A Testbed for Animating Multi-Agent Systems

Jim Cunningham, Lloyd Kamara

Logic & Automated Reasoning,
Department of Computing,
Imperial College,
London, SW7 2BZ
{rjc,ldk}@doc.ic.ac.uk

Jeremy Pitt

Intelligent Communication Systems,
Department of Electrical & Electronic Engineering,
Imperial College,
London, SW7 2BZ
j.pitt@ic.ac.uk

Abstract

This paper describes a testbed for animating systems of multiple rational agents. In particular, we are concerned with simulating a manufacturing scenario in which robotic agents are instructed to collect, paint, and deliver cars in a paintshop with several interconnected workplaces.

A rational agent will sense its environment, plan and execute its actions, and communicate with other agents. We briefly discuss how such behaviour can be specified using multi-modal logics and realised on a reactive base.

The testbed is intended to be a tool for research and development, so we provide facility for interactive experiment with imperfect agents. Even though a broken and incoherent agent may be the norm, it must still be animated and co-exist in a simulation of its real, incompletely known environment.

The testbed has been realised as a multi-processing Unix system, with each agent executing concurrently as a separate process and interacting with a privileged process which simulates reality. Agent capabilities and inter-process communication are implemented in ICL/ECRC ECL^{i}PS^e constraint Prolog, visualization and graphical user interfaces are implemented in Tcl/Tk, and heavy numerical computation for local path planning is implemented in C.

Although designed as a testbed, we have produced a system with many features of a full scale application of cooperating rational agents.

1 An Introduction to the Medlar Animator

1.1 Background: The Medlar Projects

The CEC Esprit basic research projects Medlar (3125) and Medlar II (6471) have sought general frameworks for mechanising deduction in the logics of practical reasoning – the philosophical term for logics of time, action, belief, knowledge and duty. These logics embody critical analysis about the forms of reasoning. In Medlar II the concept of a *practical reasoner* has been developed, an agent capable of acting autonomously and interacting flexibly with its real-world environment. Like other descriptions of rationality in the field of artificial intelligence (e.g. [4]), the core is its specification as a logical theory. For example, the specification should characterise in some ideal way what an agent believes about the effect of its actions.

A number of case studies have been examined as part of the Medlar projects, including the areas of production planning and automation for robot scenarios. From this has come the so-called 'Medlar PaintShop', an idealised manufacturing scenario in which robots, or rather, practical reasoning agents, are instructed to collect, paint and deliver cars in a 'factory' with several inter-connected workspaces [13]. The testbed described in this paper is designed to simulate this scenario using *logical animation* (whence Medlar Animator), a method of symbolic execution used to examine the dynamic behaviour of logical specifications [5].

There are subtle issues in the design of a symbolic simulator. Ours has a somewhat Cartesian epistemology, adhering to the tenets that an action can only happen if it is willed by "god", and that the consequence of the action is only known by sense experience [6]. To perform an action in the simulator an agent sends a request to 'reality'. Reality checks whether the action can happen, does happen, and what the effect is. The result of the action is communicated back to the agent in reply to the request, and this is what the agent now senses. To retain the overall coherence of Descartes' philosophy, Reality should be as trustworthy as the God of Cartesian metaphysics. If we want to simulate the effect of a faulty sensor, say, this is coded in an agent – the agent receives a true message, which is corrupted before being used to update senses and beliefs.

1.2 A Brief Overview of the Animator

The basis of the testbed is two process shells, one for simulating *reality*, the other for simulating an agent. So in a simulation with n agents, there will be n instances of the agent shell running. Each process shell becomes an executable process when it is supplemented by two items, a specification and a graphical user interface (gui). In the case of the reality shell, the specification is a complete description of the world: its locations, the connections between locations, and objects in it (obstacles and cars in the Medlar PaintShop). The reality gui is a 'god's-eye' or omniscient view of the 'actual' state of the entire micro-world. In the case of an agent shell, the specification is a description of the 'proper' behaviour of the agent, which is animated to decide the actions the agent will take. The agent gui shows a restricted view of its immediate surroundings which it is able to 'sense', any other information which it can also 'sense', and a representation of properties it 'believes' to be true of itself.

In a simulation using the animator, each process instance can be running on a separate machine connected to the Internet. The user can interactively issue instructions, cause events (e.g. by creating a car), or execute a script of instructions and events. It is intended that an agent will plan composite actions to achieve its goals, but each action may be accomplished by a series of primitive or granular actions, like `move()`, `pick-up()`, `paint()`, etc. These constitute the reactive base of the simulation. Whenever an agent attempts to perform a granular action, it sends a 'request' to reality which determines what happens "in reality". The consequence of the action is only known to the agent by sense experience, so reality returns in reply the new surroundings and/or attributes that a robot can 'sense' and which it now believes to be the case.

Agents navigate about the micro-world using the potential field method of path planning for robot motion [12] to extend the reactive base. However, we have the extra complication that the world can change, so we have a 'static' potential field calculated from the agent's 'memory' of the location and a dynamic potential field calculated from the agent's perceived immediate environment. The combination of the two determines in which direction the agent moves ('granularly') as it tries to get from its current location to a goal location.

Agents also need to be able to communicate with one another, and for this we intend to use cooperation protocols [2, 14] and the library for inter-process communication reported in [15]. (In fact this library is already used in the animator to implement the inter-process communication required to execute granular actions as described above.)

1.3 An Assessment of the Animator

Hanks *et al.* [11] specify a number of features that a planning testbed might exhibit if it is to be experimentally challenging and usable by others. These include: exogeneous events, world complexity, quality and cost of sensing and effecting, measures of plan quality, multiple agents, a clean interface, a well-defined model of time, and support for experimentation. Our animator has nearly all of these features.

Exogeneous events are allowed: cars can be added, moved and deleted during a simulation, subject to user control. The environment itself can be distorted as the user can add and remove obstacles. The user can even 'physically' pick up an agent and move it to a new location. The complexity of the world is quite high: it is beyond some of the artificial micro-worlds reported in [11], for example, and should be scaleable to realistic manufacturing scenarios. Although it is not a faithful reproduction of reality some real-world physical constraints are coded in, for instance two agents are not able to occupy the same location simultaneously.

The test-bed can be configured so that agents are not perfect sensors or effectors. For example, the message from reality back to an agent which updates its locale can be randomly or systematically distorted; actions, such as an attempt to pick up or paint a car, can be 'failed' on a probabilistic basis. The difference between an agent's belief about the state of the world and the 'actual' state of the world is maintained. What an agent believes and senses is represented in its gui, the actual state of the world as represented in the reality gui.

