A framework for building mobile robot applications. Application to multi-robot projects

Joaquin L. Fernández, Member, IEEE, Diego P. Losada and Eduardo Zalama

Abstract—The complexity of robot software systems calls for the use of some well-conceived architecture together with programming tools that support the architecture. These sets of tools are usually named robot programming frameworks. One common feature of robot architectures is the modular decomposition of systems into simpler and largely independent components. These components implement primitive actions and report events about their state. These modules are usually connected with different inter-process communication mechanisms. There are two basic communication approaches: client/server and publish/subscribe. The robot programming framework proposed here includes a tool (RoboGraph) to program and coordinate the activity (tasks) of these middleware modules. This tool is easily extensible to any architecture that uses a publish/subscribe communication approach. Project developers use the same task programming IDE (RoboGraph) in two different levels. The first one is to program tasks that must be executed by one robot and the second level is to program tasks that can include several robots and building elements.

Tasks are described using an interpreted Petri net editor and stored in an xml file. A dispatcher loads these files and executes the different Petri nets as needed. A monitor that shows the state of all the running nets is very useful for debugging and tracing purposes. The whole system has been used in two different applications: A multi-robot surveillance project (WatchBot) and a four-guide robot (GuideBot) that has been leading users in a public event (XventureGalicia.net) for the 2007 and 2008 editions.

Keywords: Mobile robot, Robot control architecture; robot programming framework.

I. INTRODUCTION

Robot control systems tend to be complex because they need to interact asynchronously and in real time with uncertain and dynamic environments. To handle this complexity, a robot control architecture and developing tools are needed. The control architecture must allow capabilities for acting in real time, control sensors and actuators, supporting concurrency and provide tools that help to develop applications. The robot programming framework also known as Robotic Development Environment (RDE) [1]

should include flexible, reusable and reliable tools for robot application programming.

During the last two decades our lab has been using different navigation toolkits to develop several mobile robot applications. While doing these projects, a framework for building mobile robot applications named Robotic Integrated Development Environment (RIDE) has been developed.

Several research groups have adopted different frameworks [1] in their applications. However, only a very few of these projects can finally work without human supervision for a long period of time, mainly because of their complexity. Nevertheless, many simple service and industrial applications can benefit from the use of autonomous robots for executing repetitive tasks performed nowadays by humans. Our main design principle developing this framework was that it should provide sufficient functionality and flexibility to meet the needs of real-time autonomous systems, being robust and reliable without weighing the control implementation down with unnecessary “bells and whistles.” Therefore, special attention has been set on programming tools that will allow us to design and build reliable mobile robot applications.

In order to build robust mobile robot applications, some of the solutions that have been successfully used in complex industrial manufacturing applications might be adapted. These manufacturing systems have to be robust control systems since they have to be working continuously for long periods of time. The framework presented here includes some of these solutions, taking into account that robot architectures have special requirements. The most important differences with industrial architectures derive from the fact that robot systems need to interact asynchronously, in real time, with an uncertain and dynamic environment.

The rest of this paper is organized as follows. Next section introduces the works related to this research. The control framework used in the mobile robot applications is presented in section III. Section IV and V describe the components of this framework and the communication system respectively. The executive layer is presented in section VI while sections VII and VIII describe the main issues creating new mobile robot applications and creating new projects of these applications respectively. Finally, section IX concludes the paper.

II. RELATED WORKS

The increasing population of robotic applications calls for
the design and implementation of robotic architectures that are becoming larger and more complicated. Therefore, a framework that integrates reusable components for which various companies and individuals contribute their technologies is required.

Companies producing and selling mobile robots provide their development libraries and software tools for building and debugging robotic applications. Examples of these frameworks are OPEN-R SDK for Sony AIBO [2] and Saphira for Pioneer robots [3]. These tools cannot be easily used for building multi-platform robotic systems because they are platform dependent.

