of auctioneer, by trying to assign all the tasks

in descendent order of the tasks' value estimated by the TaskEvaluator class, while agents' bids are computed by the CoalitionEvaluator.

Agent's bids are inversely proportional to the number of iterations they estimate necessary to start contributing to the task being auctioned (as computed by the CoalitionEvaluator), which is roughly the length of the shortest route to the $(x, y)$ coordinates associated with the task.

Due to time constraints, this year's contest was run with a greedy winner-selection policy, which the StrategySolver used to simply assign each task to the highest bidder. If the type of the task (e.g. herding multiple cows) requires more agents, the StrategySolver then considers the other bidders, until at least the minimum number of agents are assigned to the task or there are no more available agents. In the latter case, the task is simply cancelled. In addition, in this year's entry the StrategySolver did not consider tasks dependencies, for instance, between main type of tasks (herding, exploring) and opening fence tasks. Fortunately, the actual scenarios proposed during the contest were not very challenging in this regard, as the number of fences was limited. As such, even if non optimal, in the majority of the cases it was enough to give a greater priority to opening fence tasks to assign these tasks ahead of the main exploring and herding tasks which depended from them.

The team strategy could be improved in many ways, for example:

1. by improving the heuristics used to estimate the benefits and costs associated to every task;
2. by taking in account the costs to be incurred in opening fences positioned between the current location of the agents and the $(x, y)$ coordinates associated to the task, or between the location of the herd and the corral;
3. by performing a more optimal search of the task/coalition space in the StrategySolver class.

## 6 Technical Details

Our herder agents did no background processing, and the allowed time-slices were more than adequate for their lower-level activities. However, the Strategist agent was free to deliberate between cycles, and to asynchronously assign herder agents to new tasks. 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 perception-action cycles.

The crash-detection system developed for Multi Agent Contest 2008 was felt to be overly complex, and was removed for this year's competition. Instead, crashes were detected manually, and in each case the entire system was simply restarted. To avoid time-consuming re-exploration in the event of a crash, the Strategist agent regularly saves the static elements of the world model (obstacle locations, etc.) to disk. If a restart then occurs while a match is in progress, the known static elements of the map are reloaded into the shared world model.

Several bugs were found using the Agent Factory debugger [24] and removed. However, the overall stability of our final system was poor, and crashes occurred

frequently; it was felt that this lack of stability prevented many of our system's more advanced features from being properly exercised. The stability could almost certainly have been improved by devoting more attention to general software engineering issues, for example those outlined in Section 2. Due in part to the lack of an automated test suite, we found the stability problem to be self-replicating; a rush of quick fixes introduced as the contest deadline was approaching only led to greater compartmentalisation of development team knowledge, and the introduction of more bugs.

## 7 Discussion and Conclusion

As noted in Section 2, difficulties were experienced with the lack of formal specifications and unit tests for the system. We believe that the experiences gained in participating in this contest emphasise the need for multi agent methodologies, testing and development tools. Despite the acknowledged difficulties in specifying unit tests for all the functionality of a multi agent system, certainly some aspects of the system would be appropriate for unit testing. For instance, the low-level behaviours should individually be predictable and reliable.

The additional complexity associated with a multi agent system makes it unusually difficult for new developers to join an existing project. In addition to an improved development process mentioned above (which would result in clearer specifications of the system's functionality), we believe that there is a clear need for tools to auto-generate documentation from agent code, especially diagrams to describe the system's architecture. Such tools exist for other programming paradigms (especially Object Oriented Programming) but, in general, tool support is comparatively lacking in the multi agent domain.

A further insight is that not all styles of agent programming are suitable for every scenario. For example, when using a hybrid architecture such as ours, it is important for the lower level to be able to trigger events that are available to the deliberative mechanism of the agents. AFAPL typically uses pull-style perceptors (that are fired by the scheduler to fetch percepts) rather than providing native support for event queues. While this approach is useful for a variety of domains, it was necessary to implement a custom event queue handling mechanism for the purposes of this contest. It is important that creators of agent programming languages be aware of types of agent architectures that they themselves may not be immediately familiar with when supporting particular architecture styles.

The process of designing and developing the hybrid control architecture brings up an interesting problem that has yet to be fully addressed. Much recent attention has been focussed on programming a high-level deliberative process in an agent programming language while abstracting lower-level actions and environmental issues to a lower level [2, 25]. However, although such approaches are gaining a following, no clear guidelines exist as to where the line between these two levels should be drawn. In our system, the aggregation of primitive behaviours into more complex ones was left to the Java-based behaviour layer, with the deliberative layer being responsible for the selection of appropriate be-

haviours from those available. However, it could also be argued that a more dynamic assembly of these hybrid behaviours would be more suited to the intelligent, deliberative layer of the agent. This type of separation can still be considered an open question.

