Pattern-oriented Implementation for Automatic and Deliberative Skills of a Mobile Robot

M. Salichs, F. Rodríguez, D.M. Rivero*, A. Khamis

Department of systems Engineering and Automation, Carlos III University of Madrid
E-mail: {salichs, urbano, akhamis}@ing.uc3m.es
* Los Andes University, Avda. Tulio Febres Cordero, Mérida, Venezuela
Email: mrivero@ing.uc3m.es

Abstract: This paper presents a pattern-oriented implementation for automatic and deliberative skills described in AD architecture [1], addressing skill generation, sequencing and interfaces. The skill development framework is based on the concept of distributed active objects, which are autonomous execution entities with their own thread of control, and can interact with other using event notification. The visual proxy architecture used to build the skill interfaces, provides flexible user interfaces with minimal coupling relationships between subsystems. The implemented skill interfaces are being currently used in an international cooperation project in the framework of remote laboratories in the field of autonomous and teleoperated systems [2].

I. INTRODUCTION

Robot and automation systems are each day more important for the human being, helping us in the everyday life. Modern mobile robots tasks are usually tedious and error-prone. In many of these tasks, the robot software is composed of heterogeneous hardware and software components, which are interrelated. A large number of different skills must be integrated, into a sequential-engineered pieces of software, like the ones used for processing sensor information, for controlling actuators, for performing complex computations such as planning, and for automatic skills (go to goal avoiding obstacle, corridor following).

Recently, several research teams [3][4] are working to develop software components and framework that can be used in any architecture for control software. These components are based on new software technologies. Within the research project OROCO [3] a basic component object to real time control has been developed. Mobility [4] is a basic component framework to build a complex application, but its use is difficult, due to its poor documentation.

Complex applications on mobile robotics are rarely built from scratch. They usually derive from the reuse of similar applications, or parts of them. These applications can be used as a reusable components, and form part of other applications.

In this paper, we present a pattern-oriented implementation for automatic and deliberative skills for mobile robots described in the Automatic/Deliberative Architecture, proposed by Barber [1]. Human-robot interfaces are also described to activate or deactivate these movement skills remotely.

The paper is organized as follows. In section II the components, framework, and design pattern concepts are presented. Section III discusses the movement skills development based on AD architecture. The skill interfaces required to activate/deactivate the skill is presented in section IV, and finally, section V presents the main conclusions and future developments.

II. SOFTWARE PATTERNS AND FRAMEWORKS

Object oriented technology supports the development of software applications by offering three basic techniques: Encapsulation, Inheritance and Polymorphism. These properties have the advantage of simplifying the development of highly modular systems.

Framework, components and product line architectures are new techniques that offer partial views and solutions to the problem of developing modular, reusable, stable and efficient software systems [5][6].

A component is a piece of software that delivers a service that other programs can use [3]. The delivering of this service can be done with several degrees of flexibility to permit tailoring specific applications. Each class of the proposed framework in this article has been designed and implemented taking into consideration the above concept.

The application’s behaviors are the result of the collaboration among its component objects. For example, a task performed by a mobile robot can consist of several skills executing multiple activities in a concurrent way, e.g., go to point, look for obstacle, schedule, etc.

A framework is a design and implementation providing one possible solution in a specific domain. This is a set of components interacting under some predefined rules and conditions [5]. A framework provides functionality, with fixed aspects that can not be changed, and variable aspects that can be changed.

Different systems can be created from a framework depending on how variable aspects are configured. Thus, a framework defines a family of programs. The variable aspect is called hot spots of the framework, and the fixed aspects, the frozen spots [7]. To design a flexible
framework is necessary to define hot spots that the user will rewrite to tailor a specific application. The interaction of these classes is, typically, part of the frozen spot. In the proposed work the variable aspects are used to adapt our framework to the specific required skill. A framework is more than a library of software components: it defines the common architecture underlying the concrete application built on the framework [1][8]. Frameworks can be classified by their scope as follows:

- System infrastructure frameworks: simplify the development of portable and efficient system infrastructures such as operating systems and communication frameworks.
- Middleware integration frameworks: enhance the ability of software developers to modularize, reuse, and extend their system infrastructure to work seamlessly in distributed environments. Common example includes message oriented middleware and Object Request Brokers (CORBA) [9][10][11].
- Enterprise application frameworks: are structured collections of software components conceived for specific application domains. They solve most of the design and implementation problems common to applications in that domain and are usually built on top of middleware frameworks. One example on mobile robotics is Mobility [4].
- Global information frameworks: are used on large scale and global application domains. The concept of global information framework is still a research issue in robotics and automation.

