Ann Math Artif Intell
DOI 10.1007/s10472-010-9182-1

A BDI agent system for the cow herding domain

Nitin Yadav · Chenguang Zhou · Sebastian Sardina · Ralph Rönnquist

Abstract We describe the current state of our multi-agent system for agents playing games in grid-like domains. The framework follows the BDI model of agency and is used as the main project for a seminar course on agent-oriented programming and design. When it comes to design, the Prometheus methodology has been used by relying on the Prometheus design tool (PDT) that supports the methodology. In terms of programming language, we have used the JACK agent platform. We believe the domains developed as part of the Multi-Agent Programming Contest are the right type of settings for evaluating our research, in that it provides enough complexity to explore various aspects of multi-agent development while being sufficiently bounded to make the development effort worthwhile.

Keywords Multi-agent systems · BDI agent-oriented programming · Agent-oriented software engineering · Multi-agent contest

Mathematics Subject Classification (2010) 68T42

N. Yadav · C. Zhou (✉) · S. Sardina
School of Computer Science and IT, RMIT University, Melbourne, Australia
e-mail: czhou.ij@gmail.com

N. Yadav
e-mail: drnitinyadav@gmail.com

S. Sardina
e-mail: sebastian.sardina@rmit.edu.au

R. Rönnquist
Intendico Pty Ltd, Melbourne, Australia
e-mail: ralph.ronnquist@gmail.com

Published online: 01 June 2010

1 Introduction

We report on our multi-agent system, developed in the context of the *Multi-Agent Programming Contest*.[1] The contest task involves developing a multi-agent system (MAS) to solve a cooperative task in a grid-like dynamically changing world where agents in a team can move from one cell to a neighbouring cell.

Since the year 2006, we have been using the domain from the contest for our "*Agent-Oriented Programming and Design*" (AOPD; COSC2048/1204) advanced seminar-run course. Back then, the domain in the competition involved two teams of agents collecting gold pieces scattered around the world and delivering them into a depot [4, 5]. A core agent platform, called RACT-G,[2] is provided for students to build on. Recently, a group of students have adapted the system to the new "*Cow Herdering*" domain introduced in 2008, resulting in what we call the RACT-C agent system (letter C stands for cows). It is worth noting that even though the one objective of the project within the AOPD course is indeed to obtain a working system able to play the game, special focus is placed on the *design* of such a system. Due to that, it happens that many advanced agent features, and even extensions to the domain itself, are accounted for at the design level without following it through to the implementation level.

In a nutshell, RACT-C is an agent system built following the BDI agent-oriented paradigm [8]. In designing the system, the Prometheus [10] methodology and its corresponding (new ECLIPSE-based) PDT [6, 9] design tool—both developed within our research group—were used. The actual system implementation was carried out in the JACK [2] programming language, which is an extension of Java to support BDI agent features and is developed by our industry partner Agent Oriented Software[3] (AOS).

All development and running of agents were done on standard PC hardware, with the system comprising all agents running in a single process. This, though, is just a runtime choice, as JACK provides a Distributed Communications Infrastructure (DCI). By varying the runtime parameters, it allows other system constellations to be used, with the agents distributed over multiple processes and multiple hosts, should need arise.

In the rest of the paper, we briefly describe our system, from its system specification to its internal design and architecture, to its final implementation. We also describe the most important algorithmic strategies that were used to tackle different features of the domain, such as herding a group of cows or going across fences.

2 System analysis and specification

We use the Prometheus [10] agent-oriented methodology to design the RACT-C multi-agent system. The methodology was developed within our group and it is still one of the group's main lines of research. Prometheus facilitates software engineering in agent systems by providing detailed processes for system specification,

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

[2] **RMIT** Multi-**A**gent **C**ontest **T**eam (**G**old domain); www.cs.rmit.edu.au/agents/ract

[3] http://www.agent-software.com/

design and implementation. In addition, we carry out the design by making use of the Prometheus Design Tool (PDT), a design toolkit supporting the Prometheus methodology and also developed within our group [6, 9].

The Prometheus methodology consists of the following three phases:

1. *System specification:* defines the interface between the system being developed and the outside world and identifies the basic functionalities of the system.
2. *Architectural design:* identifies the agents to be used in the system and the interaction among them.
3. *Detailed design:* defines the internals of each of the agents and how they will fulfill their tasks.

In this section, we discuss the first phase for our RACT-C system. This phase consists of a number of interleaved iterative steps, to identify the percepts (inputs), actions (outputs), system goals and the primary functionalities of the agent system. The primary functionalities of the system are described using scenarios and system roles. Scenarios define what our system does—its operation—and the notion of scenarios is similar to the notion of *use cases* in object-oriented design. The key activities in the system specification phase involve:

– identifying system percepts from environment and actions on environment;
– defining scenarios describing the system operations;
– specifying system goals;
– describing roles responsible to fulfill system goals.

2.1 Analysis overview

The *analysis overview* step captures how the agent system interacts with the environment. That is, it identifies the *interface* between the system and the external world wherein the system is situated. By doing so, one can also elicit some of the requirements of the system. Figure 1 shows the analysis overview diagram for system RACT-C.

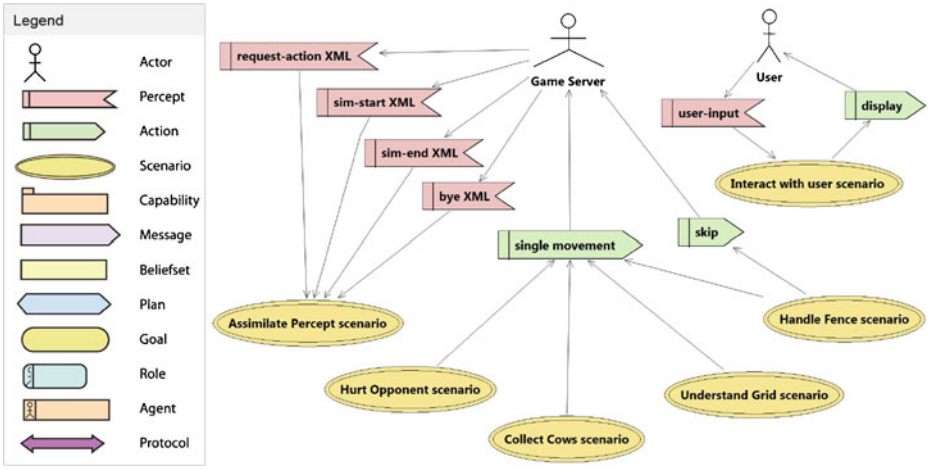


Fig. 1 Analysis overview

The *actors* are the external entities that the system interacts with. In our case, the simulator game server (*GameServer*) and the human user (*User*) are the two external actors with which our system interacts. Whilst interaction with the game server is obvious, we have also considered that the agent system may allow human operator (actor *User*) interaction through a graphical interface, which displays the current state and various log messages from the system to the *User*, and allows for user inputs to the system.

