PERGAMON

Robotics and Computer Integrated Manufacturing 15 (1999) 431–440

Robotics and Computer Integrated Manufacturing

www.elsevier.com/locate/rcim

The NEXUS open system for integrating robotic software[☆]

Juan A. Fernandez*, Javier Gonzalez

Departamento de Ingenieria de Sistemas y Automatica (ISA), E.T.S.I. Informatica, Universidad de Malaga, Campus Teatinos - 29080 Malaga, Spain

Received 1 October 1998; received in revised form 29 March 1999; accepted 30 April 1999

Abstract

In this paper a framework for constructing flexible, robust and efficient software applications for robots is described. The basic concepts needed to integrate complex, multidisciplinary robot software architectures are identified, and the methods to achieve them are taken from different areas of research (programming languages, network communication systems, real-time systems, etc.). The result is an open software system called NEXUS which includes the basic characteristics needed for the integration of very different software modules, minimizing the effort of integration and maximizing the reusability, efficiency and robustness of the resulting software applications. This software has proven to be a basis for more sophisticated tools that help in reducing the cost of modifications to and the complexity of multidisciplinary projects, allowing highly structured and reusable designs to be implemented. Although it has been currently implemented for mobile robots, it is a sufficiently generic framework suitable for use in other control systems. © 1999 Elsevier Science Ltd. All rights reserved.

Keywords: Robot control; Real time operating systems; Open control architecture; Robot programming

1. Introduction

In multidisciplinary areas of research such as robotics, a need exists for using a framework that enables a complex software application to connect its elements easily and flexibly, while providing the designers with well-defined methods for upgrading the overall structure of the application through its entire development cycle. This becomes especially necessary due to the number of people from different disciplines who can be involved in the research project. An example is mobile robotics, where software must be developed for device controllers, perception systems, remote operation, artificial intelligence, etc.

The elements involved in the implementation of a robotic application can be divided into *software for the application*, *hardware devices*, and *operating systems*. The *software for the application* can take advantage of a common integration system, while the *hardware devices* and *operating systems* tend to be more vendor-dependent and rigid in this respect. Therefore, a suitable solution to the problem of integration is an open software system [1] which is located between the operating systems and the software for the application (see Fig. 1).

We have identified the following basic characteristics that an open software system for integrating parts of a robotic application should provide:

Portability: Any program should be able to run on any platform with minor changes in its internal structure.

Upgradeability and Reusability: Any program should be able to extend its capabilities without affecting existing applications, or to be substituted by other programs with a similar functionality.

Interoperability and Distribution: The different software modules of an application should be able to exchange data, and to run in different machines from which each one obtains the maximum efficiency.

Efficiency: The use of software that integrates other software should guarantee the achievement of certain efficiency requirements, i.e., real-time response. From a different perspective, the integration process should also be efficient in the sense that it should be performed in minimum time and with minimum effort.

Robustness: Current robot control architectures require a high degree of robustness, since they are often

* Corresponding author. Tel.: + 34-5-213-2724.
E-mail address: jgonzalez@ctima.uma.es (J.A. Fernandez)

[☆] This work has been supported by the Spansih Government under the research contract CICYT-TAP96-0763.

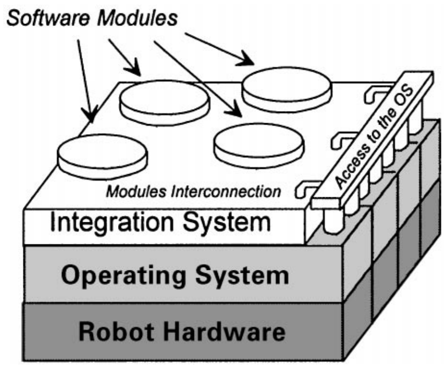


Fig. 1. An open system can be located between the OS and the software for the application. This joints the flexibility of software for implementing it with the portability on a wide range of different physical platforms.

used in real, critical situations. The integration system should provide appropriate mechanisms for guaranteeing it.

This paper describes an open software system called NEXUS that provides all these characteristics:

- It is based on techniques from object-oriented programming languages. The software parts of an application are defined as modules that hide their implementation details while providing their functionality through a simple interface. In addition, it provides a basic library of functions that hide the operating system protocols for many operations (disk operations, interprocess communications, memory management, etc.). In this way it achieves *portability*, *upgradeability* and *reusability*.
- It is a tool for obtaining *distributed* applications. The communication system between software modules is an intermediate scheme lying between a subscription/production network and a client/server paradigm. It joins the efficiency in communications of the former with the flexibility and implementation simplicity of the latter. Moreover, module distribution, although controlled by the implementors of the application at the design stage, is completely transparent for the modules at running time.
- Since the framework has been implemented on real-time operating systems, it inherits real-time mechanisms from them that are provided to the modules of the application. On the other hand, *efficiency* is also achieved at the design stage by using simple interfaces to integrate very different modules.
- Finally, it implements a hierarchical scheme for detection of run-time errors that allows the programmers to implement very *robust* applications.

Our system has been designed for reducing both the cost of development as well as further modifications to complex robotic software applications. Although it has been used in mobile robots, it is a sufficiently generic framework suitable for use in other control systems (for example, CIM applications). The main contribution of our work is to provide a general-purpose framework that implements all the basic concepts needed for integrating a control architecture, and that serves as a basis for more sophisticated tools.