Multiple agents are allowed by the test-bed. The same mechanism for inter-process communication that allows agent processes to communicate with the reality process can also be used to enable agents to communicate with each other. The message specifications and co-operation protocols defined in [14] can be used to investigate coordinated behaviour between cooperating and even competing agents.

The animator is largely data driven, so different specifications of agent behaviour can be supplied, and different environment specifications can be used to investigate alternative scenarios. There is a well-defined interface between the user and reality which makes it clear what actions are available to the user. These may change reality or cause an agent to act.

The animator does not have a well-defined model of time, except that it runs in real time 'red in tooth and claw'. Thus an agent can form a plan to act (for example to pick up a car) only to get there and find that some other agent has got there before it. We expect this model to be "realistic", but variation between simulation and a real world application could force revision of this position. A trace can be put on each agent which shows how they have moved. This can be used to record the sequence of decisions and actions that an agent took. So a user can observe, monitor, and record the behaviour of agents, to explain why events turned out as they did.

The missing features from Hank et al.'s list are measures of plan quality and support for experimentation. However, it has always been envisaged that the software would facilitate experimentation with a variety of communication and co-operation protocols due to the issues of autonomy and uncertainty inherent to the simulation environment. The issue of remote (central) control vs local intelligence is also related to these concepts. The testbed might allow the experimenter to derive sufficient data for a variety of scenarios, leading to results comparable to a much longer construction/investigation of 'real' robots.

2 Agent Specification

The specification of an agent can take place at different levels of abstraction. The coarse layering of the agent architecture in the simulator separates reasoning from an extensible reactive base and its implementation by communicating processes. A rational agent will sense its environment, plan and execute its actions, and also communicate with other agents (see, for example [4, 16]). Such behaviour can be specified as a theory in a multi-modal logic with a language which includes qualifications for seek (a goal), sense, believe, doing etc.

The intended user's specification of an agent is a theory in a multi-modal dynamic logic. The concepts of such a theory should enable planning to achieve a sophisticated goal (such as "produce a red car") by means of compound actions such as "find a car", "carry to free area", "paint the car red", "carry to finished area", etc. Compound actions will themselves be composed ultimately of granular actions which are part of the *reactive base* base of the simulation. A hierarchy in levels of abstraction over dynamic actions is familiar from procedural programming, where sub-routines are called in accordance with task structure. A similar hierarchy of predicative properties is natural in computational logic, where a synthetic property, perhaps `finished()`, will have concomitant implications from its definition in terms of more primitive sensory properties like `painted()` and `hascar()` which are provided by the reactive base of the simulator. Procedural and predicative abstraction can be combined in a suitable action logic. (Some of the ingredients are already present in [1].)

For want of a more precise demarkation we will call this layer of specification the reasoning level, although only discuss it briefly in the next section. In general this level presumes a specification of preconditions for actions, information for reasoning about ramifications, and may include both epistemic and probabilistic reasoning. Figure 1 is a specification of the granular actions in terms of an agent's beliefs about particular attributes of the reactive base.

To admit reasoning about them, the effect of the granular actions of each agent on the attributes of the reactive base must be expressed as basic properties in the language of the user specification. So we specify the properties of the granular actions and attributes. We might expect these to entail "obvious" things, that a robot is carrying a car after it has picked it up, that a car has the colour it has been painted, etc. However there is epistemic qualification in even supposing a robot to have a car after executing a primitive action to pick it. As a rational agent, the robot can only *believe* this so. It may confirm this belief through other sensors, or enter into introspective reasoning. In specifying behaviour there is a case for more frequent use of epistemic qualification, to make simple and realistic local distinctions and gradations in the states of affairs.

While a specification of the granular level is primitive for reasoning about agents, for implementation of an agent by a process, the need for communication with reality introduces a more intricate level of specification. This level comes from the Cartesian model for the underlying simulation, where each granular action is actually realised as a thread of inter-process communicative actions and tests which are even more primitive. So to implement the granular action to pick up a car, a robot agent must request reality to change the world in this way. Reality in turn will check that the robot agent does not already have a car, has moved to a location where a car is situated, and make the appropriate change to reality and the agent's attributes etc. This is a more general way of composing actions than is familiar from conventional procedural programming, but de Greef *et al* [9] have suggested a natural notation for specifying inter-process actions (a brief overview is given in Appendix A).

We have previously used the de Greef notation to specify cooperation protocols for specific

$$
\begin{array}{lll}
hascar_{Ag}(C) & \leftrightarrow [granule\_putdown] & \neg hascar_{Ag}(C) \\
 & & \\
\neg hascar_{Ag}(C) & \leftrightarrow [granule\_pick\_up] & hascar_{Ag}(C) \\
 & & \\
hascar_{Ag}(C) & \leftrightarrow [granule\_paint(Col)] & hascar_{Ag}(C) \\
\wedge colour(C, \_) & & \wedge colour(C, Col) \\
 & & \\
\mathcal{B} location_{Ag}(R, X, Y) & \leftrightarrow [granule\_move(D)] & offset(D, Dx, Dy) \\
\wedge \mathcal{B} been(L) & & \wedge((hascar_{Ag}(C) \\
\wedge \mathcal{B} locale(E) & & \wedge location(C, Cx + Dx, Cy + Dy)) \\
\wedge((\mathcal{B} hascar_{Ag}(C) & & \vee calB \neg hascar_{Ag}) \\
\wedge location(C, Cx, Cy)) & & \wedge \mathcal{B} location(R, X + Dx, Y + Dy) \\
\vee \mathcal{B} \neg hascar_{Ag}(C)) & & \wedge \mathcal{B} been([(X, Y)|L]) \\
 & & \wedge locale((R, X + Dx, Y + Dy), Enew) \\
 & & \\
\mathcal{B} location_{Ag}(R, X, Y) & \leftrightarrow [granule\_exitenter] & exit\_from\_leads\_to((R, X, Y), (R', X', Y')) \\
\wedge exit\_at(R, X, Y) & & \wedge((hascar_{Ag}(C) \\
\wedge \mathcal{B} been(L) & & \wedge location_C(R', X', Y') \\
\wedge \mathcal{B} locale(E) & & \vee \mathcal{B} \neg hascar_{Ag}(C)) \\
\wedge((\mathcal{B} hascar_{Ag}(C) & & \wedge \mathcal{B} location(R', X', Y') \\
\wedge location_C(Cr, Cx, Cy)) & & \wedge \mathcal{B} been([(X, Y)|L]) \\
\vee \mathcal{B} \neg hascar_{Ag}(C)) & & \wedge locale((R', X', Y'), Enew) \\
 & & \wedge \mathcal{B} locale(Enew)
\end{array}
$$

Figure 1: Dynamic Modal Logic Specification of Agent Actions

CSCW-tasks for semi-autonomous agents in the CEC GOAL project (Esprit 6471) [14]. In the Medlar Animator it has been used to specify structured interactions between agent and reality which constitute the granular actions of an agent, as these necessarily involve inter-process communication (i.e. an agent process with the reality according to the Cartesian model). The notation can also be used for specifiying agent–agent interactions.

