Distributed artificial life toolkit

design

Vlad Mereuta

January 31, 2002

Contents

Chapter 1

The problem revisited

1.1 Problem overview

There are many arguments supporting an agent-based architecture. Some of the arguments for this particular approach include: [3]

- problems are physically distributed
- problems are widely distributed and heterogeneous in functional terms
- complexity of problems dictates a local point of view
- systems must be able to adapt to changes in structure or the environment

Most of the artificial life simulations are entirely suitable for the agent based approach as the researchers are trying to study the behaviour of several autonomous programs.

The problem in designing a toolkit for this kind of simulations is not to implement the agents or the senses themselves, but to lay down the foundations for someone else to do it. This has to be accomplished in such a way that an experimenter could easily extend on it without being constraint by design of the toolkit.

We need to find acceptable abstractions for agents, senses and actions and a way of measuring resources taken up by agents in order to be able to distribute the workload.

1.2 Design overview

1.2.1 Design choices

1.2.2 Discrete vs. time-frame simulation

There are two main methods used in agent-based simulations. One is to divide the simulation time into set time-frames. Each action will need a certain number of these frames to be executed. The other method is to use a discrete simulation, in which actions trigger other actions.

The discrete simulation is not suitable for distributed processing. The processing loads of various modules/agents would vary greatly at different stages of simulation, which would render the task of distributing the processing requirements across several physical location extremely difficult. The varying speed of different programs and the nature of computer networks has the potential of creating speed-imbalances in between agents.[1]

This is why this toolkit will support only the time-frame simulation mode.

[1] consider the case in which two agent attempt to use an object in the world at the same time; even if one of the agent has the physical advantage of being stronger and quicker, the weaker agent could still outrun the stronger one in manipulated the object if the stronger agent runs on machine which is more heavily loaded than the other one, or if due to network lag the action requests of the stronger agent are received later.

1.2.3 Peer to peer vs. client/server

Another choice that has to be made from the start deals with the way the programs in different physical locations interact. One possibility would be to have self-sufficient programs, containing locally all the information needed to run the simulation for the data set present. The programs would only communicate when they need access to data managed by a different program. This means that each program would need to have a copy of the world being simulated, which in turn requires every single program to be notified with every single change that occurs in the simulated world. This communication overhead would be added to the normal inter-program communication. Another issue raised by this approach is resource arbitration. If two or more programs access the same resource a way to prioritise this would be needed. Managing this in a completely distribute fashion would be difficult and quite probably not efficient.

A better choice is to store the shared resources (i.e. the top-level information about the simulation world) in a single location, which is to be used by all other programs. This central location can also perform a fair arbitration when programs compete for the same resources.

1.2.4 Design

The following sections will details the main modules which will be the components of the toolkit.

Each component has a unique perspective on the system viewing the other components as black boxes. For example, the client does not and should not know how the server works. This isolates the implementation details of one from another, making public only the known communication interface.

The server

The main task of the server is to provide a mean for resource arbitration and a central location from where the simulated world is managed.

The server will also act as a communication channel as it will be the only component aware of the location of all other components. An agent for example will not be able to communicate directly with another agent, all the communication will have to go through the server. While this might seem as a potential bottleneck, it gives the server the ability of creating a complete image of what is happening in the simulation, which is a vital feature if any results are to be gathered from the process.

The client

The client is the “active” part of the toolkit. It contains one or more agents which are the most important part of simulation as their behaviour as individuals or collectives is studied by the person running the simulation.

Communication

Though it is not a module per se the communication is extremely important, as it is the mean of interaction in between all separate modules.

As we are aiming for portability and machine and language independence, the communication protocols and procedures will have to be well defined. If the client for example would have to be re-implemented in a different language the design of the new client would have to be based on the same communication.

The observer

The main task of the observer is to monitor and log the experiment. It can use data provided by the server and possibly extra data provided by the client or agents themselves. The task of the observer can range from displaying the world as it changes to interpreting complex social relations in between agents. The actual implementation is expected to vary widely in between different experiments. The toolkit should provide a thin frame for handling communication with the server and a sample map display mode.

Map editor

All simulations need to specify the individual characteristics of the environment which is to host the simulation and a starting position for their entities. All of this can be done with the map editor.

Chapter 2

The map editor

2.1 Purpose

The purpose of the map editor is to specify the main parameters of the simulation environment and possibly the initial distribution of the available agents.

The result should be an XML file containing the following pieces of information:

- the size of the world
- the positions of various entities

The map should have a graphical editor for specifying these things, but due to the nature of the toolkit it will probably have to be customised for most experiments in order to provide the required functionality.

2.2 Design

2.2.1 The map DTD

In order to keep the toolkit flexible, the map should be able to specify n-dimensional world. Each *map* has a set number of *dimensions* and a set of *dim*s, each of them specifying a the span of the map in that dimension. For example, on a two dimensional map, the value of *dimensions* is *2* and there should be two *dim* entries with attributes of *1* and the span of the map on the first axis and *2* and the span of the world on the second axis.

The entities are enumerated one after the other, each one having a unique *id*, a *name*, a *colour* and an enumeration of positions on each of the axis. The name is the only attribute that should be taken into consideration by the server as the rest of the attributes are purely cosmetic for the use of the map editor.

Here is the DTD for the map:

```
<?xml version="1.0"?>

<!-- the map has 0 or more entities and at least one dimension -->
<!ELEMENT map (entity*, axis+)>
<!ATTLIST map dimensions CDATA #REQUIRED>

<!ELEMENT axis EMPTY>
        <!ATTLIST axis id CDATA #REQUIRED>
        <!ATTLIST axis value CDATA #REQUIRED>

<!-- each element has a position in the world specified as at least one dim -->
<!ELEMENT entity (pos+)>
        <!ATTLIST entity type CDATA #REQUIRED>
        <!ATTLIST entity colour CDATA #IMPLIED>

<!-- dimension has an axis is and the location -->
```

```
<!ELEMENT pos EMPTY>
<!ATTLIST dim id CDATA #REQUIRED>
<!ATTLIST dim value CDATA #REQUIRED>
```

2.2.2 The map editor

The design given in Figure 2.1 is for an editor capable of handling two dimensional maps together with entities. It should be very easy to add entities - all one needs to do is to add more entities inside the *MapEntities* class and the map editor should be able to use these entries.

Using this map editor to edit 3D maps would require a lot more work.