The *scenarios* define the operational behaviour of our system. Scenarios can also be thought of as identifying the typical processes that occur in the system. For example, collecting cows and understanding the world—scenarios *Collect Cows* and *Understand Grid*, respectively—are two processes that will generally happen in the life of our RACT-C system.

Finally, *percepts* and *actions* provide the connection between the external world (the actors) and the main processes of the system (the scenarios). The former represent the *inputs* to the system, i.e., what the system can *observe* from the outside world, and the latter stand for the *outputs* of the system, i.e., what the system can *do* in the world.

The system receives XML messages from the game server corresponding to the different game phases: the start of a game (*sim-start*); the end of game (*sim-end*); the end of a simulation run of many games (*bye*); and during a game, where the game server repeatedly requests a next move (*request-action*). The handling of these percepts is encapsulated in the *Assimilate Percept Scenario*, which defines how the messages from the game server are incorporated into the system. The system itself can act in the environment by performing any of the allowed game actions, or by displaying information to the human user. Based on its reasoning, the system sends action demands to the game server; either to move a player to an adjacent cell, or a "skip" action, i.e., no movement (the game server also implements a response timeout, which defaults the action to be a skip action if the system fails to issue its action choice within the time limit).

As shown in the analysis overview (Fig. 1), the system has four broad level scenarios which send actions to the game server:

- *Collect Cows* stands for the process of collecting cows and pushing them into our corral.
- *Understand Grid* is the process of (better) understanding the grid, by sensing the world and exploring it.
- *Hurt Opponent* involves engaging with the opponent team.
- *Handle Fence* stands for the process of going through fences, typically involving certain level of coordination.

In addition, the system interacts with the human operator by displaying information in the graphical interface as well as possibly receiving inputs from the user. This interaction process is captured with the *Interact with User* scenario.

2.2 Goal hierarchy

The *System Goals* diagram (Fig. 2) captures the system functionality in terms of the *system goals and subgoals* that the system needs to achieve. This goal description at the system level has the advantage of articulating the system objective, and explicate

its penetration and mapping to the goals at the agent level, which are the ones that get realised in the implementation.

Goals are specified in a hierarchical structure, with goals being broken down into sub-goals. An *AND* sub-goal decomposition indicates that all sub-goals need to be achieved in order to achieve the parent goal, and an *OR* decomposition states that the parent goal can be realized by achieving *any* of the sub-goals.

Figure 2 depicts the goal hierarchy for our RACT-C system. The top level goal is *Play Game*, that is, the goal of playing the cow herding game. To successfully play the game the system needs both to *Win Game* and *Interact with user*. The game can be won by understanding the grid (*Understand Grid*), which means to collect information about the grid environment, collecting cows (*Collect Cows*) and pushing them to corral, and "hurting" the opponent team (*Hurt Opponent*), e.g., offensively by stealing their cows or defensively by providing backup to our agents.

We note that these goals correspond to the scenarios developed in the previous step. Indeed, the Prometheus methodology states that scenarios are a good starting point for building the goal hierarchy [10].

Understanding the grid involves finding cows (*Find Cows*), exploring the grid to find fences and boundaries (*Find Fences*), which in turn will facilitate navigation in the grid (*Reach Destination*), and locating the opponent's corral (*Find Corrals*). We note that understanding the grid does not directly win the game, but rather it allows the system to achieve its other goals more efficiently. Therefore, one might say that *Understand Grid* is a "weaker" goal as compared to *Collect Cows* and *Hurt Opponent*. Nonetheless, this is not represented in the design.

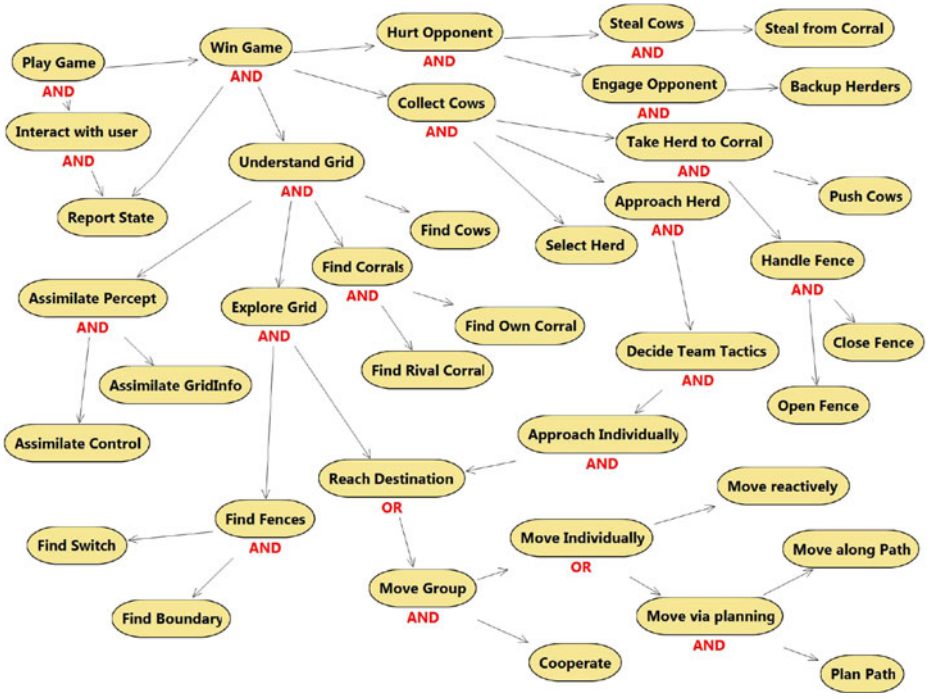


Fig. 2 Goal overview

The system collects cows by selecting a group of cows to be herded (*Select Herd*), positioning a group of agents around the herd (*Approach Herd*), and finally, push or actually herding the cow into inside the corral (*Take Herd to Corral*).

Finally, to achieve the *Hurt Opponent* goal, the system *(i)* engages a group of opponent agents by deploying more agents as backup (*Backup Herders*); and *(ii)* steals cows from the opponent (*Steal Cows*), by either dispersing a herd (*Disperse Herd*) or even stealing from the opponent's corral (*Steal from Corral*).

2.3 System roles

The system specification phase is concluded by identifying the *system roles*, that is, the basic functionalities of the system. Roughly speaking, a basic functionality—a role—is built by grouping sets of goals, percepts, and actions that describe a chunk of behaviour capturing a coherent responsibility within the system. Though in general goal sharing between roles is allowed in Prometheus, it is often a sign of poor design.

The *System Roles* diagram in Fig. 3 shows how the top level goals, percepts and actions are tied together into system roles. The roles *ManageGameRole* and *GamePlayerRole* are responsible, for instance, for interfacing with the game server. We have broken up the game server interfacing into the two roles based on the difference in kind of the percepts and actions, namely, managing the overall game (e.g., start and end of a simulation) versus playing a particular simulation (e.g., receiving sensing information from the world and moving across the grid).

Another important role is that of actually herding the cows along the grid, that is, role *HerderRole*, which should be able to approach the herd and push it towards a destination. The functionality *CoordinatorRole* is in charge of selecting a group of cows to be herded and of providing an overall path to be followed. We note that this role can be carried out eventually by an agent that is not in itself a player in the actual game.