On the other side, a number of open source robotic development frameworks have also been developed. The first approaches were based on the SPA (Sense-Plan-Act) paradigm [4]. Later, several new control architectures paradigms emerge, including reactive planning [5] and the subsumption architecture [6]. Layered control architectures appear later mainly due to the difficulty to compose behaviors to achieve long-range goals and because it proven very difficult to optimize robot behavior. Most of the frameworks used nowadays are based on layered robot control architectures. OROCOS (Open Robot Control Software) is a free software project that includes an application framework. Player/Stage [7] is a general framework for controlling a robotic system that a significant portion of the community has found useful, becoming a de facto standard in the open source robotics community. CARMEN is also a popular navigation toolkit for robotics based on a set of modules that use IPC to communicate. These toolkits do not provide specific tools to represent executive-level capabilities. However, some of the general languages designed to represent executive functions such as TDL (Task Description Language) commonly used with IPC, RAPs (Reactive Action Packages),PRS (Procedural Reasoning System) or PLEXIL (Plan Execution Interchange Language) can be used.

Petri Nets have also been adopted for representing executive functions. Some researches have already point out that several plan representations used in mobile robots can be mapped to Petri nets. For example, J. King et al. [8] described the conversion of a plan representation, using partially ordered plans (POPs) [9], into a Petri net model. Caccia et al. [10] presented a methodology for automatically transforming conventional task-variable graphs, representing the execution levels of intelligent control architectures, into Petri nets. Miliutinovic et al. [11] introduced a Robotic Task Model (RTM) based on Petri nets for a distributed robotic system.

Some applications based on mobile robots [12][13] use Petri nets to model and evaluate plans at different levels that go from motion control [14] to task supervision [15]. A few papers have also reported the use of Petri nets in mobile robots for coordination, hardware resource handling and planning [16] combining some AI planning system with Petri nets.

The framework presented here is based on a modular three layer control architecture that uses IPC for module connections. Using IPC allows adopting some robot navigation solutions from other IPC-based architectures like CARMEN. A Petri net programming environment named RoboGraph allows the use of Petri nets for representing executive functions.

Most of the works mentioned above use Petri nets to model, analyze and even test [17] the control of mobile robots. We propose to extend the use of Petri nets as a high level task application programming language through an IDE, in a similar way industrial automation companies use IEC 61131-3 compliant programming environments. This way, besides defining the Petri nets to model, analyze and test, they will also be used by a dispatcher to manage the different functional modules of the control architecture.

III. CONTROL FRAMEWORK

Our research addresses mobile robot applications that include the control, supervision and collaboration of several mobile robots (fig. 1). More specifically, it is based on a set of robots guided in dynamic environments, which are usually non-fully specified and working under real-time restrictions. The robots must be able to carry out plans remotely and react to unexpected events.

Fig. 1 shows the proposed architecture for two robots but it is extensible to multiple robot systems. It is a centralized multi-robot and multi-user system where all the robots and all the users are connected to a Central Server computer that executes the off-board planning, scheduling and resource sharing among other tasks. Users can access from anywhere on the web using a web GUI.

![Multi-robot and multi-user architecture](image-url)

Fig. 1. Multi-robot and multi-user architecture.
This kind of systems requires low and medium perceptual capabilities in order to extract, understand, and manipulate spatio-temporal information from the environment. However, this information will be used in a different manner by the on-board local control, which is mostly reactive control, and off-board remote control, which is mostly deliberative.

In general, on-board local control has to deal with safety tasks such as obstacle avoidance according to the directives of the remote global planning. Moreover, the local control response time is critical and it needs hard real-time restrictions.

On the other side, global off-board control has to deal mainly with planning tasks for multiple robots in the most efficient way. That is, it copes with the deliberative computing. However, it still needs some information about the accomplishment of the tasks in order to change the plan when it is necessary. In many cases, some other sensory information, such as camera images, is needed for the user interfaces.

IV. CONTROL COMPONENTS

The on-board control architecture is organized as shown in more detail in figure 2. Even though the different modules are organized in four sets, they can be mapped in the three layer architecture popularized by Bonasso et al. [18]. The hardware servers and control set implement the functional layer while RoboGraph Dispatch implements the executive and planning layer. Finally, the architecture includes a set of processes to interact with the users and connect to the Central Server using a wireless communication system.