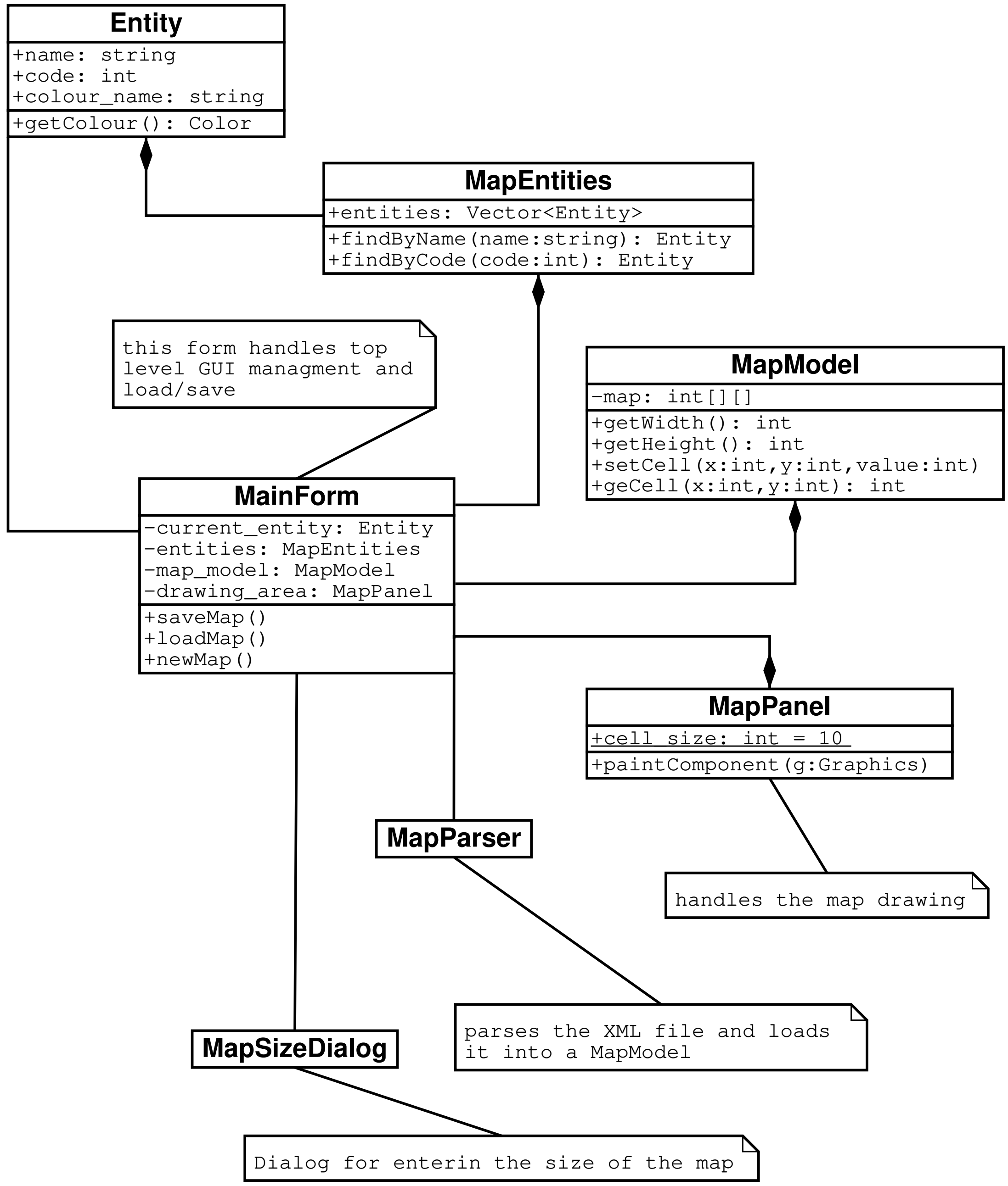

Figure 2.1: Map editor (class diagram)

Chapter 3

The client view

3.1 Agents

3.1.1 What is an agent?

There is no precise definition of an 'agent', as the term is used in a vague way. However, the following common characteristics can be identified [3]:

Definition 1 *An **agent** is a physical or virtual entity with the following properties:*

1. *it is capable of acting in an environment*
2. *it can communicate directly with other agents*
3. *it is capable of perceiving its environment (but to a limited extent)*
4. *is driven by a set of tendencies (in the form of individual objectives or a satisfactory*
5. *it possesses resources of its own*
6. *it has only a partial representation of its environment (and perhaps none at all)*
7. *it possesses skills and can offer services*
8. *it may be able to reproduce itself*
9. *its behaviour tends toward satisfying its objectives, taking account of the resources and skills available to it and depending on its perception, its representations and the communications it receives.*

The implementation of different agent types is central to any artificial life experiment, thus the possible range of agents is very large. The only way to produce a flexible useful toolkit which can provide a sound basis for a range of experiments is to consider the agents as black boxes. More specifically, we shall only consider the implementation of the first three items in Definition 1. So we have to provide tools for implementing the following:

- agents performing actions in their environment
- sensory perception
- communication

The rest of the points are directly related to *internal* agent behaviour and will differ widely in different experiments. While the exact details of action, perception and communication are going do be different for various experiments as well, they are based on the same general principle which can be formalised in the toolkit.

The actions and perception mechanisms will be detailed in the sections below. The communication will be described separately in Section 6 on page 21.

A very important issue to keep in mind while designing this is that the applicability of the toolkit depends on the agent type. In order for the distributed approach to be more effective than the monolithic one the agents have to be computationally intensive. Most of the experiments based on *cognitive* agents would be suitable for this, as each agent requires a fair amount of computing power. The *reactive* agents on the other hand would not be suitable. While they still can be implemented using the toolkit (as they follow the general architecture described above) the speed bottleneck would be created by the external events (i.e. communication, perception, actions) as opposed to internal ones and the fact that all of the communication happens over a network would only make things worse.

3.1.2 Basic agent design

An can be mainly characterised by its type, name and a set of coordinates specifying its position in the world as it can be seen in Figure 3.1. The *Entity* describes at the most basic level any object that exists within

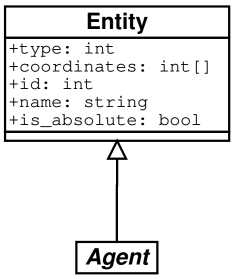

Figure 3.1: Entity (class diagram)

the simulated world. As it can be seen in Section 4.1 on page 14 the world itself can be described as a collection of entities. Each entity can have either absolute or relative coordinates (the type of set currently carried in indicated by the *is_absolute* attribute). The absolute coordinate are natural numbers, indicating the offset from the upper-left corner of the world. The relative coordinates are integers, indicating the offset from some position. As we shall see later, agents are not aware of their absolute position in the world so all the *Entities* they refer to have positions relative to themselves. The server on the other hand is aware of the absolute position of every *Entity* in the world.