Design patterns are design solutions [12] (not implementations) that can be used on different problems, and they have many implementations. Design patterns help software reuse, and allow the description of the structure and the collaboration scheme between components at a high abstraction level. Design patterns help to understand and develop frameworks [4], and are good tools for documentation, maintainability and portability. The most important patterns used in our framework are:

- Active objects [13]: are objects that possess, create and internally manage one or more threads of control called activities. Each complex skill is viewed as an active object.
- Publisher-subscriber [12]: define a one-to-many dependency between objects, so that when one object change state (EventManager class), all its dependents are notified and updated automatically.
- State [12]: allows an object to alter its behavior when its internal state changes. The execution-engine skill (IdleState, ExecState, ErrorState and de AbortState class) is based on a state pattern.
- Abstract Factory [12]: Dynamical object creation. The FactoryState class is used to dynamical creation of state class.
- Strategy [12]: defines a family of algorithms that are encapsulated separately, and can be interchanged easily. Strategy allows the use of several algorithms by clients in a transparent way. This pattern is used in our framework to obtain a data skill in standard format.
- Visual proxy [14]: used to build the human-robot interfaces and it provides flexible user interfaces with minimal coupling relationships between subsystems. The generation of the user interface is entirely separated from the abstraction layer object to provide the reusability and extensibility facilitates.
- Object Request Broker (ORB) pattern [15]: used for middleware/server interaction.

### III. MOVEMENT SKILLS DEVELOPMENT FRAMEWORK

A skill represents the robot’s ability to perform a particular task. The skill servers are implemented based on two level control architecture called AD (Deliberative and Automatic levels) proposed by Barber [1].

#### A. Automatic Level

In this level, there exist low-level control modules, which act directly upon the actuators, as well as the modules that collect data from the different sensors of the system. The sensorimotor and the sensorial skills are found in this level. The first are in charge of the robot motion. The second ones detect the events needed to produce the sequencer transitions, which manage the task performed by the robot.

#### B. Deliberative Level

In this level we find modules that require reasoning or decision capacity. Those modules do not produce immediate responses. They need to process the information they work with. Those modules will form the deliberative skills, and they will be activated by a sequencer, that will be in charge of managing the correct performance of these skills. This level is formed by a series of skills named deliberative skills, a long-term memory where information is obtained as well as a sequencer that activates and deactivates the deliberative skills. The path planner, the environment modeler and the task supervisor are some of the skills included in the deliberative level [16].

#### C. Skill Structure

In the AD architecture, skills are client-server modules. Each skill is implemented as a distributed object. Each module contains an active object, an event manager object and data objects as shown in Fig.1.

Objects are separated units of software with an identity, interfaces and state. The active object has its own thread of control and it is in charge of processing. The processing results are stored in the data objects. These objects contain different data structures depending on type of data stored but the interfaces are similar. During the processing, the active object can generate events. Events are sent to the event manager object, which is in charge of notifying them of skills, which have registered on it. In order to communicate among objects of the same module or different modules Common Object Request Broker Architecture (CORBA) is used [15].
When a skill is activated, it connects to data objects of other skills or to sensors' servers as required by the skill. Then, it processes the received input information, and finally, it stores the output results in its data objects. If the skill is sensorimotor, it can connect to actuators' servers in order to send them movement commands.

A skill can send a report about its state while it is active or when it is deactivated. For example, the skill called gotogoal can inform on whether the robot has achieved the goal or not. When this skill is deactivated it might inform about the error between the current robot position and the goal [17].

**D. Skill Sequencing**

The framework presented in this paper, permits the construction of deliberative and automatic skills in an easy way. A basic skill is defined as the capacity of a robot to process sensory information and select the adequate action. The skills can be classified into perceptive and sensorimotor skills [17].