![Onboard control architecture](image)

The navigation platform is based on CARMEN [19] and some modules such as localize, navigator and base hardware server interfaces remain basically the same. Unlike CARMEN, motion control is divided into high-level (strategic) planning [20] and lower-level (tactical) collision avoidance using the Beam method [21].

A. Hardware server modules

The hardware server modules govern hardware interaction, providing an abstract set of actuator and sensor interfaces and isolating the control methods from the hardware details. Most of the hardware devices are connected to a CAN bus using RoboCAN [22]. Some of these devices are used in navigation such as the laser and sonar while others are specific for the application such as the robot head, sound and speech systems.

There are some modules not shown in figure 2 such as module RF_Server used to keep the robot connected to the wireless [23] in two different ways. The first one is forcing roaming between APs and the second is to avoid non-coverage areas if possible.

Finally, some applications such as surveillance [24] need an interface with the different building devices.

Thanks to this layer, changes in hardware components can be made without having to change the higher layers modules while keeping the same interface.

B. Control modules

The control modules integrate sensor and motion information to provide improved sensor odometry, provide basic navigation capabilities (localization, path planning, follow path, etc) and basic application specific functions (say text, make expression, etc).

C. Executive modules

All the modules in this layer belong to the RoboGraph application that includes two modules: RoboGraph Gui that is used only for application development and RoboGraph dispatch. These modules will be explained in detail later.

D. Graphic and web interface modules

There are several interface modules for application developers to debug and trace the control and hardware server modules. However, there is only one interface module on board that allows the user to interact with the robot. Robot_Web Interface is the module that exchanges information between the robot and the Central Server station.

V. COMMUNICATION FRAMEWORK

The information gathered by sensory nodes must be available for both reactive and deliberative levels. However, the response time requirements are different in each level. Thus, while a significant perception must have an immediate response in the reactive level, this is not necessary in the deliberative level. For example, a sensor module can obtain a perception about an obstacle that suggests an action to avoid it. This perception has to be communicated to the control module that runs collision-avoiding algorithms as soon as possible. Therefore, these modules should be running in the
in the computer where we want to use the IPC set of modules of figure 1 to communicate any Java coded module with the central server. However, this package includes a C native library that needs to be installed in the computer where we want to run the Java module.

2) Multi-robot: The publish/subscribe IPC mechanism is broadcast. Modules subscribed to a kind of message will receive all the messages of that kind that any module will publish. This is a nice feature when thinking of a single robot control (figure 2). However due to the symmetry of the multi-robot multi-user framework (figure 1), a problem arises when dispatch wants to send a command (such as “go to point A”) to only a robot since all the robots will get the message.

3) Multi-user: Again, because of the broadcast nature of messages in IPC, a GUI could not subscribe only to messages from one robot. For example, if it subscribes to the laser message it will get the laser message of all robots.

4) Access control: According to figure 1 any user can try to monitor and send commands to any robot. However in a surveillance application this can not be allowed. When the Web user interface starts, the user needs to be identified.

A. IPC level

Each module running on-board the robot (figure 2) is a Linux process that exchanges information with other modules using IPC [25]. Developed at Carnegie Mellon's Robotics Institute, IPC provides among others, a publication-subscription model. An IPC-based system consists of an application independent central server and a number of application-specific processes. Each process connects with the central server and specifies what types of messages it publishes and what types it listens for. Any message that is passed to the central server is immediately copied to all other processes subscribed.

The framework implemented here uses IPC in centralized-routed mode. For this mode, IPC provides:

- **Central**: IPC uses an application-independent central server module to maintain system-wide information and to route and log message traffic. Before starting any module, a program named “central” must first be started. The most basic service provided by the central server is message passing. A message sent from any module connected to the server will be forwarded by the server to the module containing the handler for the message.