Each *Entity* has an unique identification number *id* which is supplied by the server on creation.

3.1.3 Perception

Along with communication[1] the senses are the only means for an agent to get any kind of information from 'outside'. If the *world* is a set W (see Section 4.1 on page 14 for details on what the world is and how this function was obtained) then the set of possible senses contains all the functions

$$s : W \to W' \ , W' \subset W$$

In other words, each sense can only return to the agent a subset of the objects in the world. It is up to each experiment to define each sense in order to create a set of senses S which will give the agent a mean of perception.

The abstract class in Figure 3.2 on the following page shows the details needed to describe such a sense. It is up to the experimenter to implement this class for each sense by defining the function *sense* which populates the *sensed* set with the *Entities* that can be sensed by each particular sense.

A important property is that both the server[2] and each agent have to be aware of each of the senses and actions. The server handles the computational tasks of calculating the affected set of *Entities* when requested by the agent, while the task of the agent is to interpret these results. As already has the answer of the computation as it is aware of the whole simulation world, returning a subset of this is less expensive (at most in $O(n)$) then processing it.

We have two particular sense classes: one for the agent and one for the server. The *SenseDispatcher* is simpler, as it only has to be able to dispatch a 'sensing' request to the server (along with the *Entity* owning this sense) and receive the results and store them in the *Sensed* vector for further analysis. The *ServerSense* class does the actual sense computation in the *sense()* method. Most of the applications will

[1] which is really, just a special case of perception, but in order to simplify the development we shall consider them separately
[2] as we shall see in Section 4 on page 14

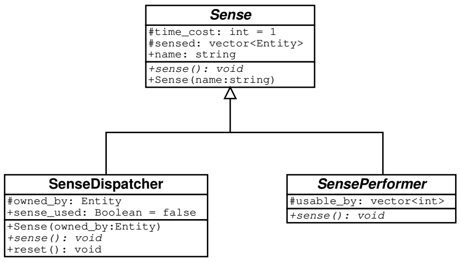

Figure 3.2: Sense (class diagram)

only have to implement the *SensePerformer* class for different senses, as the *SenseDispatcher* will suffice unless complex behaviour is required.

The *usable_by* attribute specifies a set of entity *types* which are capable of using this sense. In the *DispatcherSense* the *owned_by* attribute specifies which *Entity* owns that particular sense (specified at instantiation) this is useful as each sense request to the server will have to be presented with an originator.

Each *Sense* has a time cost (by default one).

3.1.4 Actions

An *action* is the only way for an agent to manipulate itself or its world. An action is initiated by an agent can have one or more of the following consequences:

1. The world (as seen by a third party) is taken from a state S before the action was accomplished to a state S' if the agent performs an *observable* action. That is, the action has to modify the state of any of the entities in the world. Examples of such actions would be movement, creation and destruction of agents.

2. The agent originating the action changes its state internally due to the success or failure of an action. For example an agent tries to eat from a source of food because it's hungry. If it succeeds, then it stops being hungry. The change of state from $hungry \rightarrow not\ hungry$ is not observable in the world.

3. The agent (unobservably) changes the internal state of one or more other agents.

We shall use a design similar to the one used for the *Sense* classes. Each action can be split into two distinct sides: the server side and the agent side.

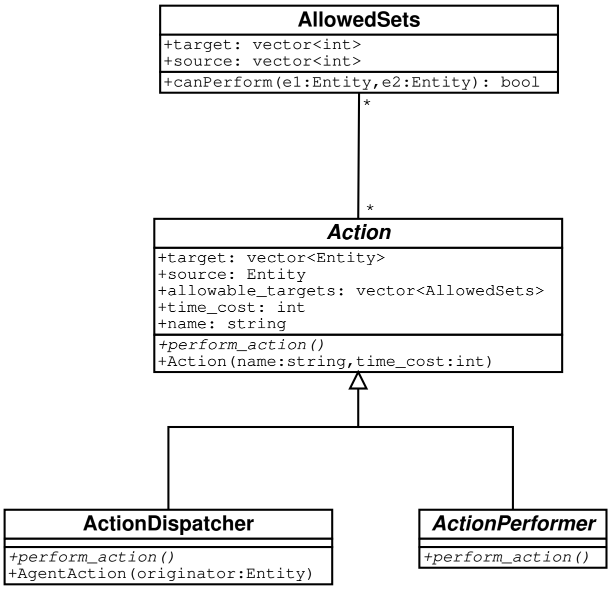

Figure 3.3: Action (class diagram)

The agent side (*ActionDispatcher*) is merely an interface which facilitates placing an action request [3] to the server. The agent will need to specify a target (which can be one or more other *Entities*) and it needs to declare itself as the *originator* of the action. The action should also provide means for checking whether the current configuration is performable. That is, we need to know whether the originator agent is allowed to perform that specific action on its declared target(s). This constraints are part of the specification for each action and are represented as a vector of *AllowedSets*, where an allowed set is a pair of vectors (*target*, *source*) meaning that any *Entity* of a type in source can perform a given action on any *Entity* of type in *target*. Once everything is checked and the agent called *perform_action()* the action request is sent to the server and the function will wait until the server answers whether that action can be performed or not. This is than indicated via the *bool* return value of the function. The agent are expected to interpret a successful action and change their internal state if needed or chose another action if the action was unsuccessful.

The server side (*ActionPerformer*) checks again to see if all the constrains are met and tries to perform the action, returning success or failure. The actual chain of events in this case is detailed in Section 4 on page 14.

3.1.5 Agent design

An agent has to be able to respond to its sensory inputs (Figure 3.4) and to external actions having itself as a target (Figure 3.5). Note that an agent may perform an action on itself. The set of actions that can be performed 'locally' by the agent are kept in the *performable_actions* vector.

