A Broker based architecture to control robots in a simulated environment

Luis Moniz[1], Paulo Urbano[1], and Helder Coelho[1]

Faculdade de Ciências de Lisboa
Campo Grande, 1749-016 Lisboa, Portugal
{hal,pub,hcoelho}@di.fc.ul.pt

Abstract. We present an extension to the Aglets platform in order to support brokering services agencies in a transparent process for the agent designer. This architecture is constructed to provide a set of functionalities, not originally available in the core Aglet environment. These include, white page service, group formation and yellow page support in a distributed and mobile agent environment. The extension was developed in three directions, new environment agents, new agent architectures and an extended agent live cycle model. We also extend the Aglet message model in order to support KQML like directives and speech acts. With these modifications, we encapsulate the Aglets environment in a set of new facilities that provide the designer with a set of higher-level tools to develop his agents. We use this platform to control a team of robots in a simulated environment, providing a complex communication channel to the robots, extending the original features providing a white pages service and a GPS like facility.

1 Introduction

The number and variety of agent-based systems have been growing rapidly in the last few years. The advent of e-commerce and the widespread use of Internet made the agents one of the hot technologies in the present, and as a result, the computer software industry is beginning to use this technology to develop its own products. The absence of an industrial norm of what is or could be classified as an agent gives rise to the abuse of the term and concept, and consequently, to the appearance of a wide variety of different heterogeneous and incompatible systems [10, 9, 14].

The main objective of some of these systems is to provide the user with a simulated environment (for instance the Swarm environment [3]) where his agents are to be executed, but almost none provides tools to aid the development of the agents. Other example of these systems is the RoboSoccer [6] environment, although it provides the user with a complex simulated execution environment, it is not simple to directly use the agents developed in this system in a real application, the specificity of the environment limits the application of the agents produced.

Other systems provide the user with a rigid and complex architecture [5], which forces the agents in a pre-determined track of development and design. In the end, the effort putted in developing these agents is hardly justified for small applications and examples. Our proposal consists in a framework to develop scalable agents and execute them in an open environment. The agents can be designed with the needed capabilities and architecture, interact with each other, and access foreign services.

Our main objective is to build a tool to provide the user a set of features and components that help him in constructing and assembling new agents. The user can then aggregate those agents into teams of agents that are executed in a platform. We also add to the system some predefined agents that support some services not available on the original platform. This approach do not impose any specific structure to the final system, the user has freedom to choose which tools, features and agents are to be active in his experiment, using only the necessary resources.

We use the Aglet platform to support the execution of our agents, we extended the original workbench in three directions in order to support the new functionalities: (1) we changed the Aglet core architecture in order to hide some low-level construction aspects from the designer, (2) we modified the Aglet life cycle model to support the integration and usage with the new services, and (3) we added special purpose agents to support these services. The new services include location service, message delivery, group management and yellow pages. In order to support more complex message exchange protocols, we modify the Aglet message handling model. This allowed us to support different types of messages, like directives, commands and speech acts. These modifications gave us a series of new facilities assembled to the Aglets workbench, that provided the designer with a set of higher-level tools to develop his agents.

As an application of these extensions we built a workbench based on two different tools: the extended Aglets multiagent platform [4] and the Player/Stage environment [2, 13] for robotics simulation. The framework links these two heterogeneous environments into a single platform, providing us with a tool for construct and an environment to experiment agents. The resulting testbed merge the Aglets platform features and the dynamic and unpredictable characteristics of the Player/Stage environment producing a tool capable of combining social and physical aspects of agents into a single experiment. The usage of the Aglets platform presents some technical advantages over other alternatives. The fact of supporting mobile agents provides us with a tool for distributed computation within a network; another advantage lies in the portability of the platform and its low computational cost, which allows the construction of a lightweight and reliable system. The Aglets platform is extended in order to support new services (GPS and communication) and control the robots of the simulated environment. This extension is composed of three main components: an extension to the core agent architecture, the addition of some new agents in the Aglet platform and the encapsulation of some low-level tasks in an abstract architecture [7]. The Player/Stage environment is used as an example of an application of the agent

paradigm to control of robots in a real world. In order to facilitate the robot control, and to provide the user with a simple tool to give commands to the robots in real time, we choose to define their behaviour control structure in the CLIPS language. In the figure 1, we present a system overview of the Aglets platform and the simulation tool.