Section 2 discusses related works of importance. Section 3 describes all the components of NEXUS, and the ways they guarantee the characteristics of an open system. Section 4 explains the process needed to implement applications. Section 5 shows a real example of the integration achieved in a real mobile robot system, and the performance measurements that can be obtained from the behavior of the application. The paper ends with conclusions and several lines for future research.

2. State of the art and related work

Several tools and techniques that facilitate the integration of robotic control architectures have been developed over the last few years. Most of them provide meaningful insights into very specific domains or particular levels of abstraction. The contribution of NEXUS is that it implements all the basic concepts needed to obtain maximum flexibility in the integration of software (portability, upgradeability, reusability) along with good performance and robustness. The main objective has been to obtain the simplest system that provides these characteristics, serving as a complete basis to implement more sophisticated tools.

Fig. 2 shows a possible classification of the parts of the software architecture which are involved in a complex, multidisciplinary robotic application. They range from the highest level of abstraction of the architecture, the *decision level*, which contains the components that allow the application to make intelligent decisions, develop long-term plans, etc., to the lowest level of abstraction: the one which deals with the hardware devices and the operating system. The *functional level* contains the components which implement control loops, behaviors, and, in general, the basic processes which are not concerned with long-term issues.

In the following, we review relevant works on integrating systems that cover some or all these levels. Since NEXUS is mostly concerned with the *functional level*, we pay more attention to the works done in this area.

2.1. The decision level

A survey of the most important knowledge representation schemes used in robotics, such as frames, scripts, feedback approaches, and rule-based systems can be found in [2]. One of the tools that has been developed to implement these high-level knowledge representations is

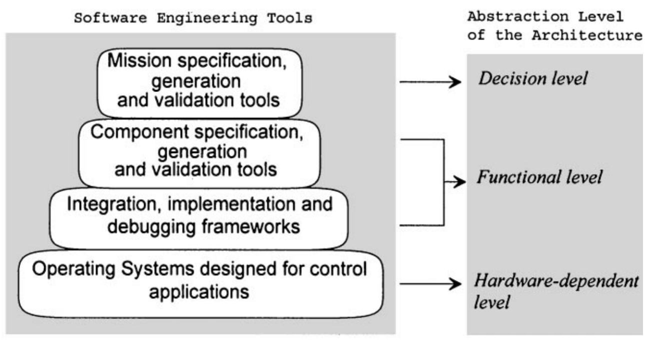

Software Engineering Tools
Abstraction Level of the Architecture
Mission specification, generation and validation tools
Component specification, generation and validation tools
Integration, implementation and debugging frameworks
Operating Systems designed for control applications
Decision level
Functional level
Hardware-dependent level

Fig. 2. The pyramid of software engineering tools for robot control architectures. The elements generated by these tools fit into the several levels of the architecture. In the figure three abstraction levels are distinguished: a decision level (planning, knowledge modelling), a functional level (behaviors, control loops) and a hardware-dependent level (basic software to manage the hardware devices).

the MAESTRO language [3], which has been designed to specify, validate and implement robot missions efficiently. The programs generated by MAESTRO can be proven to satisfy their specifications before executing.

In [4] another system is presented for generating code for the decision level. It is based on a temporal planning system called IxTeT which can reason on symbolic and numeric temporal relations between time instants. The plans are executed as event-driven automata.

2.2. The functional level

Several robot architectures that allows the reconfiguration of their functional components can be found in the literature. Among the most interesting ones are the TCA [5], CODGER [6], and NASREM [7]. TCA and CODGER are centralized architectures, while NASREM is a hierarchical one. There are several differences between NEXUS and these systems, the most important being that the former is completely non-centralized (a wide range of control architectures can be implemented: from decentralized to hierarchical), and it is focused on the modular and portable aspects of the application.

In [4], the guidelines of a complete system for designing a robot control architecture are presented. The problem is also addressed in [8], where the ORCCAD development system is described in detail. They provide the implementors with a human–machine interface based on an object-oriented paradigm suitable for designing robot tasks. The supervision of these tasks by the components at the *decision level*, i.e., the connection between both levels, is performed via a finite-state automaton. However, the problem of integrating the components at the *functional level* is not addressed.

Some commercial applications exist which include the most important characteristics mentioned in the previous paragraphs. The CODE programming interface by Cimetrix [9] provides an API with function calls for the control of mechanisms, input/output, trajectory planning, etc., in a system that is suitable for CIM architectures. Another framework for real-time system development is ControlShell by RTI [10]. It combines CASE tools and object-oriented techniques to generate, implement, and debug applications consisting of separate objects. It also provides a scheme for integrating and reusing these objects, as well as reducing the costs of implementation. However, it does not address the problem of integrating the components of the *functional level* with components of the *decision level* that are not generated.

Of special interest is the $G_o^{en}M$ framework [11] because of its similarity to our system. Both of them specify the components of the architecture as *modules* that pro vide *services* to the rest of the application. This leads to an efficient, flexible and robust way of designing, integrating and reusing the elements of the *functional level*. In $G_o^{en}M$ the code of the modules is generated automatically from templates that are filled by the programmers. Also, it provides tools for debugging the code. The connection with the *decision level* is performed by means of the Kheops rule-based system, although they do not focus their work on this issue. The communication system between modules is not addressed either.