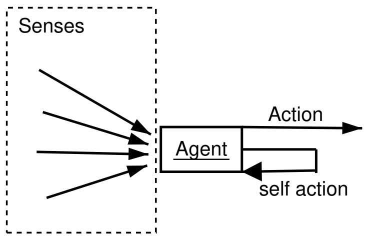

Figure 3.4: Agent, sensory inputs

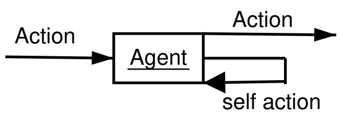

Figure 3.5: Agent, action input

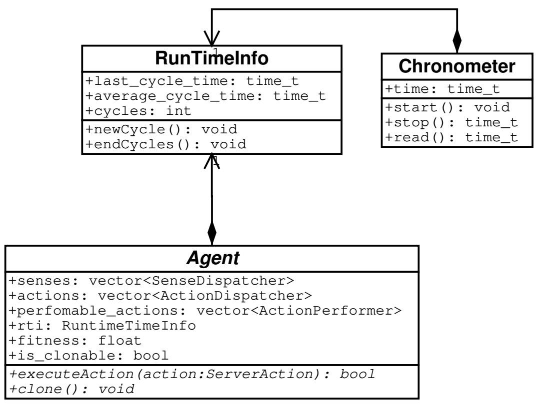

Figure 3.6: Agent (class diagram)

[3]the actual communication process is detailed in Section 6 on page 21

The main difference in between the two is that the agent will check its senses voluntarily but will be acted upon independently of its internal processes. The *executeAction* method provides an entry point for external actions acting on an agent. Each agent has all the available *senses* and *actions* stored in two vectors.

We need to be able to time each cycle in the execution of an agent to be able to find out the performance of each client, which in turn will enable us to distribute the workload evenly.

The *fitness* attribute shall be explains in section 4 on page 14.

Some agents should be able to clone themselves for resons explained in section 4.2 on page 16. This is accomplished by sending a message to the server asking for the creation of a new agent with some pre-specified data which should be enough to recreate the original agent, maintaining its internal states.

3.1.6 Consolidated agent design

By combining all the classes designed so far, we come up with the diagram in Figure 3.7 which sums up the design of an agent.

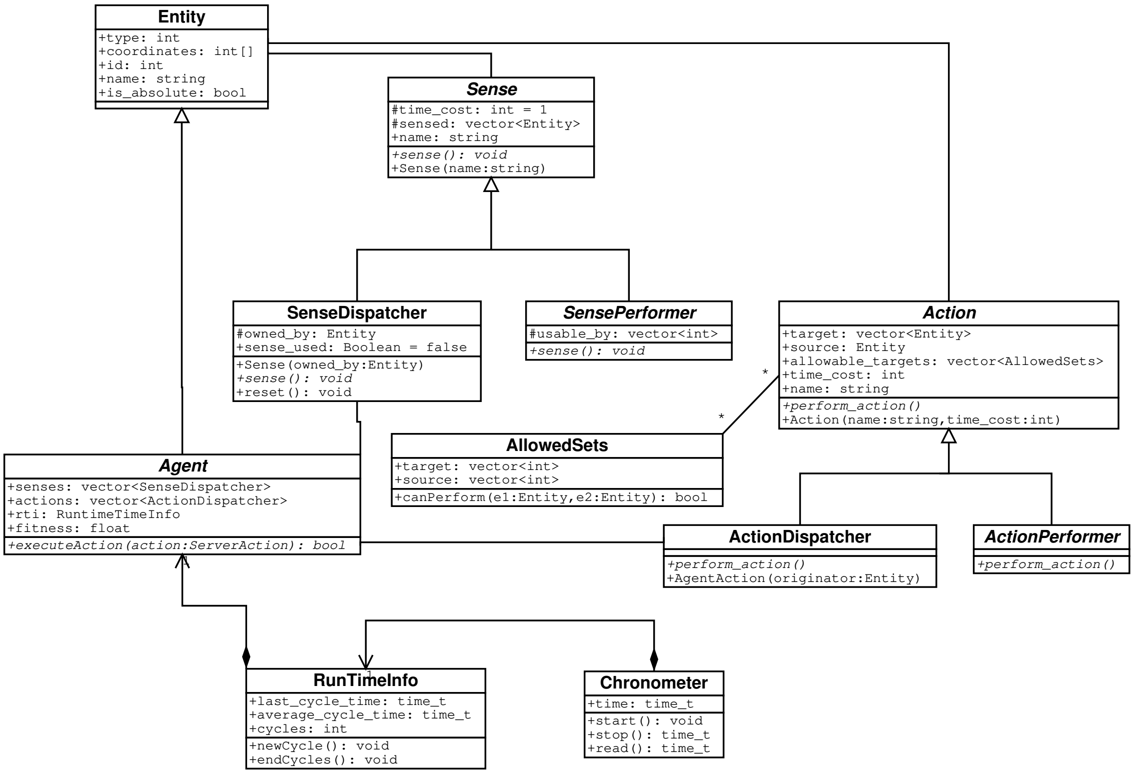

Figure 3.7: Consolidated agent design

3.2 The client

3.2.1 Role

The client acts as a container for all the agents ran on the same physical machine. The clients also acts as a communication gate for incoming communication, isolating the agents from lower level communication details. The client only handles the incoming communication, as it is the only part which raises problem (as the server will not be able to target each agent separately; it will only be able to dispatch a message to the right client).

The client should also be able to create new agents (agents should be able to destroy themselves as a response to a certain action or internal state) and it should also be able to keep a record of the total time required for each cycle.

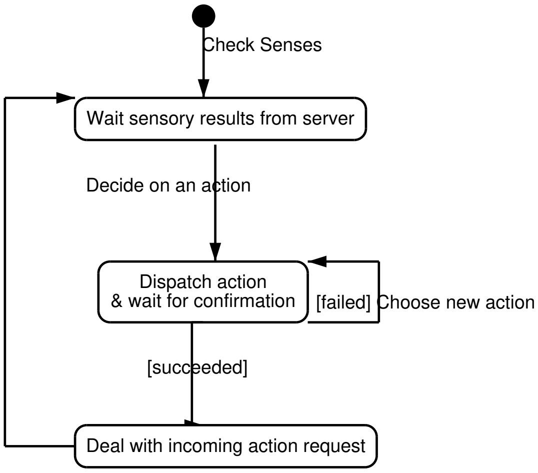

Figure 3.8: Agent (state diagram)

3.2.2 Design

All the incoming agent communication is handled by the *dispatchAgentAction* function. The client makes use of the *RunTimeInfo* class to calculate various run-time statistics.

A vector of all the agents managed by the client is kept in the *agents* attribute. The class diagram for the client is illustrated in Figure 3.9.