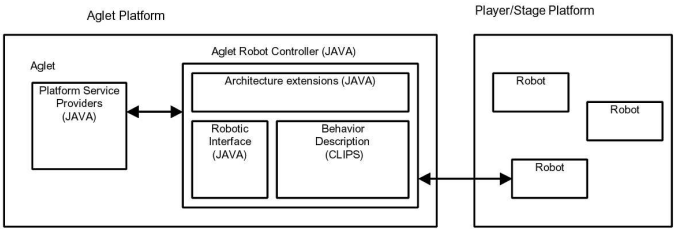

Fig. 1. The system overview

The remainder of the paper is structured as follows. In section 2, we present an overview of the Aglets platform and the extensions made to the Aglets architecture. In section 3, we present and discuss the special agents added to the system, their advantages and drawbacks. Finally, in section 4, we present the simulation tool, the integration with the aglets and some examples of robot behaviours.

2 The Aglets architecture

2.1 The Aglets environment

The Aglets environment is a platform inside which all the agents, mandatory written in JAVA, are executed. This platform provides the agents with a set of services (message transport, mobility, cloning) and an execution model that must be followed. The Aglet life cycle is event driven, and based in the aglet response to these events. An aglet must be constructed regarding the model that regulates its life cycle, we must define what to do when it is created, when it moves or when it is removed from the platform. In the platform, each agent is identified by an agletproxy (a kind of internal address), associated with it at creation time. The platform supports a message transport service based on Aglets proxies. The knowledge of these proxies is fundamental to the agents in order for them to be able to communicate. The original environment does not provide a central mechanism to track the agents and know who is active and when. For instance, when an agent moves from one platform to another its proxy changes, making that all the other agents lose its contact. The system provides a control interface and Aglet viewer (Tahiti) with which the user can launch aglets, control their state and remove them from the platform.

The Aglet architecture structure is based on its life cycle, which is ruled by events and actions. An event can be defined as an internal occurrence in the aglet, the actions are its response to that occurrence. For instance, when an aglet is created inside the platform (event), its *onCreation()* method is invoked (action), then when it is activated (event) the *run()* method is invoked (action). The same happens to all main events on the Aglet cycle, for each one there is an aglet behaviour pre-determined to be invoked.

The definition of an Aglet behaviour is made by redefining some of these behaviours, identifying which actions should be performed in each situation. This architecture and the use of the JAVA language to define agents have some advantage when redefining aglets based on previous ones; it is only needed to define what is different. For instance, if we have an aglet that checks whenever email arrives, we can use this agent to define other that remove all spam mail, based on some criteria.

2.2 The extended environment

The original support of the Aglet environment is made based in low level primitives. Is up to the agent designer to construct the aglets with tools to deal with the community. Our proposal extends the Aglet environment in order to provide a set of support tools to encapsulate these low level primitives. The extension is made two different levels: modifications on the aglet architecture, addition of special purpose agents to the environment. We modify the Aglet architecture on two components: we extend the Aglet model in order to hide some low-level construction aspects from the designer, and we modify its life cycle model to support the integration and usage of the new services added. The new special purpose agents we add to the platform (see section 3) encapsulate and extend some services, originally provided to the community by the core platform. These services are mainly of two categories: agent location management (Broker agent) and message delivering (BrokerSlave).

The Aglet architecture Our aglet architecture is built based in a layered structure. Each agent is defined by incorporating new features in a simpler agent originating a more complex and specific entity. These new features can be defined by adding new characteristics or by redefinition of the original feature. Based on the core Aglet architecture, we constructed our extended Aglet architecture adding new features, replacing others and constructing new features. This layered construction method generated a collection agent capable of using the enhanced platform services, without necessity of redefining all the capabilities each time we create a new agent.