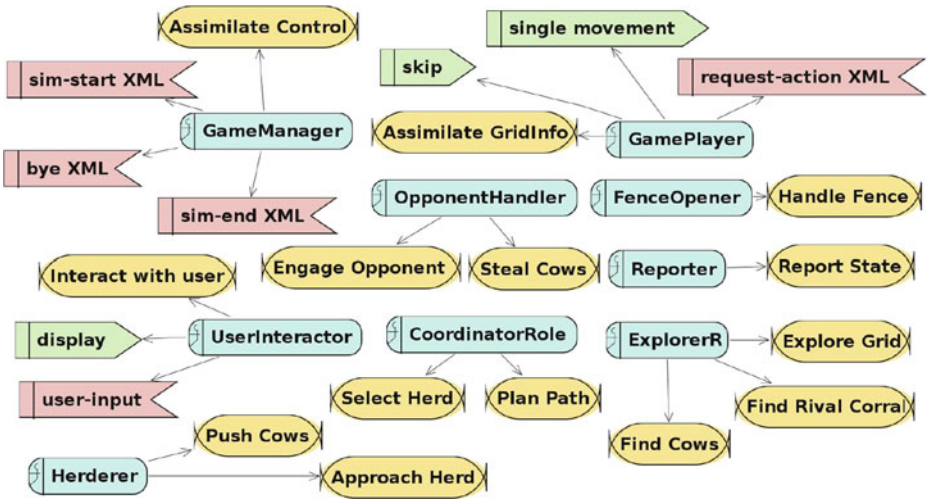


Fig. 3 System roles

With the system specification finalized, we have captured the basic functionalities that the system should provide. This is described through the systems roles with their associated goals, and their relations to the inputs and outputs (i.e., percepts and actions, respectively) of the system with.

In accordance with the Prometheus methodology, agents have not been introduced yet. All terms developed so far are associated with the system as a "whole", and little is said about how the basic functionalities eventually will be realized. The next design phase will start addressing this.

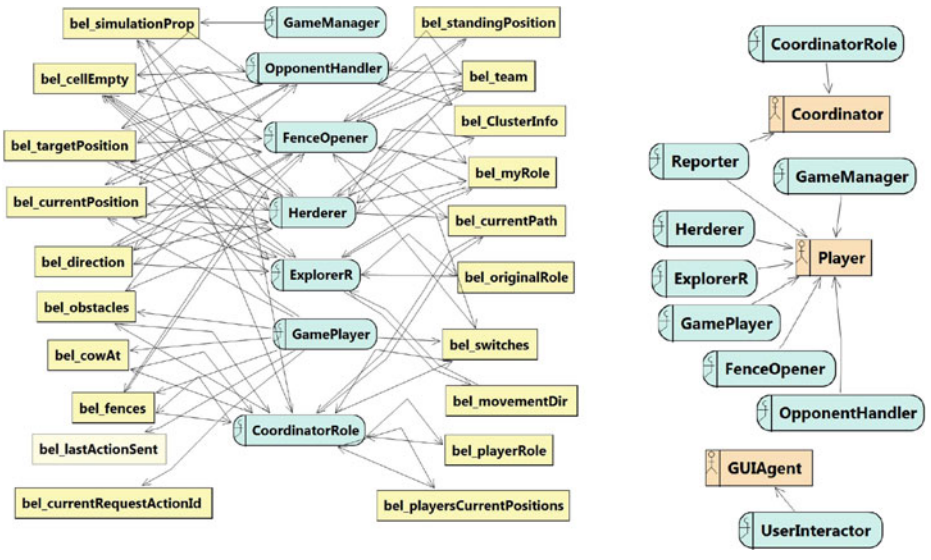
3 System design and architecture

Starting from the system roles developed in the first phase, the second phase in the Prometheus methodology aims to produce a high-level design of the actual agent system. More concretely, this phase aims to:

1. Identify the *agent types* to be used in order to realize the system.
2. Specify the *interactions* between these agents.

A major decision that is made during the architectural design involves determining which agents will build up the whole system. Roughly speaking, agent types are formed by grouping basic functionalities (i.e., system roles): *an agent is a concrete entity in the system that will autonomously and proactively realize one or more system roles*.

One strong reason for grouping system roles together is data coupling: they use the same data. The *data coupling diagram* (Fig. 4a) in the PDT toolkit supports the *data coupling* step in the Prometheus methodology. The idea is to associate each system



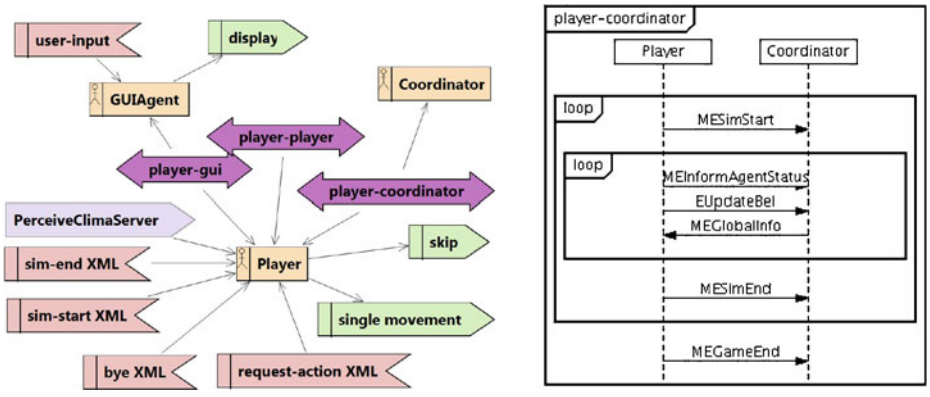
(a) Data Coupling (b) Agent-role Grouping

Fig. 4 Identifying agents in the architectural design phase

role with the data types that the role will require. An arrow pointing towards the system roles indicates that the data is *used*, while an arrow pointing towards the data states that the data is produced by the role. We observe that not only persistent data is specified and linked, but also data that the system roles require to fulfill their jobs. For example, the current position of a player in the grid (data *bel_currentPosition*) is used and produced by all roles that would involve actual movement in the grid. We note that, in our system, some data is used to store some control logic of the system. For example, the data *bel_myRole* states what the current "role" of a player agent. At a given time in the game, a player may be a herder, an explorer, a fence opener, etc.

Once data has been identified and related to functionalities, that is, to system roles, the next step involves grouping several system roles into a single cluster, thus yielding different *agent types*. When combining functionalities, one tries to lower *coupling* while increasing *cohesion* [10]. Figure 4b shows the three different agent types that are used in the RACT-C system. The *Player* agent type will represent each agent registered in the game server and playing the actual game. These are the agents that can perceive and act in the game server. Because of that, these agents need to be able to fulfill the system roles that have to do with carrying on activities in the game. A *Coordinator* agent is an agent that is able to fulfill the coordination functionality, which amounts to facilitate team behaviour within a group of player agents. Finally, a *GUIAgent* is responsible of providing all the functionalities related to the interaction with the human operator.