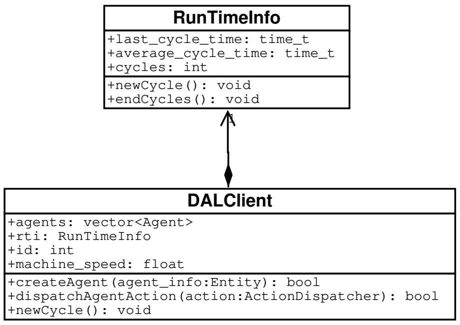

Figure 3.9: Client (class diagram)

The agents will run concurrently on separate threads of execution.

Chapter 4

The server view

4.1 The world

For the purpose of this toolkit we shall consider the world to be a n-dimensional space containing a collection of *Entities*[1]. We can view the world as a set W of entities. Remember that each *Entity* specifies its own location. As any object present in the simulation world has to be an entity, enumerating all the objects together with their location is enough to describe the simulation environment.

The server is the only component of the toolkit which is aware of all the observable components of the world. The server has precise information about the logical and physical location of each entity present in the simulation. However, the server is not necessarily aware of the internal states of these entities.

4.1.1 The map

The map is modelled by the server as a n-dimensional matrix of objects of type *Entity*. The server should be notified by the clients/agents on each visible change of a *Entity*. The visible changes are:

- agent creation
- agent destruction
- change of position for an agent

The notification is accomplished through the *stateChange* method (see figure 4.1). The method takes two parameters, an initial state and a final state. Agent creation can be represented by a void initial state and a final state containing an entity. Similarly, agent destruction is specified by a void final state and an initial state specifying an existing entity.

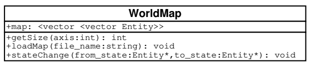

Figure 4.1: Server map (class diagram)

The main purpose of the map is being able to compute the senses for agents on request.

Map parsing

The maps are stored in XML files with the format described in section 2.2.1 on page 6. The XML files are to be parsed using a standard XML parsing library like Xerces[2].

[1] for more information on what an *Entity* is refer to section 3.1.2 on page 9
[2] http://xml.apache.org

4.1.2 Resources

The server not only needs to store information about the simulated world, but needs to manipulate certain aspects of it. Since all entities in the world can be modelled as agents [3] The server only needs to be able to managed agents efficiently (see the following sections for more details on this).

The server keeps a dictionary of all the entities involved in the simulation, together with the client they are currently residing on. This is needed because the clients themselves do not have this kind of information – all the inter-agent messages that cannot be resolved locally by a client are sent to the server, which in turn will re-route them to the relevant client/entity.

4.1.3 Time

The simulation works in time frame mode. Each cycle of the simulation follows the following chain of events:

1. the server notifies all the clients of the start of a new cycle
   (a) each client notifies all the agents of the start of a new cycle
   (b) each agent returns its 'ready' status when they are prepared for the new cycle
   (c) each client returns 'ready' when all its agents are ready
2. as soon as agents become ready, they will probe their senses trying to decide on the next action
3. the server answers all these requests in the order they arrive.
4. agents place action requests
5. the server accepts/rejects the requests
   (a) if an agent has a rejected request it will have to handle it by going back at most to step 2
   (b) if an agent has an accepted request, it will deal with the inner effects of that request and signal the server any visible changes
6. each agent notifies the server when they are ready for a new cycle
7. when all agents are ready for a new cycle, the current cycle is considered complete

At any step in between 2 and 5 the server can place its own action requests. This way simulations which require the "God" view of the world (such as simulations which use random selection of the agents and apply genetic algorithms on them) can place their algorithms on server side.

This approach is based roughly on the *contract net* [3] method, with the server being the manager and the decision algorithm for bid allocation being left open to implementations.

Cycle example

As a specific example let us consider a simulation in which the most fit 80% of the agents are paired randomly based on their fitness and mated in order to obtain a new population. The agents themselves do not have the necessary information for deciding how well that fitness classifies so they cannot initiate the mating process. However, the server has this information so after sorting and pairing the entities [4] it will send a message to one entity of each pair along the lines*'Mate with agent X'*. The agents will handle the whole process from there on via their actions:

1. the agent receiving the message would send a message to the other agent saying *'Mate with me; my genome is ...'*
2. the agent receiving the message will create send a message to the server asking it to create a new agent with some specific details
3. once the confirmation of the creation is received the agent will destroy itself and its partner

[3] This includes objects that traditionally were modelled differently, such as rocks for example. A rock can simply be modelled as an agent without any sensory or motor capacity. The only things that has to be implemented for it is the actions that other agents can perform on the rock; for example moving it or destroying it

[4] The server does not need the internal functioning details of an *Agent* so it views all the agents as *Entity*

Time manipulation

The server is able to manipulate the the simulation by executing one step of the simulation at a time. More complex behaviour can be obtained by the observer (see section 5 on page 19) by using combinations of stepping and pausing (i.e. running the simulation continously, at a given speed, stopping and resuming a simulation).

Class diagram

The class details containing the functionality described so far can be found in figure 4.2.

The *Client* classes will be instantiated in separate threads of execution as this is provides a greater advantage by being able to receive and process most of the messages in parallel.

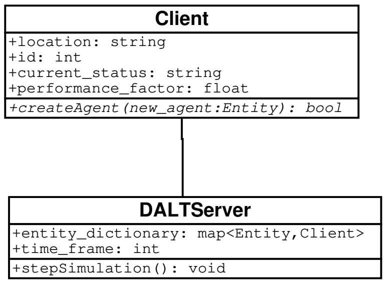

Figure 4.2: Server (class diagram)

4.2 Workload distribution

A very important feature that the server has to implement is ensuring that each client takes roughly the same time to complete each cycle. By doing this we ensure that the processing power of each client is put to full use and no significant period of time is waisted waiting for various parts of the simulation to finish.

When the simulation starts, all the initial agents are distributed evenly in between the clients. Once the simulation is running, the clients are grouped according to the average time they take to complete a cycle. A client can be added to a group if its average time is with α% of the average time taken by the group, with α defined by the simulation. The groups are then organised into a heap, with the best performing group at the top.

The aim now is to reduce the heap to just one node by allocating the agents differently. There are two ways in which this can be done, described in the following sections.

4.2.1 New agent allocation

This is the simplest way to manipulate the workload distribution. Unfortunately, it is also dependent on the simulation type. If we have a simulation in which new agents are often created, the server can attempt to balance the time taken by each client by allocating new agents into the best performing group.