Figure 2 illustrates the de Greef specification for a granular move. Note that for an agent to perform a `granule_move`, it is given (what it believes to be) its location and a goal location, from which it computes the 'optimal' direction in which to move. The agent then sends a `request` to reality to 'do' this action.

Reality computes whether or not this action can *really* be done, irrespective of the agent believing it can be done. If the move can be done and it succeeds (actions can be failed on a pre-defined – if for an example an agent is not allowed access to a certain area – or on a probabilistic basis), then reality takes four steps. First, it updates the actual environment and agent location to realise that the agent has moved in the required direction. Second it checks whether or not the agent was carrying a car, and if so, then the environment and car location are updated to realise that the car has moved with the agent. Third, it computes the new locale that the agent can sense, and fourth, it sends an `inform` message to the agent containing an acknowledgement and its new locale. The agent updates its beliefs accordingly: what it believes its new location to be (which may be different from the actual location), its locale that it can sense (which can also be scrambled to simulate faulty sensors), and it calculates a new local potential field based on what it can now sense (see the next section for a discussion of how the potential field method is used for (local) path planning).

If the action cannot be performed for some reason, then the agent is sent an appropriate

```
agent, reality:  granule_move ==
agent:  nameis(name),
agent:  compute_direction(location, goal, direction),
agent:  request(move, name, direction) ⟶ reality,
( reality:  ok_move(name, direction, environment),
  ( reality:  move_succeeds,
    reality:  update_env(name, direction, environment, new_env),
    ( reality:  hascar(name, car),
      reality:  update_env(car, direction, new_env, the_env)
    | reality:  not(hascar(name, car)),
      reality:  the_env = new_env
    )
    reality:  extract_locale(name, the_env, new_locale)
    reality:  inform(move_ok, new_locale) ⟶ agent,
    agent:  update_loc(location, direction, new_location),
    agent:  set_locale(new_locale),
    agent:  calculate_potential_field(locale, local_pf),
  | reality:  not(move_succeeds),
    reality:  inform(move_nok, _ ) ⟶ agent
  )
| reality:  not(ok_move(name, direction, environment)),
  reality:  inform(move_nok, _ ) ⟶ agent
).
```

Figure 2: de Greef Specification of Granular Move Action

`inform` message and no beliefs are changed, although the agent can know that this action has failed and may seek to take remedial action which is not specified here. For example, the agent might believe that it does not have a car, and tried to move where there was a car. The illegality of the action could be distinguished from failure of the action and cause belief revision.

3 Realisation

3.1 The Reasoning Layer

An intended application of the simulator is to validate the behaviour of logical theories about practical reasoning in experimental environments with sufficient realism and to promote the integration with production engineering research [13, 18]. However, work of the Esprit Medlar project [8], which allows the symbolic execution of multi-model theories, is not yet integrated. It requires a collection of abstract rules like "if an agent seeks goal G and believes that, provided pre-condition P holds, doing action a will possibly achieve G, then the agent may seek P" to be interpreted in the concrete environment of the simulation.

As special reasoning methods appropriate to the environment can be integrated, we have provided a reasoning module for dynamic path planning across workspaces in the presence of a potential field (see [12]). This field is provided in the reactive base.

3.2 The Reactive Base

The set of primitive senses and low-level actions specified for the reasoning layer constitute the reactive base. The agent's senses are reflected in the locale of the reactive base. Its actions change the relationship between the robot, its tasks, and the environment. Both are visible in the control interface (see below).

While these senses and operations form a non-decomposable layer within the simulator itself, they also possess modifiable external properties and attributes. In particular, the separation of cause and effect (due to the Cartesian model of the simulator) permits granular actions to be implemented with arbitrary behaviour and complexity. The result of a painting action, for example, might be dependent on a robot's location, and the type of object (if any) in its possession. The implementation of the action could employ a neural-net based technology, allowing the robot to learn and recognise the appropriate location/object to attain for the desired outcome.

The freedom exists to combine a variety of computational methods within each action (and the reactive base as a whole), opening the way for further experimentation in the software techniques underlying robot technology. In the case of the paintshop scenario, there is currently one specialised sense, a potential field direction, providing for granular-navigation of the environment.

Each agent has access to two potential fields: the first is static, and is calculated from the agent's memory of a location (the 'learning' aspect), the second is dynamic and is based upon the agent's perceived immediate surroundings at the time of the action. An overall 'contour map' of the terrain (see for example figure 3) is achieved by combining these two fields (the details of the process are described in Appendix B), and the agent uses this information to navigate through the micro-world.

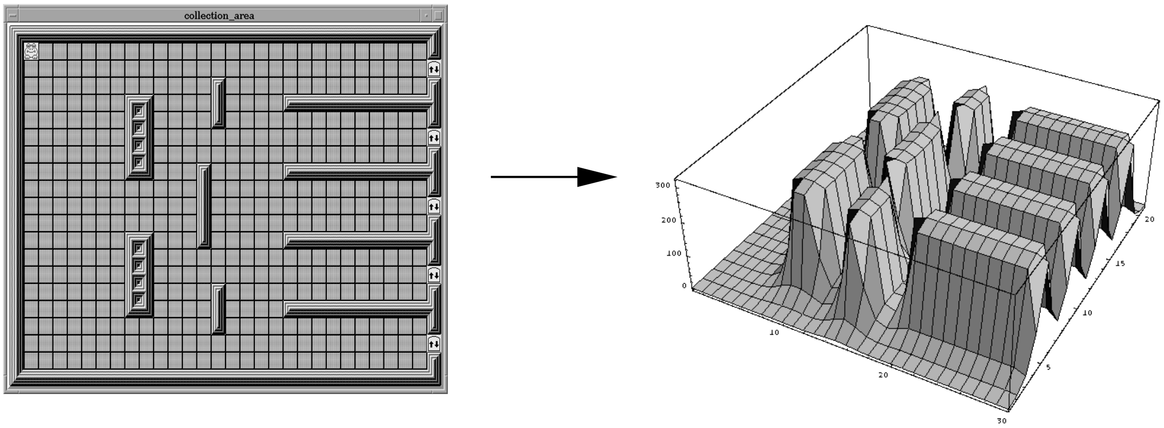

Figure 3: *Potential Field for an Example Workspace*

The potential field approach is normally used in static situations, in which all the positions of all the obstacles are known in advance, and this is used to determine an optimal path. In the PaintShop scenario, the obstacles are blocks, cars, and other agents — but the other agents can be moving independently, they will move cars about, and indeed the experimenter can arbitrarily move, add and delete cars and blocks. In short, the situation is dynamic and an optimal path cannot be pre-computed as it could lead to crashes or deadlock.

