Vrije Universiteit Brussel

Koninklijke Militaire School

CoRoBA, a Framework for Multi-Sensor Robotic Systems Integration

ir. Eric Colon

Promotors: Prof. H. Sahli
Prof. Y. Baudoin

Submitted in fulfillment of the requirements for the degree of
Doctor of Applied Sciences at:
Vrije Universiteit Brussel, Faculty of Engineering Sciences
and
Royal Military Academy, Polytechnics Faculty

VRIJE UNIVERSITEIT BRUSSEL
Faculteit Ingenieurswetenschappen
Vakgroep Elektronica en Informatica

KONINKLIJKE MILITAIRE SCHOOL
Faculteit Polytechniek
Departement Mechanica

CoRoBA, A FRAMEWORK FOR MULTI-SENSOR ROBOTIC SYSTEMS INTEGRATION

ir. Eric Colon

Submitted in the fulfillment of the requirements for the degree of
Doctor of Applied Sciences

Promotors : Prof. Dr. H. Sahli
Prof. ir. Y. Baudoin

Jury :
President: Prof. Dr. ir. D. Lefeber
Vice-President: Prof. Dr. ir. J. Vereecken
Prof. Dr. ir. J. Cornelis
Prof. Dr. ir. M. Acheroy
Prof. Dr. ir. P. Dehombreux
Prof. Dr. V. Jonckers
Prof. Dr. ir. A. Winfield

November 2006

Abstract

In order to increase efficiency in software development for controlling robots, tools facilitating the implementation of distributed control applications are required.

This thesis proposes a solution to this problem with a framework called CoRoBA (Controlling Robot with CORBA). CoRoBA is made up of component based execution units. It comes with a 3D simulation application and utility programs for distributing and managing the live and run cycle of multi-process applications.

The implementation of the framework is based on several Design Patterns that make the design flexible, elegant and ultimately reusable. The execution unit in CoRoBA is a component. Components are independent execution units that have separated interfaces for the configuration and the actual functionality they provide. According to the classical control theory, components are divided in three categories, Sensors, Processors and Actuators. They form a chain along which information is transferred and like in classic control schemes, the data flow is unidirectional. Sensors read data from external devices and transmit them to other components. Processors process received data and forward results to components linked to output devices, which are called Actuators. This division provides a clear view of the functionality of each component and consequently facilitates their reuse in new applications.

Communication between components relies on the industry standard CORBA. Using such a standard simplifies the development and improves the interoperability with existing software. The framework offers two different communication mechanisms, the first one is based on classical synchronous communication while the second relies on Events. Event based communication increases the flexibility of the application because it decreases the coupling between components.

In order to test and tune applications a simulator is required. As no existing software did satisfy our needs a 3D multi robot simulator has been developed. It relies on Java3D for the modelling and rendering of the virtual world. The simulator is responsible for the realistic motion of the robot by using geometric, kinematic and dynamic models, and takes care of the collision with fixed and moving obstacles like other robots. The simulated sensors produce measurement data that are injected in the application control loop. The software integrates seamlessly with the components of CoRoBA because all robots and sensors have a CORBA interface. The utilisation philosophy is to develop and tune control algorithms in simulation and to simply replace simulated by real components once satisfying results have been reached, no further modification of the Processor components being required.

Several distributed control applications have been implemented in order to validate the framework. Shared control and autonomous navigation applications involving different robots have been successfully tested in simulation. Development of multi-robots applications, distributed simulation and real robots and sensors has also been addressed. A qualitative and quantitative evaluation of the framework have shown that the proposed solution is efficient, usable and stable.

Foreword

When you begin working on a Ph.D. you don't imagine what it means and how it will influence your life not only during the Ph.D. but also probably for the remainder of your life.

Computer engineering is a fantastic but also a frustrating discipline. You can spend hours and days to solve intricate problems and finally discover that the solution was trivial. Most of the efforts and spent time are hidden in lines of codes that have been written during weeks and months. The difficulty is to find methods for presenting this work without showing long boring listings. Graphics are generally the good way for summarising the ideas lying behind the work. In this text, the Universal Modelling Language has been used where it seemed appropriate. However, as this communication tool is not universally known, non standard graphics have sometimes been drawn to clarify the text.

This work gave me the opportunity to learn in many aspects and to improve my knowledge in Computer Science. Concepts like Design Patterns and Object Oriented languages provide tools and guidelines that helps in improving software quality.

I wish to thank all the people who have believed in me and who have left me freely choose the orientation of this work.

I thank my promoters for their implication and their support during this long period.

I also want to thank my wife and my daughter for their support and their patience.

Contents

Acronyms

Acronym	Meaning
ACE	ADAPTIVE Communication Environment
API	Application Programming Interface
BCM	Behaviour Coordination Mechanism
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CoRoBA	Controlling Robots with CORBA
DCOM	Distributed COM
DES	Discrete Event System
EC	Event Channel
GPL	(GNU) General Public License
GPS	Global Positioning System
GUI	Graphical User Interface
IDL	Interface Definition Language
IOR	Interoperable Object Reference
IPC	Inter-Process Communication
IR	Infra-red
MFC	Microsoft Foundation Classes
OO	Object Oriented
ORB	Object Request Broker
OS	Operating System
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RT	Real Time
SDK	Software Development Kit
SSL	Secure Sockets Layer
TAO	The ACE ORB
UML	Universal Modelling Language
US	Ultrasonic
XML	Extensible Markup Language

Chapter I Introduction

1 Preamble

As introduction to this thesis we give some definitions that will be useful for a good understanding of this text.

> A **class** is a unit of abstraction and implementation in an Object Oriented (OO) programming language.
>
> A **component** is an encapsulation unit that delivers services and which is reachable through well-defined interfaces. Components are reusable building blocks which can be called at run-time and which are unaware of the clients' implementation.
>
> **Software Design Patterns** are proven software design solutions to general problems.
>
> An **architecture** is composed by sets of related patterns and components.
>
> A **framework** is an integrated collection of classes that collaborate to produce a reusable architecture for a family of related applications. It is a design and an implementation providing one possible solution in a specific problem domain. It provides generic components that need to be customised and extended in function of the application.

Classes exist at design time and are instantiated at run-time to form objects. Objects collaborate in components to achieve the tasks of the application.

Design Patterns capture experience of expert designers. They describe recurring problems and the core solution to those problems.

A framework is composed by patterns and components; its architecture defines how the different components are integrated into the framework and how they are interrelated. It defines also how components communicate with each other. A framework dictates the architecture of applications and reverses the control paradigm; components written by the programmer are called back by the framework mechanisms in function of network and User Interface (UI) events. This is illustrated by Figure 1. The Microsoft Foundation Classes (MFC) [PROS99] and the wxWidgets [SMAR05] are examples of popular frameworks.

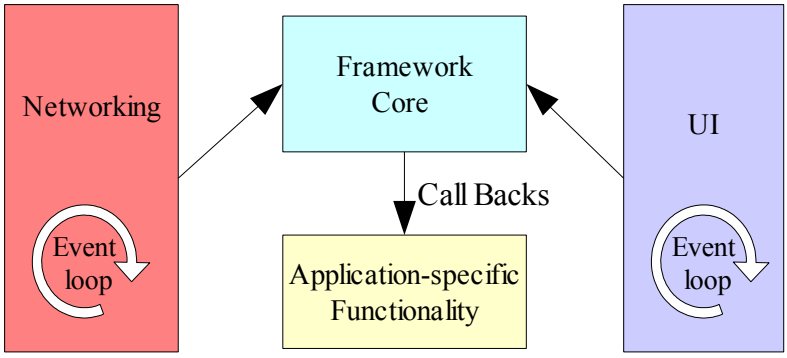


Figure 1. Framework Architecture

A Control Architecture defines the design of a set of components in which perception, reasoning and action occur. It also specifies the functionality and interface of each component as well as the interconnection topology between them. A Control Architecture specifies which components are used and how they collaborate in a concrete application. As Control Architectures inherit their functionality from the Framework (Figure 2), the framework architecture must be flexible enough to allow the design of different control architectures (Classical control, fuzzy logic control, behaviour based control, etc.).

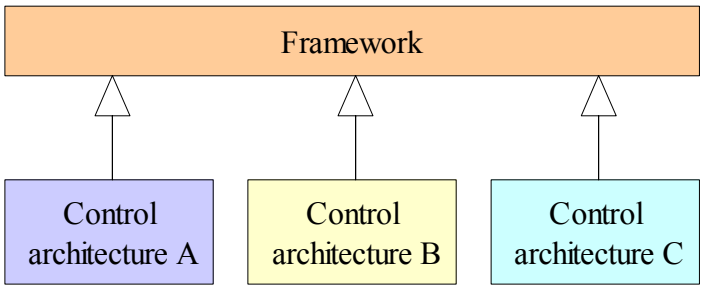


Figure 2. Inheritance diagram for Control Architectures

Rem. Appendix A contains a summary the Unified Modelling Language notations used in this text.

2 Motivation

Many researchers in robotics are nowadays faced with a recurring problem: they have at their disposal many excellent algorithms but, due to the lack of appropriate standards, it is almost impossible to easily reuse them in new applications or platforms. Existing programs have to be modified, translated, ported or even completely rewritten when changing or updating the robotic platform. If we look at what happens the last years in software engineering, we observe the emergence of new software techniques. Object-oriented languages, software components and software Design Patterns have greatly improved software re-usability. What is needed in robotics is a software framework that enables flexible and dynamic composition of resources and permits their use in a variety of styles to match present and changing computing needs. Since a couple of years, some researchers have begun to work in this direction.

For years, researchers have focused on embedded intelligence providing ad hoc implementation. Solutions have been tied to existing software and hardware, limited by software and hardware constraints (processor, memory, OS, communication, ...) or implementation costs. Everything had to be on-board. Nowadays hardware is far more affordable and wireless communication has become fast and reliable. It has consequently become easier to communicate and to implement distributed applications and with the recent progresses of the Internet, the notion of service has become familiar to many of us. Generally speaking a robot is already by itself a complex system but in order to perform useful tasks it must be equipped with additional sensors and actuators. These have in most cases their own control system resulting in a de facto distributed architecture. For non-specialists, developing software for a single robot, without speaking of multi-robots systems, can rapidly become a nightmare. What is needed in robotics is a software framework that eases the development of distributed applications by providing functions that hide and automate low level mechanisms and provide the developer with a higher level development environment and let him concentrate on intelligent aspects of the application.

Several organisations are attempting to create standards for the interaction between unmanned vehicles and control mechanisms to increase interoperability. This is specifically of importance in the military field. The United States has issued a Joint Architecture for Unmanned Systems[1] (JAUS) and NATO is now ratifying a proposed common interface for unmanned air systems (STANAG 4586). The UK Ministry of Defence is currently finalising its own Common Interface Protocol (CIP) specification. In France, the DGA has launched in 2004 a 4 year program aiming at the development of an open standard for interoperability of autonomous systems.

As other laboratories, we have been facing with the above described challenge. It has been difficult to leverage existing systems and integrate existing code that has been produced in previous projects. We have developed many interesting applications involving mobile robots, computer vision systems, tracking and location systems, 3D modelling,... that cannot be (easily) integrated. One could state that it is due to a lack of organisation or long term vision. These applications use techniques and knowledge that were available when they were developed. In the mid 90's no standard implementation was available for writing distributed applications. There was no universal language as Java, the Internet was only known by a couple of specialists and Linux was less than one year old.
Les us have a look at some of our past projects.

Corode

The Hudem project (1997-2002) focussed on the development and implementation of techniques for enhancing the landmine detection. During this project, different robotic systems have been developed. One of this system is a Cartesian scanner mounted on a mobile robot that was used to acquire data with different sensors on dummy minefields [COL02a]. An application named Corode (Control of Robots for Demining) has been written to control the robot, the scanner and to acquire and display data. While it has given (and still gives) satisfaction to its users, the approach adopted for implementing the software showed its limitations in terms of flexibility and reuse.

The main drawbacks of Corode are:

- It is written for Windows with Microsoft Visual C++. While the Application Programming Interface (API) proposes a Model-View-Control paradigm, this one is limited to a single application.

1 http://www.jauswg.org/vision.shtml

- The Graphical User Interface (GUI) is based on the widgets provided by the MFC and only runs on Windows.
- The control and visualization codes are mixed and located in several classes.

FuzzyNomad

With the indoor mobile robot Nomad200 interesting results, including navigation algorithms, remote control and virtual representation of the robot have been obtained. The navigation algorithms are written in C and run on a Linux platform using the API of the robot [COLO96].
The main drawback of FuzzyNomad is:

- The developed algorithm has been tailored to a specific robot and is mixed with the function calls to the robot API.

VRNomad

The purpose of the VRNomad application was to control and visualise a mobile robot Nomad200 via the Internet [COLO98]. Technologies used for implementing this application are VRML[2] for the 3D visualisation, Java for the robot control GUI and C for the socket communication between the Java applet and the robot.
The main drawbacks of VRNomad are:

- Sockets provide a low level communication library. Programmers have to implement the byte streaming operations for each object.
- Socket libraries have different syntaxes depending on the platform and programming language.

Vizir

With Vizir the goal was to develop an augmented reality control of the Nomad200 [COLO99]. This application mixed 3D views with real time images from on-board cameras. 3D representation was implemented with OpenInventor.
Drawbacks:

- OpenInventor is a commercial product that is expensive. Paying maintenance licenses is not a sustainable solution for universities when such a software is used episodically by students for their thesis.

3 Objectives of the thesis

From the preceding discussion and the definitions it clearly appears that to improve robot control software and to reduce its development time we need a generic framework that lets different robotic systems communicate and collaborate.

From the preceding list of limitations and drawbacks, we can already propose a raw list of requirements for an "ideal" framework:

- Multi-platform communication
- High level communication
- Multi-platform GUI
- Open source software
- Object-oriented

2 Virtual Reality Modelling Language

- Robot independent algorithms implementation

This list will be refined and extend in the next chapter.

It is evident that this effort is not unique and that other researchers and laboratories have been faced with similar frustrations when developing their control software and have consequently developed their own frameworks based on similar requirements. However, after having reviewed the State of the Art in 2001, we did not find any framework that fulfilled all the requirements listed above and we consequently decided to develop our own.

> The main goal of my thesis is **the design, implementation and evaluation of a framework for multi-sensor robotic systems integration.**
>
> To validate the chosen approach, **typical modules used in robotic applications have been implemented and tested.**

The framework name is CoRoBA, which stands for **Co**ntrolling **Ro**bots with COR**BA**[3]. It is not intended for developing real-time control applications like closed-loop actuator control but instead for integrating different systems at a higher level. It has been designed to run on high performance computing systems, that is, normal or embedded computers and not light weight systems equipped only with micro-controllers.

In the next section we present the State of the Art in control frameworks and simulators. It is evident that since the start of this thesis, some frameworks have further been developed and have reached maturity while some new ideas and projects have emerged. The State of the Art presented is consequently a mix between the situation at the beginning of the thesis and at the writing of this dissertation.

4 State of the Art

4.1 Frameworks

Not so many tools are freely available for developing generic robotic applications. The ones available are often limited to specific applications and/or hardware and Operating System. The following review is limited to the most popular ones.

Telematics Applications

Before beginning any development, we conducted a review of relevant applications presented at the conference "Telematics Applications" in 2001 [COL02b]. From this analysis it appeared that **Java** was the preferred programming language and **CORBA** (Common Object Request Broker Architecture – see Chapter 3 for more explanations) the most used software middleware. **Web Browsers** were generally used as containers for user interfaces. No generic tools was available for developing teleoperation applications.

3 CORBA: Common Object Request Broker Architecture

GenoM

One of the best known framework is certainly the module generator GenoM. It is a tool that helps building real-time software architectures. It allows an easy and rapid integration of functions in communication-enabled independent modules. Functions can be dynamically started, paused, and parametrised by asynchronous invocations. Modules are standardised servers which are automatically generated from a synthetic description. The structure of a module has two parts: a controller that manages the module according to the clients' requests and the current state and the execution engines that carry out the activities required by the controller. GenoM corresponds to the functional level of the architecture developed at the LAAS_CNRS and presented in [ALAMI98]. This architecture has proven to be efficient and flexible [ALAMI00] and it is clearly devoted to real-time applications.