The use of fences in the 2009 contest introduced an extra dimension of coordination which was a welcome addition to the scenario. The ability to 'steal' cows from opponents (due to the cows no longer disappearing when herded to a corral) was also an interesting change from the 2008 contest. However, as the final score of a match is judged solely on the number of cows retained at the exact moment of the end of the scenario, it can be argued that too much emphasis has been placed on well-timed destructive behaviour to the detriment of the successful coordination of a herding effort. A team may display good understanding of the scenario and have a successful implementation (of herding, fence opening, exploring, etc.), yet be undone by another that simply drive cows out of the enemy corral towards the end of the match. We believe that the scoring system should be revisited, while adding some suggestions:

1. Revert to the scoring from 2008 where cows are counted towards the team's final score once they enter the corral. Disruptive behaviour is still rewarded, but only continues while the cows are in the field.
2. Measure the number of cows in each corral at various intervals throughout the contest, rather than just at the end of the scenario. This rewards teams who are quicker to gather their cows and also those who maintain a high number of cows in their corral throughout.
3. Record an additional metric (while still scoring the contest in the same way) purely for academic purposes. In the event of the scenario not changing, the number of unique cows herded at any point during the match would be one possibility for such a measure.

## 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. Dragone, M., Lillis, D., Collier, R.W., O'Hare, G.: SoSAA: A Framework for Integrating Components and Agents. SAC '09 (2009)
3. Cohn, M.: User stories applied: For agile software development. Addison-Wesley Professional (2004)
4. Beck, K., Andres, C.: Extreme programming explained: embrace change. Addison-Wesley Professional (2004)
5. Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: Unit testing in multi-agent systems using mock agents and aspects. In: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems, ACM (2006) 90
6. Dragone, M., Lillis, D., Muldoon, C., Tynan, R., Collier, R.W., O'Hare, G.: Dublin Bogtrotters: Agent Herders. In Post-Proceedings of the Sixth International Workshop on Programming Multi-Agent Systems, ProMAS (2008)

7. 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
8. Ross, R., Collier, R.W., O'Hare, G.: Af-apl: Bridging principles & practices in agent oriented languages. Programming Multi-Agent Systems, Lecture Notes in Computer Science (LNAI) **3346** (2004)
9. Shoham, Y.: Agent-oriented programming. Artificial intelligence **60**(1) (1993) 51–92
10. Collier, R.W., O'Hare, G.: Modeling and programming with commitment rules in agent factory. Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approachess (Giurca, Gasevic, and Taveter eds), IGI Publishing (2009)
11. Rao, A., Georgeff, M.: An abstract architecture for rational agents. In: Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92). (1992) 439–449
12. Muldoon, C., O'Hare, G., Collier, R., O'Grady, M.: Towards Pervasive Intelligence: Reflections on the Evolution of the Agent Factory Framework. Multi-agent Tools: Languages, Platforms and Applications (2009) 187
13. 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)
14. Ulrich, I., Borenstein, J.: Vfh+: Reliable obstacle avoidance for fast mobile robots. In International Conference on Robotics and Automation, pages 15721577, Leuven, Belgium (1998)
15. Rosenblatt, J.K.: Damn: A distributed architecture for mobile navigation. Journal of Experimental and Theoretical Artificial Intelligence 9(2-3): 339360 (1997)
16. Koren, Y., Borenstein, J.: Potential fields methods and their inherent limitations for mobile robot navigation. In Proceedings of the IEEE International Conference on Robotics and Automation, pages 1398.1404, Sacramento, CA. (1991)
17. Lerman, K., Jones, C., Galstyan, A., Mataric, M.J.: Analysis of dynamic task allocation in multi-robot systems. The Int. Journal of Robotics Research (2006) 225–242
18. Zavlanos, M.M., Pappas, G.J.: Dynamic assignment in distributed motion planning with local coordination. IEEE Transactions on Robotics (2006)
19. Ji, M., Azuma, S., Egerstedt, M.: Role-assignment in multi-agent coordination. Int. Journal of Assistive Robotics and Mechatronics (2006) 32–40
20. Parker, L.E.: Alliance: An architecture for faulttolerant multi-robot cooperation. IEEE Transactions on Robotics and Automation (1998) 220–240
21. Botelho, S., Alami, R.: M+: a scheme for multirobot cooperation through negotiated task allocation and achievement. In: IEEE International Conference on Robotics and Automation. (1999) 1234–1239
22. Werger, B.B., Mataric, M.J.: Broadcast of local eligibility for multitarget observation. New York: Springer-Verlag (2000) 220–240
23. Dias, M.B., Zlot, R.M., Kalra, N., Stentz, A.T.: Market-based multirobot coordination: A survey and analysis. Technical Report CMU-RI-TR-05-13, Robotics Institute, Pittsburgh, PA (April 2005)
24. Collier, R.: Debugging Agents in Agent Factory. Lecture Notes in Computer Science **4411** (2007) 229
25. Ricci, A., Viroli, M., Omicini, A.: CArtAgO: A framework for prototyping artifact-based environments in MAS. Lecture Notes in Computer Science **4389** (2007) 67