Our solution is to make movement context-sensitive. The reactive base provides the agent with a global potential field for a workspace based on the agent's knowledge or memory whenever it enters that workspace. It also provides a local potential field based on the cars and agents (which are treated as obstacles) that are sensed in the agent's locale. The sum of the global and local potential fields at each co-ordinate gives the total repulsive force at that co-ordinate, the lowest repulsive force indicating the optimal direction at that moment. An agent then moves from its current location to a goal location one 'step' (or granule move) at a time.

Note that there is a direct correspondence between each granular action, and the concept of an elementary operation in real-life autonomous robots: actions which manipulate effectors. In such robots, each elementary operation is governed by a module which determines in what way the operation is performed (a 'strategy module').

3.3 Agent Implementation

The agent shell has been designed modularly, with each module implementing a particular agent capability, user interface requirement, or other low-level function. Each module has been implemented in a language considered most appropriate to the required functionality, so the agent state, capabilities and inter-process communication are implemented in ICL/ECRC ECL^{i}PS^e constraint Prolog, visualization and the graphical user interface are implemented in Tcl/Tk, and heavy numerical computation for the potential field path planning is implemented in C.

The state of an agent is represented by the values of its attributes. In the Medlar PaintShop scenario, the agents attributes are its name, location, locale and 'hascar'. The locale is the agent's surroundings as determined from the environment, i.e. what the agent 'visually senses'. The hascar attribute is what the agent 'tactilely senses', i.e. whether it is carrying a car or not. Note that this is the *belief state* of an agent, the *actual state* is kept by reality. An agent also has two implementation-dependent attributes, which are an internet address of the form:

`HostName/PortNumber, e.g. 'medlar.doc.ic.ac.uk'/5000`

which the agent uses to listen for incoming messages, and a *colour* which is used to aid visual identification.

For the paintshop scenario, the following is the minimal set of high-level actions of which each agent should be capable:

- *sense*: each agent can sense its immediate surroundings, called its *locale*, and whether or not it is carrying a car; a faulty sensor can be simulated;
- *act* (physically): an agent can physically pick-up, paint or drop a car, and can physically move from one location in the environment to another; completing an action may result from performing a sequence of *granular* actions;
- *communicate*: agents can communicate with each other by message passing, possibly, but not necessarily, using cooperation protocols;
- *reason*: the process whereby an agent can reason symbolically about its next action.

Currently, the agents can only sense and act (the implementation of communication and reasoning is discussed as further work in section 5). The basic agent architecture, showing the main implementation modules, is illustrated in figure 4. Note that the Agent Planner exists only in a very rudimentary form. It will be developed when we lift the architecture given here to something more like the Archon agent architecture [17] to include a situation assessment module for pro-active decision making, and an acquaintance model for maintaining information about other agents in the micro-world.

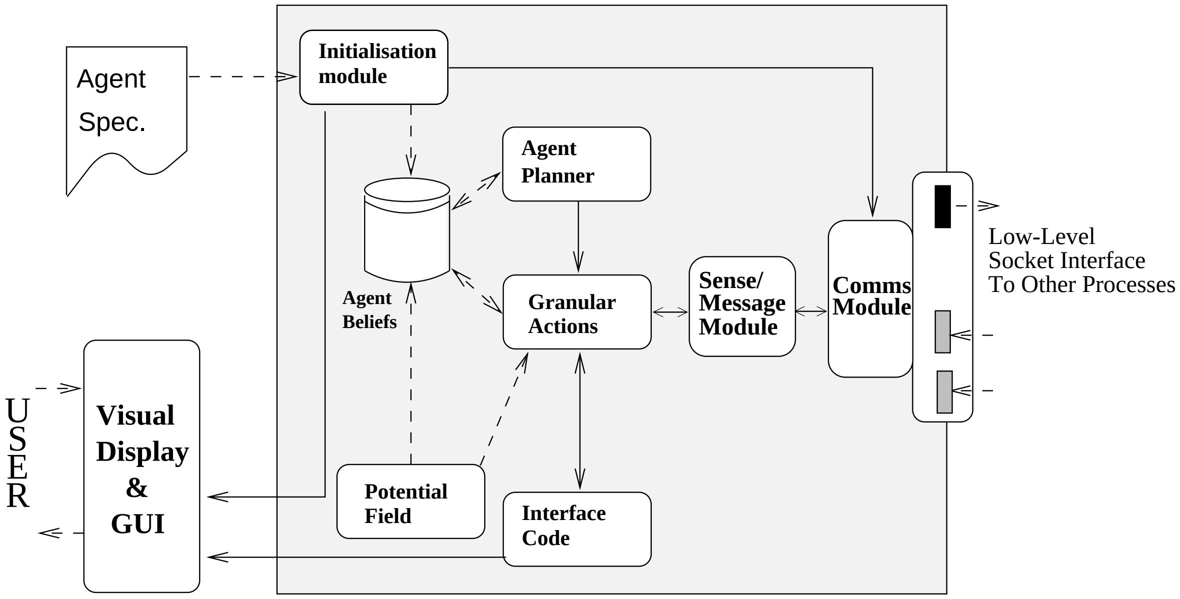

Figure 4: *Agent Architecture*

For example, components of the agent gui include a display of its locale, a control interface for invoking granular actions, and an event recorder. These are illustrated in figure 5.