Once the set of agents to be used in the system are identified, the next step is the development of a high-level specification of the overall architecture of the system, by stating how these agents interact with each other in order to meet the required functionality of the system. To that end, the so-called *protocols* are developed at this stage, capturing such interactions *dynamic* aspects of the system. Figure 5a depicts the current architecture of or RACT-C system. Observe that besides the protocols between the different agent types, the percepts and actions received and performed by each agent type are also displayed. These, however, are automatically inherited



(a) System Overview (b) Player-Coordinator Protocol

Fig. 5 The system overview diagram—architectural design phase

from the system roles (Fig. 3) and their grouping with agents (Fig. 4b): if a system role requires the performance of an action or certain perception, then the agent in charge of such functionality does as well. See that only *Player* agents can perceive and act in the game server, by exchanging appropriate XML messages; neither of the other two type of agents can do so. In fact, the *Coordinator* agent can be seen as a completely internal agent to the system, as it does not interact with none of the actors from the external world—not every agent will interact with the environment.

In order to describe all possible interactions between two or more agents, we use the so-called *interaction protocols*; depicted using the Agent UML (AUML) notation [1]. Figure 5b shows the particular protocol that is followed between a *Player* agent and the *Coordinator* agent. A protocol is built from basic messages that are passed from one agent to another, plus different complex constructs. In our *player-coordinator*, the *Player* first informs the *Coordinator* when a new simulation has started. After that, the two agents will exchange several messages repetetively: the *Player* will inform the *Coordinator* of its status (e.g., its current location) as well as information it has sensed from the world (via event *EUpdateBel*); the *Coordinator* will then send global information that the agent may not see by itself. At some point, the *Player* will inform the *Coordinator* that the simulation has come to an end. At this point, the *Player* may inform of a new simulation game that has just started and the whole process is repeated, or it may inform the *Coordinator* that the whole tournament is over (via message *MEGameEnd*).

Observe that as protocols are developed among the different agents, the *dynamic* aspects of the system required to meet the functionalities of the whole system are captured. However, how exactly those agents will realize those functionalities and such interactions is still not specified at this stage. That is the aim of the next and last design phase.

4 Programming language and execution platform

When it comes to the actual programming of our agent system we have relied on the PDT toolkit [6, 9] and the JACK BDI-style agent programming platform [2].

The PDT design toolkit was used to carry out the so-called *detailed design* within the Prometheus methodology, the last phase in the design of an agent system. In the detailed design, one fleshes out the internal details of each agent so that agents can fulfill their responsibilities (i.e., their roles) and realize the architectural design already developed. This involves stating which event goals, messages, percepts and actions, plans, and data each agent will have and use, and how all these are to be put together to realize the agent. In particular, *capabilities* provide a convenient mechanism to group a set of entities together providing a special functionality akin to "library routines." The detailed design for our *Player* agent can be seen in Fig. 6. As one can observe, the agent is built around three capabilities providing three different general functionalities. The *ClimaTalking* encapsulates everything that has to do with the low-level communication with the game server so that agents can "talk" to the server; this capability will be discussed below in Section 4.1. The *GameSyncing* capability is concerned with the management of the game, such as starting or finishing a simulation game. Finally, the *GamePlaying* encodes everything related to how an agent *plays* an actual simulation game, and will be discussed in Section 4.1.

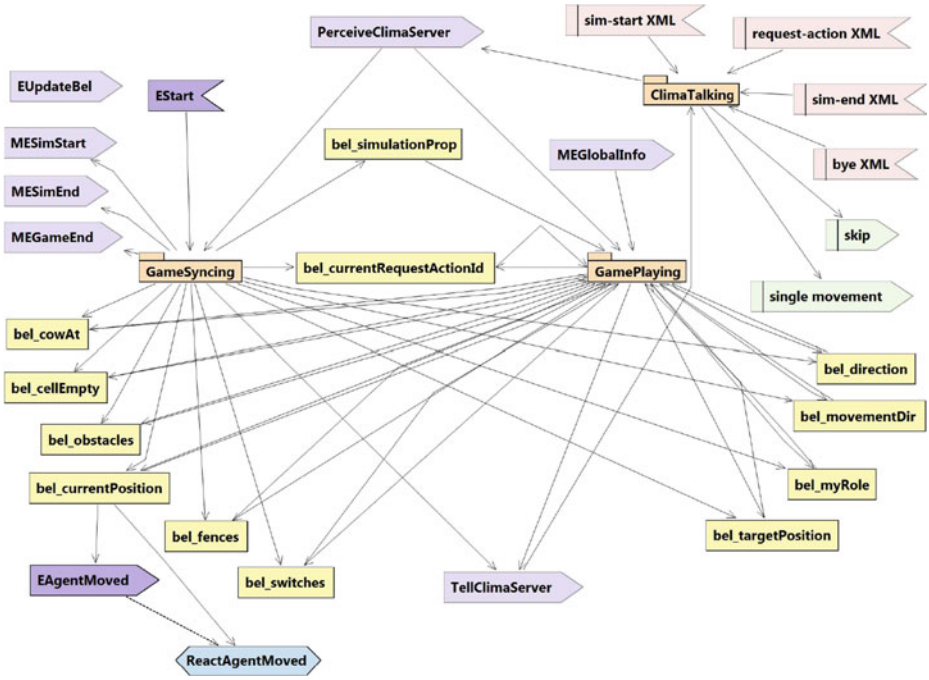


Fig. 6 Detailed design for the *Player* agent

Once the internals of each agent type have been designed, the PDT toolkit is able to automatically generate *skeleton* source code in the JACK programming language.[4] The skeleton code will include the corresponding files for each entity in the system (e.g., agents, plans, events, etc.) as well as all the relations among these entities that are already specified in the detailed design of the agent (e.g., the fact that a certain plan is relevant for an event goal and uses certain data). This automatic code generation not only provides a head start for the BDI programmer, but it also minimizes the chances of introducing inconsistencies when realizing the actual architecture in the programming platform. Of course, once the basic framework is generated, it is up to the BDI programmer to fill the "gaps," including the procedural knowledge of each plan.

As shown by the agent-role grouping in Fig. 4b, our current RACT-C system is built using two agent types *Player* and *Coordinator*. Specifically, a system *instance* consists of a single *Coordinator* agent and multiple *Player* agents. The former agent is responsible for achieving the *Select Herd* and *Plan Path* goals, which we identified as a *team-level* support functionality; the latter agents are the ones playing the actual game in the game server and so the system shall have one such agent per existing player in the team.

[4]Future version of PDT will be able to generate code in other mature BDI-like programming languages such as Jason.

4.1 Communication infrastructure & general architectural overview

As described earlier, the overall system is based on a central game server, which maintains the common "environment" for all the teams of agents, and controls the simulation advance. The game server is designed as a simulation back-bone that interacts with control participants (agents) via TCP/IP sockets using an XML based messaging protocol. During a game, the server issues messages to participants describing their current perceptions and allowing each participant to respond to make an action within a fixed period of time.