- **IPC interface functions**: IPC includes functions to connect to Central, subscribe to messages and register message handlers. These functions take care of opening sockets, registering messages, and sending and receiving messages. The IPC library contains functions to marshall (serialize) and unmarshall (de-serialize) data, handles data transfer between machines with different Endian conventions, invoke user-defined handlers when a message is received, and invoke user-defined callbacks at set intervals.

B. JIPC level

Regarding the multi-robot communication topology, the framework shown in figure 1 is very similar to the one using inside each robot. Each IPC set of modules of figure 1 controls one robot and is running in the onboard computer. JCentral, the building interface, Task Manager, Robograph multi-robot and the video streaming are usually running in the Central Server computer.

JIPC provides also a communication model very similar to IPC. Regarding the executive layer this is a very important issue because both provide a publication-subscription model with a similar interface. We will see later that this is a key point in order to use RoboGraph to coordinate both: the modules connected via IPC for autonomous robot tasks and modules connected via JIPC for more general tasks, including multi-robot tasks.

However, JIPC was fully developed on Java and extended for multi-robot and multi-user applications. The main differences with IPC are:

1) **All developed in Java**: The Web User Interface is, for most of the applications, a Java applet in a web page that can be started from any web browser. IPC provides a nice Java package to communicate any Java coded module with Central. However, this package includes a C native library that needs to be installed in the computer where we want to run the Java module.

2) **Multi-robot**: The publish/subscribe IPC mechanism is broadcast. Modules subscribed to a kind of message will receive all the messages of that kind that any module will publish. This is a nice feature when thinking of a single robot control (figure 2). However due to the symmetry of the multi-robot multi-user framework (figure 1), a problem arises when dispatch wants to send a command (such as “go to point A”) to only a robot since all the robots will get the message.

3) **Multi-user**: Again, because of the broadcast nature of messages in IPC, a GUI could not subscribe only to messages from one robot. For example, if it subscribes to the laser message it will get the laser message of all robots.

4) **Access control**: According to figure 1 any user can try to monitor and send commands to any robot. However in a surveillance application this can not be allowed. When the Web user interface starts, the user needs to be identified.
Afterwards, all the commands he sends are checked by JCentral referee for privileges. In a similar way, subscription to different robot resources (view camera, robot position, etc) must be approved to JCentral referee.

5) Polling mechanism: In order to use the user interfaces behind NAT mechanisms and solve the private/public connectivity problem.

Central module and JCentral module have quite different implementations. Differences range from the implementation language (C for Central and Java for JCentral) to communication functionalities. For example, JCentral needs to differentiate between temporal and final robot disconnections. A final disconnection might occur shutting down the Robot Interface module without disconnecting from JCentral. In this case, JCentral has to cancel the connection. On the other side, a robot can loose temporal connection and JCentral should be able to resume communications afterwards. For example, when a robot moves through an area without coverage, it will lose communications temporally but should be able to resume afterwards. JCentral implements a timeout mechanism to disconnect modules when communication has been stopped for longer than a threshold time.

VI. EXECUTIVE LAYER

The executive layer is responsible for task decomposition, dispatching, monitoring and error recovery. Petri nets are a popular choice for representing executive functions.

A. Petri net plan representation

Petri nets have been widely used to model, design, execute and evaluate tasks in manufacturing dynamic systems. As we have seen in the related work section, some researches have also point out that several plan representations used in mobile robots can be mapped to Petri nets. In this work we use hierarchical binary interpreted Petri nets. For the interpreted Petri nets, we associate actions (places and transitions) with the publication of messages and events (transitions) with the reception of subscribed messages.

As a simple example, figure 3 shows a couple of the Petri nets that implements the same task GO POINT. The Petri net on the left is the simplest implementation while the one on the right is able to deal with some incidences. In both cases, there is only one initial mark in the place labeled Set Goal, while the END place has been selected as a final place. If no final places were defined, the task will end when there were no marks on the Petri net.

The “publish message set_goal” action is assigned to Set Goal place. The parameters for this message are the goal coordinates. Module Navigator (figure 1) will handle the set_goal message planning a new path and publishing the navigator_plan message that contains the planned path.