3.4 Reality Implementation

The Reality shell is also implemented in a combination of Prolog and Tcl/Tk. There is Prolog module which updates the state of the micro-world according to actions requested by agents: this is coupled with a Tcl/Tk module which displays the environment and changes to the environment on the screen. There is another Prolog module which maintains an agent directory. This records and displays their actual states (irrespective of an agent's internal beliefs) and can act as a name server for inter-agent communication. It is also coupled with a Tcl/Tk module which displays the agent information on the screen. Finally there is a Tcl/Tk module which implements the reality gui, allowing the user to interact with simulated microworld. Reality also uses the Prolog library for inter-process communication, so that reality and agent processes share a common mechanism for communication.

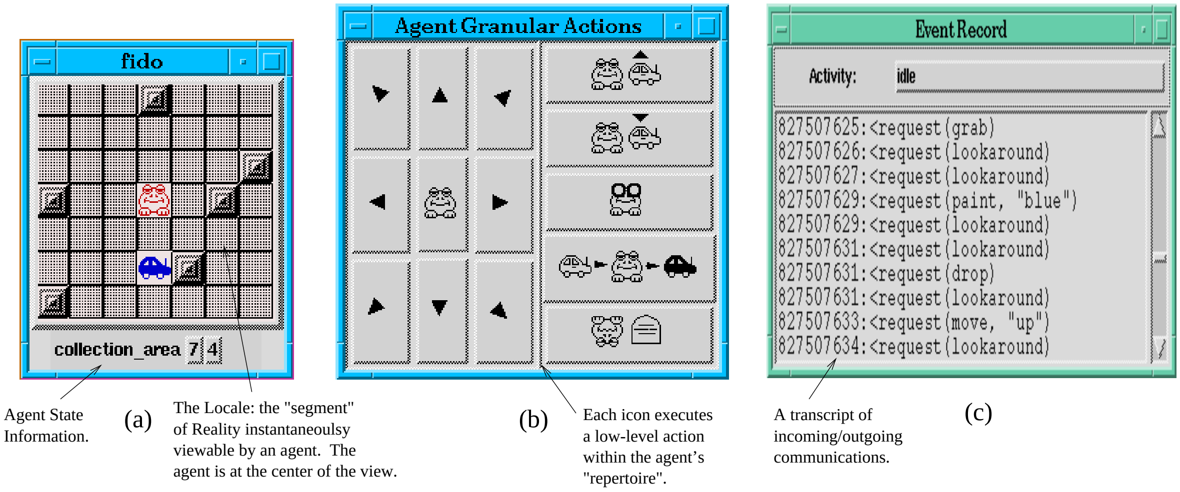

Figure 5: *A Tcl/Tk Agent gui*

3.5 Animator Architecture

In the animator, each agent is executed as a kind of 'active object', i.e. an 'agent' is a self-contained, concurrently executing process that encapsulates some notion of a state, communicates with other processes by message passing, and is primarily pro-active in determining its next action and changing its own state. It runs on its own host machine and is connected to reality by a network. Reality also runs as a separate Unix process identified by an Internet address and port number on its host machine. (This information is required by agent processes to send messages to reality.) On initialization, it requests a environment specification, initialises the micro-world, and displays this to the user. It then enters its event loop, and waits for agents to register, processes action requests from those that have registered, and responds to user actions.

Two-way reality–agent and agent–agent communication channels are implemented via the inter-process communication library. This package provides high-level predicates which enable agent and reality processes, executing on different machines connected to different networks, to be connected via bi-directional sockets and to pass messages via datagram connection. This communication model is essentially a classical client/server one in which the reality shell, housing the centralised data, performs the role of server. The agent processes are the clients, whose requests to reality result in 'state transitions' within the associated process. However, the name server functions enable reality also to act as a 'broker' which enables peer–peer connections to be established between agents.

Usually, there will be several agents operating within a simulation, each of which will be performing (or trying to perform) actions in real time. Successful actions will have an immediate effect on the environment from every agent's point of view. This might mean that two agents are trying to move onto the same location, which is, of course, not possible. Thus there is an issue of resource control to be addressed, with the environment itself the resource, and a requirement that each agent has 'fair' and exclusive access to the resource.

The issue is resolved (in part) by the process model of interaction: the environment resides within the reality shell, and the corresponding process is under the shell's control. The shell maintains communications channels to each agent, and accepts messages from these channels at any time. However, each message is processed individually; other messages are queued (in the order they arrive), until the current transaction is completed. Assuming the 'liveness property' of the network connecting reality and agents, any agent request will be processed eventually. The actions of the user are treated in a similar fashion, but they may 'preempt' the agent requests to an extent, since they are generated within the reality process itself.

Note that network delays may have an effect on the perceived response times of the animator: a fast-running, local agent process will be able to place more requests in the communications queue than an equally fast-running remote one. This is an unavoidable feature of the architecture, but the communication latency can be seen as a characteristic of the simulated agent: a remote agent is just a 'slow' agent.

4 A Working Demonstration

The typical set-up and visual display of running a simulation on the testbed is illustrated in figure 6. However, it isn't necessary to have separate machines for each agent and for reality; as each is a Unix process they could all be run on one processor. As the inter-process communication is required regardless of how many machines are used, and the simulation runs faster with more machines (over local networks), we feel it is more interesting, and adds an extra element of verisimilitude, to distribute the processes across various machines.