Another example of an open system at the functional level is the OSACA architecture [1]. It is intended to set a standard for the implementation of control systems, providing an open framework that hides vendor-specific details. The basic components of OSACA are analogous to those of NEXUS, although the latter provides a more general network protocol than a client/server paradigm and those characteristics for real-time response needed in more dynamic and unstructured applications, such as mobile robotics.

2.3. The hardware-dependent level

Several real-time operating systems have been designed for implementing control architectures. For completeness we mention some of them along with their most important features. ALBATROSS [12] provides hard real-time communications between tasks through shared memory. The Harmony OS [13] supports transparent multiprocessing that enables finding the optimal load balancing among a number of processors. Chimera II [14] provides great efficiency in distributed applications and is the basis for more abstract works mentioned in the previous subsection.

The use of an operating system designed explicitly for robotic applications facilitates the development of very efficient applications, although it involves implementing large amounts of code that is not directly related to the integration problem.

3. Description of the NEXUS system

In our framework, an *application* is defined as an integration of different software modules that provide services to each other, aimed at accomplishing a given robotic task. The modules hide their implementation details: only the services and the format of their input and output parameters are known by other modules. The format of these data is a predefined one, very simple but general enough to capture the functionality of a wide class of services. Libraries of functions are provided to the implementors of the modules in order to access both the other modules and the OS. This basic scheme guarantees many of the characteristics needed in an open system: *portability*, since the modules access NEXUS and the OS through general and simple interfaces that can be adapted to very different hardware platforms and operating systems; *upgradeability*, since a change in any module that does not change its functionality (its set of services) does not affect the rest of the application; and *reusability*, since any module can be plugged into another application that needs its functionality without knowing its implementation details.

As shown in Fig. 3, our system is divided into two subsystems. The *Task-Dependent Subsystem* contains the sets of modules of the application. These sets, called *Conceptual Units*, are used by the implementors for clarity in the design of the application. The *Management Subsystem* is common for every application: it contains the managers needed for maintaining the structural information about the modules and for supporting communications. In the following subsections a more complete description of these subsystems is given.

3.1. The task-dependent subsystem

The task-dependent subsystem varies for each application. It includes the entities called *ICE modules*,[1] or simply *modules*. Each module provides some *services* that define its functionality. ICE modules can be grouped into *Conceptual Units*. The ICE modules are plugged into NEXUS before the application execution begins. This plugging-in stage is an automatic operation consisting just in executing the executable file of the module, and is simple enough to allow any module to be substituted by another that provides the same functionality at any time.

In the following, the components of the Task-Dependent Subsystem are described in detail.

3.1.1. Conceptual units (CUs)

For achieving a clear design of the application, a CU should contain all the modules concerning a given conceptual area of operation of the robot system. The definitions of these conceptual areas depend on the application being implemented. For example, it may be desirable to group into the same CU all the modules that deal with the sensors of a mobile robot (laser scanners, sonars, cameras), or the modules related to the operation of navigation.

[1] "ICE" stands for the initials of the three main components of a module: Interface, Code, and Error-recovery code.

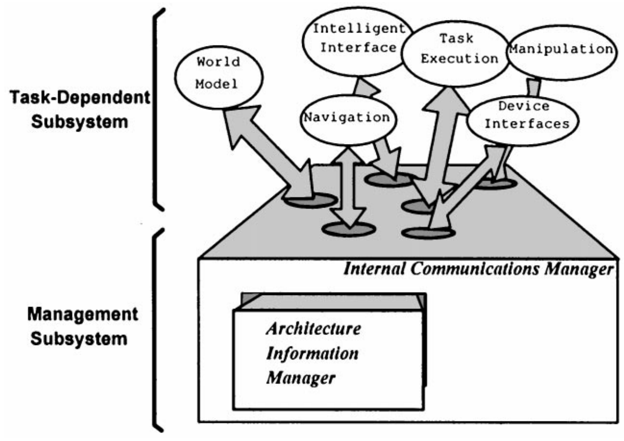


Fig. 3. Components of NEXUS. The Conceptual Units represented in the task-dependent subsystem are an example (this subsystem depends on the application being implemented).

3.1.2. Services

Each service of a module is defined through three parameters: its input data, its output data, and its characteristics. The characteristics of a service specify its *functionality* (whether it is a service for consulting -*query*-, updating -*modifier*-, or monitoring -*monitor*- the module status), its *duration* (whether the requests last until the architecture is deactivated -*permanent*-, or not -*temporary*-), and its *concurrency* (whether several requests can be served at the same time or not -*reentrant* or *non-reentrant*-). Any service is described by a subset of these characteristics. This allows the programmers to implement a wide range of services, for example those that run periodically, or those that execute only when a given asynchronous signal is received.

In order to guarantee real-time response, a request for a service may hold information about the temporal requirements to be met, i.e., the maximum time allowed to serve the request. If these temporal constraints are not satisfied, the Management Subsystem automatically replies to the routine which sent the original request, including information about the error.

3.1.3. ICE modules

ICE modules are the main components of the CUs. Each module is executed on a single computer, and therefore all the services that it provides are served from that machine. The possibility of distributing the ICE modules among the computers of the robotic system permits optimal exploitation of the available hardware resources. For example, one computer in the system may be more appropriate for image processing than others, so the ICE modules which offer services related to visual perception should execute in that machine.