- **Perceptive Skills:** are those skills which interpret the information obtained from sensors or other perceptive or sensor-motor skills.
- **Sensorimotor Skills:** are those skills which obtain as input the information provided by sensors or other perceptive skills, and based on this information, choose the most adequate actions for the actuators.

Skills can be combined, in order, to obtain more complex skills. A skill hierarchy is defined placing in the lower level the skills in charge of the communication among the actuators and sensors. These skills combination will allow:

- To reuse the software components of the skills already developed making it possible to use each skill in more than one different skill.
- To use the principle of divide and conquer, when building complex skills,
- splitting them into simpler skills and sequencing their execution.
- High integration between perceptive and sensor-motor skills.
- Distribute processing of the skills

The framework can be classified as a white box and enterprise application frameworks, its design and construction has been performed incrementally, using the unified process of software development [18] and the methodology proposed by Fayad [6].

The Class Diagram shown in figure 2 represents a software architecture of the framework. This diagram establishes that each skill (ComplexSkill) is an active object, and it is composed of data object (DataSkill), that are used to do internal computing. The setSample() interface method is used to store the internal data whereas the getSample() method is used to obtain a internal state of the data in a particular representation.

**Fig. 1 Skill Structure**

When a skill is activated, it connects to data objects of other skills or to sensors' servers as required by the skill. Then, it processes the received input information, and finally, it stores the output results in its data objects. If the skill is sensorimotor, it can connect to actuators' servers in order to send them movement commands.

A skill can send a report about its state while it is active or when it is deactivated. For example, the skill called gotogoal can inform on whether the robot has achieved the goal or not. When this skill is deactivated it might inform about the error between the current robot position and the goal [17].

**D. Skill Sequencing**

The framework presented in this paper, permits the construction of deliberative and automatic skills in an easy way. A basic skill is defined as the capacity of a robot to process sensory information and select the adequate action. The skills can be classified into perceptive and sensorimotor skills [17].

- **Perceptive Skills:** are those skills which interpret the information obtained from sensors or other perceptive or sensor-motor skills.
- **Sensorimotor Skills:** are those skills which obtain as input the information provided by sensors or other perceptive skills, and based on this information, choose the most adequate actions for the actuators.

Skills can be combined, in order, to obtain more complex skills. A skill hierarchy is defined placing in the lower level the skills in charge of the communication among the actuators and sensors. These skills combination will allow:

- To reuse the software components of the skills already developed making it possible to use each skill in more than one different skill.
- To use the principle of divide and conquer, when building complex skills,
- splitting them into simpler skills and sequencing their execution.
- High integration between perceptive and sensor-motor skills.
- Distribute processing of the skills

The framework can be classified as a white box and enterprise application frameworks, its design and construction has been performed incrementally, using the unified process of software development [18] and the methodology proposed by Fayad [6].

The Class Diagram shown in figure 2 represents a software architecture of the framework. This diagram establishes that each skill (ComplexSkill) is an active object, and it is composed of data object (DataSkill), that are used to do internal computing. The setSample() interface method is used to store the internal data whereas the getSample() method is used to obtain a internal state of the data in a particular representation.

**Fig. 2 Framework Class Diagram**

The DataSkill class is associated with a common data class (CommonData) based on strategy pattern design. The getData() method in this class may be rewritten by the developer of the new skill. This method is requested by other ComplexSkill objects interested on knowing the DataSkill state in a standard format.

The complex skill execution will be carried out by the state pattern design in the NetState class. The NetState has associated to it a net implementation (NetImplCon). Different kind of net implementation can be used. A Petri net implementation was used in our project, but the user can use the abstract class INT-NetImpl and create his/her own implementation by rewriting each interface method.

The net implementation has two types of nodes: the location nodes and the transition nodes.

The location nodes are used to define the skills sequence which must be controlled during the execution, being most of them sensorial-motors skills. The transition nodes are related to sensorial skills. Each node will have associated a logical expression which is evaluated by the Expression class during the perceptive skill execution.

Each logical expression is evaluate in a fork process. The logic execution is represented in the State Diagram, as shown in figure 3. Initially, the skill is in an idle state (IdleState). In this state the net implementation of the sequencer is built. If the developer of the complex skill need to tailor a specific behavior the onIdle() hot spot method will be rewritten. Once the skill is activated, it moves to the execution state (ExecState), the skill related
to the location and transition node in each process are concurrently executed. Each transition has associated logical expression, which must be evaluated to know if it is possible to pass to a new state or not.