MCA2

MCA2[4] (Modular Control Architecture) is a software framework with real-time capabilities that is rapidly gaining in popularity It targets control applications of autonomous robots and enables developers to focus their work on developing control methods. MCA2 is neither an automatic code generation tool nor does it contain a visual programming tool. All methods are realized by simple modules with standardized interfaces ("edges") that can be grouped. Input and output interfaces are limited to arrays of floating point values. MCA modules communicates through low level sockets API. This architecture offers an homogeneous structure at all system levels. As modules can be integrated both on Linux and on RT-Linux without changes, they can be developed on Linux-side and then transferred later to RT-Linux.

DCA

DCA (Distributed Control Architecture) has been developed to control a mobile manipulator. In [PETE01] Peterson lists and analyses the requirements of this architecture. Actually we find in DCA many similarities with other projects. The originality relies in the adoption of a process-algebra for specifying tasks. Concerning the implementation, DCA offers a development environment with a communication library inspired by ACE[5] (ADAPTIVE Communication Environment), and a number of services. The execution relies on a tree organisation containing supervisors and controllers. The controller contains a process algebra interpreter that organises the execution of the controller modules.

The main drawback in GenoM, DCA and MCA2 is the proprietary and quite limited communication mechanism. The frameworks presented hereafter avoid these limitations by building up on communication middleware.

Since the beginning of this thesis in 2001 some frameworks that are very close to CoRoBA in the requirements and software implementation have emerged. It reinforced our conviction that the choices that we have made in the design phase are the good ones.

MIRO

MIRO (Middleware for Robots) is a distributed object oriented framework for mobile robot control, based on CORBA technology. MIRO core components are developed in C++ for Linux. MIRO development began earlier than CoRoBA and consequently more GUI visualization and configuration tools are available. However, in 2001 when I conducted the State of the art review, Event-based

4 http://mca2.sourceforge.net

5 http://www.cs.wustl.edu/~schmidt/ACE.html

communication was not yet implemented in MIRO [MIRO01]. A particular strong point of MIRO is its configuration capabilities through XML files.

MARIE

MARIE (Mobile and Autonomous Robotics Integration Environment) is a programming environment allowing multiple applications, programs and tools, to operate on one or multiple machines/Operating System and work together on a mobile robot implementation. The aim is to develop an integration framework based on the Mediator Design Pattern[6] for distributed systems. Each application adapter interacts with existing applications independently. MARIE uses ACE as its communication middleware. All interactions between applications are done asynchronously [COTE04].

Orca

Orca[7] started as part of the EU-funded OROCOS Project which purpose was to develop an Open-Source Robotic Control System. ORCA is an open-source set of tools for developing component-based robotic systems. It provides the means for defining and developing components which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks. In addition it provides a repository of pre-made components which can be used to quickly assemble a working robotic system.

Player

Player is a device server that provides a powerful, flexible interface to a variety of sensors and actuators (e.g., robots) [TOBY05]. It defines a set of standard interfaces (Interface specifications), each of which is a specification of the ways that you can interact with some class of devices. Because Player uses a TCP socket-based client/server model, robot control programs can be written in any programming language and can execute on any computer with network connectivity to the robot. In addition, Player supports multiple concurrent client connections to devices, creating new possibilities for distributed and collaborative sensing and control. Player control code that works with one robot will work (within reason) on another robot.
Player makes a clear distinction between the programming interface and the control structure, opting for a maximally general programming interface, with the belief that users will develop their own tools for building control systems. Further, most robot interfaces confine the programmer to a single language, providing a (generally closed-source) language-specific library to which the user must link his programs. In contrast, the TCP socket abstraction of Player allows for the use of virtually any programming language. In this way, it is much more "minimal" that other robot interfaces.
Stage and Gazebo are two simulators that present a standard Player interface and comes with popular robot and sensor models. Gazebo is presented in the next section.

As we will see in the following chapters, CoRoBA tries to merge the strong points of the frameworks mentioned above while minimizing their weak points.

4.2 Simulation

Having a simulator is essential when developing robot control software because it allows refining control strategies. Part of my thesis has been devoted to the development of a 3D simulator for multi-robotic systems that seamlessly interacts with the control framework. The Simulator is called

6 A Mediator promotes loose coupling by keeping objects from referring to each other explicitly.
7 http://orca-robotics.sourceforge.net/index.html

MoRoS3D (Mobile Robots Simulator 3D).

In the reminder of this section, we give an overview of some developments in 3D simulation tools for mobile robots. This review does not include the numerous Software Development Kit's and game engines that are available to develop 3D applications.

GSV

A project similar to MoRoS3D is GSV (Graphical Simulation and Visualisation) that has been developed at the University of Auckland by the Robotics Research Group as a module of their robot programming environment. Simulation services are exposed as CORBA interfaces (it is actually the only 3D mobile robot simulator having CORBA interfaces we are aware of). Most of the requirements of the GSV presented in [TREPA03] are also met in MoRoS3D but as a commercial game engine Torque® has been selected for the 3D visualization this simulator does not meet one of the aforementioned requirements. MoRoS3D offers equivalent capabilities but is based on the free library Java3D.

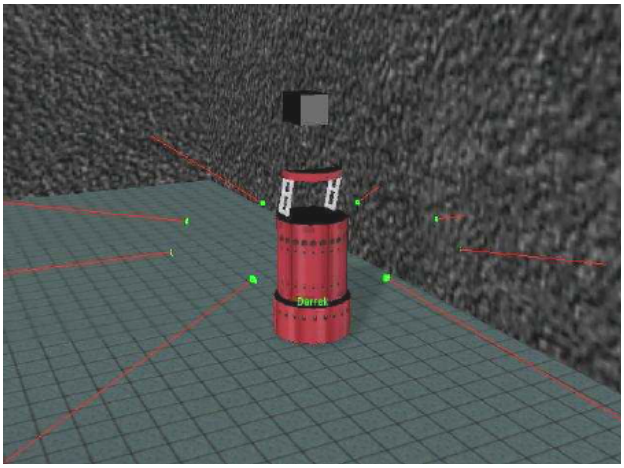


Figure 3. A B21r mobile robot with laser sensors in GSV

Other examples of free available simulators that do not use CORBA are:

STAGE/GAZEBO

The Player/Stage project provides two multi-robot simulators: Stage and Gazebo. Since Stage and Gazebo are both Player-compatible, client programs written using one simulator can usually be run on the other with little or no modification. The key difference between these two simulators is that whereas Stage is designed to simulate a very large robot population with low fidelity, Gazebo is designed to simulated a small population with high fidelity. Gazebo[8] is a multi-robot simulator for outdoor environments. It is capable of simulating a population of robots, sensors and objects in a three-dimensional world. It generates both realistic sensor feedback and physically plausible interactions between objects (it includes an accurate simulation of rigid-body physics). Gazebo implies the use of the Player framework[9].

8 http://playerstage.sourceforge.net/gazebo/gazebo.html
9 http://playerstage.sourceforge.net

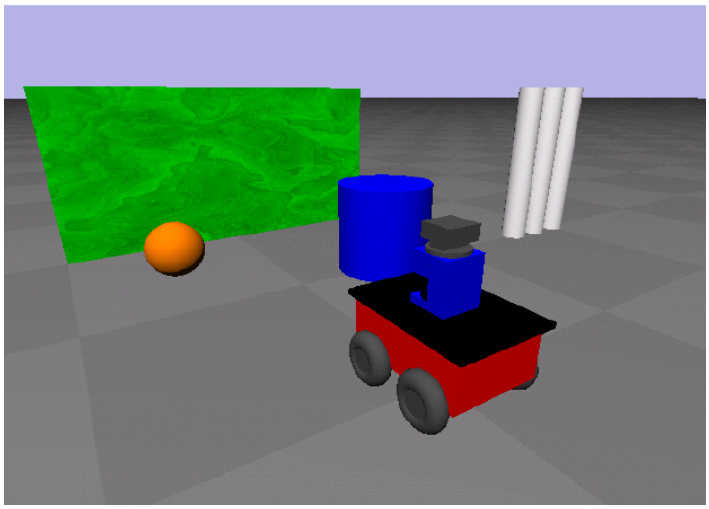


Figure 4. A Pioneer mobile robot with a Sick laser in GAZEBO

OpenSim

OpenSim[10] is a 3D simulator that uses OpenGL for real-time rendering of the robot environment as realistically as possible. It uses a physics engine to simulate dynamics in real-time (collision between arbitrary polyhedral objects with friction). Development has been ongoing for a while, however the simulator is a long way from rendering realistic scenes and only has a limited set of simulated sensors. It has been mostly used for research into inverse kinematics of redundant manipulators with constraints for tool use (for environmental restoration, disassembly and dismantlement tasks).

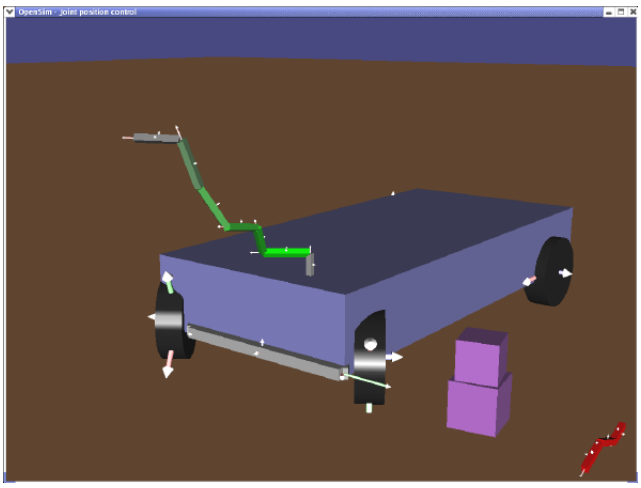


Figure 5. A mobile robot with a manipulator in OpenSim

After having reviewed existing software and on-going projects it clearly appeared that implementing our own simulator would be the easiest way for rapidly getting results and keeping the control of future developments.

10 http://opensimulator.sourceforge.net/

5 Originality and output

A deep analysis of requirements (user, developer, application) has allowed us to identify Robotics Control Patterns that represent almost all possible robotic applications. This approach has helped us to capture generic requirements for the framework.

The design of the framework relies on classical Design Pattern [GAMM95] and Real Time Design Pattern [DOUG03]. The coding is based on proven Object Oriented methods in order to improve software quality. This approach provides a solid base for future developments of distributed robotic applications.

The main output of this work are:

- CoRoBA, a versatile framework for developing distributed multi-sensor robot control applications.
- MoRoS3D, a 3D multi-robot simulator that seamlessly integrates with CoRoBA.

6 Thesis outline

This text is divided in six chapters and a concluding chapter. The present chapter has given the motivations and the goals of the thesis as well as a review of the State of the Art in the relevant domains. An overview of other chapters is given below.

Chapter 2 specifies the software requirements for a framework allowing the development of distributed mobile robots control applications. Two different approaches are considered to identify the requirements for the framework. The first approach takes into account the functionality of the applications we want to build with the framework, whereas the second one considers the needs of potential users. In order to identify reusable components, we introduce use cases that allow us to derive Robot Control Patterns. From different surveys, we produce a list of general characteristics owned by telematics applications. In the reminder of this chapter we detail them and infer from them the consequences for the framework design.

Chapter 3 begins with a presentation of the Design Patterns that form the theoretical foundation for the software design. It is followed by a section devoted to communication middlewares and justifies the choice made by showing that it fulfils the requirements presented in Chapter 2. The chapter continues with a discussion on architecture support for deployment.

Chapter 4 deals with the components architecture, that is their internal structure, representations, how they communicate with each other, which model they use to synchronize, to handle events, to store, retrieve and share data,.... The different types of components as well as the component interfaces and their implementation are covered in this chapter.

Chapter 5 first explains how the simulator interacts with the framework. In the second section, details of the environment model and scene graph representation are given. The chapter continues with sections devoted to distance sensors, robots simulation and collision detection. This part deals more particularly with geometric, kinematic and dynamic modelling for different robotic platforms. The CoRoBA integration and the description of the simulator engine conclude this chapter.

Chapter 6 is devoted to applications developed to validate the CoRoBA framework. The presentation describes simple applications like direct Joystick control and more elaborated ones, like autonomous

navigation and multi-robot control. Components involved in each application are explained in detail and reuse of components is emphasized throughout the chapter.

The last chapter relates the requirements presented in the first two chapters with the actual framework implementation. It is followed by the results of a detailed performance analysis. Finally, future research directions are suggested and global conclusions are drawn.

Chapter II Analysis

1 Introduction

This chapter specifies requirements for a software framework allowing the development of distributed mobile robot applications. The use of frameworks is beneficial for reaching a high quality level and accelerating the software development process. Furthermore, frameworks allow developers to concentrate on applications rather than on ancillary code.

Developing reusable software is an incremental and iterative work as illustrated by Figure 1. It begins with the conceptual design. In this step, requirements are captured, a high level architecture is produced and the purpose and function of the components are described. In the second step, the specification design is created. At this stage, the object model may be created, interfaces and sequence diagrams defined. In the final step, the implementation design, the final details are laid down and physical systems and technologies are selected. At this stage we have sufficient details for starting the implementation. These steps may be repeated several times during the development cycle.

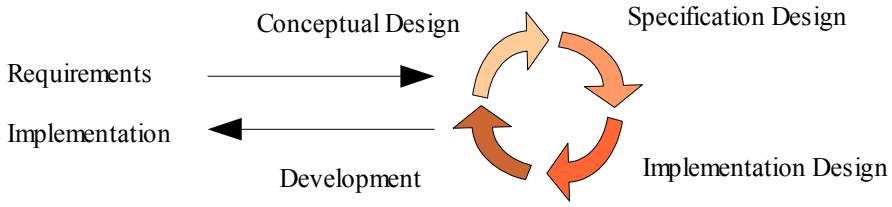


Figure 1. Software development steps

As the needs change, the code must consequently also be adapted. The life cycle of object-oriented software has typically several phases, namely prototyping, expansionary and consolidating phases (Figure 2).

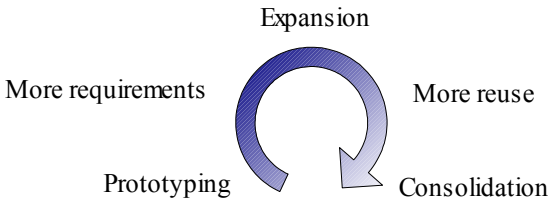


Figure 2. Software life cycle phases

During the different phases the code is often re-organized or re-factored. This means that from the beginning, software has to be designed with change in mind. In order to minimize those modifications or to simplify them, we need at least to focus on two points: requirements and Design Patterns. As we will see in the next chapter, using Design Patterns in early stages of software development avoids later re-factoring. Design Patterns anticipate specific changes by letting some aspects of system structure vary independently of other aspects [GAMM95]

This chapter deals with requirement gathering. Classically, user requirements define the functionality of the applications. Here we don't consider a specific application but rather application patterns from which we capture requirements for the framework.
In section 2 we discuss requirements that apply to every networked applications, that is distribution, communication, computing and performance issues, etc.
Section 3 presents a decomposition of applications based on increasing complexity that leads to the definition of Robot Control Patterns from which requirements are inferred.
Section 4 considers the needs in development and deployment phases. In this section we will see that requirements have multiple origins and that users' requirements are totally different from programmers' ones.

2 Computing and communication issues

The requirements and architecture design should lead to the proper identification of technologies. Whenever possible the choice of a technology or tools should not constraint the architecture. Therefore, the following important software aspects must be considered:

- Operating Systems
- Distribution and Networking technology
- Communication model
- Programming model and languages
- Portability
- Integration of existing systems

2.1 Operating systems

Low level control modules may require real-time[11] (RT) capabilities as delays in communication can introduce instabilities in the control loop. In a hierarchical architecture, the closer the modules are to the hardware layer, the more they have increasing real-time constraints.
Real-time systems are divided in hard real-time (an event is reacted to within a strict deadline) and soft real-time (will not suffer a critical failure if time constraints are violated). For closed-loop control of motors, hard real-time systems are required. The control period for such systems lies generally between 1 and 10 ms. For higher level tasks (path following, obstacle detection, navigation ...) we can admit control periods varying between 50 an 500 ms. For the highest level (Path planning, terrain modelling,...), time constraints are less critical (typical control period are between 1 and 10 s).