An ICE module consists of three parts:

Interface: The interface part of the module defines the services implemented in the module and receives the requests for services, executes the code that serves them, and sends back the output data to the requesters. It also provides the rest of the application with the public information about the functionality of the module. In Fig. 7 an example of the definition of services is shown.

Code: In addition to the code of the routines that supply services, this part also contains two special routines for both initializing and finishing the internal status of the module (internal static variables, configuration of the module, etc.).

Error-recovery code: This part deals with the errors that the module may find during its execution. Some of them can be overcome by the module, while others must be propagated to the module services requesters. The mechanisms provided to deal with errors are described in the following paragraphs.

3.1.4. *Hierarchical error recovery system*

Error recovery usually involves some sort of replanning, that is, re-structuring the execution of a failed operation. This process should be performed as near as possible to the point at which the error is detected. The hierarchical scheme used by NEXUS guarantees this. It is possible to integrate very robust applications if this mechanism is used appropriately.

The different classes of errors that can occur during the execution of a service request, and the ways by which they could overcome, are shown in Fig. 4. Errors that the service routine can manage by itself are called *recoverable errors*. When a recoverable error occurs the very routine which is serving the request will replan its execution to overcome it. In contrast, those errors which the service routine cannot deal with on its own are called *unrecoverable errors*. They have to be reported to the module that issued the service request. The code in charge of this is in the *Error-recovery-code* part of the module.

There are some special unrecoverable errors called *critical errors* caused by unexpected failures in a process or a machine that prevents sending a report to the requester, and cannot be completely solved by the module programmers. For example, when a module[2] is killed, all the routines which are providing services in that module may be killed. NEXUS detects most of the critical errors

[2] In the LynxOS implementation, a module is implemented as a single process which consists of many subprocesses (tasks or *threads*).

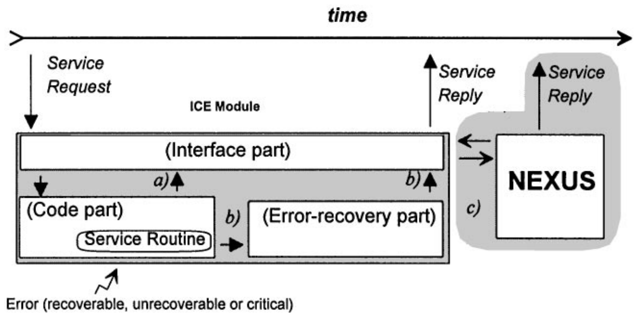


Fig. 4. Classes of run-time errors that can occur during the execution of an application. (a) Recoverable errors can be managed by the routine which detects them. (b) Unrecoverable errors have to be propagated to the requesters of the service which makes the detection. (c) Critical errors that cannot be managed by the application are detected and solved by the system managers.

and communicates their occurrence to the other parts of the robot system automatically (via *events*, described in Section 3.2). Among other things, it detects and tolerates module killing, out-of-time service replies, and wrong communications formats. Due to its distributed nature, an application can also continue executing despite a machine shutting down.

Other critical errors which might lead the application to an unstable status (for example, when a module uses most of the CPU time because of a corrupted code or a wrong algorithm) are handled by the multi-priority and preemptability features inherited from the underlying real-time OS. The ICE modules are prioritized with respect to other modules, and the services of a module are prioritized with respect to other services in the same module. This is a simple way to design the priorities scheme for any application in order to limit the system resources that each routine can consume.

3.2. *The management subsystem*

This subsystem contains the common facilities required in any robotic application. Currently two managers are implemented: the *Internal Communications Manager* (ICM) and the *Architecture Information Manager* (AIM) which run in every machine of the robot system. All the communications between the different components of the architecture are managed by the ICM, and the structural information about the architecture is stored and managed by the AIM.

3.2.1. *The Internal Communications Manager* (*ICM*)

The ICM is in charge of distributing information among the different components of the robot system. If the source and destination modules of an information package are not being executed on the same computer, the communication is routed to the ICM of the destination machine, and then it is directly sent to the suitable module. The ICM can also register the message flow of the application for further study.

Two types of communications exist in the architecture: *messages*, that contain an arbitrary amount of data (for example, *requests* to any component of the system, or *replies* to these requests) and *events*, that do not contain any information and are sent to all the modules of the application at the same time. With these basic types of communications, both synchronous and asynchronous concurrent processing can be implemented. Messages are used to request and reply service requests, while events are used for asynchronous operation. For example, if there is a module called "Battery Sentinel" offering a monitor service which supervises the charge level of the batteries of a mobile robot, an event can be defined for informing the other modules about whether the charge falls below a certain level.

The communication system might be seen as a client/server one because every module becomes a *server* for the service requests of other modules (the *clients*). However, the existence of an AIM (with all the information that it holds) in every computer makes it more similar to a subscription/production one: the information exists *in the network*, not in a particular machine. The modules register their services at plugging-in time, and the transmission of this information to all the computers becomes a sort of *subscription* to these services. The difference with a more flexible subscription/production system (such as the NDDS [25]) is that all the machines subscribe to the services of the rest. However, the communication system of NEXUS still provides a higher level of robustness and more efficient communications than a pure client/server one.