The agent life model In order to use the Broker services, each aglet must know at least one Broker agent. When an aglet is created, its first action is to register with the broker agent; it must repeat this action every time it arrives to a different platform or context. When an agent leaves a platform, it informs the

broker that it is momentary unavailable, or if is disposed, it informs the broker of its own destruction. We extend the Aglet model to execute these actions in a transparent from the Aglet designer; these tasks will be automatically performed when each of the events occurs.

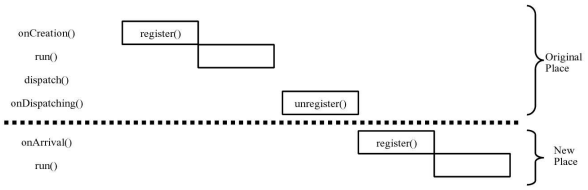

Fig. 2. Example of the execution of a moving sequence

These hidden actions provide the agent designer with a transparent level in the agent architecture, which associates complex sequences of actions into simpler ones. For instance, moving an agent from a platform to another corresponds, in the designer view, to issue the command *moveTo(destination)* to the agent. On the lower level, this command is translated in a sequence of actions: send an *unregister* message to the Broker, move to the new location, send a *register* message to the broker with the new proxy (figure 2). Our interference on the agent life cycle creates a level of encapsulation that hides the low level protocol, allowing the system to automatically manage the activity and location of the agents.

2.3 The message handling protocol

Another level of encapsulation is the message transport level. The agent only needs to know the name of the receiver of the message, for it the broker does not exist, the message protocol is only sendMessage(destination, message). How the system handles the message is completely transparent to it, even the subconscious knowledge of the broker existence. The directives of message handling are defined at different levels in the agent, each one handling different kinds of messages. The lower levels manage all system messages types and reactive events. For instance what an agent should do if it receives a dispatch or a retract message. The message handler is extended in the higher level to process other messages received by the agent. This permits to implement more complex protocols on higher levels, separating them from the low-level control messages. For instance, in the high level we can support KQML message protocol [12], and how the agent processes and react to these messages, and in the low level more reactive based messages like dispatch(machine) because the current location is going down. This option is not reduced to fixed number of levels; it is the designer choice if he needs more complexity to this behaviour. In our base Aglet architecture the lower level behaviours of message handling (layers 1 and 2) are

predefined, but the designer is not limited, if needed he can redefine the agent reactions to these predefined events in the higher levels.

2.4 Special primitive messages

Another tool built in this system is a set of new message primitives that can be used by the agents. These primitives are a kind of social commands, directed to access to society and to other agents specific information, we can view these as simplified KQML directives. For instance, the lookup agent message permits to the agent to know if someone is still accessible by the society, or the listing message allows to access to all the current members of the society. Other primitives available are used to send messages to other agents. Originally, when a message is sent the sender must know the receiver proxy; these primitives allow send a message to a specific agent using its name. Another is the broadcast type message that distributes the message contents to all active agents at the moment.

3 Service Providers Agents

3.1 The Broker Agent

This agent has several functions like central mail deposit, receiving and sending all the messages in the Aglet platform, agent state and address manager, and special messages interpretation. The aglets do not have a direct knowledge of the existence of this service, it is hidden in the message service. When the agent sends a message to another, this message is sent first to the Broker and it is the Broker mission to deliver the message to its final destination.

The main role of the Broker agent is to maintain a simple white pages service. The Broker agent identifies the other agents by their name and associates it to their current state and agletproxy. This allows sending a message directly to an agent, without needing to know its agletproxy or it state, the broker will forward the message to the correct recipient in its actual location. For instance, when an agent moves from a platform to another, it becomes unavailable for a time, during this time all his messages must be kept to posterior delivery. When the agent arrives to its new location, it must register with the Broker providing its new proxy, the Broker then changes its state to active allowing the messages to be delivered.

The message interpretation task allows, as seen in section 2.4, some agents to have special primitives to control the other agents. For instance, it is possible to send a message to Broker to dispose another agent and this action is performed directly. It is possible to use the Broker to support group creation, and using special messages, the agents can ask the broker which groups are currently active, to be added or removed from a group, and to send a message to group. In addition, a simple yellow pages service can be supported by the Broker agent. As with group, this service identifies the agents by what kind of services they can provide to the community.