Transition labeled Path has associated the “receive navigator_plan message” event. When this event occurs and Set Goal place is marked, the transition will be fired.

Place Go has the “publish message follow_path” action assigned. The BEAM module (figure 2) will handle this message, activating the follow_path behavior. Once the goal is reached, the autonomous_stopped message is issued by the BEAM module. This message is associated to the STOP transition. The END place is the only final place; therefore, the task will finish publishing the corresponding end_task message.

Since the robot is working on a real complex dynamic environment, several problems can arise while following the path. A Petri net that takes into account some of these problems is shown on the right part of figure 3. While following the path (Go place), the navigation system can publish a message autonomous_stopped, but with a parameter setting that the goal has not been reached. This event is associated to transition No Goal. Another mechanism to detect possible problems is the use of a timer for the follow_path action. The timer is initialized in the Path transition and validated in the Time out transition. Therefore, while the Go place is marked any of the three transitions can be fired. Two of the transitions denote an incidence and special actions could be taken in this Petri net for each transition. However, in this simple example, the incidence is not handled here; instead, it is propagated to the parent task using the end_task message. This message has a parameter where a return value with the kind of incidence can be stored. This parameter is used in the Petri net of figure 4 by transitions OK and ERROR to check if the “GO POINT” Petri net has reached the goal.

Figure 4 defines SHOW POINT task that uses the task GO_POINT defined above. In the initial place a start_task message is issued with several parameters, such as the name of the Petri net GO_POINT, Petri net identity (PN1), user identity (by default the same as requested “SHOW POINT”).
running mode and goal coordinates. RoboGraph dispatch is subscribed to this message as well as the end_task message.

![Petri net diagram](image)

Fig. 4. Petri net that implements the task “SHOW POINT”. Initial place includes the execution of task defined in figure 3. When the subnet finishes it execution a message is issued and a string return value is used to know the outcome of the execution.

If the end_task message form PN1 is received reporting success in the GO_POINT task, transition OK will be fired and actions associated to say text and face person are executed in parallel. Both threads of execution are synchronized in transition STOP because places say wait and face wait need to be marked for the transition STOP be fired.

If the end_task message for PN1 is received reporting failure in the GO_POINT task, transition ERROR will be fired skipping the presentation on this point.

Resource sharing, scheduling, and any other of the Petri nets representation features can be used to define almost any complex task.

B. Task Manager

Petri nets are a powerful tool to model, analyze and design distributed sequential and/or concurrent systems. They have been successfully used describing and analyzing tasks with sequential and parallel actions in complex industrial manufacturing applications. However, they have not been equally accepted for coordination, hardware resource handling and planning in complex manufacturing systems. Instead, modules developed in conventional programming languages are more popular. For this purpose we developed the Task Manager.

For example, in a surveillance application, a patrol task for a mobile robot can be easily defined with a Petri net using a sequence of actions that include moving to the different points to visit. However, deciding which robot should we use can be easily decided using different criteria in a conventional programming language. Therefore, if during the execution of a task (Petri net) a hardware resource is needed, a request message is sent to the Task Manager that should reply with the assigned resource. The name of the resource can be stored in a Petri net variable for later use.

VII. Creating New Mobile Robot Applications

According to the framework presented in figure 1, in order to develop a new mobile robot application, different programming levels must be handled:

1. Basic modules level. These modules range from hardware drivers that control new devices to modules that provide new functionality necessary for the new application. Fortunately, we have found that with this framework we can directly migrate modules from other applications and very few modules need to be added. IPC modules from other frameworks such as CARMEN are also easily integrated. Even though most of these modules are running on the robot, in some applications we needed to develop modules for the Central Server computer. For example, a building control module that provides building sensor information and basic primitives to control elevators and doors.

2. Task definition level. Once the modules to provide the basic primitives are built, tasks are defined as Petri Nets using Robograph. There are some basic tasks such as go_to_point that are general to all applications. However, this level is more application dependent. Fortunately, programming new tasks is done quite fast using RoboGraph.