The components that allow to establish the network communications needed for distributing the applications are inside each ICM, and they are called *network-exchanges* [15]. The network-exchanges of the ICMs are not visible for the programmers of applications, but this abstract data type can be used separately in any module for implementing other communication schemes, for example, communications from a module to a web page in the internet, or to a platform on which NEXUS does not run.

Network-exchanges are very easy to use, and at the same time are powerful tools that offer modularity, flexibility, reliability, and robustness. The purpose of a network-exchange is to make the underlying network transparent for the programmer. Currently, they are implemented using the TCP and UDP protocols, but other versions may be developed. Network-exchanges establish a security context between local and remote ICMs. It is guaranteed that every communication through network-exchanges is sequenced and unduplicated. They also provide a query interface to perform network administration tasks, including query services for the current state (number and characteristics of connections, traffic flow) and event facilities which are useful to assure real-time response. A network-exchange consists of four agents and a connection set that contains zero or more one-way connections to or from other network-exchanges.

3.2.2. The Architecture Information Manager (AIM)

The AIM maintains all the structural information about the application: CUs, Services, and ICE modules. In our distributed system, the existence of many AIMs (one in each computer) might lead to inconsistency problems between the data stored in each one. This is solved by reporting to each AIM the modifications made in the other ones. Therefore, any change made in the database of one machine must be sent through the network. Although this process can certainly be slow, it is limited to occurring only before activating the architecture, and consequently it does not affect the performance of the application during its normal operation.

4. Implementing applications

Our system has been written in C and C++ and implemented completely on LynxOS. The implementation consists of an executable file that contains the ICM and the AIM and two C libraries for linking with the code of the modules. One of these libraries contains all the routines for accessing NEXUS and for requesting services, sending replies to service requests, declaring event handlers, etc. The other one is an interface with X/Motif and is optional.

The process of implementing a module just consists in defining the services that it will provide (the *Interface* part of the module) and programming the routines that serve requests for these services (the *Code* and *Error-recovery-code* parts of the module). These parts are linked together with the libraries in order to form a single executable file. A function from the libraries is in charge of connecting the module and registering its services in the system database automatically.

The requirements that a routine for serving service requests must satisfy to be included in the *Code* part of a module are:

(1) Error propagation to the requesters.
(2) Interfacing with the underlying OS by using the libraries.
(3) Semaphore protection in the reentrant portions of the code.

Note that this simplicity in designing and connecting modules facilitates the integration of software programmed by different people while offering a simple interface to the common resources needed in any control system.

4.1. Access of the applications to the Operating System

The libraries provided to the programmers are aimed at implementing portable accesses to the OS. They offer the following interfaces:

- Disk operations interface.
- Memory management interface.
- Multiprocessing interface (threads, processes, semaphores, etc.).
- Several inter-process communication interfaces (message queues, signals, etc.).
- Real-time interface (real-time events, timings, preemptability, etc).
- Network communication interface (based on network-exchanges).
- Graphic windows-based interface.

They can be easily adapted to run on a wide range of platforms, so the applications become very portable. The basic requirements for the OS on which NEXUS is to be implemented are satisfied by any UNIX-like OS with multiprocessing and real-time features such as accurate timings, multiple priority levels, and preemptability. Currently, it is completely implemented on LynxOS, and a reduced version has been implemented on Windows NT. In the future, other versions will be developed (e.g., RT-Linux).

5. A real application

Up to now, our system has been implemented and tested in several mobile robots. One of these is the RAM-2 mobile robot (Fig. 5), an autonomous mobile robot designed and built at the University of Malaga for research applications in indoor environments [16]. It has a variety of sensors such as two laser scanners, a sonar ring and a camera, and a manipulator arm placed in front of the platform. The robot contains two onboard Pentium cards, one of which supports the Lynx real-time operating system, while the other supports Windows NT.

The overall objective is to obtain a robot delivery system capable of taking small objects from one place to another inside a structured office-like environment. A small application that partially achieves this objective in a semi-autonomous fashion has been implemented. For this purpose, most of the physical components of the robot are used: a radial laser scanner for constructing 2D representations of the rooms, a frontal laser scanner for detecting dynamic obstacles, a manipulator arm to grasp and drop objects on the commands of a remote operator (guided by a camera), and motor, steering and odometric systems to perform navigation and dead-reckoning.

More than 10 different ICE modules have been implemented for managing these systems and performing complex tasks. In parallel, simulation modules with the same functionality as the ones dealing with physical devices (motors, lasers, etc.) have been implemented in order to test the system intensively. Fig. 6 shows the topology of the application with all the modules and conceptual units, and their location on each computer motherboard. Some of them were already developed as simpler, individual experiments, while others have been designed explicitly for this application. The former ones have been adapted to NEXUS quickly (typically, generic programs can be adapted in a few hours). Most of the time has been spent in solving several bugs and errors that were not detected in the original experiments. When a large number of modules are integrated and work together, these errors tend to appear rather easily. Our system has demonstrated its robustness in these cases, allowing the application to keep running with a reduced performance, or to finish the execution in a controlled way.