This work is more concerned with soft real-time and non real-time systems. If real-time capabilities are required, we may consider using RT OS (RT-Linux, RTAI, VxWorks, QNX,...). Higher levels and components having a long planning horizon do not need to be real-time and generic OS can also be used. Graphical User Interfaces (GUI) are typical non real-time components that can run on common OS and platforms or on Personal Digital Assistants. GUI's can also be based on interpreted languages as they do not need to be real time.
At this stage we differentiate two kinds of components: synchronous components (time-driven) and asynchronous components (data-driven). Synchronous components are generally designed to run on a

11 Wikipedia definition: “In computer science, real-time computing is the study of hardware and software systems which are subject to a "real-time constraint" —ie. operational deadlines from event to system response. By contrast, a non-real-time system is one for which there is no deadline, even if fast response or high performance is desired or even preferred.”

single processor (however there exists development tools to distribute real-time processes over several processors). Asynchronous components can be distributed over a network because they depend less on real-time synchronisation.

How well synchronous components perform mostly depends on the OS real-time capabilities while asynchronous components are influenced by communication performances. A common approach to separate real-time from non real-time tasks is the use of distributed events communication in order to decouple the data flow from the execution control. This decoupling does not mean that the arrival of data does not influence the execution control; it only means that it is the called component that decides how and when to service the event, and not the calling component. As a matter of fact, **the framework must provide asynchronous communications**.

In order to keep reactivity, components must be multi-threaded but when different threads have to access common data the use of synchronisation mechanisms is required. Therefore, **the framework must offer support for easy development of multi-threaded applications.**

2.2 Distribution and network technology

In multi robot and sensor networks applications processes are necessarily distributed over different networked processors. Specific algorithms can run on dedicated hardware (specialized subsystems) or be split between many computers for better performances. Distribution can also contribute to increase fault tolerance (a task can be solved in different places and data can be saved on different servers).**The framework must make the distribution of an application over multiple nodes easy for the developer.**

Developing robust, extensible and efficient communication applications is challenging. In particular, developers must master a number of complex operating system and communication concepts such as:

- Network addressing and service identification.
- Presentation conversions, such as encryption, compression, and network byte-ordering conversions between heterogeneous end-systems with alternative processor byte-orderings.
- Process and thread creation and synchronization.
- System call and library routine interfaces to local and remote interprocess communication (IPC) mechanisms.

It is possible to alleviate some of the complexity of developing communication applications by employing higher-level communication libraries that reside between clients and servers and automates many tedious and error-prone aspects of distributed application development, including:

- Authentication, authorization, and data security.
- Service location and binding.
- Service registration and activation.
- Demultiplexing and dispatching in response to events.
- Implementing message framing atop byte stream oriented communication protocols like TCP/IP.
- Presentation conversion issues involving network byte-ordering and parameter marshalling.

The previous discussion leads to the fourth requirement: **the framework must rely on a higher level communication library.**

2.3 Communication models

Communication regulation between distributed components can be implemented in different ways. Classical communication models used in distributed applications are:

- Master/Slave : The Master initiates and controls the communication with the Slaves.
- Client/Server : The Server provides services to the Clients request. The communication is performed using a well defined interface.
- Peer-to-peer: A peer process contains both a client and a server. Requests and replies go in both directions between the peer processes.
- Group: Message passing is used to talk to all members of the group. Unicast, multicast and broadcast are common mechanisms.

The framework should not impose the communication model to the application.

2.4 Programming model and languages

For Services requiring real-time performances or for those who are computation intensive, C is generally preferred over C++. However, the availability of Object Oriented (OO) design tools based on UML (Unified Modelling Language) facilitate the modelling phase when using OO languages. Platform-independent languages like Java or Python are interesting alternative for GUI's and configuration tools.

It must be possible to mix different programming language in an application.

2.5 Portability

Portability can be reached as long as an abstraction layer for hardware devices and operating systems functionalities is used. We distinguish two different cases: universal languages (Java, python, ...) using native interpreters meaning that binaries (byte code) can be used without any modifications on different computing systems or universal libraries that abstract OS calls to libraries and require the same (portable) source code to be compiled for different targets (ACE, wxWidget, QT, ...). While most of the robots are provided with libraries running on Linux, not everyone wants to develop on this OS and many developers are used to Windows. In order to be able to develop applications running on both OS, **the framework code must be portable**.

2.6 Modularity

An important consideration when designing any large, complex system is to break it into pieces for development and testing. As different users require different sets of features, dividing functionality in small units allows a developer to select exactly what is needed. Modularity is also essential when considering simulation and hardware in the loop capabilities that consists in replacing some hardware by simulated ones or running simulation in parallel with a real system for parameter estimation.

The framework must be modular.

2.7 Integration of existing systems

Beside developing new components, existing systems (Sensors, Robots, Algorithms, Applications) will need to be integrated into the framework.

Robots can be controlled by computers or micro-controllers. In the former, in order to connect to and control existing robots, we need to be able to call functions of the robots' native libraries that are generally written in C or C++.

In the case of micro-controllers, we would need to link them to a more capable computer in order to integrate the robot into the framework. This generally can be done through serial ports (more and

more with field-buses and means that we must be able to communicate through this kind of connection. New robots can be developed in such a way that they take into account in their design the characteristics of the framework. Consequently, **the framework must allow using native libraries.**

3 Robot Control Patterns

3.1 Definition

In order to gather additional requirements for the framework we should consider a large number of typical robotics applications. Such applications require the following functionalities:

- Mobility: control of actuators, proprioceptive sensors,...
- User Interface (in and out): control, reporting, visualization and monitoring,...
- Sensors for environment perception: Ultrasonic, Infra-red, vision,...
- Navigation: obstacle avoidance, path following,...
- Localization: dead reckoning, inertial systems, GPS, ...
- Planning: path and trajectory generation, sequencing of actions,...
- Task: specific actions carried on by the system in order to fulfil the mission

The combinatorial explosion that would result when considering all these characteristics has lead us to take another approach. We propose to define Robot Control Patterns. The list we propose is actually inspired by the Man-Machine Interaction Patterns proposed by R. Graves in [GRAV00].
Control Patterns classified from the simplest to the most complex one are:

- Direct control
- Monitoring
- Sensor data processing
- Direct telecontrol
- Supervised and autonomous control
- Multi-robot control
- Multi-user control

This classification makes it possible to identify requirements for typical components involved in each pattern.

3.2 Direct control

Description

The user controls the system through a User Interface (UI) (Figure 3) that can be made up of a Graphical UI and/or an haptic interface like a 3D joystick or a master arm (mice and keyboards are included in the GUI).

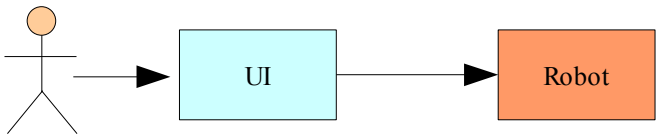


Figure 3. Direct Control

Problems

The following problems limit the flexibility and reuse of components in control applications:

- GUI's are specifically developed for one application.
- GUI's run on specific OS/platform
- Haptic inputs are generally processed in a global control application.
- Commands are specifically coded for one robot.

These problems have been for example reported by Graves in [GRAV00]. To solve them, he developed a generic GUI that operates a number of different types of remotely operated robots.

Requirements

From the aforementioned problems we can specify the following requirements:

- **Robot control GUI's must be independent of the robot.**
- **GUI's should run on most popular platforms.**
- **Motion commands must be independent of the robot.**

3.3 Monitoring

Description

We define monitoring as the visualization of relevant data coming from services (Figure 4). Monitoring allows a person to check the working of a given system. Depending on the circumstances, this person may or may not influence the monitored process. In both cases the operator should be able to customize the visualization process.

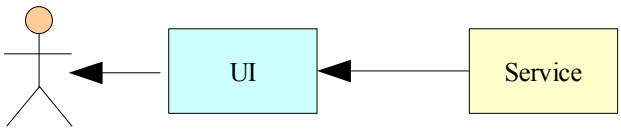


Figure 4. Monitoring

Examples of services that can be monitored are:

- robots: position, configuration, 3D graphics,...
- sensors: cameras (fixed images, streams), distance sensors, proximity sensors,...
- process: results of data processing or mission status.

Problems

The data visualization is generally embedded in a global application GUI that requires the installation

of specific software and if new sensors and/or new data types have to be shown, the application must be recompiled and redeployed.

Requirements

The listed problems suggest the following requirements:

- **It must be possible to select the monitored service at run-time**.
- **The GUI must display and save data in various formats: video, audio, text, 2D vector graphics, Bitmap graphics, 3D, etc.**
- **The GUI must be independent of any application and must adapt itself automatically to the system capabilities.** For instance, if we add a camera component to the application, the GUI should adapt itself to display the images.

3.4 Data processing

Description

Data produced by sensors must be processed in order to extract relevant information from which decisions on robot actions can be taken (Figure 5). This case encompasses two different domains: processing of sensors data that is required for the control of the robot and processing of application specific sensors data.

Problem

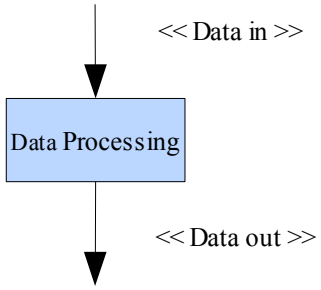


Figure 5. Data processing

Algorithms are often embedded in a global application and tailored to specific sensors and robots. In order to change algorithms, programs have to be recompiled and redeployed on distant machines.

Requirements

The framework must allow interchanging algorithms easily. The developer must have the possibility to add and select processing modules in order to evaluate different solutions and choose the ones that best suit his needs. It could be interesting to launch parallel data processing that exploit different algorithms and have the possibility to compare and select outputs from different processing components. We can summarise these requirements by saying that **the architecture of the whole application must be defined at run-time**.

The first three patterns address the basic requirements of the framework and can be combined and refined to build more complicated patterns. The next pattern is a direct combination of the first two ones.

3.5 Direct telecontrol (teleoperation)

Description

If the user has no direct visual contact with the robot and its environment, viewing services are used (Figure 6). We assumed in this application pattern that the user directly interacts with the robot. We also suppose that the control latency and refresh rates are good enough to assume that the close-loop user–command–view is fast enough to allow real-time control of the robot(s). The figures depend on the application and the speed of the robot. Latencies of 0.2 sec and refresh rates equal to 10 images/seconds are practical limits to control a slow moving robot (5 km/h). Image and sensor resolution are other important limiting factors.

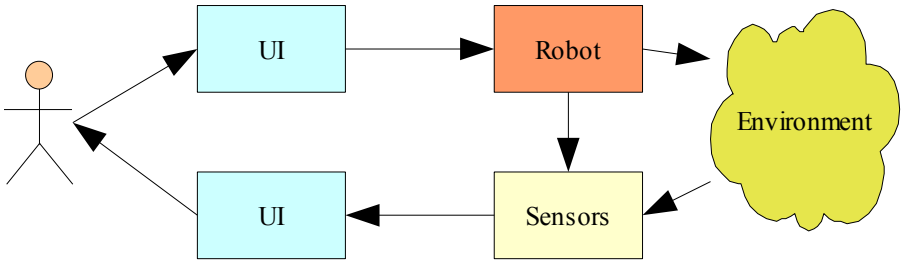


Figure 6. Direct telecontrol

Problem

Control and visualization are embedded in a global application.

Requirements

The command and visualization data flow must be separated from each other.

3.6 Supervised and Autonomous Control

Description

Supervised and autonomous control

Traded, shared and assisted control are different forms of supervised control as defined by Sheridan in [SHER92].

In the traded control pattern the operator acts as a supervisor but may from time to time assume direct control. This can be done by giving him the highest priority or by temporary disabling other behaviours.

In the shared control pattern, the operator may act as a supervisor with respect to control of some variables and direct controller with respect to other variables. For example he could teleoperate a pan and tilt camera mounted on a autonomous mobile robot.

Assisted control is another form of supervised control. Navigation assistance is based on proprioceptive and exteroceptive sensor data processing. Speed regulation and path following is generally based on dead reckoning or inertial platforms but it could also benefit from external sensors like GPS or visual tracking systems. Obstacles avoidance and obstacles following requires perception sensors like IR, US, cameras, radar, lidar,...

We can also provide the operator with assistance through perception (augmented reality and force-feedback control).

In the autonomous control pattern the operator can observe but not influence the process (other than pushing an emergency stop button).

Deliberative and reactive control architectures
Hierarchical decomposition is the classical approach for developing motion control for autonomous or semi-autonomous robots. Typical deliberative robot control architectures comprise three levels: Planning, Executive, and Functional (or Control) (Figure 7). These levels are usually organized according to the level of abstraction at which they operate.

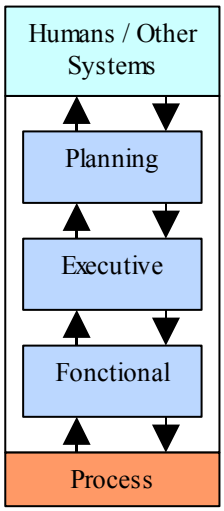


Figure 7. Deliberative architecture

The Planning level constructs high-level plans utilizing Artificial Intelligence planning search techniques. In the past, these algorithms have typically been computationally intensive and required a significant amount of time to respond to new updates or changes. Domain knowledge for this level is encoded in a declarative model, where it can easily be utilized by different search techniques.

The Executive level is responsible for execution of plans produced by the Planning level. The Executive level typically performs further expansion of planned activities based on current execution context. This level is also responsible for monitoring activities and robot conditions as execution proceeds and for handling exceptions as they arise. This level must quickly react to changes, so it is usually more responsive than the Planning level. Domain knowledge at the Executive level uses procedural representations such as looping constructs, conditionals, etc.

The Functional level is responsible for low-level control of the robot. This level typically consists of real-time control loops that directly command the robot hardware, and that tightly couple sensors to actuators. This level is not addressed in this work.
Deliberative architectures are usually used to implement autonomous robots. The user interacts with the highest level and cannot act at lower ones.

In Behaviour Based control sensors are directly coupled to reactive modules that provide basic behaviours. Behaviour Coordination Mechanisms (BCM) are necessary to produce effective motion commands. If behaviours are viewed as operands, then BCM's are the operators used to combine

behaviours into higher-level behaviours. BCM's can be divided into two main classes: arbitration and command fusion, which are complementary.

Arbitration mechanisms select one behaviour from a group of competing ones and give it ultimate control of the system until the next selection cycle. This approach is suitable for arbitrating between the set of active behaviours in accord with the system's changing objectives and requirements under varying conditions. It can focus the use of scarce system resources (sensory, computational, etc.) on tasks that are considered to be relevant. Two possible implementations are:

- Priority-based arbitration: which is a subsumptive-style, where behaviours with higher priorities are allowed to suppress the output of behaviours with lower priorities.
- State-based arbitration: which is based on the Discrete Event Systems (DES) formalism, and is suitable for behaviour sequencing.

Figure 8 illustrates a typical subsumption architecture as originally proposed by Brooks [BROO85].

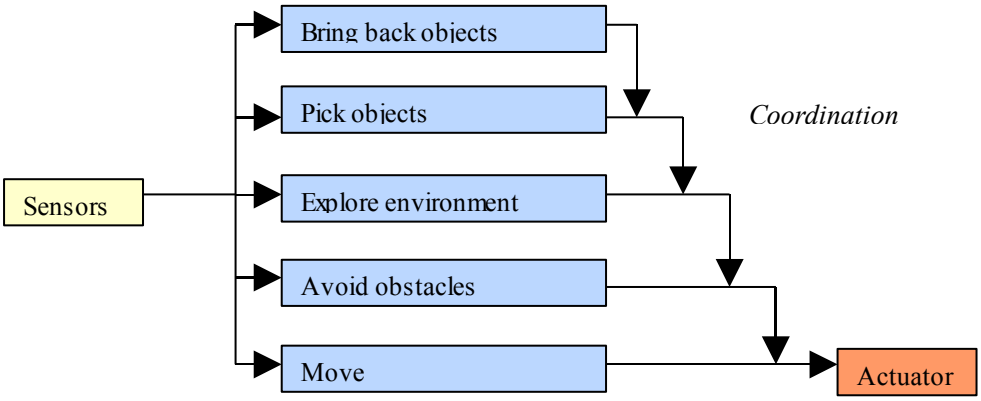


Figure 8. Typical subsumption Architecture

Command fusion mechanisms combine recommendations from multiple behaviours to form a control action that represents their consensus. Thus, this approach provides for a coordination scheme that allows all behaviours to simultaneously contribute to the control of the system in a cooperative rather than a competitive manner. This makes them suitable for tightly-coupled tasks that require spatio-temporal coordination of activities.