During the skill execution a known error can occur. In this case the execution method of the error State class (ErrorState) is enabled. If an unknown error occurs, the failure state (AbortState) class is activated. In this state the user need to rewrite an execute method to recover the process. If recovering process is not possible, an error or idle State is activated. Each state has several hot spots, the user can rewrite to tailor a specific skill.

Additionally, the ComplexSkill and the ExecState classes have associated an event and error manager class, which are implemented as a publisher subscriber pattern. These classes are used to send notification to any other skill interested in being notified. The ExecState class inherits from the EventChangeHandle class, allowing the handling of the incoming events. The change events are a discrete data type.

Every skill is implemented as a CORBA component, that can be viewed as a client of basic skill and server for more complex skills. Each skill is built as a component module (Module Component), that has its own net state, data skill, net implementation, event manager. The own expressions are evaluated in concurrent mode. When some of the expressions give a true value, the process node related to this expression is executed.

Each complex skill has associated plan to execute. This plan is stored as a text file, and the built() interface method is charged to build a specific structure. A plan language was defined to specify the skill sequence that must be executed to build a complex skill. This language allows to indicate what skills must be executed, their execution parameters and the expressions describing the transition conditions that must be reached.

This language syntax was expressed in a grammar independent context, using a Backus Naur Form (BNF). Two examples written in this language are presented in the next subsection.

### E. Complex Skill Examples

The skill generation framework is based on the concept of distributed active objects, which are autonomous execution entities with their own thread of control, and can interact with other using event notification.

The framework development was built in C++ using CORBA (OmniORB 3.0) [19] and Linux as OS. The name service is used to find other CORBA components, whereas a Dynamic Interface Invocation (DII) was used to request a service to other components without the need to know their stub.

The experimental results were carried out in a RWI-B21 robot, equipped with a vision system. The robot has two on board computers; one for image acquisition and processing and the other to execute the motion commands and obtain data from a laser and the other robot sensors.

Several complex skills have been developed and tested successfully. Some of these complex skills are: Find a Door using a Laser sensor (FindDoorLaser) and Go to Lab (GotoLab). The first example (1a) presents code listening to execute FindDoorLaser complex skill. This skill looks for a door while the robot is moving through a corridor. In this example the onEndProc() hot spot method is rewritten, and each time the transition node is triggered the onEndProc() method is executed to evaluate if the FindObjectLaser has triggered or not. A counter was used to calculate a number of hits. If this number is equal to some value, an event associated with this skill is modified to notify process. The example 1b presents the complex skill Go to Lab.

**Listening 1a: FindDoorLaser**

```
PROC1 { CorridorFollow(2.0,0.0,0.05,0.5,YES) [CorridorFollow(0,0,0.0):(EndCorridor, YES)] PROC 3 [FindObjectlaser():(FindObjectLaser, FOUND)] PROC 2 }
PROC2 [ FindObjectlaser() [FindObjectlaser():(FindObjectLaser, FOUND)] PROC 3 ] PROC3 { stop() }
```

**Listening 1b: Go to Lab**

```
PROC1 { corridorFollow(6.0,0.0,0.0, [findLaserDoor:[findLaserDoor, FOUND]) PROC 2 [corridorFollow ( ):CorridorEnd:YES) PROC 5 ] PROC 2{ [approDoor(6.0,0.0,0.0,0.05,0.5,YES) [approDoor(6.0,0.0,0.0,0.05,0.5,YES) :ApproachDoor:YES]PROC 3 [approDoor(6.0,0.0,0.0,0.05,0.5,YES) :ApproachDoor,NO)PROC 4] ) PROC 3 { [crossDoor () [crossDoor() (DoorCrosed:CROSSED) PROC 4 [crossDoor() (DoorCrosed:NO_CROSSED) PROC 5 ] PROC 4{rotate()} [rotate() ROTATE:YES] PROC 5 ] PROC 5 {stop()}
```
IV. SKILL INTERFACES

Carlos III University of Madrid is participating in a broader international cooperation effort in the framework of remote laboratories in the field of mechatronics and telematics [2]. Our contribution in this project is mainly to provide an Internet-based remote laboratory in the field of indoor mobile robotics, addressing different approaches to solve the main problems of mobile robots such as movement skills, environment mapping, localization, path planning, etc...[20]. An intuitive user interface is required for inexperienced people to control the robot remotely.