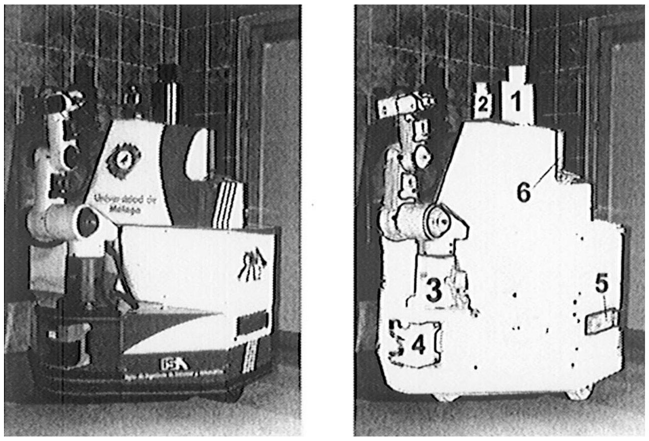


Fig. 5. The RAM-2 mobile robot, on which NEXUS has shown its flexibility for integrating software programmed by several researchers. (1) Radial laser scanner. (2) Camera. (3) Robot arm. (4) Frontal laser scanner. (5) Sonars. (6) On-board display.

The *Device Interfaces* CU contains the modules needed to deal with the physical components of the mobile robot: lasers, motors, and manipulator. All of them have a simulation counterpart in order to test the application before involving the system in physical movements. The *World Model* CU includes a module that merges 2D local maps of the environment into global ones (useful for self-location of the robot [17,18]) and a module that integrates all these maps and other kinds of information from the environment into a highly structured topological and hierarchical model based on Multi AH-Graphs [19,20]. The *Navigation* CU contains modules that generate paths on the 2D geometrical maps for going from one place to another and execute these paths while detecting and avoiding dynamic obstacles [21]. The *Teleoperation* CU enables a remote operator to move the robot arm and a pan/tilt camera system in order to manipulate objects. For this purpose, it contains a module that communicates with a remote station (that does not run

NEXUS) via radio ethernet, using network-exchanges, and another module that moves the camera plugged into the Windows NT motherboard. Finally, the *Task Execution* CU contains a module which executes high-level tasks such as navigating from one room to another, constructing maps and topologies of the environment, or delivering objects to any location in the building. These complex tasks are implemented internally as state machines that send requests for services from the other modules.

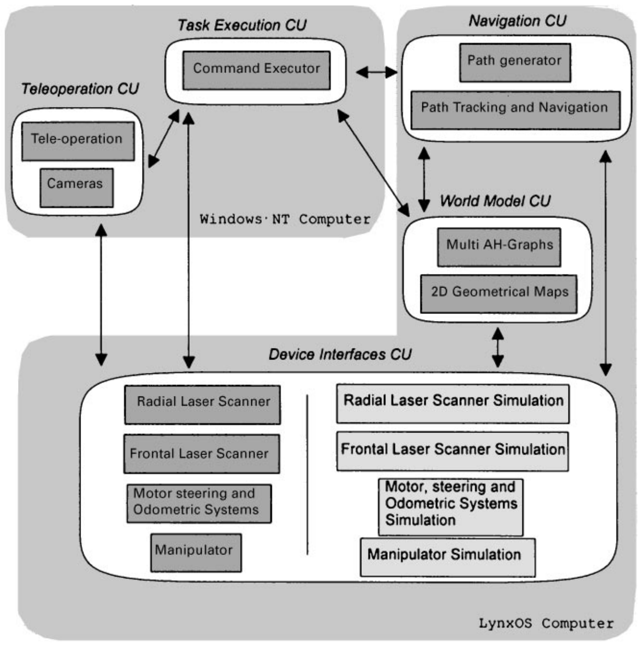


Fig. 6. Conceptual Units and modules implemented for the application described in Section 5. Since the RAM-2 mobile robot contains two motherboards, one running LynxOS and the other windows NT, the modules are spread on them maximizing the suitability of each platform for their tasks. The Windows NT platforms is used for managing teleoperation and dealing with the cameras, while the LynxOS platform is in charge of performing more critical real-time processes.

In Fig. 7, definitions for some of the services provided by the modules of the application are shown. The following examples illustrate the wide class of services that have been implemented in this case:

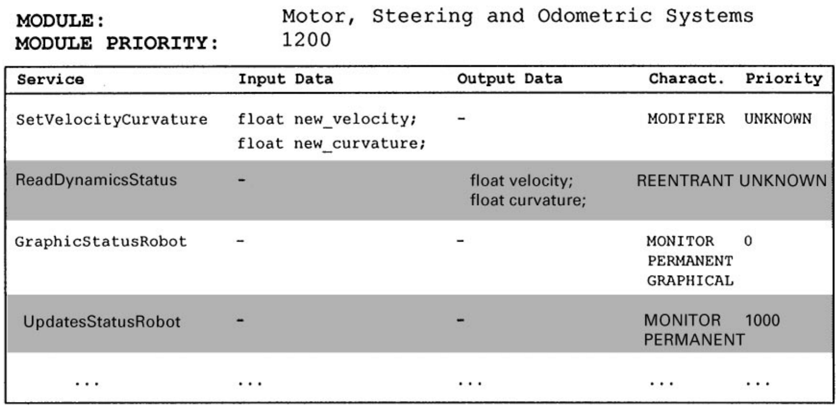
Periodical monitor services: An example is the service in charge of reading data from the frontal laser at a fixed frequency of 20 Hz. These types of services are highly prioritized with respect to other services in the application in order not to lose possibly critical data. Often they are the services that send asynchronous events to the architecture when delicate situations occur.