3. Task Manager level. This level has been recently added for resource allocation, coordination and tasks assignment.

A. Basic modules level.

These modules provide the primitives that are going to be used in the tasks (Petri nets). Basic modules are programmed using different programming languages (C, C++, Java, Lisp). Since the modules communicate via a socket based interface (IPC and JIPC), they can be written using any language with socket support. The interface with a module is defined by the messages that publishes and subscribes.

Robot Web interface module is generic for all the applications. This module acts as a gateway between IPC and JIPC subscribing to messages in IPC (JIPC) and publishing them in JIPC (IPC). This module is application-independent and only an xml configuration file changes among applications. The configuration file contains the messages that should be exchanged, in both directions, between both communication systems.

For the applications where the robots need to interact with the building, sensors and actuators of the edifice are connected to the Central Server computer using a Modbus fieldbus. The building control module (Building interface in figure 1) manages the Modbus communications providing sensor readings and actuator commands to JIPC modules.
Graphical user interfaces are application-dependent even though we have found some general graphical elements that are common to most of the applications. For example, mobile robot applications monitored from the web usually include the map of the area where the robots move, robots positions and information from different sensors such as proximity sensors or images from cameras.

There are two different kind of GUIs:

- Robot onboard GUI. Usually web monitored applications such as surveillance have a rather simple onboard GUI visualized in small touch screens. However, for more robot-human interaction applications such as tour guide robots, they are usually more sophisticated and are visualized in bigger touch screens since they play a key role in user acceptance.

- Remote access GUI. Most of these GUIs are used to monitor the application from the web or some local area network.

GUIs are programmed in JAVA since they can be easily executed from everywhere on Internet and can also be included in a web page. Robot onboard GUIs are IPC modules and remote access GUIs are JIPC modules.

B. Task definition level. RoboGraph

Figure 1 shows the programs that form RoboGraph. The GUI (Graphical User Interface) is a development tool that make possible to create, edit and monitor the execution of the different tasks while the dispatch module is in charge of executing those tasks.

1) GUI

This program can work in three different modes: Editor, Monitor and Play Logger.

In editor mode, the user can create new tasks using a simple and intuitive Petri net graphical editor. Figure 5 shows the GUI while editing a Task. Petri net structure is created by selecting and dragging the different elements: places, transitions, arcs and marks. Then actions, associated to places and transitions, and conditions, associated to transitions, must be defined.

Actions can be commands implemented in any module in the control architecture of figure 1. These commands can be selected from a menu list automatically generated by the GUI. For example, the second menu list on figure 5 deployed when clicking in a transition includes, among others, the modules with primitives available. Selecting one module (pantilt module was selected in figure 5) a third menu list shows the commands (also events in the case of a transition) for the selected module. Each command is an IPC (JIPC) message and the user must define the command parameters that will automatically appear in a new window when that command is selected in the editor.

When Dispatch executes the Petri net, the IPC messages assigned to places and transitions will be published as the net progresses. There are also available some special commands such as start and stop another task (Petri net) or start a timer.

Conditions can be events produced by any module in figure 1. These events can be selected from the menu list generated automatically by the GUI. An event can be the arrival of an IPC message, a condition on some IPC message parameter or any logical expression on several parameters over the same or different IPC messages. RoboGraph GUI allows defining any logical expression over the message fields. However, complex conditions over message fields are sometimes more naturally expressed using other programming languages. For these cases, a Java-like editor is also integrated in RoboGraph to program conditions and actions associated to places and transitions.

Timers are a tool widely used in automation that comes very handy here. In addition, in our applications we have also used them as an error detection mechanism in order to time some actions of different modules. Actions can start a timer while conditions can test the value of a timer.

Global variables are used to get starting data and store information to share conditions and events in different places and/or transitions.

GUI in monitor mode connects to central (IPC) and subscribes to different dispatch messages that show the status of the different running or waiting Petri nets. Every running Petri net is shown in a different tab with the current marking as in figure 6. When dispatch evolves a Petri net marking, an IPC message is issued and GUI will update the monitor tabs. Therefore, using monitor we can see in a snapshot the status of the system, since the marking of the running Petri nets represents its status. This is a very helpful tool when debugging an application. An information window with the queued tasks (Petri nets) and IPC messages can also be displayed on the left tabbed pane of figure 6.