To implement these interfaces, event-based control approach has been used to guarantee the system stability in the presence of network time delay. In this approach, non-time based motion reference is used. This reference is usually related directly to real time sensor measurements and the task and thus time delay will not have effect on the stability [21]. The role of the user in the closed loop is just to activate or deactivate the skill so the control loop is not sensitive to communication delay. The idea of overcoming communication constraints by communicating at more abstract level and increasing the robot’s autonomy is fundamental to remote control via constrained communications.

The user requests sent by the client tier are received by a servlet in the middleware tier (web server) as http requests. After receiving the http-based user requests, the servlet defines Object Request Broker (ORB) required to communicate with the server tier using IIOP protocol, then the servlet defines a CORBA object, which may be sensor object or control object.

The server tier consists of subsystems based on the described skill control framework. By using these subsystems the ORB objects are implemented by an encapsulated and modular manner. The object to be implemented may be a sensor object to invoke the sensor data or a control object to send control commands to the robot actuators. The following subsections describe these concepts in more details.

A. Graphic User Interfaces Design

The user interfaces are designed based on visual-proxy architecture [14] to obtain high degree of scalability and reusability. The visual-proxy pattern is in some ways a specialization of the Presentation/Abstraction/Control (PAC) architecture [22], which can be used to build user interfaces for object-oriented systems and can guarantee high degree of extensibility and reusability of the software components.

The objective of PAC architecture is to separate the generation of the user interface entirely from the abstraction layer object to provide the reusability and extensibility facilitates. A PAC control object, is passive with respect to message flow. The messages go directly from the visual proxy (presentation layer) to the abstraction-layer object that manufactured the proxy. In the visual proxy architecture, the encapsulation is still intact in the sense that the implementation of the abstraction-layer object can change without the outside world knowing about it. Fig. 4 shows the main components of the graphic user interface (GUI) implemented in Java.

![Fig.4 GUI Components](image)

As shown in Fig.4, there are two main classes, the CommandHandler and the DynamicRepository. The CommandHandler class can be direct control, automatic skills control, deliberative skills control, or a combination of different types of control classes. This class may use the DynamicRepository class to invoke sensory data, which may be necessary to complete the control task, such as in the case of obstacle avoidance skill by using sensory data.

The DynamicRepository class provides information about sensor state and can be used to remotely acquire the sensory data. This class is implemented as a thread by implementing a Runnable interface to provide updated sensory data in real time. This data may be odometry, which indicates the actual robot position and its translational and rotational velocities or ultrasonic sensor data or laser sensor data.

The ExperimentTool class implements User_interface so it can produce visual proxies when asked. It asks the other CommandHandler and the DynamicRepository classes for visual proxies as simple Components. This class contains a constructor and implementation of all methods required by the interface but it isn’t a God class, where object-oriented systems tend to be networks of cooperating objects with no central God class that controls everything from above. It positions the asked proxies within the panel but does absolutely nothing else with them.

This class is simply a passive vehicle for holding visual proxies. The proxies communicate directly with the abstraction-level objects that create them and these abstraction layer objects can communicate with each other. If the state of an abstraction-level class changes as a result of some user input, it sends a message to another of the abstraction-level classes, which may or may not choose to update its own user interface (its proxy) as a sequence.

B. Description of a Skill Client/Server Interaction

As shown in Fig. 5 to activate or deactivate a movement skill, the user has to use the SkillGUI to send
Http-based commands to the SkillServer through SkillServlet, which serves as an intermediate server between client and server.

SkillServlet doesn't call the SkillServer directly but through an Interface Definition Language-IDL interface (SkillIDL). This class calls methods of a proxy object that implements the SkillIDL interface. Because it calls methods through an interface, it doesn't need to be aware of the fact that it is calling the methods of a stub object that is a proxy for the SkillServer object rather than the SkillServer object itself.