Graphical monitor services: Most of the modules open graphical windows to show their internal status and for enabling the user to change them. The code associated with each of these windows is part of low priority monitor services whose execution does not affect the rest of the application's performance.

Root services: They are services that are usually requested by an operator action on a text terminal or graphical window (that is, on a monitor service). They originate other requests in order to successfully execute a complex task. The *Task Execution* CU contains most of the services of this type.

Normal synchronous services: Most of the services of the application belong to this class. They are executed as the result of an incoming request from another module and after a (typically) short processing, they yield some result.

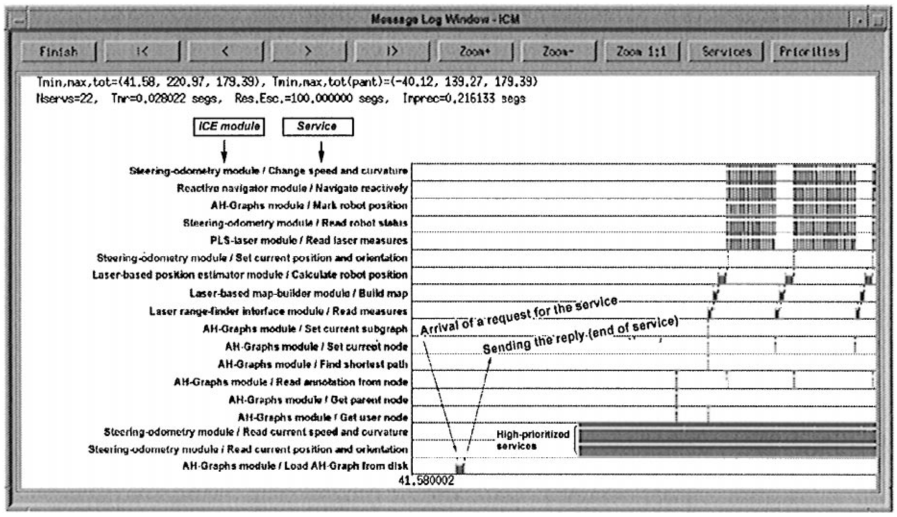
Regarding the performance of the application, Fig. 8 shows the timing diagram for the message flow logged by the ICM while RAM-2 was navigating. Several timing parameters appear, such as the total time elapsed



MODULE: Motor, Steering and Odometric Systems
MODULE PRIORITY: 1200

Service	Input Data	Output Data	Charact.	Priority
SetVelocityCurvature	float new_velocity; float new_curvature;	-	MODIFIER	UNKNOWN
ReadDynamicsStatus	-	float velocity; float curvature;	REENTRANT	UNKNOWN
GraphicStatusRobot	-	-	MONITOR PERMANENT GRAPHICAL	0
UpdatesStatusRobot	-	-	MONITOR PERMANENT	1000
...	...	...	...	...

Fig. 7. Definitions of some services of the real application commented in Section 5. They include short names for the services, format of the input and output data, characteristics and relative priorities with respect to other services of the same type in the same module. The module is also assigned a relative priority with respect to the other modules of the applications. As shown in the figure, monitoring permanent services are running from the activation of the application to the end of it, therefore they do not need input or output data. However the rest of the services are executed by request and they may provide output data or need input data for their internal processes.



```
Tmin,max,tot=(41.58, 220.97, 179.39), Tmin,max,tot(pant)=(-40.12, 139.27, 179.39)
Nservs=22,  Tm=0.028022 segs,  Res.Esc.=100.000000 segs,  Inprec=0.216133 segs
```

Message Log Window - ICM
Finish
|<
<
>
>|
Zoom+
Zoom-
Zoom 1:1
Services
Priorities
ICE module
Service
Steering-odometry module / Change speed and curvature
Reactive navigator module / Navigate reactively
AH-Graphs module / Mark robot position
Steering-odometry module / Read robot status
PLS-laser module / Read laser measures
Steering-odometry module / Set current position and orientation
Laser-based position estimator module / Calculate robot position
Laser-based map-builder module / Build map
Laser range-finder interface module / Read measures
AH-Graphs module / Set current subgraph
AH-Graphs module / Set current node
AH-Graphs module / Find shortest path
AH-Graphs module / Read annotation from node
AH-Graphs module / Get parent node
AH-Graphs module / Get user node
Steering-odometry module / Read current speed and curvature
Steering-odometry module / Read current position and orientation
AH-Graphs module / Load AH-Graph from disk
Arrival of a request for the service
Sending the reply (end of service)
High-prioritized services
41.580002

Fig. 8. Timing diagram generated by the ICM while executing the application designed for the experiment in Section 5. The 2D-bar segments indicate the time elapsed in serving each service request.

in the experiment, the number of services involved, etc. The average time between the arrival of a service request to a module and the response to it is 0.028 s. This value depends on each application and serves for debugging purposes. The timing diagram obtained by the ICM consists of 2D horizontal bars that measure these periods. Due to the preemptive features of LynxOS, services with high priorities (the "Read current speed and curvature" and "Read current position and orientation", for example) maintain their timings in spite of others with lower priorities (such as "Calculate robot position").

6. Conclusions and future work

In this paper we have described NEXUS, a software extension of a real-time operating system that allows us to integrate the software elements of a robot system in a flexible and efficient way. Its main feature is that it decouples the programs designed for the application from the fixed software and hardware of the robot system. To achieve this, it inherits some important features from high-level programming languages (encapsulation, hierarchical error recovery, modularity, etc.) and from the subscription/production paradigm.