In our design, the game server interaction is abstracted into input *percepts*, which arise asynchronously as messages are received, and an output side *event*, abstracting the message delivery to be achieving a "tell game server" goal. This component of the system is encapsulated in a *ClimaTalking* [5] capability, which is incorporated in all *Player* agents, since these are direct game participants. The *Coordinator* agent type is not a direct participant, and it therefore lacks the *ClimaTalking* capability.

Although the communications wrapping is a low level aspect in the system, it has a special interest from the design perspective as it constitutes the foot hold for the cognitive model of *Player* agents. At a low level, the game server interaction is described as a perceive+act cycle, where the agent repeatedly (once every time step) receives a percept and responds with an action. That is, the server interaction is a handshaking protocol, where the server repeatedly presents current state and requests the agent to respond with its action.

However, conceptually a player agent is driven by more long-term objectives, and its response action is generally the fruit of a series of deliberations relating to the history. Thus conceptually, there is a separation between the perceive+act cycle of the server protocol and the purposeful behaviour of the agent. The *ClimaTalking* capability implements this separation in a way that ties in with the lowest levels of Rasmussen's Decision Ladder [7], allowing for the higher levels of decision making to be temporally detached from the request-response cycle of the game server interaction.

In this specific design, the simulation control messages end up as management percepts triggering *Assimilate Control* goals, while the "request action" messages cause *Assimilate GridInfo* goals. The latter result in the build up of the grid knowledge both for the individual *Player* that receives the percept, and via intra-team messaging also for the *Coordinator* agent.

Receiving a percept enables the *Player* to make a new action for progressing on its long-term behaviour goals, and the *ClimaTalking* capability includes achieving the goal *TellClimaServer*, which represents how a *Player* commits to an action. The *GamePlaying* capability, which is detailed below, includes the management of tentative lines of actions, as formed and proposed so as to achieve the player's longer term goals (such as "Collect Herd", "Explore Grid" and ultimately "Win Game"). It is here the decision is made about which among the agent's current tentative next actions to commit to, in response to the developing situation.

[5] The initial design of this capability was for the CLIMA 2006 Contest, which is how it got its name.

4.2 Playing the game: *GamePlaying* Capability

The game playing capability is responsible for processing the current game step information and sending every player's action back to the server. The *ClimaTalking* capability processes the server input and posts the *PerceiveClimaServer* message. This contains the data that the server sends, e.g., the player positions, the cells each player can see, etc. The *HandlePercept* plan handles the *PerceiveClimaServer* message and internally posts messages for three key tasks:

- *EUpdateBel* to update the agent's beliefs.
- *EAct* to cause deliberation so that the agent can choose its next action.
- *EUpdateGUI* and *EShowBeliefs* to update the GUI and output the agent's beliefs.

The *ActionDecision* capability consumes the *EAct* message and posts the message *EExecuteClimaAction* which contains the next move the agent wants to make. The reasoning to choose the next action happens inside the *ActionDecision* capability, shown in Fig. 7. This capability in turn comprises three capabilities; namely, *Explorer*, *Herder*, and *Opener*, similar to agent roles. The current active role of the player agent, as allocated by the coordinator (through the *MEGlobalInfo* message), is stored in the *bel_myRole* beliefset. The *EAct* is handled by the capability corresponding to the active role of the agent. For example, if the agent is playing the role of a herder, the *EAct* is handled by the *Herder* capability and the agent's next move is posted via the *EExecuteCLIMAaction* message. In order to reason about the next action the capabilities use environment information such as the location of the switches, obstacles, fences, are stored in beliefsets *bel_switches*, *bel_obstacles*,

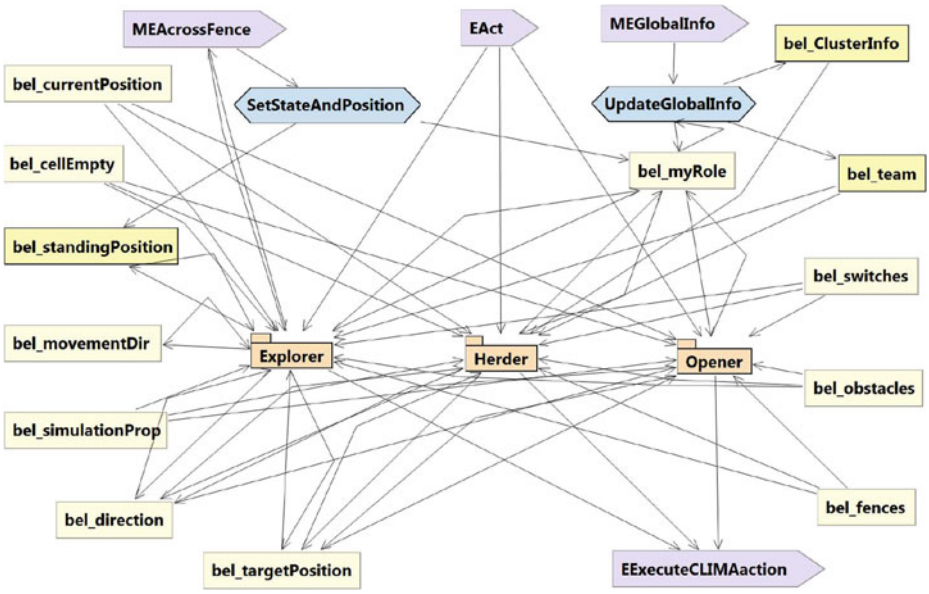


Fig. 7 *ActionDecision* capability

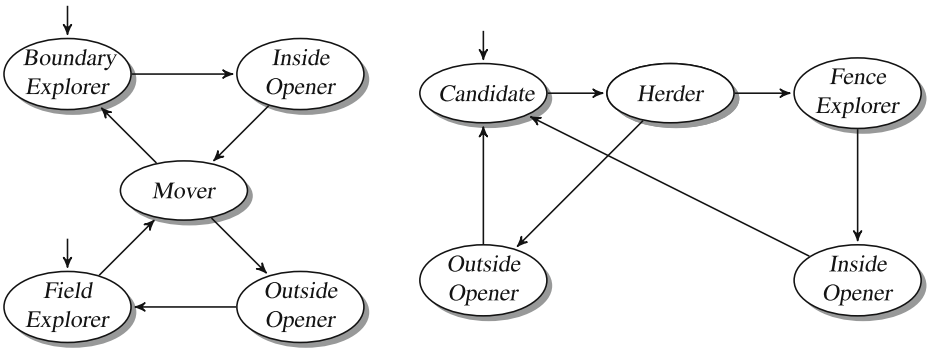


Fig. 8 Logic of *Explorer* and *Herder* roles

bel_fences respectively. Note, the *Herder* and *Explorer* roles are team activities. The team related information is stored in the *bel_team* beliefset and the cow cluster information for the herders is stored in the *bel_clusterInfo* beliefset. Section 5 describes the deliberation that happens inside the the capabilities which result in the next action.