Examples of complementary mechanisms for fusion:

- Voting techniques (Action selection architecture, ...)
- Fuzzy command fusion mechanisms
- Multiple objective behaviour fusion (Schema's based architecture , ...)

Both architectures have advantages and drawbacks and it seems legitimate to combine both approaches to profit from their advantages while trying to cancel their drawbacks. Behaviour based architecture can be used in different ways to obtain various control scheme. P. Arnaud proposed in [ARNA00] the Generalized Actions Fusion Architecture that integrates different BCM approaches in one framework.

On top of the behaviour layers we also need supervisors that can perform scheduling, context switching, error reporting, ...

Problems

Recurrent problems are:

- Control architectures are tailored to a given application.
- Links between components are fixed.
- User interaction is limited to a given control level.

Requirements

The framework must allow the implementation of generic control components that can be customized at design and configured at run-time, that are reusable and that can be easily combined with each other. The framework must provide the capability to fix priorities or enable/disable behaviours running in components, to add or remove behaviours, to group and discover them at run-time, that is to modify the application's functionality at run-time. The framework should make it possible to implement different BCM's and to interchange them seamlessly. To summarize, **the framework may not impose the control architecture.**

3.7 Multi-robot systems

3.7.1 Coordination

Description

When several robots need to work together, it is necessary to manage a given number of supplementary tasks that are not directly productive but serve to improve the way in which those activities are carried out.

The coordination of actions is one of the main methods of ensuring cooperation between autonomous robots. Actions have to be coordinated for four main reasons:

- The robots need information and results produced by other robots.
- Resources are limited.
- We want to optimize costs.
- We want to allow robots having separate but interdependent objectives to meet their objectives while profiting from this inter-dependence.

Among all coordination methods [FERB99], the coordination by synchronization is one of the easiest solutions. To synchronize several actions it is necessary to define the manner in which actions are time-related, in order to time them in the right order and carry them out just at the right moment. Synchronization constitutes the lowest level of the coordination of actions. Petri nets are generally used to describe and solve the problems of synchronization.

Problem

Processes run on different machines, having each their own clock that are not synchronized.

Requirement

The framework must provide mechanisms to synchronise processes.

3.7.2 Scalability

Description

A system scales well if its performance reduces not more than proportional to the applied load.

Problem
The communication and control of processes easily become a bottleneck certainly when a central supervisor or a blackboard concept is used.

Requirement
The system must be scalable.

3.8 Multi-user systems

Description

In some applications, it could be necessary or advantageous to split the control of the robot(s) between different operators. We consider for example the command and control of a group of reconnaissance robots for which different functions have to be accomplished:

- Motion control
- Obstacle avoidance
- Navigation
- Observation
- Self-protection

Different approaches are possible:

- One person controls one system
- One person is responsible for one level
- A pyramidal approach is used
- Some functions are shared or autonomous

In such complex applications it could be required not only to divide the workload between different operators but also to modify this division in function of the performed tasks and of the circumstances. Obviously coordination between operators will be required.

Problems

Common limitations to flexibility in user control are:

- Commands are provided by a single control centre.
- Only one user can take the control of the whole system.
- Users are not able to communicate with each other to coordinate control actions

Requirements

- **The framework must allow splitting the robots' control between different users.**
- **In order to coordinate control actions, communication between operators may be required (instant messaging with text, data, voice, video).**

4 Development and deployment support

4.1 Development

Besides the needs from users we also have to consider those from other categories of people involved in the development process. In [BRUY02] Bruyninckx proposes a four level organisation: Framework

builders, Component Builders, Applications Builders and End Users. Lars Peterson divides the users in 5 groups, namely End users, Application programmer, Module programmer, Interface programmer and Hardware designer, and describes in [PETE02] what they should expect from a framework. The correspondence between the two approaches is summarised in the following table.

Table 1. Categories of users

Lars Peterson [PETE02]	**Hermann Bruininckx [BRUY02]**
User	User
Application programmer	Application Builder
Module programmer	Component Builder
-	Framework Builder
Interface programmer	-
Hardware designer	-

Users use programs developed by Application developers. Their focus is on the functionality of the application. They need therefore an intuitive User Interface. This interface should be developed by Application Developers and will not be provided by the Control Framework. However, **technology selection made in the design of the Framework should not constraint the development of application UI's**.

Application developers assemble and customise components provided by Components developers to build an application. Tools must be provided to to facilitate the development of new components and the integration of existing components into applications.

In order to perform extensive tests of applications, a simulator is required.

In order to identify possible problems monitoring tools are necessary.

For assuring maintainability, high-quality documentation of the system design and implementation is of prime importance.

Framework developers work on the infrastructure code that will support development of other categories of developers. They do not target any specific applications but they must keep in mind the needs of the other categories and the applications that will be developed with the framework.

Interface programmers are closer to the hardware. They need to be sure that device drivers they develop can be seamlessly integrated in the framework.

Hardware designers have no direct interaction with the framework.

4.2 Deployment

Once it has been developed, a software has to be deployed. A system manager with administration rights has to install and configures services.

- Once in operation, **logging tools are useful to track software activities**.
- And finally, **the system must be stable and reliable**.

5 Summary

This chapter has reviewed and presented requirements for a framework that must facilitate the development of distributed robots and sensor networks applications. Some requirements have been derived from Robot Control Patterns while other have been based on general software development considerations.

In the following table the requirements have been grouped in different categories: Meta-requirements (M), Functionality requirements (I), Development requirements (D), Application requirements (A), UI requirements (UI) and Use requirements (U).

Developing a framework that meets all the aforementioned requirements is certainly an utopia. In the architecture and specification design we will have to make choices and compromises. The Implementation requirements are off course the key ones because they essentially define the framework structure and implementation. This work obviously focuses on these requirements. The Meta-requirements are related to the quality of the software and some measures of effectiveness will be defined in order to evaluate how these criteria have been met.

Most of the listed requirements are totally or partially addressed in this work excepted those related to the User Interface and some from the Development and Application categories. In the next chapters we will see how they are satisfied.

Table 2. requirements

#	Title	Category
1	Applications developped with the framework must be stable and reliable	M
2	The framework must be modular	M
3	The system must be scalable	M
4	The framework must allow using native libraries	F
5	The framework must rely on a higher level communication library	F
6	The framework must provide asynchronous communication.	F
7	The framework must offer support for easy development of multi-threaded applications	F
8	The framework must provide the flexibility to make the distribution of an application over multiple nodes easy for the developer	F
9	The architecture of the whole application must be defined at run-time	F
10	The framework must provide mechanisms to synchronise processes	F
11	The framework should not impose the communication model to the application	F
12	The framework must allow splitting the robots' control between different users.	F
13	The framework must allow access to several users at the same time.	F
14	The framework code must be portable	F
15	It must be possible to mix different programming language in an application	F
16	Technology selection made in the design of the Framework should not constraint the development of application UI's	UI
17	Robot control GUI's must be independent of the robot.	UI
18	The GUI must run on most popular platforms	UI
19	The GUI must display and save data in various formats: video, audio, text, 2D vector graphics, Bitmap graphics, 3D, etc.	UI
20	The GUI must be independent of any application and must adapt itself automatically to the system capabilities.	UI
21	It must be possible to select a monitored service at run-time	UI

#	Title	Category
22	Tools must be provided to to facilitate the development of new components	D
23	In order to perform extensive tests of components a simulator is required	D
24	Tools must be provided to to facilitate the integration of existing components into applications.	D
25	The command and visualization data flow must be separated from each other.	A
26	Motion commands must be independent of the robot	A
27	In order to coordinate control actions, communication between operators may be required (instant messaging with text, data, voice, video)	A
28	For assuring maintainability, high-quality documentation of the system design and implementation is of prime importance	U
29	In order to identify possible problems monitoring tools are necessary	U
30	Once in operation, logging tools are useful to track software activities	U

Chapter III Framework Architecture

1 Introduction

The previous chapter focusses on requirements that will serve for the design of the framework architecture and the main functionality of components. Analysis is driven by what the system should do while design is characterized by how the system must achieve the requirements. In this introduction we give some design guidelines and we consider the consequences of granularity and partitioning on the framework. Section 2 presents Design Patterns implemented by the framework. Section 3 is devoted to communications. The framework architecture is covered in Section 4. Section 5 considers architecture support for deployment.

1.1 Design guidelines

Besides requirements it is just as important to have a clear set of design guidelines that we can use to guide the technical decisions that must be taken as we develop the solution that meets those requirements. The following guidelines concerning the choice and utilisation of tools have been considered during the development of the framework:

- The selection of open source software (GNU General Public License,...)
- The use of standard tools and technologies (UML, C++,...)
- A framework design based on Design Patterns
- The development of validation applications

With the requirements gathered in the previous chapter and the system guidelines listed above, we can consider the framework design.

1.2 Granularity, partitioning and interfaces

Granularity, partitioning and interface design are key features of any distributed object model [BALE00].

Granularity refers to the level of abstraction that is provided by each component through its interfaces while partitioning is concerned with the location of the objects, that is in which process they run and where they are located in the network (Figure 1). Interfaces are the public faces of the systems.

Objects in coarse-grained models represent higher level concepts. This representation simplifies the implementation of applications because it reduces the code and the number of processes but limits the freedom of the developer. On the other hand a fine-grained model offers more possibilities to the programmer but also requires more work because the number of interactions is larger. He must learn more because he has to understand the internal mechanisms of the object model.

A fine-grained model affects performance because more interactions between objects are required to perform a single task. Even if computers are always improving in speed and gigabit network interfaces are available, each remote method invocation adds some overhead and increases the latency of the global system. A general rule is that the cost of the communication should not go beyond the cost of execution.

On the other hand, a coarse-grained system reduces the flexibility when more than one concept is embodied in a single interface. A just compromise has consequently to be found to allow developed components to be reusable in new applications.

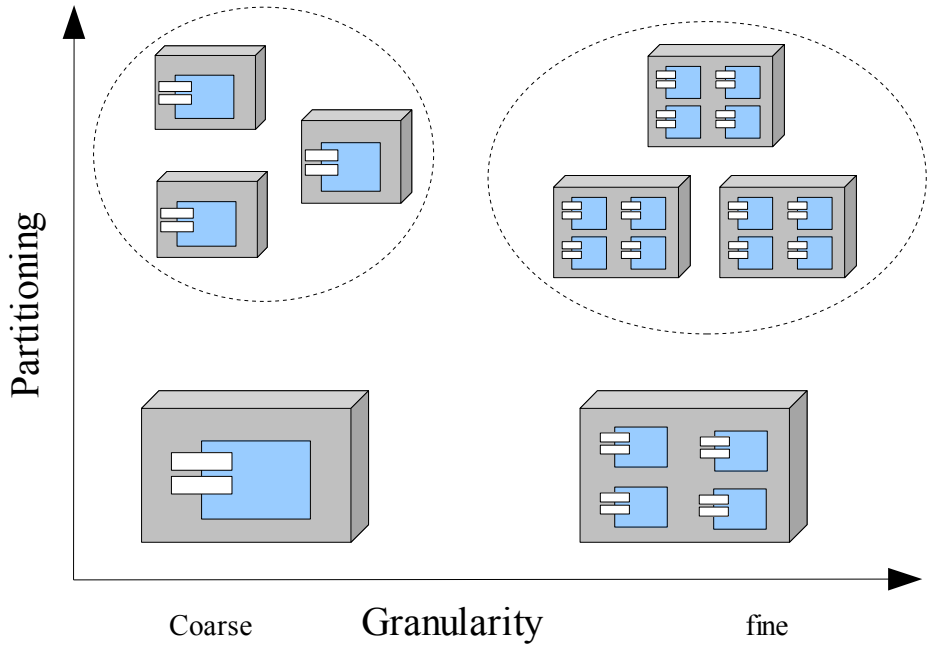


Figure 1. Effects of granularity and partitioning on system components

A distributed system must be partitioned so as to support evolution. It can be decomposed into a set of subsystems where each subsystem provides a well-defined service. Effects of partitioning can be evaluated by using sequence diagrams, which are good indicators of the amount of communication necessary to perform the tasks required by the system. Partitioning has not only an influence on deployment but also on interfaces because objects must implement interfaces in order to be remotely accessible.

When designing an interface the following principles have to be followed:

- Interfaces should support a single concept (cohesion).
- Coupling between interfaces must be kept minimal.
- Exceptions have to be defined.
- A polymorphism strategy has to be chosen.

The next chapter shows how the proposed implementation deals with these issues.

2 Design Patterns

A framework captures the design decisions that are common to its application domain. Applications based on frameworks can be built faster and have similar structures; they are consequently easier to maintain. The main drawback is that applications are particularly sensitive to changes in the framework interfaces but Design Patterns actually help reducing these changes.

2.1 Definition

Design Patterns capture good design practices and present them in a systematic way. A pattern describes a recurring problem and the core solution to that problem. In general, a pattern has four essential elements:

- The **pattern name**: it facilitates the discussion and lets us design at a higher level of abstraction.
- The **problem** describes when to apply the pattern. It explains the problem and its context and might include a list of conditions that must be met before it makes sense to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities and collaborations. It does not describe a particular concrete design or implementation because it is a template that can be applied in many different solutions.
- The **consequences** are the results and trade-off's of applying the pattern. The consequences of a pattern include its impact on a system's flexibility, extensibility or portability.

Many Design Patterns have been used by different authors but they became very popular after the publication of the famous book [GAMM95] by what is known as the "gang of four". In this book Design Patterns can be classified according to their purpose: creational, structural and behavioural. Each category can yet be subdivided according to the scope criterion that specifies if the pattern applies primarily to classes or objects. Patterns for designing control frameworks have been presented in [DOUG03].

2.2 Design patterns and framework

Sets of interrelated patterns tailored specifically to work well together are called frameworks. Patterns and frameworks have similarities but are different in three major ways:

- Design Patterns are more abstract than frameworks
- Design Patterns are smaller architectural elements than frameworks
- Design Patterns are less specialized than frameworks

A framework is a set of cooperating classes that make up a reusable design for a specific category of software. Frameworks provide four primary usage strategies: instantiation, generalization, parametrization and extension. In a framework-based development effort, the majority of the application is provided by the instantiated framework.

The disadvantages of frameworks are that they limit the freedom of developers and that they are much more difficult to design and construct than applications, even though they greatly simplify application development. [DOUG03, p128-129].

Choosing the right patterns from a catalogue is not straightforward and relies mainly upon the experience of the developer. A short description of selected patterns is given below.

2.3 Architectural Design Patterns

Architectural Design Patterns that are presented here are divided in two categories: *Subsystem and Components Architecture Patterns* on one hand and *Distribution Patterns* on the other hand [DOUG03].

2.3.1 Subsystem and component architecture Patterns

- **Name**: Hierarchical Control Pattern

Problem: We want to separate the interfaces for the control and configuration of the objects and the interface for the actual functionality provided by the object.

Solution: This pattern is based on composition. The Hierarchical Control Pattern uses two types of interfaces: control interfaces that monitor and control how the behaviours are achieved and functional interfaces, which provide the services controlled by the other set of interfaces. This pattern is illustrated by Figure 2. The control interface provides services to manage how the functional services are performed. The functional interface provides the services of the Controller according to the parameters selected via the control interface. Leaf elements participate in the realisation of the functionality.