The stub object encapsulates the details of how calls to the SkillServer object are made and its location. These details are transparent to SkillServlet objects. Also stub objects assemble information identifying the SkillServer object, the method being called and the values of the arguments into a message. On the other end of the connection, part of the message is interpreted by a CallDispatcher object and the rest by a skeleton object. The connection classes are responsible for the transport of message between the environment of a remote SkillServlet (Web Server) and the environment of the SkillServlet (Robot Server).

The CallDispatcher receives message through the connection object from a remote stub object and then passes each message to an instance of an appropriate skeleton class, which are created by the CallDispatcher. The CallDispatcher object are responsible for identifying the SkillServer object whose method will be called by the skeleton object.

The skeleton classes are responsible for calling the methods of SkillServer objects on behalf of remote SkillServlet objects. The skeleton object extracts the argument values from the message passed to it by the CallDispatcher object and then calls the indicated method passing it the given argument values. If the called message returns a response, then the skeleton object is responsible for creating a message that contains the return value and sending it back to the stub so that the stub object return it. If the called message throws an exception, the skeleton object is responsible for creating an appropriate message.

The SkillServer classes (implemented in C++) implement SkillIDL interface. Instances of this class can be called locally through the SkillIDL interface, or remotely through a stub object that also implements the same skill interface.

C. Implementation of a Skill Client/Server Interaction

The following steps can be used to implement skill client/server interaction as shown in Fig 6.

1A: The SkillClient object creates an HTTP-based connection with the skill servlet.

```java
URL SkillServletURL = new URL(url);
URLConnection SkillServletConnection = SkillServletURL.openConnection();
SkillServletConnection.setDoOutput(true);
SkillServletConnection.setUseCaches(false);
```

2A: The client sends to the servlet the necessary parameters to activate/deactivate the skill.

```java
PrintStream out = new PrintStream(SkillServletConnection.getOutputStream());
out.println(param1);
out.println(param2);
out.println(param3);
```

1B: SkillServlet receives the parameters sent by the client to activate/deactivate the skill.

2B: A call will have been made by the servlet to initialize the object request broker (ORB).

```java
Properties props = new Properties();
props.put("org.omg.CORBA.ORBInitialPort", #);
String[] args = new String[] {"""};
ORB orb = ORB.init(args, props);
```

3B: The servlet obtains the stub object to call the remote server object. ORBs provide a mechanism that takes the logical name of an object and creates a stub object in consultation with a mechanism that uses the Registry pattern to provide location of the server object.
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
NamingContext ncRef =
NamingContextHelper.narrow(objRef);
NameComponent nc = new NameComponent("Skill",
"Framework");
NameComponent path[] = {nc};
org.omg.CORBA.Object objRefSkill =
ncRef.resolve(path);
FrameworkComponents.SystemModule system =
FrameworkComponents.SystemModuleHelper.narrow(objRefSkill);
FrameworkCore.ObjectContainer container =
system.lookup("ObjectContainer");
String[] skill_i = new String[]{"Skill_i"};
org.omg.CORBA.Object obj =
container.find_object(skill_i);
SkillRef = RobotSkill.SkillHelper.narrow(obj);

4B: The servlet calls the SkillStub object to
activate/deactivate the skill with the intention of calling
the skill object that may or may not be remote.

SkillStubRef.activate (param1, param2, param3, ...);

4B.1: The SkillStub object asks the ServletOut object to
write a message that includes the skill class name, the
method name and the arguments that the servlet object
passed to activate/deactivate the skill. The ServletOut
object passes the message through a network connection
to the ServerIn object.

1C: The CallDispatcher object extracts the class of the
object to call from the message. It then obtains the actual
object whose method is to be called. Using different
thread, the CallDispatcher object asynchronously calls the
invoke method of the SkillSkeleton object. It passes the
message and the server object whose method is to be
called to the invoke method.

1.C.1.1: The SkillSkeleton object extracts the name of the
method to call and the arguments from the message. It
then calls the server objects’s activate/deactivate method.

1.C.1.2: The SkillSkeleton object constructs the response
produced by the call to the skill object’s
activate/deactivate method. This can be a returned value
or a thrown exception. It then passes the message to the
ServerOut object’s write method, which passes the
message through a network connection to the ServerOut
object.