The *ActionDecision* capability models the game player roles, as detailed in the next section. We use beliefsets to store the player's role and role related information, as allocated by the coordinator. Based on the assigned role the player chooses its next set of move(s). The game server accepts one move per cycle, in contrast our system is capable of generating a set of moves (in advance) based on the player's intention. The *ActionDecision* capability bridges this step-wise move requirement with long term planning by player agents.

The initial *Player* behaviour model is expressed as a collection of state machines that are aligned to the perceive+act cycle of the game server interaction. Figure 8 presents the state machine logics of the *Mover* and *Herder* roles. This means that the agent's composite tasks are modeled by using explicit state representations that define the enabling contexts for the behaviour model plans, which essentially serve as state-to-state transition rules. The BDI plan selection mechanism, then, serves as the method of selecting which state transition to apply.

5 Agent team strategy

As stated above, when the system is started, a single instance of the *Coordinator* agent type is created and multiple instances, one per actual player in the team, of the *Player* agent type are created. The former is responsible for achieving the *Select Herd* and *Plan Path* goals, which we identified as a team level support functionality.

The *Player* agents takes part in the game as individual game players, by basically fulfilling the game playing roles. During a game, each *Player* commits to and acts within any one of its roles one at a time under the direction of the *Coordinator*. In our current strategy, one *Player* is dedicated to finding the "home" corral and operating its fence switch. This *Player* will stay close to the fence switch, to defend it from the opponent and to step up to it in order to open the fence whenever a herd is about

to be pushed into the corral, and then step away again. The other remaining *Player* agents alternate between *exploring* the grid and *herding* cows. Both such activities involve groups of agents which act cooperatively to achieve the corresopnding goals.

5.1 Avoiding obstacles

Individual *Player* agent uses a *reactive movement algorithm* to avoid obstacles. The idea is to store the history of *Player*'s movements by means of a *priority table* so that obstacles can be traversed. This table maintains priorities of each possible direction (i.e., left, right, up and down). Each step, these directions are checked in the order of their priorities until finding an available direction. Then *Player* moves towards this direction and updates the table by different rules depending on the priority of the chosen direction.

5.2 Exploring the grid

An explorer group involves three *Player* agents acting together to explore the grid, field by field (a field being a closed area with possible fences connecting to other fields). Two *Player* agents—the boundary explorers—navigate along the field walls in opposite directions to understand the field's boundaries and locate fence gates. The third *Player* moves across the field, trying to locate cow herds and understand the internal shape of the field.

The boundary explorer who perceives a fence first will then go to find the switch and open it. Meanwhile, it notifies the other two explorers to come to the fence and go through, passing to the next field. One of these two explorers then goes to the switch from the inside so as to hold open the fence while the first switch opener comes through. Thereafter, they split up again to explore the new field. If the explorers find the rival corral at any point in time, they may shift their roles to become "stealing" herders: they only herd cows out from inside the rival corral.

5.3 Herdering cows in the grid

Once a herd of cows has been found, the *Coordinator* agent, who receives information from all game players, determines the *boundary* and *size* of the herd. Then, it computes the path from the centre of herd to the team corral by applying the A* algorithm, with appropriate clearance values to guide the herd as a whole around any obstacles, c.f. the Brushfire algorithm [3].

Based on the size of herd, the *Coordinator* agent decides on the number of *Player* agents requijred for pushing the herd, and selects appropriate ones to form a herding group. Each member of the herding group is given its own relative location in the herding pattern, which is a semi-circle sector behind the herd and opposite to the herding direction. Based on this and the path, each herder agent computes and move to its target spot. Once all herders have reached their new spots, the *Coordinator* updates them with the herd boundary and size details, from which agents calculate their next target locations to move to so as to "push" the herd. This process continues repeatedly.

To go pass across fences, the herding agents cooperate in the following manner. First the herder nearest to the fence switch moves up to its switch to open the fence.

If the location of the switch is not known yet, the herder first moves along the fence to find it, and then opens it. As the fence opener leaves for operating the switch, the remaining herders are re-deployed as a smaller herdering group, and they all push the herd through the gate. After having gone through the gate, the nearest of the herders goes to the switch from the inside, to hold open the fence and let the outside switch operator to come through as well. Thereafter, the whole group is re-deployed in the herd once again and they continue herding the cows together along the herdering path.

6 Technical details

Although we have already discussed some technical details of our agent system, here, we shall elaborate on robustness, background processing, and BDI failure recovery.

In terms of stability, our current system is very reliable when it comes to the low-level connectivity and communication with the game server. This is due to a solid implementation of the *ClimaTalking* capability. Nonetheless, more robust mechanisms are needed to deal with non-connectivity issues. For instance, if an agent is found to be non-responsive, it may be reset by the coordinator (or even by the agent itself). In some cases, e.g., the agent was fully reset from scratch, the agent may decide to query teammates for information that may be relevant to her (e.g., knowledge on the surrounding area where she is located).

Probably more interesting are the issues of background processing and BDI failure recovery. Being a fully reactive system, in its current version, the RACT-C system uses neither. However, we point out that the system was specifically designed to accommodate both aspects in a natural manner. Indeed, the decoupling of the perception/actuator mechanism (via capability *ClimaTalking* and events *PerceiveClimaServer* and *TellClimaServer*) from the actual decision making mechanism (via capability *GamePlaying* and event *EAct*), allows for simple modeling variations based on *long-term goals*, which would be able to exploit both JACK's powerful plan/goal failure-handling mechanism plan as well as computational processing that is independent of the game perceive+act step-wise regular cycle.

So, in what follows, we discuss design principles for modeling the pursuit of long term objects. One might say that a state machine approach as discussed in Section 4.2 arises naturally from the perceive+act presentation of a player agent with respect to the game server interaction. At the same time, however, a player agent behaviour model naturally includes processes that may span multiple game server interactions. In the state machine approach the long term intentions, and the flow of process steps, are reduced to arise incidentally by the layout of that state transition logic. In that sense, the BDI reasoning aspects of continuing an intended plan and plan-goal failure are completely unused.

We note that the contrast between the state machine approach and having long term intentions in plans is more conceptual than technical. For instance, the *GamePlaying* capability, which is presented in Fig. 9 would remain intact and provide the same action execution framework regardless. The *EAct* event still arises for the purpose of choosing which action to commit to. The difference in approach would then be found in the modeling of the decision making behind which actions to choose from.