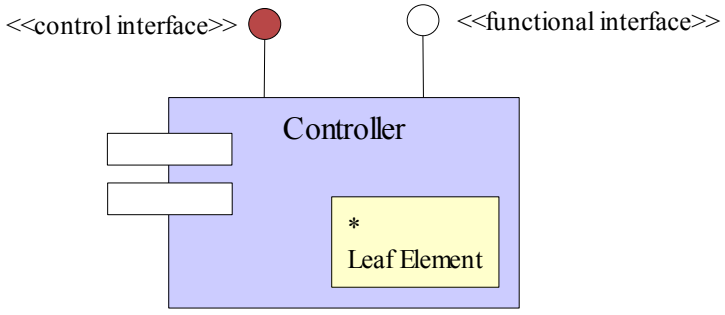


Figure 2. Hierarchical Control Pattern structure

- **Name**: Component-based Architecture Pattern.

Problem: We need an architecture that is robust in the presence of maintenance and is highly reusable in a variety of circumstances.

Solution: The Component-based Architecture Pattern organises a system into replaceable units with opaque interfaces. Interfaces are generally divided in Client and Management Interfaces (Figure 3).

Consequences: Systems may be constructed via assembly, components being selected at run-time. Opaque interfaces hiding implementation details can be seen as an advantage or a disadvantage. Components tend to be heavy in terms of required resources (memory, size on disk).

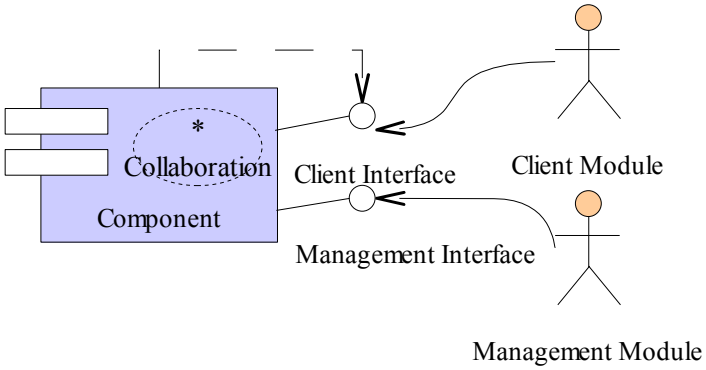


Figure 3. Component Based Architecture Pattern structure

- **Name**: Channel Architecture Pattern.

  Problem: We would like an architectural structure that improves throughput capacity with the replication of units allowing efficient processing of multiple data in different stages of processing. We would also like an architecture that improves reliability and safety through the simple addition of redundant processing units.

  Solution: The Channel Architecture Pattern is useful when data within a stream is sequentially transformed in a series of steps. A channel can be thought of as a pipe that sequentially transforms data from an input value to an output value (Figure 4). It is possible to find multiple elements of the data stream in different parts of the channel at the same time.

  Consequences: The Channel Architecture Pattern improves the flexibility provided by the Component-based Architecture Pattern by lowering the coupling level between components. It also greatly facilitates the implementation of Safety and Reliability Patterns because similar components can run concurrently on different machines.

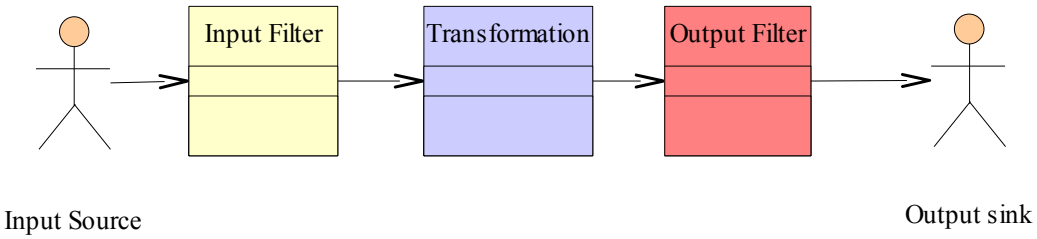


Figure 4. Channel Architecture Pattern structure

2.3.2 Distribution Patterns

Distribution, which is an essential aspect of architectures, comes in two primary forms: asymmetric and symmetric. In asymmetric distribution systems, the binding of objects to the address space is known at design time while in symmetric distribution system it is not known until run time. Symmetric distribution is more flexible and allows dynamic load balancing. The patterns presented

below deal with the collaboration aspects of the architecture and focus on how the objects find and communicate with each other.

- **Name**: Remote Method Call Pattern.

  Problem: The programming model used to invoke local services is very well understood and what is needed is a means to do the same thing even when the client and the server do not reside in the same address space.

  Solution: The Client does not communicate directly with the Server but via a Client Stub that contacts the Server Stub, which invokes the specified method on the Server. The low-level network operations are hidden to the Client and Server (Figure 5).

  Consequences: this pattern simplifies the process of Client Server communication over a network. It offers a *pull* approach; the server merely responds to a request from a client. There are many implementations that are based on this Design Pattern. Most of them propose a mechanism (compiler, macro, ...) that automatically generates the stubs.

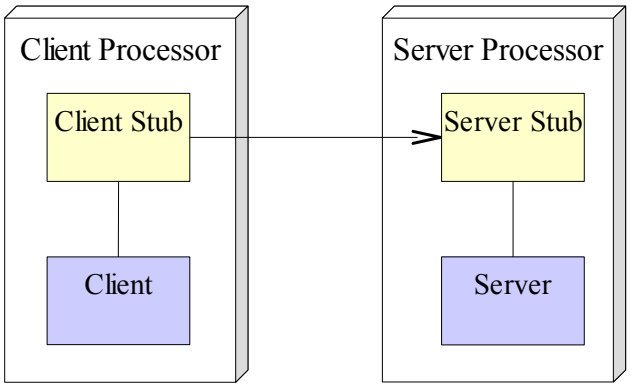


Figure 5. Remote Method Call Pattern structure

- **Name**: Broker pattern.

  Problem: Most of the distribution patterns require a priori knowledge of the location of the servers what limits their use to asymmetric distribution architectures. Ideally, the solution should provide a means to locate and invoke services at the request of the client.

  Solution: The Broker Pattern may be though of as a symmetric version of the Proxy Pattern. It provides a Proxy Pattern in situations where the location of the clients and servers are not known at design time. The Broker is an "object reference repository" globally visible to both clients and servers (Figure 6).

  Consequences: The Broker Pattern is a very effective means for hiding remoteness of clients and servers. While not completely successful in hiding all the details, it nevertheless greatly simplifies the creation of systems with symmetric distribution architectures.

Commercial Object Request Brokers (ORB) do require a minimum amount of resources that may exceed those available in small computing systems. For those cases, it may be possible to use smaller, less capable ORB's or write one from scratch that includes only the desired capabilities. In the Broker Pattern, the clients may dynamically discover the available services. This makes the Broker Pattern more scalable than the Proxy Pattern but also somewhat more heavyweight.

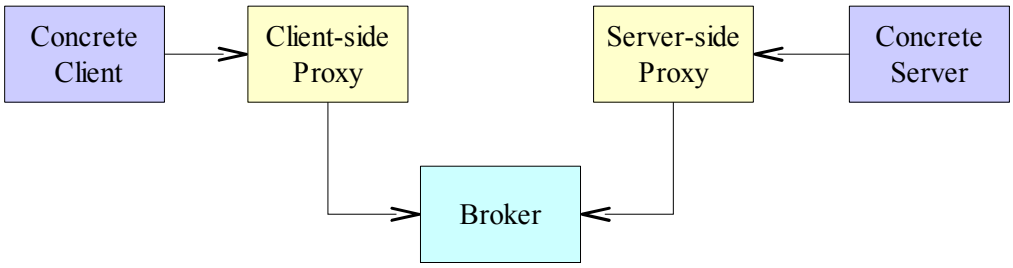


Figure 6. Broker Pattern structure

- **Name**: Data Bus Pattern.

Problem: Many systems need to share many different data among a mixture of servers and clients, some of them might not be known when the client or data is designed.

Solution: The Data Bus Pattern further abstracts the classic Observer Pattern [DOUG03, pp370-376] by providing a common (logical) bus to which multiple servers post their information and where multiple clients come to get various events and data posted to the bus. The Data Bus Pattern is basically a Proxy Pattern with a centralized store into which various data objects may be plugged along with metadata that describes its contents (Figure 7). It serves to further decouple the client implementation from the server's. The pattern comes in both "push" and "pull" versions.

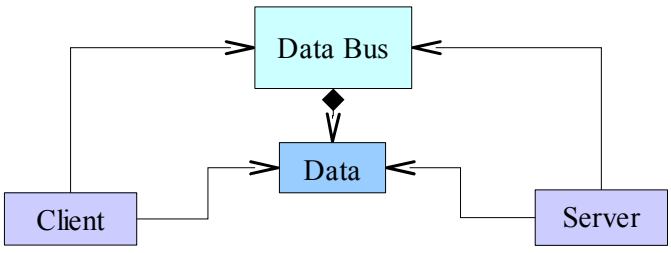


Figure 7. Data Bus Pattern structure

Consequences: The Data Bus pattern offers a single location for clients to go and acquire data and for servers to publish their data. The Data Bus is extensible; it does not have to understand the semantics of the data. New servers and clients can be added at run-time.

2.4 Behavioural Patterns

Behavioural class patterns use inheritance to distribute behaviour between classes [GAMM95].

- **Name**: Template Method Design Pattern

  Problem: Several classes implement an algorithm having a common structure but a slightly different implementation. We want to reduce the code redundancy.

  Solution: The Template Method Design Pattern defines the skeleton of an algorithm in an operation of a base class and lets subclasses redefine certain steps of this algorithm without changing the algorithm structure.

  Pattern Structure: see Figure 8.

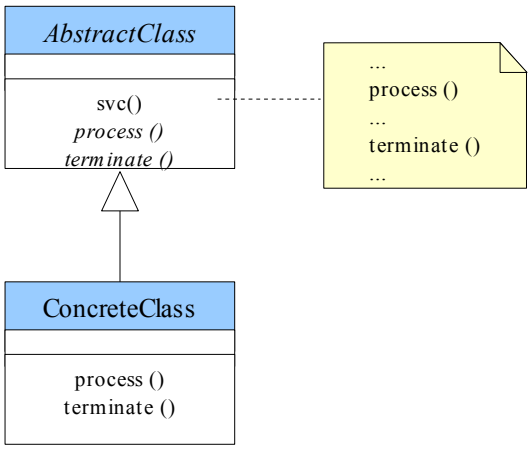


Figure 8. Template Method Design Pattern structure

  Collaboration roles:
  AbstractClass:
  - defines abstract primitive operations that concrete subclasses must implement (example: process, terminate)
  - implements a template method defining the skeleton of an algorithm (example: svc)

  ConcreteClass:
  - implements the primitive operations to carry-out the invariant steps of the algorithm.

  Consequences: This pattern is a fundamental technique for code reuse. It is used when common behaviour among subclasses should be factored and localized in a common class to avoid code replication. It also leads to an inverted control structure because the parent class calls the operations of a subclass and not the other way around.

2.5 Concurrency Patterns

Concurrency Patterns provide solutions when the different threads of an applications are not independent and share resources that must be managed carefully to avoid corruption.

- **Name**: Message Queuing Pattern

Problem: In most multi threaded systems, threads must synchronise and share information with others.

Solution: The Message Queuing Pattern provides a simple means for threads to synchronise and communicate information among one another using asynchronous communications implemented via queued messages.

Pattern structure: The structure for the pattern is shown in Figure 9.

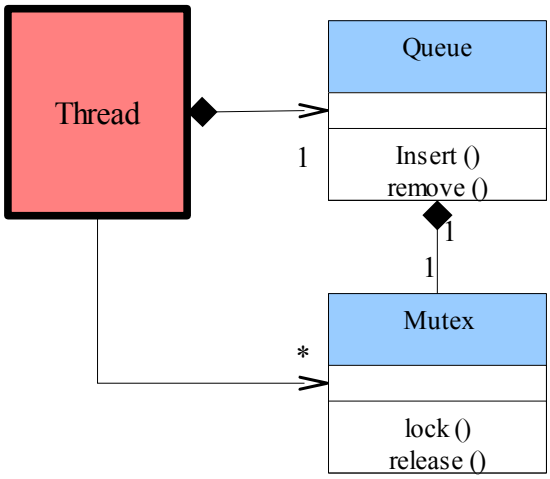


Figure 9. Message Queuing Pattern Structure

Collaboration roles:

Thread

The thread object is active. It can both create messages to send to other threads via the queue and receive and process messages when it runs.

Queue

The queue is a container that can hold a number of messages.

Some fundamental questions must be answered when implementing such a queue. What is the (maximum) size of the queue ? Do we allow increasing the queue size ? What do we do with events in excess ? Do we use a cyclic buffer ? Do we store different sorts of data in the same queue or do we use several queues ?

The answers to these questions depend mainly from the behaviour we want to obtain.

We can consider two different cases:

- Events produced by components are disposable in the sense that we may lose some without jeopardising the system stability.
- All events must be processed.

Mutex

The mutex is a mutual exclusion semaphore. It provides non interruptible lock and release operations that protects data against simultaneous access.

Consequences: This pattern has many advantages and is supported by virtually all real-time operating systems. The primary disadvantages are that it is a relatively heavyweight approach

to information passing among threads and information must be shared by value instead of by reference.

The way these Design Patterns are used in the framework is described in Chapter IV.

3 Communication

3.1 Communication libraries

Native Application Programming Interfaces (API's) for writing communication software are available on all platforms. They propose the well-known socket mechanism, which provides an endpoint for communication between processes. Those API's are written in C or C++ and offer different abstraction levels in function of the version. The main problem is that the software implementation is different for each platform. Functions have different names, parameters and initialization sequences. This situation obliges the programmer to learn different libraries in order to write hybrid network applications. This option offers the advantage that programmers can optimize the code and write very efficient and small footprint programs. It is perhaps acceptable for full-time programmers but not for control application developers who don't have the time to devote to this activity. It is preferable for them to write intelligent high level applications that perform real tasks than to spend most of the time for writing low level code. The need for a higher level communication library is therefore evident.

Multi-platform communication library's could eliminate some of the drawbacks listed above. Incidentally, the ADAPTIVE Communication Environment (ACE) [HUST04] has been considered as a possible communication library at the beginning of this project. However, it does not provide a higher level of abstraction like middlewares (see next section) do and much work is left to the programmer who still has to deal with low level socket operations, data marshalling and unmarshalling, localisation of services in a network, etc.

3.2 Middleware

A typical distributed software architecture presents a layered structure as shown in Figure 10. The first two bottom layers constitute the platform; they provide services to the layers above them. Ideally the application layer should be independent of the platform layer. That is why we find an intermediate layer named middleware. The aim of this layer is to mask heterogeneity and to provide developers with an uniform API to implement communication and resource-sharing support for distributed applications [COUL01, pp31-32].

However, middleware does not solve all problems and sometimes introduces an artificial homogeneity and delays integration problems. Moreover one should not forget that there exist many different and incompatible middlewares and that integrating them becomes in itself a new challenge. One way proposed by S. Vinoski in [VINO02] is the Web Service paradigm. Web Services are based on the ubiquitous Internet infrastructure while the communication occurs via XML-based messages.

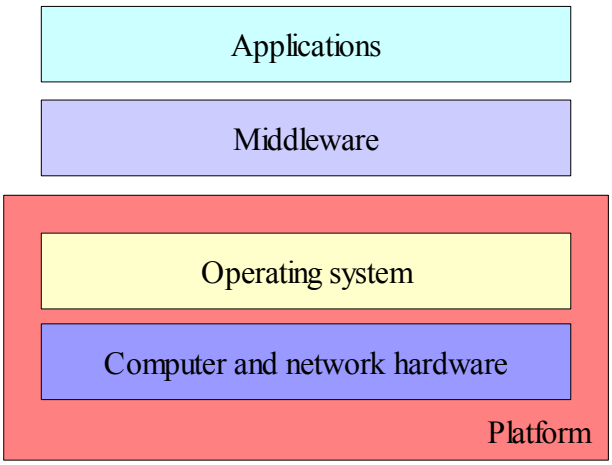


Figure 10. Layered structure of distributed applications

Another drawback of middlewares is the steep learning curve programmers have to cope with. For this reason, middleware utilisation is only valuable in long term projects.
As performance and flexibility are generally mutually exclusive, middlewares try to offer a good compromise between both requirements. They must be largely configurable in order to satisfy an heterogeneous user community. Run-time configuration separates the development from the deployment and eliminates the needs for software modification by the user.