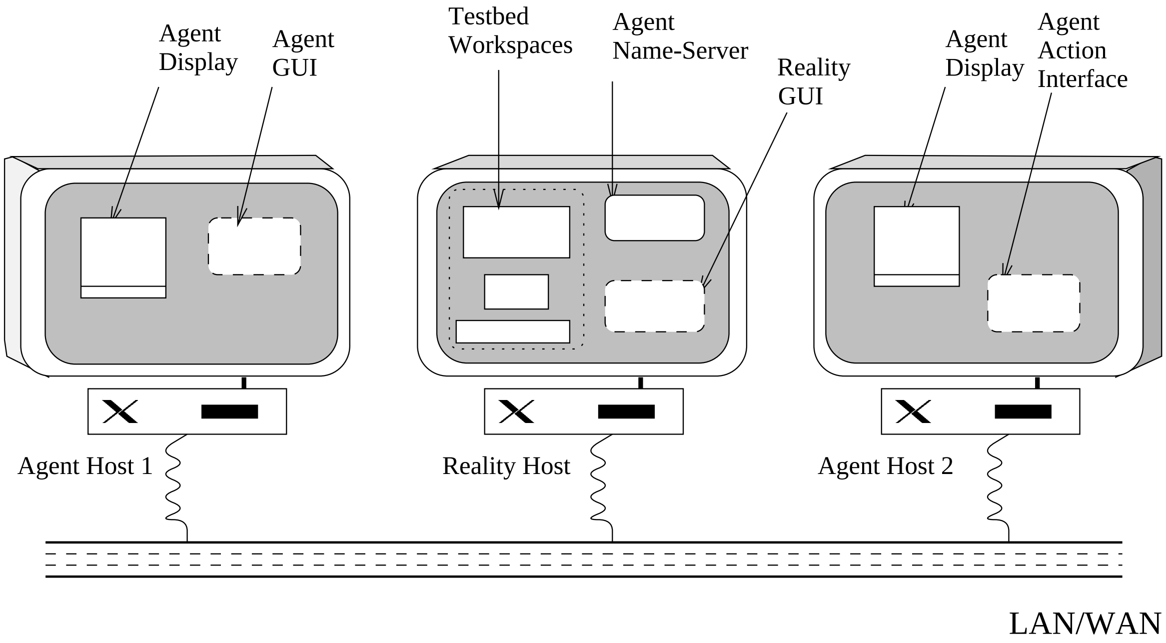

Figure 6: *Animating a Simulation on the Testbed*

Running the simulator is performed as follows. The reality shell is started from the Unix command line: there is an option to specify the environment workspace filenames. If this option is not exercised, the user (or rather, experimenter) is presented with a dialogue which is used to select the workspace specification files from any accessible directory in the file system. When the user has finished, the environment is initialised, drawn on the screen, and the reality gui displayed to the user. The process enters its event loop.

Reality must be started up first, after which the agents can be started up, although agents can be added at any time during a simulation. To add an agent, an agent shell is started from the Unix command line: again, there is an option to specify a parameters file describing the agent's inital state (name, location, etc.). If unspecified, the new agent process queries the reality process to obtain data about the size and names of workspaces in the environment. It then initiates a dialogue with the user using this information. The user must complete the dialogue by specifying a name, location, (workspace, x and y coordinates), and other options. This is then sent to reality to *register* the agent. If the registration is successful three things happen: the agent is drawn on the reality display in the workspace; the agent's details are added to the agent directory maintained by reality; and a new window is opened on the agent's display showing its initial state and 'sensed' locale.

Each agent process comes with an optional gui which allows access to granular actions, providing a limited form of proactivity and autonomy in the absence of a fully implemented situation assessment module. Currently, though, most user interaction with the simulated micro-world is through the reality gui. This allows the user to change the environment dynamically, add, move, paint and delete cars, move and 'kill' agents, issue instructions to agents, and query agents about their plans and actions. A facility to trace the movements of agents can also be selected through the reality gui. Provision of further functionality and modes of interaction are described in section 5.

We now describe a sample run in the Medlar Paintshop. There are three workspaces in the environment: a collection area, painting area, and delivery area. Cars will be produced from a production line (external to the simulation) and appear in the collection area. The user, using an `Instruct` button on the reality gui, can then instruct agents to collect a car, paint it a particular colour, and deposit it in the delivery area. The agents undertake the task through a standard decomposition into sub-tasks and execute these, as discussed earlier, by sending requests to reality to perform each granular action that contributes to completing the sub-task.

A typical execution of such tasks is illustrated in figure 7. As the agents execute the necessary granular actions, the environment display and the agent display are both updated to reflect the effect of the action 'in reality' and the agent's 'sensory perception' of that effect. (Unfortunately this can't be demonstrated without video!) An additional effect is noticeable in the environment: the movement trace option has been selected, resulting in a trail behind each agent. It can be seen how their movement has been affected by the potential fields: an agent steers clear of the obstacles in a room unless it can sense its goal in its locale. In the confines of the collection area, the repellent effect of agent proximity is also noticeable. Also, the agents may have been attempting to collect the same car. In such a case, the first agent to the car gets it: the second agent has to find another car, which illustrates how an agent can recover from 'real world' events which disrupt its plan and are beyond its control.

Having collected a car, an agent then carries it to the painting area. The process of moving between workspaces requires an agent to chose a suitable exit and adjust its potential field calculations, using the 'remembered' global potential field for the given workspace. The agents now also view cars as obstacles: they cannot move over them (as they are 'holding' cars already), so these are now included in (local) potential field calculations. The painting location itself is divided into three areas; each corresponding to a 'bay' in which cars are

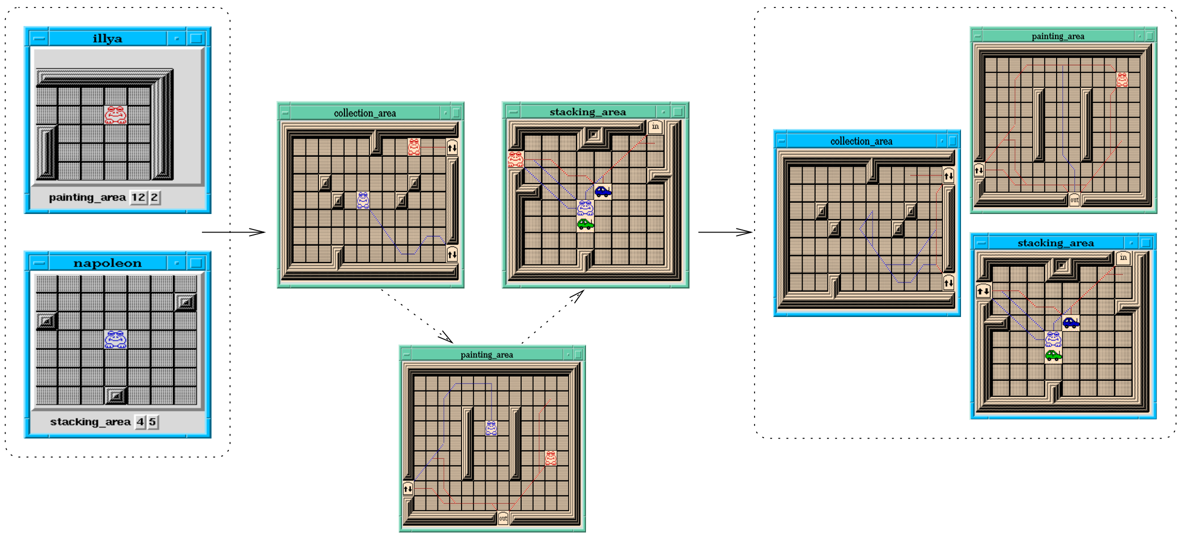

Figure 7: *Snapshots of Reality with Two Agents Collecting, Painting and Delivering Cars*

painted red, green or blue. A specific coordinate is chosen from bay area, and the painting action is performed: the 'car icon' in the agent window changes colour accordingly.