## A Summary

**1.1** *This entry has been developed using the multi-agent framework Agent Factory. While there are many different domains where Agent Factory has been successfully used, we base our system loosely on the SoSAA robot control architecture.*

**1.2** *The main motivation was to improve upon a relatively successful participation in the Multi Agent Contest 2008 and dissemination of AF experience among newer researchers.*

**1.3** *The agent platform within which the agents ran for the contest was hosted in a single IBM server, as the processing overhead was not sufficient to warrant the additional complexity of adding additional nodes.*

**2.1** *The requirements were expressed as a set of changes on the Multi Agent Contest 2008 system, in an informal story-based format. To facilitate distributed collaboration, these stories were exchanged by email.*

**2.2** *Our system was not formally specified. It was thought that the existing Multi Agent Contest 2008 design would require very few changes.*

**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 AFAPL 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 Section 3*

**3.3** *Roles are assigned to herder agents by a central strategist agent. Once assigned roles, 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 Agent Factory and the AFAPL language in the upper layer of the architecture.*

**4.2** *The AFAPL language is a AOP language that supports the fabrication of intentional agents. Key constructs in the language include: beliefs, plans, goals, commitments, commitment rules, actions, perceptors, and modules.*

**4.3** *Low level capabilities are implemented as a set of behaviours (Java classes) that are loaded into a behaviour controller. The upper level implementation is in AFAPL but makes use of an action (implemented in Java) that allows the agent to select the current active behaviour and a perceptor (implemented in Java) that allows that agent to gather information both about the current state of the environment and the progress of the current active behaviour. Coordination is realised through a combination of a shared map and FIPA ACL.*

**4.4** *Two agent programs were developed. The first encoded the behaviour of the Herders and the second encoded the behaviour of the Leader. Each Herder employed a hybrid architecture, with a lower reactive layer providing its basic capabilities.*

**4.5** *A centralised approach to coordination was employed. The Leader agent would periodically evaluate the world state and assign tasks to the Herder agents using a greedy algorithm.*

**5.1** *The sensory-motor behavioural skills of our agents are inspired by the Vector Field Histogram (VFH) family of navigation algorithms used in robotics and described in Section 5.1. 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** *A strategy agent uses a master-slave ACL protocol to distribute tasks among the agent collective. Tasks' goals are computed via a generic (problem independent) multi agent scheduling mechanism*

**5.3** *The simple scheduling strategy used in the multi agent herding contest is a greedy auction in which the most valuable tasks are iteratively assigned to the most suitable agents. No global optimization is performed*

**5.4** *All perceptions are stored in a shared World Model. The only explicit communication happens at the ACL level, where the strategist agent sends a directive to all the agents in the collective*

**5.5** *The strategist agent sends a new goal to every agent in the collective at most at every cycle of the simulation. To avoid to continuously instructing every agents at each iteration, the goals are communicated only when they would require a significant change of behaviour from the agents*

**5.6** *The team strategy could be improved by improving the heuristics used to estimate the benefits and costs associated to every task, and by performing a globally optimized search of the task/coalition space*

**6.1** *While the herder agents waited for the next set of percept data, the Strategist agent calculated new task assignments in the background, then informed the herder agents of its decision asynchronously.*

**6.2** *To facilitate manual crash recovery without re-exploration, the locations of known static map elements such as obstacles were periodically saved to disk. This map data was then reloaded from disk if the system was restarted while a match was in progress.*

**6.3** *The overall stability of the final system was poor, and could have been greatly improved with more attention to general software engineering issues such as testing.*

**7.1** *Unit testing and a better-defined development process would likely have prevented some of the problems that arose during the process.*

**7.2** *In general, tool support for the development of multi agent systems lacks behind other programming paradigms.*

**7.3** *No clear guidelines exist as to where the division between deliberative and lower level should be put in a hybrid control architecture. This is an interesting open question in multi agent systems research.*

**7.4** *We believe that the most pressing change that could be made to the scenario would be to adjust the scoring system to reward more positive actions by teams. A number of alternatives are proposed in Section 7.*