3.3 Programming models

In distributed applications, programs need to invoke operations in other processes, often running on different computers. To achieve this, the following programming models are available [COUL01]:

- The Remote Procedure Call (RPC) which allows clients to call procedures in server programs running in separate processes and generally in different computers from the client.
- The Remote Method Invocation (RMI) that allows objects in different processes to communicate with each other.
- The distributed Event-based programming model that allows objects to be notified when events they have registered interest in have been emitted.

As most of the current distributed systems are written in object oriented languages, we only consider the last two models.

3.3.1 Remote Method Invocation

RMI is mainly represented by three implementations: Java-RMI, DCOM and CORBA. A comprehensive comparison of these three middlewares can be found in [COLO02].

DCOM

The Distributed Component Object Model (DCOM) is an extension to COM that allows network-based component interaction. While COM processes can run on the same machine but in different address spaces, the DCOM extension allows processes to be spread across a network. It is a Microsoft product that runs almost exclusively on Windows and consequently does not satisfy requirement **R14**.

Java RMI
The Java platform's Remote Method Invocation (named Java RMI) system has been specifically designed to operate in the Java application environment. The choice of Java RMI would oblige us to use almost exclusively Java and it is certainly not the best choice for developing control applications. Furthermore, the use of Java makes the integration of existing libraries more difficult[12] than by using the C++ language (requirement **R4**).

CORBA
The Common Object Request Broker Architecture (CORBA) is an open distributed object computing infrastructure being standardized by the Object Management Group (OMG). It ensures interoperability across programming languages, machines and products. CORBA automates many common network programming tasks such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling and operation dispatching.

CORBA actually implements the **Remote Method Call Pattern** and relies on Interface Definition Language (IDL) compilers to generate stubs and skeletons for clients and servers (Figure 11). This Design Patterns contributes to fulfil the requirement **R8**.

An Interface definition written in IDL completely defines the CORBA operations and fully specifies each operation's arguments. Operations specified in IDL can be written in and invoked from any language that provides CORBA bindings. C++ and Java are two of the supported languages. Most of the time, Interfaces are automatically translated into concrete code by an IDL compiler. This code has then to be customized by the developer with application specific instructions.

CORBA offers different communication solutions that give the developer a large freedom when implementing distributed applications (requirement **R11**). Besides the classical synchronous method call (named 2-way in CORBA jargon), we have at our disposal the Asynchronous Messaging Invocation (AMI) or the Event based communication services.

The CORBA synchronous method call is the most familiar to the programmer because it applies to remote calls the same principles as to a local method call. It uses a synchronous communication model and consequently method invocations block until the response is received from the remote object.
The AMI allows sending processing requests to a remote object without blocking the calling process. This later receives the response when it is available and a callback or a polling mechanism have to be used to get the response data. The AMI mechanism requires modifying the client but not the server which is unaware of the change.

12 It is however possible to use C++ libraries via the Java Native Interface API.

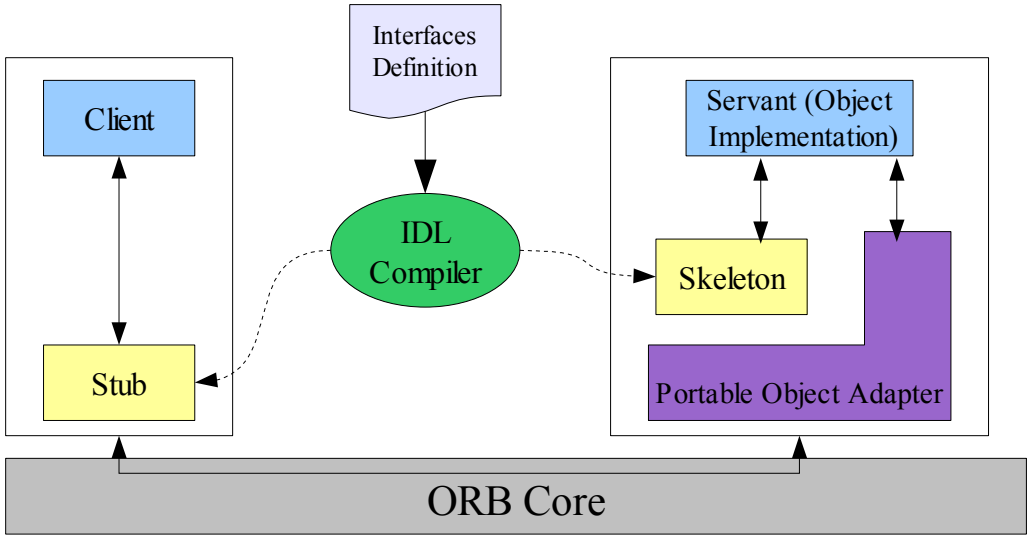


Figure 11. CORBA Remote Method Call Pattern implementation

The servant, visible only to the server, is the executing CPU and memory resource that performs an object's operation. It is activated and deactivated according to the pattern selected by POA policies. The Portable Object Adapter (POA) is the piece of the ORB that manages server-side resources for scalability. It deactivates objects' servants when they have no work to do, and activates them again when they're needed.

3.3.2 Distributed event notification

Jini

Jini cannot be compared to Java RMI and CORBA. Java RMI and CORBA can be seen as middleware technologies that enable components and objects to communicate over a network. Jini, on the other hand, provides an interaction model and the infrastructure for distributed objects to cooperate with each other, or to work in a coherent, robust, and scalable way.

The servant, visible only to the server, is the executing CPU and memory resource that performs an object's operation. It is activated and deactivated according to the pattern selected by POA policies. The Portable Object Adapter (POA) is the piece of the ORB that manages server-side resources for scalability. It deactivates objects' servants when they have no work to do, and activates them again when they're needed.

3.3.2 Distributed event notification

Jini

Jini cannot be compared to Java RMI and CORBA. Java RMI and CORBA can be seen as middleware technologies that enable components and objects to communicate over a network. Jini, on the other hand, provides an interaction model and the infrastructure for distributed objects to cooperate with each other, or to work in a coherent, robust, and scalable way.

Jini specifies functionalities that allow to set up a network of objects that dynamically link together and perform useful work. Jini does not specify how the server and client objects communicate. This

kind of work is the speciality of the other two technologies. In Jini RMI and CORBA do no compete against each other but are complementary and can be mixed to build up a complex network. The Jini distributed event specification allows a potential subscriber in one Java Virtual Machine (JVM) to subscribe to and receive notifications of events in another JVM. Java RMI is used to send events between objects.
Despite its interesting characteristics it seems that Jini has not reached the level of popularity that was awaited by its developers.

CORBA Event and Notification Service
There are many situations where the standard CORBA (a)synchronous request/response model is too restrictive. For instance, clients have to poll a server repeatedly to retrieve the latest information. Likewise, there is no way for the server to efficiently notify groups of interested clients when data change. For these reasons the Object Management Group first introduced the Event Service. The Event Service supports asynchronous exchange of messages between clients. It introduces event channels which broker event messages, event suppliers which supply event messages, and event consumers which consume event messages. The CORBA specifications define different methods for sending and receiving events: consumers and suppliers can push or pull events via Event Channels (Figure 12). Implementations of the Event Service act as "mediators" that support decoupled communication between objects. Events are typically represented as messages that contain optional data fields.

A primary goal of the Notification Service is to enhance the Event Service by introducing the concepts of filtering and configuration according to various quality of service requirements. Clients of the Notification Service can subscribe to specific events of interest by associating filter objects with the proxies through which the clients communicate with Event Channels. Furthermore, the Notification Service enables each channel, each connection and each message to be configured to support the desired quality of service with respect to delivery guarantee, event ageing characteristics, and event prioritization. The advantages of this communication method is counterbalanced by the complicated consumer registration (multiple interfaces, bidirectional object reference handshake, ...). Not all CORBA libraries implement the Notification Service.

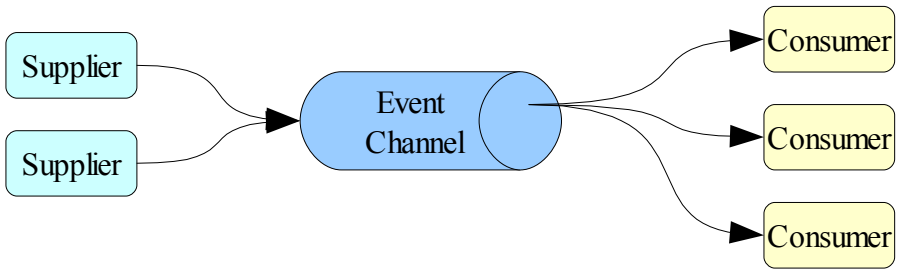


Figure 12. CORBA Event Service communication principle

3.4 Middleware selection

The CORBA middleware have been selected for implementing the framework communication functionality. CORBA is actually a specification of the Object Management Group (OMG) and presently more than 30 implementations coexist on the market. Some are freely available others are commercial products. Their common characteristic is that none of these versions implements all

specifications. While the third version of CORBA has been published, most of the CORBA implementations conform partially to the version 2.3 to 2.6 of the specifications.

The TAO[13] (The ACE ORB) implementation has been chosen among others as the CORBA library for C++ developments. For developers of distributed and embedded applications who have stringent performance demands, TAO is a freely available, open-source, and standards-compliant real-time implementation of CORBA. TAO applies the best software practices and patterns to automate the delivery of high-performance and real-time QoS to distributed applications. TAO has been ported to many operating systems including almost all UNICES, Win32, VMS, QNX, ...

One could ask the question if middleware is really usable for building distributed robotic systems ?

In [GILL02] C. Gill reports results of a comparative performance experiment that partially answer this question: using CORBA for data transmission instead of raw sockets adds an overhead that is mostly prominent for small data packets. Those results are explained by the operations added by CORBA and the extra information contained in a GIOP[14] frame.

Jay Gowdy [GOWD00] reviewed a wide variety of communication toolkits. He has qualitatively compared them based on criteria such as suitability for implementation of typical data flows in robotics, portability and ease of use. According to this review, CORBA (and particularly the TAO implementation) performs very well for every criteria but the "ease of use". According to him, it should be considered in "... long-term projects with nebulous, possibly changing goals...". Similar projects like Miro and Orca, presented in the first chapter, are also based on CORBA.

The weakness of the CORBA standard lies in the management of a system of distributed objects that has to be developed by the programmer. It lacks services that enable the user to monitor the distributed objects, manage those processes and interacts with network management tools [BALE00, p30]. Each CORBA implementation offers a proprietary solution for doing this.

Before taking the final decision, a test application has been developed. A CORBA serial server has been implemented using the different available communication models. A complete description of this application has been reported in [COLO04]. From this test application we concluded that CORBA would suit our needs.

Different language mappings have been defined in the CORBA standards and we are not limited to C++. For instance, Java provides comprehensive API's to communicate with CORBA objects. As long as we use the 2-way communication scheme, the SUN implementation coming with the Java Development Kit is sufficient but for the AMI or the Notification Service we must use another CORBA implementation as explained in Chapter V.

13 http://www.cs.wustl.edu/~schmidt/TAO.html

14 The General Inter-ORB Protocol has been defined to allow interoperability between different CORBA implementations.

4 Architecture support for deployment

4.1 Introduction

Besides application components (navigation, vision,...), which are the building blocks of an application, a framework has also infrastructure components (Name server, Time server,...) that provide services that are used by application components once deployed. TAO includes many generic services that can be directly reused as infrastructure components in numerous applications. Services that are used by the framework are described below.

4.2 Event based communication

Event based communication is not part of the core of CORBA but is supported by OMG Event and Notification Services. Once started the services offer interfaces for managing communication channels through administration objects. The Notification Service attempts to preserve all of the semantics specified for the OMG Event Service, allowing for interoperability between basic Event Service clients and Notification Service clients.

4.3 Configuration

A flexible system has more chance to survive the permanent changes of the computing world. In order to improve the flexibility of components, it is advantageous to push configuration as far away as possible in the development and deployment process. Late binding consists in selecting object implementation at run-time and is based on the Factory and Strategy Design Patterns [GAMM95].

The key to providing flexible systems lies in the amount of self-description within the system. This requires adding meta-information[15] management capabilities. Meta-information allows to increase the flexibility of the system and brings in dynamic capabilities. The downside to building a system using meta-information is the increase in complexity. The more generic a piece of code is, and the more reliance on runtime information, then the greater the chance of unexpected error situations. We need obviously to weigh the expected benefits with the cost.

Configuration information are generally provided on the command line or read from a configuration file but other possibilities are offered by CORBA for managing meta-information:

- Interface Repository
- Naming Service
- Trading Service
- Implementation Repository
- Meta-Object Facility

4.3.1 Interface Repository

Interfaces in CORBA are designed using an Interface Definition Language (IDL) and can be seen as both a contract and meta-information. The Interface Repository provides run-time information about IDL interfaces. Using this information, it is possible for a program to encounter an object whose interface was not known when the program was compiled, yet, be able to determine what operations are valid on the object and invoke requests using the Dynamic Interface Invocation (DII). DII provides the tools to create and invoke requests at runtime. This capability can be used to implement

15 Meta-information is information about information.

"intelligent" clients that discover interfaces at runtime or gateways between different protocols.

4.3.2 Naming Service

Object References[16] are used to locate the distributed objects in CORBA. The Naming Service provides a mechanism for the publication and dissemination of Interoperable Objects References (IOR). It contains a database of name-to-object-reference mappings that can be organised as a directed graph. Anything, but typically the server process hosting an object, may bind an object reference with a name in the Naming Service by providing the name and object reference. Interested parties (typically clients) can then use the Naming Service to resolve a name to an object reference. It is also possible to create federated Naming Services by cross-referencing the services running on different machines.

4.3.3 Implementation Repository

It is the responsibility of the ORB to ensure that an implementation of a distributed object is active when it receives a request for that object. It needs to keep track of which object implementations are available and whether they are activated or not. It does this by working with the server to keep track of when it is activated and stores information on how to reactivate it. Method invocations on the server will actually be sent to the Implementation Repository, which will then be able to start the server process if it is not already running and forward the invocation to the real server.

We may also have more than one implementation of a given server and each implementation may be located on different machines. This information can be incorporated within the Implementation Repository and used by the ORB for load balancing. The Implementation Repository itself may also be distributed and replicated among a set of machines, allowing for redundancy. In this case, the Object Reference of an object will contain the addresses of the various Implementation Repositories.

4.3.4 Trading Service

The Trading service allows to find other objects on the network that match a set of criteria. Within the trader an object is associated with a set of properties. Rather than locating an object based on its name, an object is located based on its capabilities. The service type name, the object reference and a list of name-value pairs describing the capabilities of the object must be provided to the trader by an exporter (server). An importer (client) will make a request for a service using a constraint language consisting of Boolean expressions. Since the properties of an exporter can be dynamic, searches can be based on the state of the object. Traders can also be federated and provide the core mechanism for the dynamic use of objects.

4.3.5 Meta-Object Facility

The Meta-Object Facility (MOF) is a recent addition to the services provided by the CORBA specifications. It provides the necessary functionality to describe relationships between the distributed objects. A standard based on XML has also been proposed to represent meta-information on a MOF repository: The XML Metadata Interchange (XMI).

4.4 Load balancing

In control applications, we can distinguish two types of processes: those which may run on any computer and those that are constrained to specific ones. This is the case for machines that are physically linked to sensors and actuators. In order to be able to distribute components without additional constraints, executables must be able to run on every available platform. Another

16 An Object Reference is an identifier that can be used throughout a distributed system to refer to a particular unique remote object.

parameter is the number of event channels created when an event based communication scheme is used. Each possible case corresponds to a given cost that comprises the processing cost and the communication cost.
Apart from trivial cases, it is necessary to study the way processes are distributed. Load-balancing tries to optimize resources allocation. Optimization requires the definition of metrics as computing load (memory, CPU utilization, file IO), network bandwidth, space storage, ... The load can be balanced statically and/or dynamically. Static load balancing relies on expected usage patterns of the proposed services and can be updated based on off-line analysis of metrics. Dynamic load balancing is based on instantaneous load metrics.
The Naming Service and the Implementation Repository can be used to perform basic load balancing [BALE00, pp237].