Fig. 5. RoboGraph GUI in editor mode.
In order to reach this project goal, a JIPC module (robot) should be sent or who request them.

The difference is that for JIPC interface functions it is necessary to define not only the commands but also to know which module (robot) should be sent or who request them.

C. Task manager Level

This is a JIPC module that handles resource sharing, coordination and planning. These functions can be quite different from one robotic application to another even thought some of them are common to most mobile robot applications. For example, tasks need to be assigned to robots in a multi-robot application. However, in some applications all the robots might have the same devices and can carry out the same tasks while in other applications robots might have different abilities and only a few of them can carry out specific tasks.

Recently, an application where the robots need to transport items labeled with RFID tags had been developed. In this case, the number in the RFID tag defines the destination of the item. Therefore, when the robot gets a new item, the identification number is obtained with an onboard RFID reader. This number is sent in a message to the Task manager that should replies with a destination address. The transport task is a Petri net with a sequence of primitive actions and events that include the action to request the destination address and waits for the assigned destination address event.

VIII. CREATING NEW APPLICATION PROJECTS

Mobile robot applications such as a surveillance system [24] or a tour guide system were developed using this framework. Even more, we have built several projects of each application. The application should be easily installed and adapted to create different projects. For this purpose, a configuration GUI for each application becomes necessary. In order to reach this project-independent feature, all the parameters that might be project-specific are stored in a xml file that can be edited by installers using the application configuration GUI.

For example, a surveillance application project differs from another in the working area (map) where the robots are...
going to move. Besides the maps, some tasks might have parameters that also depend on the project. For example, the patrol task will depend on the points to visit on the map and the schedulers that define the time when the patrol should be started. Patrol task is generic but workers that install the project should instantiate this task assigning running parameters such as the points to visit and starting parameters such as schedulers or events. All these parameters are easily defined by installation workers using the application configuration GUI and stored in a xml file that defines the project configuration.

The tour guide application presents a similar pattern. Tasks include guide people following predefined routes where the points of the routes, texts or multimedia files to reproduce and maps are project specific parameters that can be defined using the project configuration GUI.

For some projects it might also be necessary to add new primitives (basic modules), tasks (Petri nets) or new resource sharing functions (Task manager). This work has to be made by qualified application developers.

IX. RESULTS AND CONCLUSIONS

A new framework for building mobile robot applications has been presented. Three programming levels are needed to develop a new application. Basic primitives are provided by modules that can be developed using different programming languages according to their nature. Tasks are defined as Petri nets using RoboGraph IDE allowing a rapid task definition and debugging. Finally resource sharing, coordination and planning are carried out by the Task manager module.

This framework provides several features building mobile robot applications:

- Hardware abstraction. Hardware server modules provide a hardware abstraction layer.
- Enhanced scalability. The framework is modular, flexible and easily extensible.
- Real-time performance. In order to deal with soft and hard real-time requirements two communication layers have been used.
- Programming Language independence. Application developers can choose the appropriate programming language to build basic modules.
- Maintainability. Modular systems are usually easy to maintain, update and scale. Tracing and debugging problems are easier to settle when the system state can be seen looking at the evolution of a Petri net rather than monitoring a set of variables.
- Module reusability. A key requirement to promote software reuse is to loosen the coupling between software modules. Each module in figure 1 is a Linux independent process. Basic modules and even some tasks will remain unchanged from application to application.

- Reduce development time. In similar applications, most of the modules remain without changes and only the edition of the Petri nets and GUI modules will be necessary for a new application.
- Training. Almost everybody that has worked or learn to use IEC 61131-3 compliant programming environments (Siemens S7 Graph, Graphcet, etc.) will be able to program new tasks using RoboGraph.