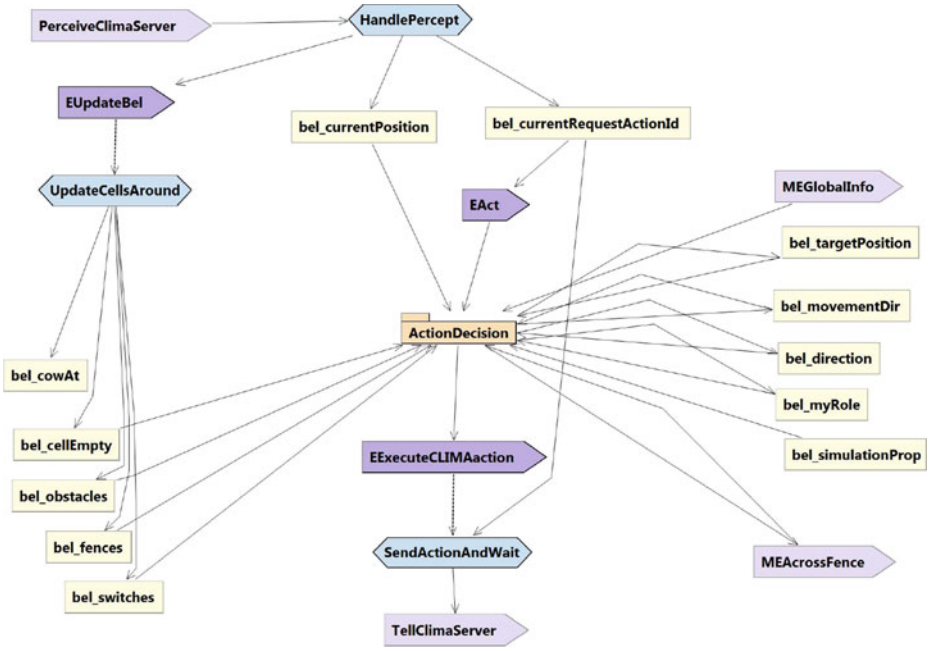


Fig. 9 *GamePlaying* capability

The framework also needs additional means that synchronizes the execution order of long-term intentions relative the *EAct* action arbitration and commitment. In particular, upon the receipt of a new percept, all long-term intentions that are affected by it should wake up and progress exhaustively, and in particular offer their new action suggestions, before action arbitration and commitment takes place. The notion of *TaskJunction* in JACK provides this kind of synchronisation, which may be thought of as a semaphore that allows multiple concurrent "readers," but only a single exclusive "writer." Furthermore, a *TaskJunction* includes a method for escaping the reader group temporarily while the intention is waiting for its next situation (or next percept) to arise. This "escape" function ties in with JACK intention execution to ensure that the intention re-joins the reader group immediately, as soon as the situation for the intention comes about and even before the intention execution is continued. In that way, the effects of a percept propagates immediately, so as to cause the required synchronisation that holds back the *EAct* intention while long term intentions are computing.

7 Discussion and conclusion

We strongly believe the actual Multi-Agent Programming Contest is highly valuable for the agent-oriented research community. The contest provides a *convenient* framework for evaluating ideas, techniques, and approaches to agent-oriented programming. By convenient we mean that the domain used is both sufficiently *rich* to

allow the exploration and evaluation of various aspects of multi-agent development as well as sufficiently *bounded* to allow research groups to develop systems with reasonable effort.

In this paper, we provided an overview of our current RACT-C agent system for the Agent Contest cow herding domain. The RACT-C platform is built following the BDI style paradigm and relies in basically two technologies, namely, the PDT toolkit for design and the JACK language as the programming language. The RACT-C system has been mainly developed to support the main project for the "Agent Oriented Programming and Design" seminar course, in which the students are to extend a basic core agent system both in terms of design and programming. In particular, since agent-oriented design is one of the main lines of research within our group, the RACT-C system is used as a case-study for the Prometheus methodology [10] and its associated PDT design support toolkit [6, 9]. As the core system get more stable and more students become involved within the project, we plan to have a more active role in the official competition. Though the domain is bounded and feasible, we found out that it is still necessary to devote substantial programming time in order to reach a reasonable working system. We believe we are currently reaching such stage as well as a critical mass of students interested for continuing the development.

In the contest, our agents managed to collect 194 cows, scored 23 points and ended up ranking fifth. We implemented complete herding and stealing components that worked well in driving cows towards our own corral or out of the rivals corral. In addition, we developed a reactive movement algorithm, which works more efficiently than A*, to avoid obstacles. However, the path planning component was not stable. As a result, we lost some cows when going around obstacles. Overall, the contest helped a lot in promoting the development of our JACK system.

There are some drawbacks and limitations in our current RACT-C system that we have learnt during its development. For example, at this stage, we have a "flat" design of goals, in that these are understood always as "achievement" goals. It is not clear how qualitative or secondary/side goals can be represented in the design, such as acting biased towards information gathering (e.g., preferring unexplored areas). Another missing feature is that of explicit "teams." At this point, teams of agents (e.g., explorers or herders) are only implicitly represented and have no special semantics. An explicit modeling and programming of such teams would decouple team specific tasks (e.g., forming a herding team or coordinating the navigation through fences) from individual agent tasks. It would also facilitate and abstract certain agent processes related to the team, such as broadcasting information to all the members of a team (without knowing the actual members). We are planning to accommodate explicit team-oriented behaviour by using the JACK-TEAMS extension to JACK, which accommodates native team reasoning entities for "team" programming.

Our system is still not fully faithful to the BDI programming paradigm as agents do not pursue long-term goals with long lasting associated intentions. However, an alternative version that represents strategies with long-term goals can be accommodate within the current design (see Section 4). Among other things, this will allow the system to take advantages of the typical goal/plan failure recovery mechanism found in typical BDI architectures, in which alternative plans are tried for a goal if the current plan happens to fail. For instance, if a particular way of herding the cows happens to fail, for instance due to the herd being too large for the space available in the grid, an alternative (maybe more conservative) herding plan-strategy could be used.

Although our focus so far with the agent contest domain has mostly been linked to our research-oriented course, we plan to become active participants of the yearly official contest as our current system becomes more mature and a group of students is formed to work on it. As it is the case with other AI competitions, it is fundamental that the students are the ones driving the participation in this type of events. The agent contest provides a sufficiently complex, but still feasible, domain for evaluating our current approaches and techniques on multi-agent development. Moreover, as done in other AI competitions (like the International Planning Competition), the domain can be incrementally extended over the years in a way that the domain is enriched with new challenges. This extensions however need to be carefully planned and introduced. First, these extensions need to be always *fully incremental*, so that newer contests are always build on top of previous versions and participants are guaranteed their efforts will span more than a single edition of the event. Second, the complexity of the extensions should be such that any existing participant can extend their current systems without a substantial amount of work relative to the overall system. One such extension, for instance, would be to restrict the field view of agents in such a way that these cannot see beyond obstructing obstacles or the possibility of agents to move/push certain type of objects.

Acknowledgements We would like to thank the many students that have taken the Agent-Oriented Programming and Design (AOPD) in recent years for their valuable feedback on using initial version of the agent system reported in this paper. We also want to thank Jimmy Sun for his always available technical support with the PDT toolkit. Finally, we acknowledge the financial support of Agent Oriented Software and the Australian Research Council (under grants LP0882234 and DP1094627).