4.5 Safety and Reliability

4.5.1 Definition

A safe system will no cause any accident or loss in case of failure while a reliable system has a high probability to function for a specified period of time. Both Safety and reliability requires redundancy in the designs of systems.

4.5.2 Failure Models

Even if programmers try to produce bug free software, failures and errors will always occur and because they cannot be totally eradicated, we must deal with them. Object Oriented Languages have a mechanism called Exception handling. Exceptions are errors that can be foreseen at design time and for which we can provide special handling code from which we can recover.
In a distributed system both processes and communication channels may fail. Failures can be classified as omission, arbitrary and timing failures [COUL01, p53].

Omission failures happen when a process or a communication channel fails to perform what it is supposed to do. For processes it generally means a crash while a communication channel produces an omission failure if it does drop messages. This is generally caused by a lack of buffer space on the receiver or by a network transmission error.
The crash of a local application can generally be easily detected; its GUI does not longer respond to users' actions or it does not produce any more output on screen. Using an appropriate mechanism the failed program can be manually stopped and started again.
Detecting that a distributed application has crashed is not a trivial task. The basic detection method relies on time-outs. If the remote process has not replied to invocations after a given period of time, we may assume that it has stopped functioning. In an asynchronous system a time-out does not mean necessarily that the other process has crashed.

Arbitrary failures refer to the worst possible semantics, in which any type of error may occur (false data is produced by the system). They are generally produce by bugs in a program and cannot be detected by seeing whether the program responds to invocations. Arbitrary failures in communication channels are rare as they can be easily rejected by the communication software.

Timing failures are applicable in synchronous distributed systems where time limits are set on execution time, message delivery time and clock drift time.

4.5.3 Dealing with failures

Patterns described in [DOUG03] provide solutions for improving safety and reliability of a software system. Most of them are based on redundancy or on watchdog mechanisms.

CORBA encapsulates communication errors in networked exceptions [HENN99]. A number of system exceptions are defined to capture common error conditions. The developer has also the possibility to define his own exceptions in IDL.

For processes controlling a physical system, a crash could leave it in a dangling state. That is why most of the robots use time-outs and stop moving if no commands have been received after a given period of time. What is true for abnormal termination also applies to normal program shut-down. A stopped program should always leave the system in a safe state.

4.6 Security

The simplest security solution is undoubtedly isolation. The more distributed and interoperable the system, the more it is open for potential compromises. Security should always be designed as an integral part of the system because it touches every component and layer. The security of a distributed system can be achieved by securing the processes and the channels used for their interaction. It should be in the infrastructure to ensure authorized access between components. Information sent over communication lines should be encrypted.

One of the goal of the CORBA Security Service is to provide security for applications and users in a transparent manner. The features provided by the OMG security service can be added to an ORB in a non-intrusive manner because it is implemented with interceptors. Existing distributed applications that make use of a CORBA ORB can thus run without alteration using a secure ORB. The Secure Socket Layer can also be easily integrated in any CORBA application.

4.7 Logging and monitoring

Knowing what is going on at any time in a system is the first step in managing it. There are three major forms of tracking:

- Logging captures runtime data and control flow information that is typically archived and often centres on one service at a time.
- Monitoring provides runtime information that is dealt with at run time and that captures the state of the system as a whole.
- Auditing records purely business-related information throughout the system.

The best pattern for implementing these tracking mechanisms is to have a single service where all information is gathered. At the life cycle points of any entity to be tracked, a message is sent to the central information-gathering service. In a more sophisticated, distributed version, local gatherers can be federated, feeding to the central gatherer, which can provide a centralised archive and user interface.
TAO provides a highly configurable DEBUG/TRACE mechanism as well as a distributed logging service named Telecom Logging Service.

4.8 Life Cycle and Persistence

Life cycle refers to the meaning of implementing an entity that may exist beyond the scope of a single process while persistence considers the fact that an entity may exist but not its implementation.

CORBA Objects can be Transient or Persistent. Often new CORBA programmers are confused by the distinction made between CORBA objects and their implementation. CORBA objects are represented by Interoperable Object References (IOR) and are implemented by servant that create code objects corresponding to these references. For objects that will only be referenced once Transient IOR's are created. The CORBA specifications assure that each IOR is unique so that after object deletion, this reference will never be used again. Persistent IOR's on the other hand must survive their implementation and the state of the objects, if any, are stored in a file or a database.

5 Summary

Basing the architecture of the CoRoBA control framework on proven Design Patterns is a good strategic choice because the implementation of these patterns will improve the software flexibility, maintainability and reliability.
The second decision concerns the choice of the communication library. Some framework developers have opted for low-level socket libraries. While this is a good choice with regard to performance, it is a bad one concerning portability and maintenance. The need for a higher-level communication library is clear. CORBA has been selected because of its language and platform independence. Among different CORBA implementations, we have chosen TAO because it is widely adopted, it implements most of the CORBA specifications and is free open-source software.

The last section of this chapter considered what the deployment implicates for the architecture. It also presented the most important CORBA services that are useful for deploying applications.

Now the theoretical foundations have been defined, it is time to present the design and implementation of the framework and of its constituting components.

Chapter IV Design and Implementation

1 Introduction

The proposed software is a solution package for developing distributed applications and is composed of:

- A framework of components called CoRoBA.
- A 3D simulation application that interacts seamlessly with the framework components,
- Utility services for distributing and managing the live and run cycle of components.

The framework and the utility services are described in this chapter while the simulator is presented in Chapter V.

From the definition given in Chapter I, we know that a framework is a generic solution to a family of problems. CoRoBA can effectively be called a framework because it proposes solutions for building distributed multi-robots control applications. Because a framework is also an implementation, CoRoBA offers classes that constitute the building blocks of components. These classes contain most of the repetitive and error-prone operations that are reused without any modification in all components.

The implementation extensively uses the property of Object Oriented languages: inheritance, encapsulation and polymorphism. The structure is furthermore based on classical Design Patterns. As all components derive from the same parent classes, they all have the same internal structure. And as the basic structure is systematically reused we are sure that it has been tested many times in different situations, it is consequently a guarantee of stability and robustness. The communication mechanisms are for instance managed by the framework. The developer has however the possibility to use classic synchronous calls or a more advanced Event based communication scheme.

As components developed with CoRoBA are independent they can easily be reused in different applications. Similar components are exchangeable; a component can be instantaneously replaced by an other one having the same Interface.

All these characteristics allow to decrease the learning and the development time and to increase the software reliability.

2 Framework Architecture

2.1 Design Patterns

A framework is actually an abstract concept that is characterized by the architecture of the components and the communication patterns. In this section we explain how the selected Design Patterns presented in Chapter III are implemented in CoRoBA. The next section is devoted to the component architecture

It is important to keep generic functionality separate from specific functionality so that changes

made on one part has a limited impact on the other one. For reaching this goal, CoRoBA implements the **Component-based Architecture Pattern** and the **Hierarchical Control Pattern** (Chapter III, section 2.3).

The execution unit in CoRoBA is a **component**. Components are independent and have separated interfaces for the configuration and the actual functionality that they provide (Figure 1). The interface allowing the remote configuration is called Service. The Client interface on the other way provides access to component functionality. Each component has a dedicated CORBA interface that allows to clearly identify it. This approach combined with the interface inheritance makes it possible to develop generic and specific tools, that is, tools tailored to specific interfaces. This design choice defines a fairly coarse granularity (Section 1.2 of Chapter III) that clearly separates functionality and facilitates partitioning of components.
These patterns provide the framework with the modularity required by the requirement **R2**.

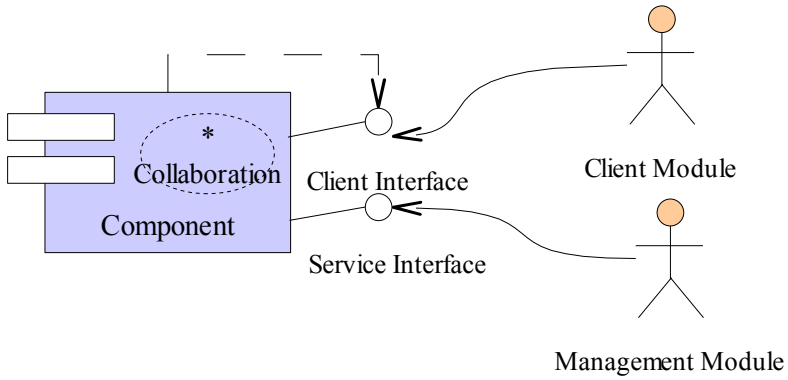


Figure 1. Component Based Architecture Pattern structure

Components form a chain along which information is transferred (Figure 2). Like in classic control schemes, the data flow is unidirectional. It is an implementation of the **Channel Architecture Pattern** (Chapter III, section 2.3). Data can be transferred synchronously if operations of the Client Interfaces are used or asynchronously as it will be explained in section 4. This Design Pattern participates in improving the modularity (**R2**) and the reliability (**R1**) of a system.

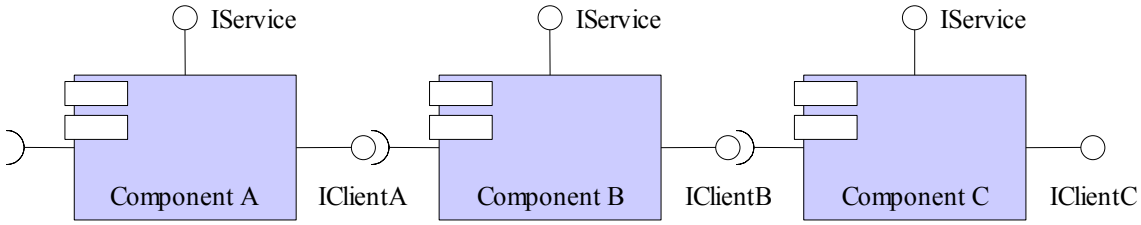


Figure 2. Chain of Components

Two other Design Patterns implemented by CoRoBA will be subsequently presented.

2.2 Component architecture

The Service interface defines the remote management operations that control the life and run-cycle of components (Figure 3). Other operations allow to change the running mode (see section 5), to read and change the period (in the Periodic mode), to get general information like the creation date, the author, etc. The complete IDL definition of this interface can be found in Appendix B. It is the base interface all other interfaces are derived from.

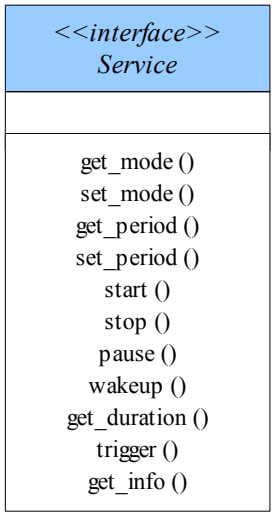


Figure 3. Operations defined by the Service Interface

The operations defined by the *Service* interface are implemented by the class RMA_Service_i that inherits from the abstract *Service* class generated automatically by the IDL compiler. In CoRoBA all servant objects are subclasses of the *RMA_Service_i* class.

The definition of separate interfaces as required by the Hierarchical Control Pattern is the first step for separating the process logic from the management logic. To complete the separation, the management data flow is decoupled from the process data flow by using different threads.

RMA_Servide_i also inherits from the ACE class *ACE_TaskBase* that facilitates the creation of portable multi threaded applications (Figure 4). The *ACE_Task_Base* is an ACE utility class that implements the Active Object pattern [GAMM95]. It contains four virtual methods that are used to manage the execution cycle of a thread: *close*, *svc*, *suspend* and *resume*.

The four management operations defined in the *Service* interface are mapped to the *ACE_Task_Base* methods allowing to control the thread remotely. The *start* activates the thread, *pause*, *wakeup* and *stop* respectively call the *suspend*, *resume* and *close* methods.
A special component called *Component Remote Control* (CRC) has been develop to interact with this *Service* interface. This is illustrated by Figure 5.

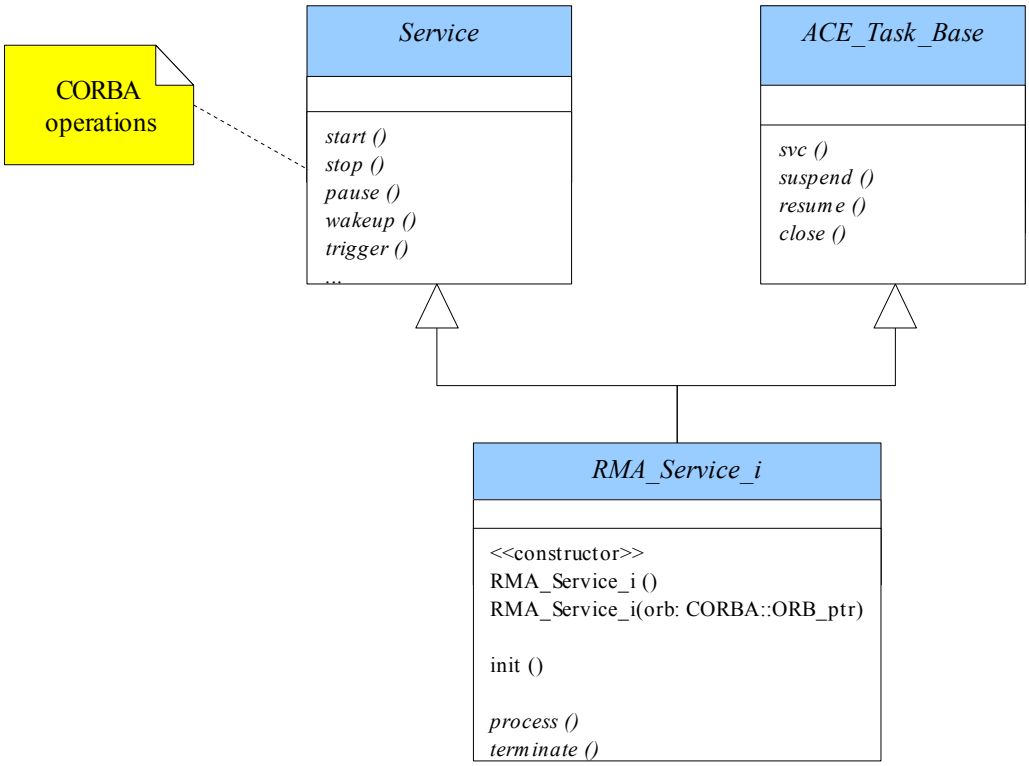


Figure 4. RMA_Service_i inheritance diagram

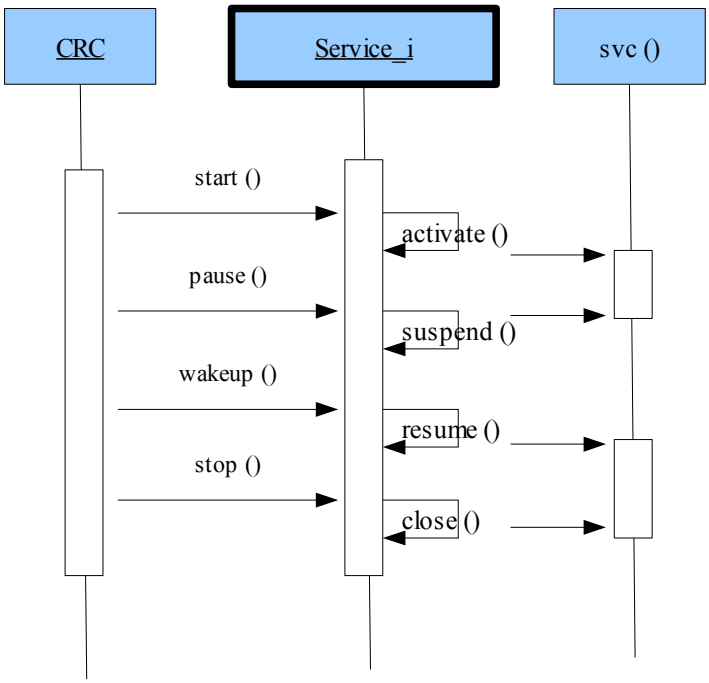


Figure 5. The interaction diagram of the RMA_Service_i class

The core functionality of all components relies on two methods: *process* and *terminate*. The process methods is called at regular interval in the loop of the *svc* method that runs in the separate thread.