Petri net properties also make them good candidates for qualitative (un-timed models) performance evaluation and quantitative (timed models) performance evaluation of robotic tasks. Significant research has been done in this area for industrial applications [26][27] and also some in mobile robots tasks [12][13]. This is one of our future research areas in order to provide RIDE with task analysis tools.

Several frameworks have been developed that support various features building mobile robot application described before. Therefore, a comparison with the already available frameworks is necessary. Also, several schemas for comparing these frameworks also known as Robot Development Environments (RDEs) have been proposed. Kramer et al [1] established a comprehensive list of evaluation criteria targeted at robotics applications to compare the RDEs strengths and weakness. The evaluation was carried out on nine open source, freely available RDEs including CARMEN. The onboard control architecture presented in figure 2 is based on a set of IPC communications modules based on CARMEN. Therefore, the basic performance evaluation [1] for CARMEN is valid for RIDE. However, a few new tools that mitigate some of the CARMEN weakness were included in RIDE. The rest of this section we will review the evaluation criteria where CARMEN performs worse and the improvement obtained with RIDE tools. According to Kramer et al [1], the main CARMEN weaknesses are:

- Monitoring/Management. CARMEN does not have coherent system-wide facilities to monitor and manage the components (modules in figure 2). RoboGraph provides mechanisms to monitor and manage the components connected to IPC as described in section VI.
- Security. CARMEN does not provide a way to securely authenticate modules that connect to IPC neither RIDE. However, in RIDE, there are mechanisms to authenticate the modules connected JIPC such as robots and users.
- High level language. CARMEN does not provide a higher-level language for behaviour description or agent communication. RoboGraph allows users to define any kind of tasks including behaviours using a graphical editor based on Petri Nets. It should also be noticed that, even not included in CARMEN and therefore not mentioned by Kramer et al [1], there are languages such as the Task Description Language TDL [28] that can be used together with IPC.
• Architecture execution. RIDE includes a graphical user module control that allows starting a single module or sequences of modules that might form a mobile robot application. These sequences and running parameters can be defined and stored in a XML file.

The framework for comparing agent systems proposed by Kramer et al [1] does not include criteria to evaluate multi-robot and multi-user RDE adaptability.

Using RIDE framework, a couple of applications have already been developed. The first one is a tour guide robot named GuideBot and the second a surveillance and security system based on mobile robots named WatchBot.

The tour guide robot application participated in the “Xunventude Galicia Net” for the 2007 and 2008 editions with different robots. For the 2007 edition a Peoplebot base was working during three days in the “Palacio de Congresos y Exposiciones de Galicia”, Santiago de Compostela (Spain). For the 2008 edition a B21 base with a robotic head (figure 7) was used. Even though two different platforms have been used, most of the modules for the 2007 edition were also used in 2008. Main changes in the basic modules were that the base control module was changed and the head control was added. In the Task definition level a few Petri nets were created in order to include the new primitives. For example, the robotic head moves the mouth while the speech module was reproducing a text. Finally, a few configuration project parameters such as the places to visit, the texts to say were changed.

![Fig. 7. Tour guide robot application (GuideBot).](image)

The main challenge in both editions was the short period of time for adapting the project. The environment is a set of stands; most of them mounted a few days before. Furthermore, some of the tasks are not fully defined until a few hours before the starting of the event. Therefore, a tool like RoboGraph to quickly create, change and debug tasks becomes necessary.

For the robot tour guide only a robot was used and it was not need for the Task Manager. However, for the security system a Task Manager was needed and added to the framework. Currently, a Task Manager is also been added to the tour guide robot.

![Fig. 8. Surveillance application robots (WatchBot).](image)

The security and surveillance application has been recently finished and presented [29] even though a former version was already presented in [24]. Figure 8 shows one of the modified pioneer based platforms used.

![Fig. 9. Surveillance users GUI.](image)

This application includes all the elements of the framework and includes an easy project configuration GUI where installation workers can define all the project parameters and store them in an xml file. The user GUI is shown in figure 9.
ACKNOWLEDGMENTS

We would like to thank all the people that have influenced this work. In particular to the people that developed and contributed to CARMEN that we use as base architecture for this research.

REFERENCES