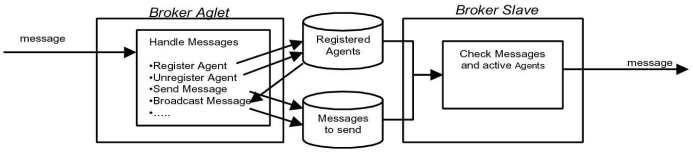

Fig. 3. Broker and Slave interaction

In this system, it is possible to have several Brokers working in the same platform or in different platforms, all connected and exchanging messages. The only limitation is at Aglet name space, each name identifier must be unique, and do not have conscious information regarding the broker where is registered. So in order to exchange messages between agents registered in different brokers, a meta-protocol of message transport have to be defined. In our architecture, we implemented the Broker agent as a set of aglets, each one responsible for a different task of the Broker functioning. The Broker agent is not a single agent, but a rigid hierarchy of agents with well-defined functions.

3.2 The Broker Slave

The role of this agent is to send the messages to the right recipients. Periodically it checks the messages received by the Broker and finds out the currently available agents. With this information, it resends each message to its correct recipient. The time interval between checks of messages can be tuned in order to avoid unnecessary consume of processor time. Internally the BrokerSlave has a access to the message queue with all messages to be delivered and to the list of current active agents. In each cycle, he checks all messages, to find out if their recipients are currently active, and if it is so, it delivers the messages to their final destination. This agent works more like a demon than an agent, activating itself at pre-determined time intervals and sending all pending messages to the current active agents.

3.3 Adicional Brokers

Although the original Broker Agent provides the workbench with a white page services and message delivering service, addressing each agent by its name, when we start to design more complex experiments, we had to create adicional brokers. These Brokers maintain a database of the active agents indexed by one specific characteristic,for example, the role of the agent in the simulation, groups to which that agent belongs, tasks it can perform or physical distance to the message originator. These approach has the advantage of divide the task of identifying the recipient of the message to different agents according to the message type. Another advantage relies on the fact that we do not need to activate all the Brokers in an experiment, the user can choose which ones he will need and activate only those.

3.4 The security/control issues

The control of the agent is made in two different levels. One is based on messages send to the agent, and is it choice the form to answer them. For instance, an agent has a normal behaviour to react to a retract message, but it is not compelled to perform it. The other is based on the agletproxy information, when the aglet receives a command through its proxy, the corresponding action is initiated. We use this characteristic to create a management level of the aglets. The main security issue is who can send this messages. In the first situation, it is the agent responsibility to decide if it should comply or not. However, in the second the agent has no choice in the action performed, it is up to the Broker to manage this control level identifying the untrusted sources. Other security problems are controlled by the aglets platform and the Java Virtual Machine. This include access to resources, memory allocation, denial of service attacks, etc.

4 The Player/Stage and Aglets

The Player/Stage platform simulates a team of mobile robots moving and sensing in a two-dimensional environment. The robots behaviours are controlled through the Player component of the system. The Stage component provides a set of virtual devices to the Player, various sensors models, like a camera, a sonar and a laser, and actuators models, like motors and a griper. It also controls the physical laws of robot interaction with each other and the obstacles. In our current environment only the sonar, laser and motors are usable. This tool provides a controllable framework to test and experiment in a simple robotic environment.

Due to limitations in the Player/Stage platform (it does not provide a communication channel among robots) we choose to integrate this tool with the Aglets platform, associating to each robot an aglet. This aglet controls the robot behaviour using the Player interface (sensing and actuating), and it is capable of communicating with the other aglets (robots) through the Aglets platform. This extension provides the robots with a communicating channel (peer to peer and broadcast) that provides them with complex message exchange capabilities. It also provides an optional interface with the system user through a window console. Each robot can have its own window console, making possible to the user to track the aglet execution and communicating directly with it, allowing to suspend its execution, examine and modify its internal state, and also resume its execution. This tool is fundamental in our environment in order to overcome the limitations of the original simulation display. Additionally we add a GPS to the system, providing the robot with the knowledge of its absolute position in the environment. We choose the CLIPS language to contruct the robots behaviour (see section 4.1). To link the two platform we add to provide some modules to the Aglets platform (see section 4.2). We also construct some special agents (see section 4.3) that can be used to send commands to all robots, and to monitor the simulation.