4.2.2 Mobile agents

A more complex way to solve this problem is to move agents from client groups with high load to client groups with low load. A separate, reversed heap of the clients is constructed, this time with the worst performing group at the top. The agents are moved from the top of this heap to the top of the one created previously. Until agents are balanced.

Agent moving is accomplished by cloning and destruction. If the agents support the *clone* mechanism then, each agent can be instructed to create a clone of itself and then destroy the clone. The clone is allocated to a client according to the method described in the previous section.

The only problem that this approach raises is that the agents in some simulations can be complex enough to make the cloning process difficult. Also, if the agents contain a lot of internal data moving them from one physical location to another can be time consuming in itself so the process should not be performed at every cycle.

4.2.3 Time and performance measuring

In order for any of the methods above to work, we need a way of estimating what the performance impact would the creation of an agent have on a given client.

For this purpose, a generic processor speed benchmark, such as the Byte benchmark, is distributed along with each client. When the client is installed on a new physical machine, the benchmark is ran. We have already provided ways for both the agents and clients to report the time each cycle takes. The only thing that needs to be done, is creating a dictionary of average normalised times for each agent type in the simulation.

Say that we have a client which has been evaluated at f times the performance of the reference for the benchmark. When the client reports the time taken by a hole cycle, this timing information is left unchanged. However, each agent will not report the number s of seconds that it took to perform a cycle, but s/f which is the number of seconds that would take for the agents to complete on the reference machine used for the benchmark.

These normalised times are received from the clients along when they notify the server of being ready for the next processing cycle. The timing information is averaged and this information can be used now estimated what performance impact allocating a new agent to a given client would have.

4.2.4 Class diagram

The class details containing the functionality described above can be found in figure 4.3.

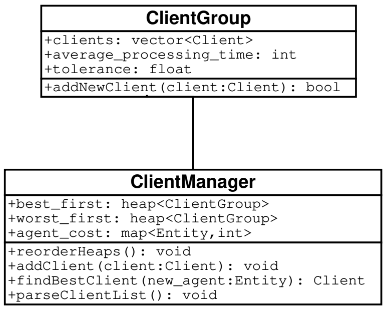

Figure 4.3: Client Manager (class diagram)

4.3 Conflict resolution; action management

In each processing there exists a period when the concurrent nature of the agents creates the opportunity for conflicts. If in sense scanning stage the server can supply information about the same entities to different agents without any problem, when the agents submit their action requests, conflicts can appear as two or more agents can submit an action request for manipulating the same entity.

It is impossible to find a completely satisfactory solution for arbitraging these resources.

The server supports the following generic approach:

1. place all incoming action requests in the *supplied_actions* vector

2. use the *processSupliedActions* method to process split these actions into two groups: actions not creating conflicts and conflicting actions

3. use the *processRejectedActions* method to resolve the conflicts

It is up to every experiment to define the specific implementation of these methods depending on the experiment needs.

The class diagram for *Action* management is given in figure 4.5 on the next page.

The *ActionArbitrage* class is the entry point for all the action requests from simulation. The action queue is being processed continuously until all clients report that they are ready for the next cycle.

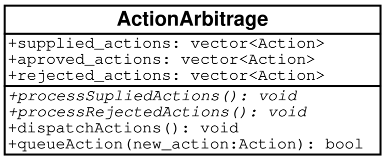

Figure 4.4: Action management (class diagram)

4.4 Client configuration/registration

A list of clients specified by the addresses of the machines they reside on is given in a file which is parsed by the server on start-up. If one or more of the clients is not available the program will not start.

4.5 Consolidated design

The consolidated class diagram for the server is given in figure 4.5.

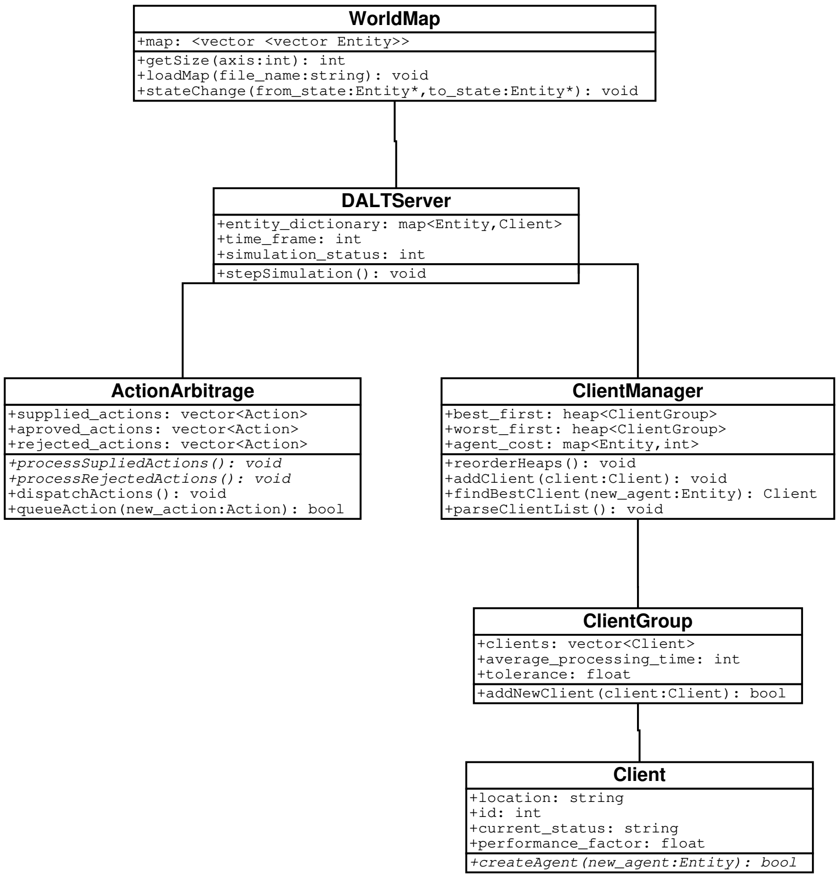

Figure 4.5: Consolidated class diagram for the server

Chapter 5

The observer

5.1 Overview

The observer acts as a window to the experiment. As the server and the client do not have any user interface, all the data is to be gathered by using this module.

By design the observer will only monitor the state changes observable by the server. It is up to individual implementations extend the observer in order to be able to monitor more internal agent states.

The other task of the observer is to provide means of managing the simulation; namely it should be able to signal the server to start, pause, restart and step the simulation.

Similarly to the map editor, the provided implementation will only support two-dimensional maps.