5B: The servlet returns the response message to the client
object.

5B.1: The ServletIn object’s read method returns the
response message to the SkillStub object.

5B.2: The SkillStub object extracted the response from
the message. If the response is a value, it returns it in
form of HTTP response. If the response is an exception, it
throws it.

3A: The ClientIn object reads the response through a
network connection (Internet) and returns it to the
SkillClient object.

D. Implemented Interfaces

Different Internet-based interfaces have been
implemented to remotely activate/deactivate movement
skills of an indoor mobile robotics.

• Direct Motion Control

This interface aims at familiarizing the user with the
mobile robot motion control and positioning as a simple
motion skill. The remote user can send direct control
commands to move the robot forward, backward or to
rotate it clockwise or counter-clockwise. Using a 2D
model for the lab and the robot, the remote user will be
able to view the effect of the sent commands.

• Mobile Devices-based Direct Control

Despite the phenomenal growth in wireless
technologies and services in recent years there has been a
relative lack of research on possible uses of these systems
in the remote control of mobile robots. This mainly a
consequence of the fact that the use of these systems has
to deal with specific problems related to the limitation of
handheld devices (slow processors, limited
memory/storage, and small displays) and wireless
network (restricted bandwidth, high latency, low
connection stability, and low predictable availability).

Once limited to the academic community, Personal
Digital Assistants (PDA)’s are now commonplace.
PDA’s are attractive interface devices because they are
lightweight, extremely portable, and feature touch-
sensitive displays.

Fig. 7 PDA-based Direct Control

A PDA-based controller has been designed to send
direct control commands to the robot’s actuator server
and to acquire the odometry data (actual location with
respect to the initial point, translational velocity and
rotational robot velocity). Fig 7 shows the PDA-
Controller interface (Compaq iPAQ 3870 PDA).
Another interface has been implemented to remotely
control the robot by using a Mobile Information Device
Profile (MIDP)-enabled cellular phone. The device user
starts a MIDlet that presents the GUI. The user would
select a control command (Forward, Backward, Stop,
Turn Left, Turn Right). Then presses a button to send the
request to the control servlet. The servlet forwards the
request to the actuator server via CORBA ORB.
• Go To Point Skill

By using this skill, the user can send the robot to a certain point in the lab. To activate this skill the user has to set the skill parameters as goal point, robot velocity and maximum error. The skill can provide information as to whether the robot has achieved the goal or not. When this skill is deactivated it might supply information about the error between the current robot position and the goal. Obstacle avoidance skill has been added to this skill, which provides the possibility to avoiding obstacles when the robot is moving toward the goal point.

• Orientation Control Skill

This skill provides a closed loop control on the robot orientation. The operator determines the desired orientation and the maximum error and then the skill can be activated to change the actual robot orientation to the desired one.

• Wall Following Skill

The robot uses wall following skill to move parallel to and offset from the laboratory’s wall according to a direction given by the user (right, left or unknown). The robot's goal is to remain a given distance from the wall as it moves. Ultrasonic sensory data are used to generate events to the actuator server according to the minimum distance determined by the user. Fig.8 shows screenshots of skill interfaces.

Fig. 8 Screenshot of Skill Interfaces

V. CONCLUSIONS AND FUTURE WORK

A framework to develop automatic and Deliberative skills has been successfully used. Several skills have been built using this framework in a very simple and easy manner and very short time. In addition, the developed basic skills have been reused to build complex ones. Moreover, different skills developed in other platform can be integrated, creating a hierarchy of skills.

The visual proxy pattern used to build the human-robot interfaces, provides flexible user interfaces with minimal coupling relationships between subsystems. The generation of the user interface is entirely separated from the abstraction layer object to provide the reusability and extensibility facilitates. A graphical interface to define plans of sequences and store it in a database is under development as a net implementation class.

In future, the way a transition node is defined will be improved, implementing each node as an agent, where each expression will be defined as a rule and each target will be defined as an action. Rules will be evaluated each time an agent realises that its belief has been changed.

VI. ACKNOWLEDGMENTS

The authors gratefully acknowledge the funds provided by the Spanish Government through the DIP2002-188 project of MCyT Ministerio de Ciencia y Tecnologíá.

REFERENCES