4.1 Jess CLIPS interpreter

To simplify the design of the robot behaviour we choose to describe it using the CLIPS language. This language is a rule-based language with a forward chaining based reasoning engine. The advantage of CLIPS lies in the fact that it is an interpreted language, giving the user control over the robots behaviour in real time. He can modify or observe the behaviour of a robot without stoping the simulation. The user can define the robot behaviour in the form of pairs conditions/actions, being appropriate for the representation of reactive behaviours.

To support this language we had to incorporate in our aglets the Jess CLIPS interpreter engine [8]. When an agent is created the file, where is behaviour is defined, is passed as a parameter, this file is loaded to the agent Jess interpreter and then the agent moves on to standby mode. The agent is activated only when it receives the run command. The console commands are also given in CLIPS. For instance, the user can command a robot to go to a specific location, run 10 rules of its behaviour or add a new rule to the agent.

4.2 The extended modules

To accede to the robot primitives and Aglets commands inside the Jess environment we had to build some libraries of CLIPS functions. These libraries give access to the physical environment and to the control the robots, to the Aglet platform and message sending commands, and to some CLIPS extra features. We divided these libraries in three groups: physical commands, Aglets commands, CLIPS extensions.

The Physical commands library give access to the Player/Stage environment from the CLIPS virtual machine. It provides functions like *position* that returns the robot GPS data or *setSpeed* that sets the translation and rotation speed of the robot.

The Aglets commands library provide a set of functions to use the Aglets environment features. For instance, the *broadcast* function that sends a message to all active agents, the *getmsg* that retrieve a message from the agent mailbox, etc. These group of features are supported mainly by our Broker agents.

The CLIPS extensions library export to the Jess virtual environment some functions written in foreign code. These functions can be divided in two categories: functions not originally available in the Jess environment, and functions that could be written in CLIPS but presented a better performance written in other language. Examples of these are the trigonometric functions and the *findall* function that allow to find out a matching fact in the agent facts base.

Our system also allows us to build a library of functions that are available to all the agents in the simulation. This library contains functions and pre-defined behaviours and commands that are common to all robots. For instance, a simple command like go to a specific position *(gohome x y)* implies a complex set of actions and behaviours: find out the current position; calculate the trajectory; choose and start the robot with the correct speed and direction; constant check for obstacles and if found any avoid it; and check if the goal is reached.

These kinds of behaviours can be constructed and became available as macro-behaviours in the CLIPS library.

4.3 The Special Agents

We also defined two special agents categories: the global controller, and the monitor agent.

The global controller is a special built-in agent that represents the observer and is used to control the other agents in the simulation. The observer can suspend and resume all robot agents. The controller can also interrupt on-line every other robot agent order to send it commands: we may want to inspect its state or to force it to begin a particular behaviour. We consider two agent classes which must be controlled: (1) the robot agents and (2) the special agents. So, the observer can broadcast commands to all agents, to all robot agents, to a particular agent, and to a list of agents.

We use the Aglets facilities to produce intrusive monitor agents, capable of generating real-time information about the environment and the robots. Instead of exhibiting a transparent behaviour in the simulation, these agents request the data directly from the robots and produce some analysis. In order to allow this behaviour, the robots must understand the Monitor requests and answer accordingly to them. We want to extend this class of agents incorporating more complex, detached and transparent agents, capable of producing real time statistics, like how many times each robot got stalled, which is the more efficient, the explored world, etc, without interfering in the simulation. We should also note that these agents are fully customized by the user, and it is possible to build Monitor agents with specific characteristics, specially tunned to observe some given experiment.

4.4 A simple example

In this section, we will present a simple example of the design of the "following-mate behaviour" [1] in our environment with four robots.