5.2 Server monitoring

As we saw in section 4.1 on page 14, the server is notified of any change of state in the observable environment, namely agent movement, creation and destruction. The observer will not only update its internal states but will graphically display this changes.

The architecture of the observer follows the same general lines as the map editor (see section 2 on page 6).

5.3 Server control

The observer provides means for the experimenter to control the running of the simulation. This is accomplished by linking events generated by the user interface to the methods provided by server via remote method invocation (see section 6 on page 21). The server only provides a method for running one step of the simulation at a time, but several calls to this method can be linked to form a smooth running simulation. Each call to the server will return all agents that have been affected by changes.

5.4 Agent monitoring

Particular experiments may need information on the internal states of particular agents. This particular architecture does not support intrusive exploration of these internal states. The agents have to report this data to the observer in order to be displayed and logged.

5.5 Design

The design for the observer is very similar to the one for the map editor. The main difference is the source for map changes. In the map editor the source is the user, in the observer the source is the server. The class diagram for the observer is given in figure 5.1 on the following page.

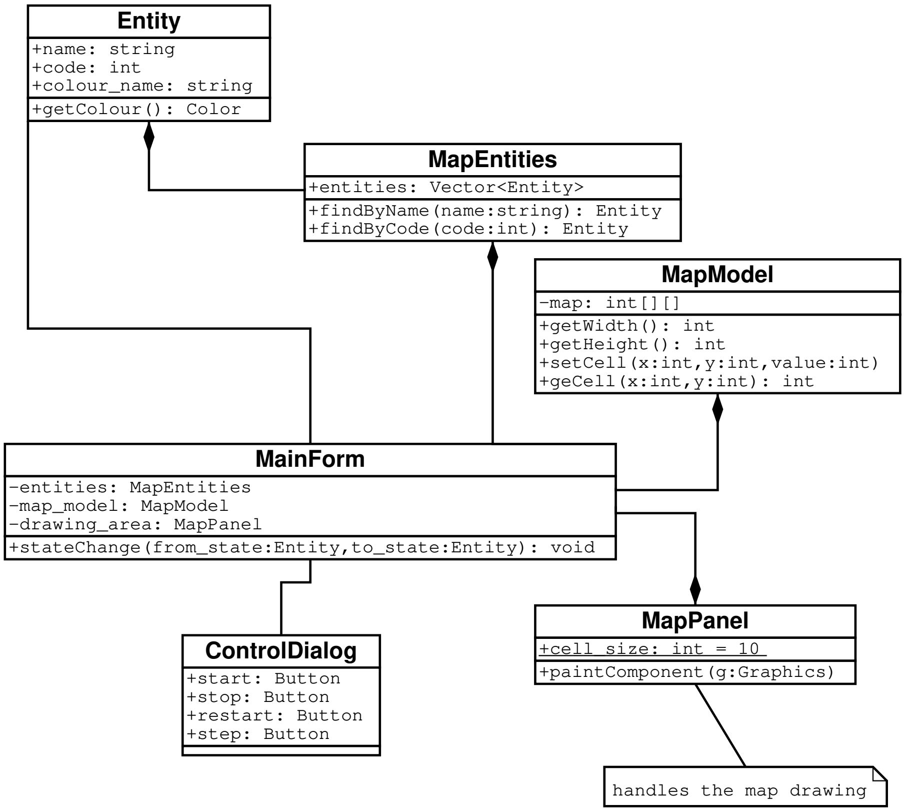

Figure 5.1: Observer (class diagram)

Chapter 6

Communication

6.1 Overview

The whole toolkit is based around the communication in between the server and the client. The communication protocol has to be language and architecture independent so clients can be easily implemented in various languages, or even a reimplementation of the server should be fairly easy.

SOAP makes an ideal protocol with regard to the issues mentioned above. It is open, it is based on XML and it has implementations for most of the popular programming languages. In order to make use of the features provided by SOAP [8] the messages and their content have to be clearly specified, acting as the main source of information for the main implementations and subsequent re-implementations.

Choosing a communication protocol is closer to the implementation stage than to the design stage. However, this is a required step which dictates big part of the design itself which could have been entirely different should a different protocol have been used.

SOAP is nothing more than a fancy method for RPC, over *HTTP*. This means that all the modules of the toolkit will have to provide a *SOAP* server and a *SOAP* client. The server provides methods that can be invoked remotely by the clients. The clients can invoke remote methods on any of the known servers.

All the types described below are the ones specified in the `xsd` and `xsi` name spaces used by the *SOAP* specification. This leaves the task of serialising objects up to the individual module implementation, but ensures a consistent message format, which could not be easily achieved if the serialising task would be left up to an automatised process.

6.2 Agent to server communication

This section will cover messages that can be sent by individual agents to the server. The methods described below will be made available by the server an will be used by the clients.

6.2.1 Sense requests

The agents will periodically need to scan their available senses. While the agents are aware by the senses possessed the actual processing is done by server. The server makes available the following method:

Method: *scanSense*

Parameters

- *entity_id* (`xsd:int`) — this is the id of the entity (agent) requesting this sense.
- *sense* (`xsd:string`) — the name of the sense being used

Returns

- *sensed_entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the sensed entities

Where *Entity* is a compound value containing the following:

- *entity_id* (`xsd:int`) — identifies the entity

- *entity_type* (`xsd:string`) — the type of this entity
- *coordinates* (`SOAP-ENC:Array, SOAP-ENC:arrayType="xsd:int"`) — an array with the position (with an origin in the current position of the agent originating the sense request) of this entity on all axes

Of course, this is only the base form of a sense. Custom senses may send more information to the server by using extra parameters, but the return values will always keep the same format (the *Entity* compound can be extended to provide extra information).

6.2.2 Action request

The action request keeps a similar format to the sense request.

Method: *actionRequest*
Parameters

- *entity_id* (`xsd:int`) — this is the id of the entity (agent) originating the action
- *action* (`xsd:string`) — the name of the action being used
- *targets* (`SOAP-ENC:Array, SOAP-ENC:arrayType="xsd:int"`) — an array with the IDs of the entities that are specifically targeted by this action

Returns

- *performed* (`xsd:int`) — 0 for performed, 1 for not performed. Other values may be added later

Again, specific actions will build on this format by adding extra parameters to the action call.

6.2.3 State change notification

Whenever an agent changes its observable state, the server is notified by this change so that the map (and indirectly the observer) can be updated. See section 4.1 on page 14 for more details on this.