The final sub-task consists of depositing the freshly painted cars in the stacking area. For the moment, the empty coordinate nearest the location's center is chosen. Again, the field interaction between obstacles, cars and other agents is pronounced in this location, resulting in an interesting agent trail. Once the coordinate is reached, the agent deposits the car and returns to the position it occupied at the beginning of the task.

5 Summary, Further Work and Conclusions

We have described a system in which multiple rational agents can be embedded in a microworld containing various objects and locations. This system can be seen as:

- a *simulator*: we can model factory situations and manufacturing scenarios like the Medlar Paintshop, in which there are a number of mobile agents given tasks to complete, and equipped with some kind of 'rationality' in order to go about completing tasks;
- an *animator*: the intended user's specification of an agent is a theory in a multi-modal dynamic logic: agents can differ according to their goals, skills and capabilities but the same technique of logical animation may be used to explore the dynamic behaviour of each individual agent's specification;
- a *testbed*: the user can specify different environments and agent behaviours, and can control exogeneous events: in this way the user can perform experiments to investigate how different agent theories effect task accomplishment, performance, cooperation, etc.

The system comprises two main shells: one for a simulation of 'reality' one for (each of) the agents occupying that 'reality'. The system has been implemented using ECL^{i}PS^e Prolog (v3.4.5), Tcl/Tk (v7.4/v4.0), and C, with the total amount of code coming to about 350K. The system runs on Sun SparcStations under Unix SunOS 4.1.3. Particular features of the system are the use of separately executing processes for each agent and for reality, the separation of an agent's beliefs and senses from the actual state of the world and its visualisation as separate displays, and the inter-process communication which enables actions to be performed and the effects to be 'sensed'.

Also worth noting is the hybrid combination of higher-level symbolic reasoning about actions (e.g. move to a goal location), with lower-level sub-symbolic mathematical computations on potential fields to decide in which direction it is currently optimal to move, which is used to simulate rational behaviour. There are analogous combinations of symbolic and sub-symbolic computation which have been used in simulation, such as the introduction of the numerical solution of differential equations, reduction of a planning search space through constraint reasoning, or neural nets for trained robot arm actions.

The current state of the system is that it is stable and usable: however, some of the of the critical ingredients have only been partially completed and are yet to be integrated with the overall system. This includes:

- *specification and animation*: if we have an agent specifcation in a multi-modal logic then we seek to animate it. We have a prototype implementation of an automated reasoning system for each of the 15 normal modal logics (a preliminary description of this system is given in [8]). We are looking to lift this system to be able handle multi-modal logics in the style suggested in [7], which will then be used as the core of our agents' situation assessment modules;
- *inter-agent communication and cooperation*: the library package for inter-process communication between an agent and reality can also be used for peer–peer communication between agents. We are intending to take advantage initially of the cooperation protocols defined in [14], but there are many issues in message passing and understanding, negotiation strategies, and agent authority and authorization that need to be addressed (cf. for example, [3, 10]);
- *user interface*: the reality gui needs to be extended, to provide: additional user commands to agents beyond 'instruct' and 'query' – in fact the user should have full access to the entire message passing and protocol range understood by agents; 'superuser' functions to freeze the experiment, retract actions, etc.; and the ability to write scripts that issue standing orders to agents, cause events to occur at particular times or regular intervals, and so on.

Although designed as a testbed, we have produced a system with many features of a full scale application of cooperating rational agents. Industry is experiencing a paradigm shift from hierarchical to decentralised management and production. This shift is already well established in the flat management structures of big organisations. The Medlar Paintshop scenario carries this powerful development into the realm of autonomous production machines. Substantial improvements in process automation can be achieved if planning goals can decouple decision making to local levels and provide clear information for local human interaction [18]. The testbed we have designed and implemented makes it possible to use the logics of practical reasoning to describe the goals and capabilities of intelligent production machines, to animate a production engineering scenario in which software agents collaborate to realise the intended goals, and to exploit high-level logical reasoning to facilitate human interaction in the integration of semi-autononous production systems.

Acknowledgements

We would particularly like to thank Mathieu Capcarrere for his contribution to the design and implementation of the Testbed. We would also like to thank Nat Pryce for permission to use icons from his Tcl/Tk game `Budge`, and Keith Barrett for discussions on Descartes (although any misinterpretation of Cartesian philosophy remain our own).

Lloyd Kamara and Jeremy Pitt were supported by the CEC Basic Research Action Medlar II (Esprit 6471). Further information on Medlar II can be found via URL:
`http://medlar.doc.ic.ac.uk/lar/medlar2.html`.

References

[1] W. Atkinson and J. Cunningham. Proving Properties of a Safety-Critical System. *IEE Software Engineering Journal*, 6(2):41–50, 1991.

[2] B. Burmeister, A. Haddadi, and K. Sundermeyer. Generic configurable cooperation protocols for multi-agent systems. In *MAAMAW'93, Neuchatel*, 1993.

[3] M. K. Chang and C. C. Woo. SANP: A communication level protocol for negotiations. In Y. Demazeau and J. Muller, editors, *Decentralized A.I. 3*. Elsevier/North Holland, 1992.

[4] P. Cohen and H. Levesque. Persistence, intention, and committment. In P. Cohen, J. Morgan, and M. Pollack, editors, *Intention and Communication*. MIT, 1990.

[5] M. C. Costa, R. J. Cunningham, and J. Booth. Logical Animation. In *Proc. of 12th International Conference on Software Engineering*, pages 144–149, Nice, 1990. IEEE Computer Society Press.

[6] J. Cottingham, R. Stoothoff, and D. Murdoch (translators). *The Philosophical Writings of Descartes*. CUP, 1984.

[7] J. Cunningham. Investigations into modular and concurrent deduction. CEC Esprit Project Medlar (Esprit 6471) Periodic Progress Report 2, Department of Computing, Imperial College of Science, Technology and Medicine, 1994.

[8] J. Cunningham and J. Pitt. Towards model building in multi-modal logics using **KE** and constraint satisfaction. CEC Esprit Project Medlar (Esprit 6471) PPR3, Department of Computing, Imperial College of Science, Technology and Medicine, 1995.

[9] P. de Greef, K. Clark, and F. McCabe. Toward a specification language for cooperation methods. In *GWAI'92 16th German AI Conference*, volume 671, pages 313–320. Springer Verlag, 1993.