Summary

1.1 *The RACT-C agent system was developed mostly as the running project for our seminar style course "Agent Oriented Programming and Design" (AOPD) as well as a case study/example for our PDT design tool for the Prometheus methodology.*

1.2 *The motivation was to test the applicability of the overall agent design and programming framework developed as part of the AOPD course by a group of students and to identify the practical challenges to be addressed.*

1.3 *In terms of software, we have used the PDT tool for design and the JACK agent programming language. In terms of hardware, we have used standard desktop and laptop machines, with all agents running in the same hardware (a runtime choice).*

2.1 *No specific requirement analysis approach was used as the basic requirements were already all specified in the game document and protocol; no extra functional requirements were given. Still, one can see the Analysis Overview as a place to elicit and display parts of the requirements (e.g., play the game in the game server).*

2.2 *At the highest level of abstraction the specification is done in terms of how (external)* actors *(e.g., the game server) interact with the* scenarios *of the system via input* percepts *and output* actions*. The specification is then further refined into a* goal hierarchy *and set of main* roles*.*

2.3 *The multi-agent system is specified following the Prometheus methodology and using the PDT that supports such methodology.*

2.4 *Autonomy, coordination/team-working, pro-activeness are all implicit within the scenarios, within which the whole system outputs actions to the game server while it assimilates observations from the server. The different roles that the system need to fulfill are also specified together with their associated goals defining the roles.*

2.5 *The system is a truly multi-agent one; each agent acts autonomously and can survive by itself. Nonetheless, the different agents communicate and coordinate at certain points to carry out joint tasks (e.g., opening a fence) or exchange information.*

3.1 *The second development stage involves the architectural design of the system. By relying on the roles from the specification phase, the require data, agents, and communication protocols among them are detailed.*

3.2 *The Prometheus methodology was used to state the architectural design of the system. First, the required data for each role is specified. Second, agents to fulfill the system roles are created. Finally, sensible protocols among agents are specified.*

3.3 *Agents are meant to autonomously and proactively be able to address the goals associated with their corresponding roles. The communication that shall be required is encoded in the protocols. Coordination and team-work arises as agents take roles with shared goals in the goal hierarchy.*

4.1 *The PDT tool supporting the Prometheus methodology was used for designing the system. The JACK agent-oriented programming language and platform was used for actual implementation.*

4.2 *An advanced BDI-type programming language was used. As such, the agent system is built around* events *and crafted* plans *for handling such events in the context of what is* believed *true in each situation.*

4.3 *The PDT toolkit is able to generate initial JACK skeleton source code compatible with the design. In that way, a minimal consistency between design and actual implementation is guaranteed. For instance, an actual JACK plan will generally belong to some specific* capability *related to a* system role *and will handle JACK* events *that model goals of the role in question in the design.*

4.4 *JACK provides special Java classes to model typical BDI agent components like beliefset, plans, events, and capabilities. In addition, the programming framework is fully integrated with Java so that standard Java code can be used as support for specific tasks (e.g., path planning).*

4.5 *The agent system has specific plans encoding simple strategies for various tasks, such as, navigating the grid, pushing cows towards a direction, going through fences in groups, and finding objects like fence's switches or cows.*

5.1 *A reactive algorithm for movement is used to avoid obstacles. The idea is to maintain a table of priorities of each possible direction. At every step, these directions are checked in the order of their priorities until the required direction is found. The same table is used to store the navigation history of the agent.*

5.2 *One extra agent is used as coordinator who can collect information from every agent and hence has a global view of the environment. Coordinator can make a better decision with respect of assigning appropriate roles for agents.*

5.3 *We do not have this functionality the moment.*

5.4 *The location of the switch and each agent is shared while passing through a fence. The opener agent holds the switch from inside so that the other agents can go across the fences. Once the rest of the agents have crossed, the agent who is the closest to the switch goes to hold the switch from outside to let the opener in.*

5.5 *We broadcast messages to communicate in a team.*

5.6 *Using a leader agent for each herder team would improve herding efficiency. The leader can also help in path planning, opening fences, or blocking rivals herds.*

6.1 *Although the system does not currently do any background processing while "idle," it was specifically designed to accommodate such computation in future versions.*

6.2 *The current system, being fully reactive, does not exploit JACK's powerful failure-handling mechanism. A simple modeling variation based on long-term goals would be able to do so. Disconnection from the game server is transparently handled by the ClimaTalking capability, which will automatically re-establish the connection.*

6.3 *The communication with the game server proved to be extremely reliable. More robust mechanisms are required, however, to deal with agents crashing due to bugs in their code as well as with agents that take too long to output their next action in the server (e.g., by resetting the agent itself or their plans).*

7.1 *The strategies encoded so far for the various tasks are currently not fully reliable. Also, some other strategies are not conceptualized in the implementation yet. However, the current agent system provides a very reliable and clean core BDI framework that can be used as a base system towards a competitive system.*

7.2 *Since the agent system developed is of some non-trivial complexity, we gained many insights both with respect to agent design and actual BDI-style programming.*

7.3 *In using the PDT toolkit, many existing limitations where discovered and resolved for its new ECLIPSE-based version. Some intrinsic limitations of the BDI programming approach as well as limitations of the JACK platform were also discovered.*

7.4 *We think that the observation model can be easily improved by not allowing agents to see beyond walls. The existence of tools in the grid that agents may use could also be a interesting extension to consider (e.g., vehicles allowing faster navigation).*

References

1. Bauer, B., Müller, J.P., Odell, J.: Agent UML: a formalism for specifying multiagent software systems. In: Proceedings of the Agent-oriented Software Engineering Workshop (AOSE), Secaucus, NJ, USA, pp. 91–103. Springer, New York (2001)
2. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK intelligent agents: components for intelligent agents in Java. AgentLink Newsletter **2**, 2–5 (1999). Agent Oriented Software Pty. Ltd.
3. Choset, H., Lynch, K.M., Hutchinson, S., Kantor, G.A., Burgard, W., Kavraki, L.E., Thrun, S.: Principles of Robot Motion: Theory, Algorithms, and Implementations. MIT Press, Cambridge, MA (2005)
4. Dastani, M., Fallah-Seghrouchni, A.E., Ricci, A., Winikoff, M. (eds.): Programming Multi-Agent Systems, 5th International Workshop, ProMAS 2007, Honolulu, HI, USA, May 15, 2007. Revised and Invited Papers. LNCS, vol. 4908. Springer (2008)

5. Inoue, K., Satoh, K., Toni, F. (eds.): Computational Logic in Multi-Agent Systems, 7th International Workshop, CLIMA VII, Hakodate, Japan, May 8–9, 2006. Revised Selected and Invited Papers. LNCS, vol. 4371. Springer (2007)
6. Padgham, L., Thangarajah, J., Winikoff, M.: Prometheus design tool. In: Proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 1882–1883 (2008)
7. Rasmussen, J.: The human data processor as a system component. Bits and pieces of a model. Technical Report Riso-M-1722, Riso National Laboratory. Research Establishment Riso, Roskilde, Denmark (1974)
8. Shoham, Y.: An overview of agent-oriented programming. In: Bradshaw, J.M. (ed.) Software Agents, pp. 271–290. MIT Press (1997)
9. Thangarajah, J., Padgham, L., Winikoff, M.: Prometheus design tool. In: Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS), pp. 127–128 (2005)
10. Winikoff, M., Padgham, L.: Developing intelligent agent systems: a practical guide. In: Wiley Series in Agent Technology. Wiley, New York, NY (2004)