The message contains two entities with the same format as the ones described in section 6.2.1 on the preceding page. For position changes, the second entity will contain the relative position to the first entity. For creation/destruction one of the entities will be replaced with an empty entity with id and type set to -1. This method returns nothing.

Method: *stateChange*
Parameters

- *initial_state* (`Entity`) — the initial entity state
- *final_state* (`Entity`) — the final entity state

6.2.4 Cycle completed notification

Whenever an agent executed all the actions it could in a cycle and is ready for a new cycle a cycle complete message is sent to the server. The *CycleComplete* method provided by the server takes as parameter only the ID of the entity sending this notification. A value of 0 or 1 is returned depending on whether the server approves the request or not.

Method: *cycleComplete*
Parameters

- *entity_id* (`xsd:int`) — the ID of the entity notifying the completion of its cycle

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

6.3 Agent to agent communication

All the inter-agent communication passes through the server (reasons for this are described in 4 on page 14). A positive side-effect of this is that not every agent is required to perform as a SOAP server, which would be computationally intensive for simulations with many agents. All the action requests are sent to the server which are in turned re-routed to the right client (possibly via the `SOAP-ENV:actor` mechanism). The client de-serialises the message and passes it internally to the right agent.

For some actions, the server might be able to perform them so there would be no need to pass the messages on.

6.4 Server/client communication

6.4.1 Methods provided by the server

The server provides only one method for the client, the registration method.

Client registration

The server is the first program module of the toolkit which started. After start-up the server waits for clients to make their presence known by calling this method. The clients have to 'present' themselves by providing the contact details for their soap server. In exchange they will receive 0 or 1 depending on whether the server accepted the registration. There is no need to keep track of the clients with unique ids as the address of the client SOAP server is unique.

Method: *clientRegistration*

Parameters

- *client_endpoint* (`xsd:string`) — URL for the client SOAP server
- *client_ns* (`xsd:string`) — the name space used by the client
- *speed_factor* (`xsd:float`) — The benchmarked speed of the machine the client is running on

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

6.4.2 Methods provided by the client

Client status

This method provides generic information about the client.

Method: *clientStatus*

Parameters

- none

Returns

- *status_info* (`ClientInformation`) — a compound containing the requested information

The *ClientInformation* compound contains the following:

- *last_cycle_time* (`xsd:int`) — time taken to complete last cycle
- *average_cycle_time* (`xsd:int`) — average time taken to complete a cycle

More fields may be added to this compound.

Agent status

This method provides generic information about an agent.

Method: *agentStatus*
Parameters

- *entity_id* (`xsd:int`) — the id of the agent to which the message is to be routed

Returns

- *status_info* (`Array:AgentInformation`) — a compound containing the requested information

The *AgentInformation* compound contains the following:

- *last_cycle_time* (`xsd:int`) — **normalised** time taken to complete last cycle
- *average_cycle_time* (`xsd:int`) — average **normalised** time taken to complete a cycle

More fields may be added to this compound.

Agent creation

This method is used by the server to create a new agent. The only information supplied is the type of the agent. For more complex simulation more information might be provided, such as an initial internal set of states for the agent or cloning information.

Method: *createAgent*
Parameters

- *type_id* (`xsd:int`) — the type of the agent to be created
- *agent_id* (`xsd:int`) — the ID of the agent to be created (allocated by the server)

Returns

- *performed* (`xsd:int`) — 0 for approved, 1 for not not approved. Other values may be added later

New cycle notification

When this method is called a client notifies all its agents of the beginning of a new cycle.

Method: *newCycleNotify*
Parameters

- none

Returns

- none

Action request

The interface for the *ActionRequest* method is identical to the one used by the server (section 6.2.2 on page 22). The client de-serialises the information provided and passes the request to the appropriate agent.

6.5 Server control

The server provides the following methods for invocation by the observer:

6.5.1 Stepping the simulation

Method: *stepSimulation*
Parameters

- none

Returns

- *rezult* (`xsd:int`) — 0 if a new cycle was initiated, a positive value greater then 0 otherwise

When the observer calls this method, the server attempts to execute one step of the simulation. This will happen only if all the entities have finished the previous cycle and are ready to start. If all entities are ready, a new cycle is started and the method will return the value 0. Otherwise a new cycle is *not* started and a positive value is returned.

6.5.2 Retrieving the entities modificated created in the last cycle

Method: *getDelta*
Parameters

- none

Returns

- *changed_entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the entites which this simulation cycle changed

All the entites which have suffered visible changes during the last cycle are returned in the *changed_entities* array. The observer should be able to distinguish in between new entities (entities whose uniques IDs have not been encountered in previous steps), entities which simply changed their state (known IDs) and entites which have been destroyed (indicated by an empty entity type).

6.5.3 Testing server state

Method: *isIdle*
Parameters

- none

Returns

- *rezult* (`xsd:boolean`) — true if the server is idle (and ready), false if the server is still running a cycle

This function has to be used to poll the server in order to determine when it is safe to retrieve the changes occured in the cycle. This also acts as a trigger inside the server for certain operations, such as delivering actions.

6.5.4 Map status

This method is similar to the *stepSimulation* method. The main difference is that no action is performed by the server.

Method: *mapStatus*
Parameters

- none

Returns

- *entities* (`SOAP-ENC:Array, SOAP-ENC:arrayType="Entity"`) — an array with the entites in the world

Bibliography

[1] Distributed artificial intelligence: Theory and praxis, 1992. Library: Q.335.D4.

[2] Bob DuCharme. *XML : The Annotated Specification*. Prentice Hall PTR, 1999. Library: QA 75.4.X6.

[3] Jacques Ferber. *Multi-Agent Systems*. Addison Wesley, 1999. Library: Q.335.F3. 1.1, 3.1.1, 4.1.3

[4] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley, 2000. Library: QA.75.1.G6.

[5] Kendall Scott Martin Fowler. *UML Distilled*. Addison-Wesley, second edition, 2000.

[6] Jan Egil Refsnes. web: http://www.xml101.com/dtd.

[7] Stephen R. Schach. *Classical and Object-oriented Software Engineering*. McGraw-Hill, fourth edition, 1999. Library: QA 75.2 S2.

[8] Scot Seely. *SOAP Cross Platform Web Service Development Using XML*. Prentice Hall PTR, 2002. 6.1

[9] Munindar P. Singh. *Multiagent Systems*. Lecture Notes in Artificial Intelligence. Springer-Verlag, 1994. Library: Q.335.S4.

[10] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

List of Figures