This behaviour implies that each robot has a mate, whom it should follow in the environment. The robot is waiting until it receives a message from its mate indicate its current position, and then it starts to follow it, avoiding obstacles at the same time. Periodically, it broadcasts its position to all robots.

The behaviour cycle is displayed in Fig: 4. The four actions on top work like the subsumption architecture, only one action can be activated and in case of conflict the lowest one is chosen. The three actions (sonar, position and messages) on lower line correspond to an update of sensory information. Next, we present a fragment of the CLIPS code describing the "Safe Wandering" and "Follow" actions. We should note the use of a global variable **mate** indicating which is the robot to follow.

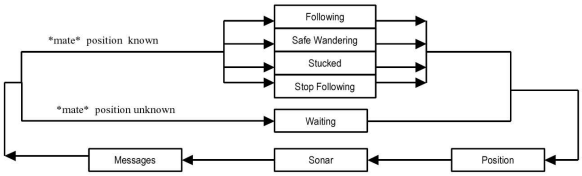

Fig. 4. The robot generic behaviour

```
; ------------------------------------------------
;       Rule to safe-wander when following
; ------------------------------------------------
; Obstacle avoidance while following the *mate*

(defrule safe-following
  (declare (salience 10))
  (goal follow)
  ?pos <- (position ? ?x ?y $ ?)
  ?sonar <- (sonar ? ? ?s1 ?s2 ?s3 ?s4 ?s5 ?s6 $?)
  (test (or (< (sonar-data ?s1 ?s2 ?s3) 150)
            (< (sonar-data ?s4 ?s5 ?s6) 150)))
 =>
  (bind ?minL (sonar-data ?s4 ?s5 ?s6))
  (bind ?minR (sonar-data ?s1 ?s2 ?s3))
  (if (and (= ?minL 0) (= ?minR 0))
      then (setSpeed 0 180)
      else (setSpeed (+ ?minL ?minR) (- ?minR ?minL)))
  (update-sensors)
  (broadcast mate-pos ?*my-name* ?x ?y)))
```

```
; ------------------------------------------------
;       Rule to follow the mate given its position
; ------------------------------------------------
; go to the *mate* robot

(defrule follow-robot
  (declare (salience 2))
  (goal follow)
  (message mate-pos ?*mate* ?mate-x ?mate-y)
  ?sonar <- (sonar $?)
  ?pos <- (position ? ?x ?y ? ? ?compass ?)
 =>
  ; Calculate the best speed parameters of the robot
  (bind ?speed (speed-to ?x ?y ?compass ?mate-x ?mate-y))
  (setSpeed (nth$ 1 ?speed) (nth$ 2 ?speed))
  (update-sensors)
  (broadcast mate-pos ?*my-name* ?x ?y))
```

We present an example of a graphical representation of the track followed by the robots (se Fig:5) and their change of bearing (see Fig:6) when performing this behaviour.

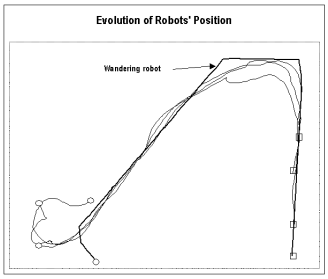

Fig. 5. The evolution of robots' position with time. The white circles correspond to the starting positions and the squares the final robot positions

In our experiment we use four robots (yellow, red, blue and green) to wander in the environment, exhibiting a line formation pattern. One robot wanders freely in the environment, and the other three try to follow it in a single line. Except for the wandering robot, all the others have a different mate to follow. To startup the experiment we use the global controller console to send commands

to the robots (for instance, we can use *(you-do red wander)* and *(you-do blue follow red)*). lp Using the consoles during the simulation, it is also possible to change the red robot behaviour, transforming it from a followed into a follower of the green robot. All we need to do is alter the red robot internal fact *(goal safe-wander)* into *(goal follow)* and change its *mate* variable to green. This can be accomplished directly in the robot console or through the global controller, issuing the commands *(follow green)* or *(you-do red follow green)*[1] respectivelly. This feature gives us a tool to, in real time, interfere in the simulation forcing the robots into the situations we want to obtain.