[10] T. Finin and R. Fritzson. KQML — a language and protocol for knowledge and information exchange. In *Proceedings of the Thirteenth International Workshop on Distributed Artificial Intelligence*, pages 126–136, Lake Quinalt, WA, 1994.

[11] S. Hanks, M. Pollack, and P. Cohen. Benchmarks, testbeds, controlled experimentation and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.

[12] J.-C. Latombe. *Robot Motion Planning*. Kluwer, 1991.

[13] J. Pfalzgraf, U. Sigmund, and K. Stokkermans. Comparison of two approaches to modelling robotics scenarios. CEC Esprit Project Medlar II (Esprit 6471) Periodic Progress Report 1, Department of Computing, Imperial College of Science, Technology and Medicine, 1993.

[14] J. Pitt, M. Anderton, and J. Cunningham. Normalized interactions between autonomous agents: A case study in inter-organizational project management. *Computer-Supported Cooperative Work*, to appear.

[15] J. Pitt and J. Cunningham. Blending the balkans: One approach to integrating and combining inference engines. CEC Esprit Project Medlar (Esprit 6471) PPR3, Department of Computing, Imperial College of Science, Technology and Medicine, 1995.
[16] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
[17] T. Wittig, editor. *ARCHON: An Architecture for Multi-Agent Systems.* Ellis-Horwood, 1992.
[18] H. Zetlmayer and C. Martin. Simulation based rescheduling and synchronisation of complex multi-stage production plants. In V. Orpana and A. Lukka, editors, *Production Research 1993.* Elsevier, 1993.

Appendix A: Specification of Granular Actions

The language proposed by de Greef *at al.* [9] can be used to specify protocols which include conversations between two or more agents, to blur the distinction between 'client' and 'server' agents, and to be more specific about what data is being passed between agents, and when. The language inherits (most) features of a logic programming language, and also provides for the notion of an agent, communication mechanisms, and sets.

The exchange of a message is written:

$$\textit{sender\_agent: message}(\textit{arg}_1, \ldots, \textit{arg}_n) \rightarrow \textit{receiver\_agent}$$

Similarly, messages are sent to a set of agents thus:

$$\textit{sender\_agent: message}(\textit{arg}_1, \ldots, \textit{arg}_n) \rightarrow \textit{Receivers}$$

and received from a set of agents by writing:

$$\textit{Msgs} = \{(r, \textit{arg}_1, \ldots, \textit{arg}_n) \mid r\text{: } \textit{message}(\textit{arg}_1, \ldots, \textit{arg}_n) \rightarrow \textit{sender}\ \textbf{within}\ \textit{timeout}\}$$

The value of *timeout* is measured on *sender_agent*'s clock, and ensures the instruction will terminate because there is no guarantee that all receiving agents will respond.

Message passing can be included in "plan schemas" which assign sub-tasks to agents or agent variables. For example, a plan schema to calculate a percentage might be:

$$\begin{aligned}
&\textit{agent: percentage_of(part, whole, percentage)} = \\
&\qquad \textit{agent_database(divide, Agent1),} \\
&\qquad \textit{agent: request(divide, part, whole)} \rightarrow \textit{Agent1,} \\
&\qquad \textit{Agent1: inform(divide, fraction)} \rightarrow \textit{agent,} \\
&\qquad \textit{agent_database(multiply, Agent2),} \\
&\qquad \textit{agent: request(multiply, fraction, 100)} \rightarrow \textit{Agent2,} \\
&\qquad \textit{Agent2: inform(multiply, percentage)} \rightarrow \textit{agent.}
\end{aligned}$$

Assuming *agent* has no mathematical skills, it looks in an agent database for an agent offering the appropriate service, and sends a request to that agent asking it to perform the operation on the accompanying data. Having done so, that agent returns the result. This is done for both the divide and multiply to calculate the percentage of the original input.

The main drawback of this specification language is that it does not have a semantics, although de Greef *et al.* believe "a formal semantics can be defined". We have found the notation useful in writing 'formal', precise specifications of 'dialogues' whose intended meaning, at least, is intuitively clear.

Appendix B: Calculating Potential Fields

Two constants need to be defined: η, a scaling factor on the repulsive potential force at any point; ξ, a scaling factor on the attractive potential force at any point, and ρ_0, the *distance of influence* factor, beyomd which a C-region has no repulsive effect on an agent.

The procedure for calculating the global potential field is as follows. For each coordinate q in the workspace:

1. *Derive the C-regions:* for the global potential field this means locating the blocks (but not the walls) in the workspace. Each separate block yields a *C-region* which is represented by a set of coordinates;
2. For each coordinate q in the workspace, *calculate the repulsive potential at q:* this done by applying the following function to each C-region C:

$$\mathcal{U}_{rep}(q) = \begin{cases} \frac{1}{2}\eta\left(\frac{1}{\rho(q)} - \frac{1}{\rho_0}\right)^2 & \text{if } \rho(q) \leq \rho_0 \\ 0 & \text{if } \rho(q) > \rho_0 \end{cases}$$

   where $\rho(q)$ is the distance from q to the nearest coordinate in the C-region given by:

$$\rho(q) = \min \ \|q - q'\| \ \text{ for all } q' \in C$$

To calculate the local potential field for an agent the procedure is:

1. *Derive the C-regions:* as above but for the local potential field this means identifying the cars and other agents sensed in its locale and treating these as the C-regions;
2. For each coordinate q in the locale: *calculate the repulsive potential at q.* As above biut for the local potential field identified as $\mathcal{U}_{rep_L}(q)$.

An agent moving from a current coordinate q to a goal coordinate q_{goal}, given the particular locale at the time (the global potential field will have already been calculated), will move to a coordinate q' adjacent to q as follows:

1. *Calculate the local potential field.*
2. For each coordinate q' adjacent to q, *calculate the attractive potential of the goal coordinate.* This is derived from the following equation:

$$\mathcal{U}_{att}(q') = \frac{1}{2}\xi\rho^2_{goal}(q')$$

   where $\rho_{goal} = \|q' - q_{goal}\|$ is the distance of the agent from the goal.
3. For each coordinate q' adjacent to q, *sum the values obtained from the previous calculations*:

$$\mathcal{U}(q') = \mathcal{U}_{att}(q') + \mathcal{U}_{rep}(q') + \mathcal{U}_{rep_L}(q')$$

The coordinate q' with the lowest value of $\mathcal{U}(q')$ is the optimal coordinate to move into, *under the circumstances.* Note though that certain other factors are involved (like has the agent 'visited' this coordinate before in this 'move'), in order to avoid problems with concave obstacles which can create local minima.