We have experimentally tested the advantages of this framework by designing applications in which our mobile robot RAM-2 serves as a delivery agent in an indoor environment. The application consists of more than 10 modules developed by different people. Some of them were available as part of previous applications and others have been implemented for this test. The process of adapting the existing code has been surprisingly short: only a few hours for each module. The robustness of the whole system has been demonstrated. When critical errors made some modules break down the rest of the system was able to continue a limited but correct operation. In addition, the performance of the applications has been satisfactory.

NEXUS has been designed as a complete basis on which to implement more sophisticated tools for the integration and implementation of complex control applications. Currently we are working on some extensions:

- Implementing versions of the system for other platforms, such as RT-Linux.
- Enabling secure accesses to applications from the internet (WWW).
- Providing more powerful visual tools for designing, programming, and debugging applications.
- Collecting a number of libraries for the robotics domain, such as kinematics, plan execution, petri-nets plan specification, etc.
- Extending NEXUS to manage dynamic binding, that is, assignment instances of server routines for service requests at run-time.

References

[1] Sperling W, Lutz P. Enabling open control systems — an introduction to the OSACA system platform. Robotics and Manufacturing, vol. 6, New York: ASME Press, 1996.

[2] Wolfe WJ, Chun WH. Robot architectures and design paradigms. Proceedings of the SPIE Mobile Robot VII Conference, Boston, vol. 1831, 1992. p. 307–17.

[3] Coste-Manière E, Turro N. The MAESTRO language and its environment: specification, validation and control of robotic missions. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'97), Grenoble, France, 1997.

[4] Alami R, Chatila R, Espiau B. Designing an intelligent control architecture for autonomous robots. International Conference on Advanced Robotics (ICAR'93), 1993.

[5] Simmons R, Lin L-J, Fedor C. Autonomous task control for mobile robots (TCA). Fifth IEEE International Symposium on Intelligent Control, Philadelphia PA, September 1990.

[6] PLCopen. PLCopen. Standardization in Industrial Control Programming. http://www.plcopen.org.

[7] Albus JS, Quintero R. Towards a reference model architecture for real-time intelligent control systems (ARTICS). Proceedings of the IEEE Third International Symposium on Robotics and Manufacturing, Burnaby, Canada, July, 1990.

[8] Simon D, Espiau B, Castillo E, Kapellos K. Computer-aided design of a generic robot controller handling reactivity and real-time control issues, Rapports de Recherche n° 1801, Programme 4: Robotique, Image et Vision, INRIA, November 1992.

[9] Cimetrix Incorporated. The CODE Programming Interface. http://www.cimetrix.com/Software/CODE.html, 1997.

[10] Real-Time Innovations Inc. (RTI) and Stanford University. ControlShell: Object-Oriented Framework for Real-Time System Software. http://128.102.240.17/products/cs.html, 1996.

[11] Fleury S, Herrb M, Chatila R, $G_{o}^{en}M$: a tool for the specification and the implementation of operating modules in a distributed robot architecture. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'97), Grenoble, France, 1997.

[12] von Puttkamer E, Zimmer UR. ALBATROSS: an operating-system under realtime-constraints. Real-Time magazine, Diepenbemmd 5 - 1650 Beersel - Belgium, vol. 5, nr 3 91/3, 1991.

[13] Gentleman WM, MacKay SA, Stewart DA, Wein M. An introduction to the harmony realtime operating system. Newsletter of the IEEE Computer Society Technical Committee on Operating Systems, Summer 1988. p. 3–6.

[14] Stewart DB, Schmitz DE, Khosla PK. The Chimera II real-time operating system for advanced sensor-based control applications. IEEE Transactions on Systems, Man and Cybernetics, (1992); 22(6):1282–95.

[15] Fernández JA, González J, Martín A. Communicating and integrating the modules of a robotic software application. Fourth International Symposium on Distributed Autonomous Robotic Systems (DARS'98), Karlsruhe, Germany, May 1998.

[16] Ollero A, Simon A, García F, Torres VE. Integrated mechanical design of a new mobile robot. IFAC Symposium, Málaga (Spain), New York: Pergamon Press, 1992.

[17] Gonzalez J, Stentz A, Ollero A. A mobile robot iconic position estimator using a radial laser scanner. Journal of Intelligent and Robotic Systems 1995;13:161–79.

[18] Gonzalez J, Ollero A, Reina A. Map building for a mobile robot equipped with a laser range scanner. IEEE International Conference on Robotics and Automation, San Diego, 1994. p. 1904–10.

[19] Fernandez JA, Gonzalez J. Mobile robot path planning and navigation using hierarchical graphs. International ICSC Symposium on Engineering of Intelligent Systems (EIS'98), Tenerife, Spain, 1998.

[20] Fernandez JA, Gonzalez J. A general world representation for mobile robot operations. Seventh Conference of the Spanish Association for Artificial Intelligence (CAEPIA'97), Malaga, Spain, November 1997.

[21] Muñoz VF, Cruz A, García-Cerezo A. Speed planning and generation approach based on the path-time space for mobile robots. IEEE International Conference on Robotics and Automation. Leuven, Belgium, 1998.