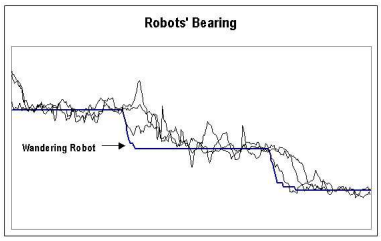

Fig. 6. The evolution of robot bearing with time. We should note that the major differences in the lines, correspond to the appearance of obstacles (wall or robots) in the robots paths

With the capacity of getting statistics from the simulation using the monitor agents, we get a tool to full control the running experiment and also to get the data we need from the individual robots. All these features can be customized by the user according to the experiment objectives. The user can build agents to get specific data in specific situations from the experiment and the other agents, and also direct the experiment in order to get the situation he want to achieve.

5 Conclusion

This paper presented an extention to the Aglets platform, in order to support a simpler form of addressing agents by its identification. This solution overcome the original necessity for each agent to know the other agent proxy. We supported this mechanism automatically, by adding special agents to the original Aglet architecture, extending the core platform. The framework was constructed to provide a set of functionalities, not originally available in the Aglet environment. These include, white page service, group formation and yellow page support in a distributed and mobile agent environment. The extension was developed in three directions, new environment agents, new agent architectures and an extended agent live cycle model. With these modifications, we encapsulate the

[1] These commands are specific of the robot CLIPS architecture developed to this example, they are not part of the core of the platform

Aglets environment in a set of new facilities that provide the designer with a set of higher-level tools to develop his agents.

We joined the platform with a foreign tool for robot simulation and illustrated the application of the tool in a simple example of simulating behaviour based robots in a dynamic environment. The resulting framework allowed us to combine the Aglets platform potential of interaction and management of agents, and the dynamic and unpredictable characteristics of the Player/Stage environment in a tool capable of modelling social interactions of agents with physical representatives.

The tool developed can aid the designer to create his agents from a higher perspective, without concerning with low-level details. Also it made possible to overcome some limitations of the Player/Stage environment, adding a global positioning system to the robots and a peer-to-peer communication. The inclusion of monitoring and global controlers agents gives the user a new management level over the experiment. The advantage of using agents instead of fixed interfaces to control and monitoring, allowed the user to tune these agents to the specific experiment objectives.

References

1. Ronald C. Arkin. *Behavior-based Robotics*. The MIT Press, 1998.
2. R. T. Vaugham B. Gerkey, K. Stoy. *Player Robot Server*, version 0.8c user manual edition, 2001.
3. Roger Burkhart. The swarm multi-agent simulation system. In *OOPSLA'94 - Workshop on the Object Engine*, 1994.
4. Mitsuru Oshima Danny B. Lange. *Programming and Deploying Mobile Agents with Java Aglets*. Peachpit Press, 1998.
5. F. Bellifemine et al. Jade - a fipa-compliant agent framework. In *Proceedings of PAAM'99*, pages 97–108, 1999.
6. Mao Cheny et all. *RoboCup Soccer Server*, version 7.07 user manual edition, 2002.
7. Michael Fisher. Representing abstract agent architectures. In *ATAL*, 1998.
8. E. J. Friedman-Hill. Jess, the java expert system shell. Technical Report SAND98-8206, Sandia National Laboratories.
9. Robert S. Gray. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents*, Baltimore, Maryland, December 1995.
10. Mark R. Cutkosky Heecheol Jeon, Charles Petrie. Jatlite: A java agent infrastructure with message routing. *IEEE INTERNET COMPUTING*, 2000.
11. Yoav Shoam. Agent oriented programming. *Artificial Intelligence*, 60:139–159, 1993.
12. Yannis Labrou Tim Finnin and James Mayfield. Kqml as an agent communication language. In Jeffrey M. Bradshaw, editor, *Software Agents*. The MIT Press, 1997.
13. Richard T. Vaugham. *Stage: A Multiple Robot Simulator*, version 0.8c user manual edition, 2000.
14. James E. White. Mobile agents. In Jeffrey M. Bradshaw, editor, *Software Agents*. The MIT Press, 1997.