The method *svc* is executed by the thread when the object is activated. The loop calls the method *process* which is declared virtual in the Service base class and must be implemented in child classes. The loop is run conditionally and flags allow pausing or stopping it. These flags are set and reset by the Service methods already mentioned. The sequence diagram of the *svc* loop is represented in the figure 6.

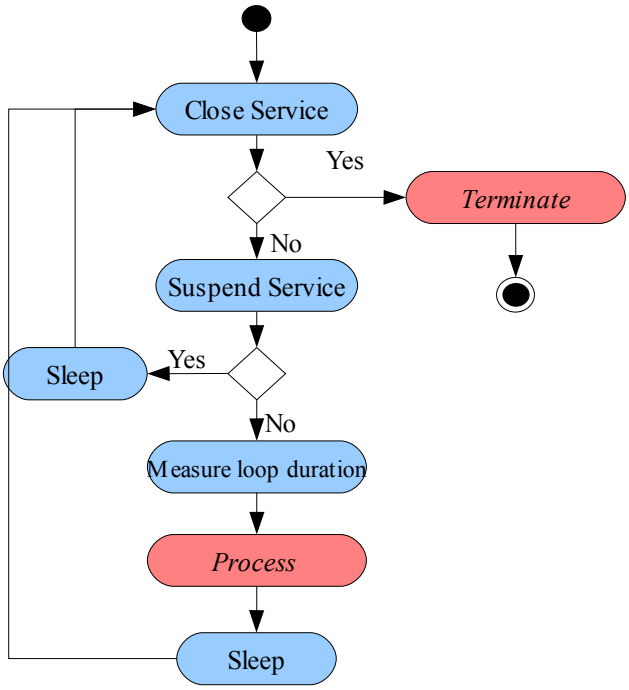


Figure 6. Sequence diagram of the svc loop

Defining the structure of an algorithm in the base class is advantageous if some of the methods are declared virtual, the concrete implementation being deferred to subclasses. This mechanism is actually known as the **Template Method Design Pattern** that has been presented in the preceding chapter in the section 2.4. It leads to an inverted control structure because the parent class calls the operations of a subclass and not the other way around.

The *svc* loop is therefore implemented in the class *RMA_Service_i* and is reused without any modification by derived classes. The two methods performing the concrete work (*process* and *terminate*) are virtual and implemented by subclasses. Because those methods are virtual the most heavily derived version will be invoked at run-time. This mechanism allows to implement an immutable algorithm structure while allowing an easy extension of its subparts.

The run cycle of the CoRoBA components is controlled by a Finite State Machine (Figure 7), which is a common way to give structure to the execution of computer tasks.
A process can be in a number of possible states, performing a particular function in each of these

states and making a transition to another state caused by an external event or internal state logic. This object state machine does guarantee that all state functions are executed atomically within the context of the state machine, state functions are properly serialised with state transitions. The state transitions are caused by the invocation of the four CoRoBA operations *start*, *pause*, *wakeup* and *stop* already mentioned.

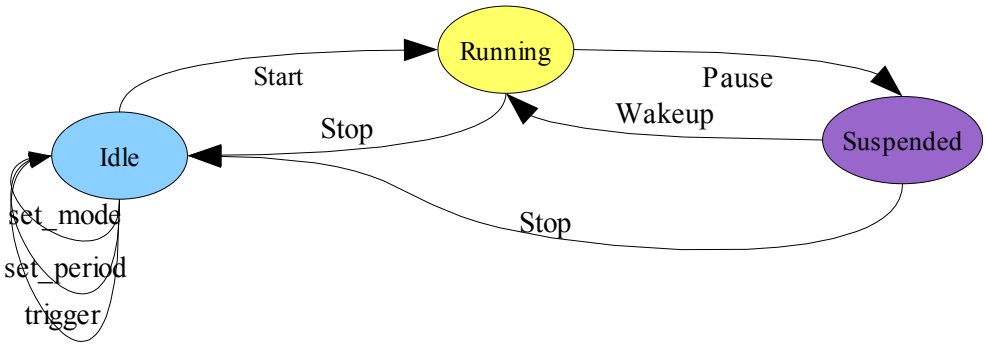


Figure 7. Finite State Machine of the CoRoBA components

3 Component categories

3.1 Definition

There are obviously many different ways for developing applications but it seems appropriate to define three fixed categories of components in order to provide a default implementation for each of them. The components composing the CoRoBA framework are split in Sensors, Processors and Actuators that form a chain along which information is transferred (Figure 8). Like in classic control schemes, the data flow is unidirectional.

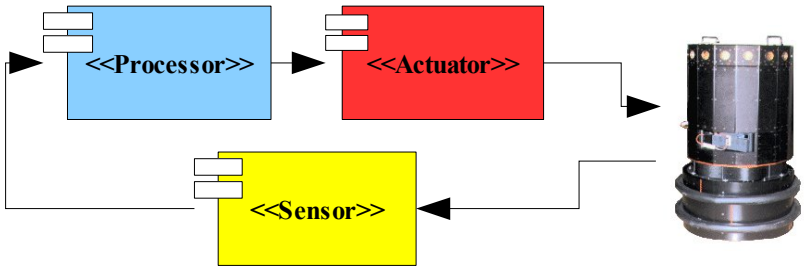


Figure 8. Closed-loop chain of Sensor, Processor and Actuator components

Sensors and Actuators make the link between the Processor components and the physical elements. Sensors are connected to physical systems and retrieve information that is forwarded to processing components. For instance, in a robotic control application, Sensor components read data from navigation and environment perception systems and transfer them to Processors that use this information to produce navigation commands. These commands are then sent to Actuators who are connected to physical output devices. Actually in the context of CoRoBA, the term Actuator regroups all components that produce data going out of the network of components. Displaying data on a screen is for instance performed by an Actuator.

3.2 Interfaces and implementation

At the second level of the interface hierarchy we find three Interfaces corresponding to the three types of components defined above (Figure 9). They inherit from the *Service* interface and consequently a component implementing a derived interface will automatically implement the base interface, allowing to invoke management operations on it.

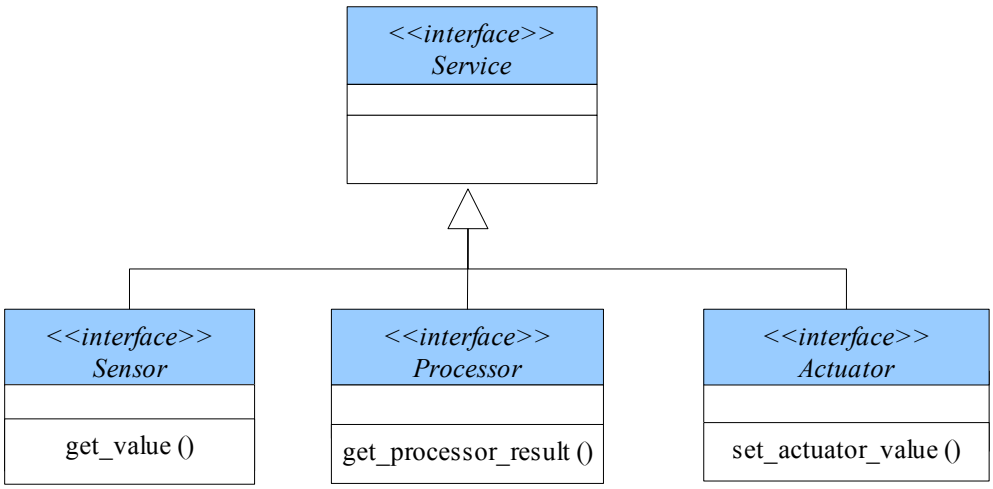


Figure 9. Sensor, Processor and Actuator interfaces inheritance diagram

The operations defined in those three interfaces are generic operations that can be used to implement default synchronous 2-way calls for components.

We distinguish classes inheritance from CORBA interface inheritance. Interface inheritance gives rise to polymorphism and has the same semantics as for C++; a derived interface can be treated as if it were a base interface. The main difference is that C++ inheritance means inheriting implementation while IDL inheritance applies only to interfaces; the implementation of inherited interfaces is completely unconstrained, that is derived interfaces are not necessarily implemented by derived classes.

The framework proposes classes implementing common operations required by the three types of components. For instance the Interface Sensor is implemented by the class *RMA_Sensor_i* (Figure 10). This class inherits from the class Sensor (automatically generated by the IDL compiler) and from the *RMA_Service_i*. The class *RMA_Sensor_i* is associated with the class *RMA_StructuredPushSupplier* that is responsible for sending data to Processor components. This class will be presented in details in section 5.3.4.

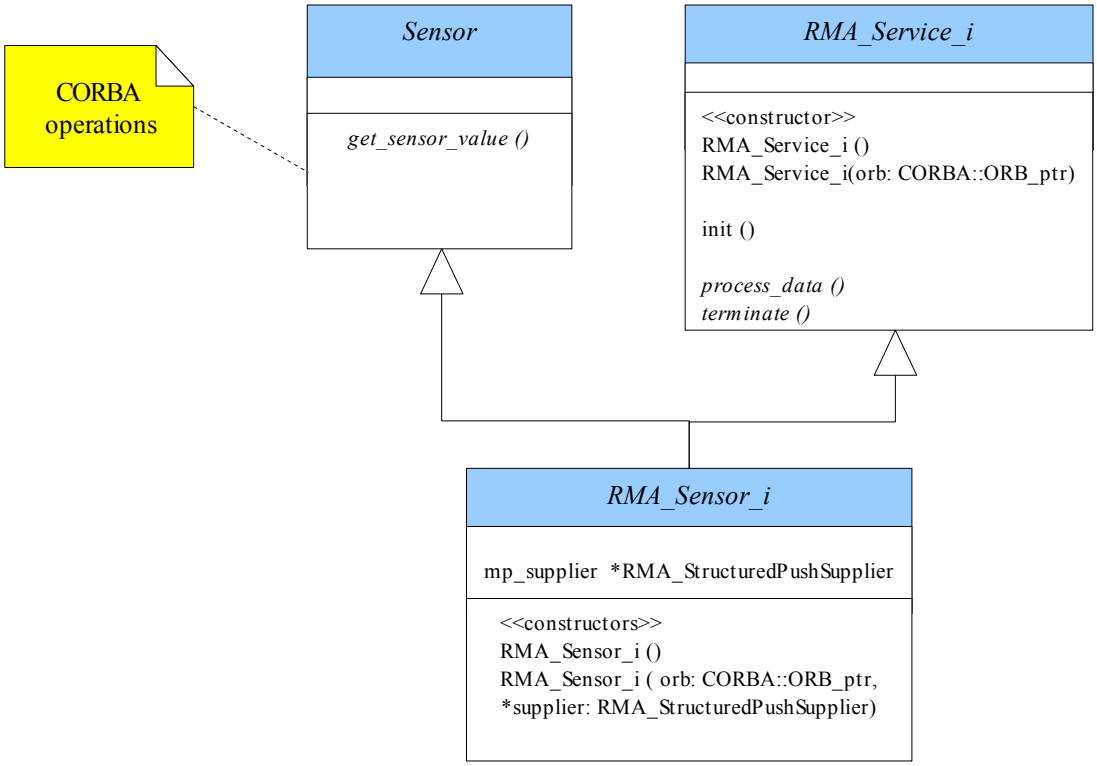


Figure 10. RMA_Sensor_i inheritance diagram

3.3 Component development

The first step for developing a new component using synchronous calls is to define the corresponding interfaces with the required data structures, the operations and the exceptions. This Interface must be derived from one of the three Interfaces presented above. Each concrete component must then implement the operation defined in this new interface.

On the other hand, the Event based scheme only requires the definition of data structures used for representing data embedded in Events. For a component using events, the developer defines the data structure in IDL; the structure of the events being directly defined in the implementation code.

We give the example of a Sensor that reads commands from a joystick. The interface definition is listed below. The data structures *MotionCommand* and *McmdSeq* are used for transmitted data in events. The *Motion_Command_Sensor* interface inherits from the interface Sensor. The interface contains two synchronous methods, *get_motion_command* and *get_number_of_axis*.

```
#include "./../CoRoBa_Sensor_Lib/RMA_Sensor.idl"
module RMA {
        typedef long MotionCommand;
        typedef sequence<MotionCommand> McmdSeq;

        interface Motion_Command_Sensor : Sensor {
                typedef short NumberOfAxis;
                MCmdSeq get_motion_command();
                NumberOfAxis get_number_of_axis();
        };
};
```

The following picture represents the class and interface inheritance diagram for the *Motion_Command_Sensor* interface
The class *RMA_Motion_Command_Sensor_i* implements this interface. It must also implements the virtual methods *process* and *terminate*.

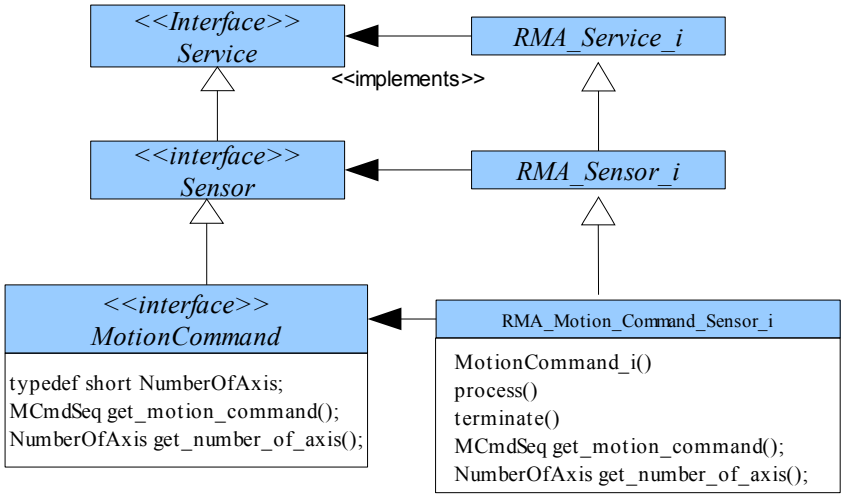


Figure 11. Interface and Class inheritance for the Sensor RMA_MotionCommand

The following code presents the implementation of the operations of this class. Exception handling code has been omitted for simplicity reasons.

```
// Constructor
RMA_Motion_Command_Sensor_i::RMA_Motion_Command_Sensor_i (...)
{
        //  Initialize variables and acquires the Joystick
 }

// Destructor
RMA_Motion_Command_Sensor_i::~RMA_Motion_Command_Sensor_i (void)
{
        // Unacquire the joystick
}

::RMA::MCmdSeq * RMA_Motion_Command_Sensor_i::get_motion_command ( )
  ACE_THROW_SPEC ((    CORBA::SystemException ))
{
        // Read joystick data and put it in the variable js
        g_pJoystick->GetDeviceState( sizeof(DIJOYSTATE2), &js ) ) )

        // Copy the data into a MCmdSeq variable
        m_MCseq.length(3);
        m_MCseq[0] = js.lX;
        m_MCseq[1] = js.lY;
        m_MCseq[2] = js.lRx;

        return  m_MCseq;
}

RMA::Motion_Command_Sensor::NumberOfAxis
RMA_Motion_Command_Sensor_i::get_number_of_axis ( )
  ACE_THROW_SPEC (( CORBA::SystemException ))
```

```
{
        read NofAxis of the joystick;
        return NofAxis;
}

void RMA_Motion_Command_Sensor_i::process(){

        // Read joystick data and put it in the variable js
         g_pJoystick->GetDeviceState( sizeof(DIJOYSTATE2), &js ) ) )

        //copy the data into a MCmdSeq variable
        m_MCseq.length(3);
        m_MCseq[0] = js.lX;
        m_MCseq[1] = js.lY;
        m_MCseq[2] = js.lRx;

        // Put the sequence in a CORBA any
        any <<= m_MCseq;

        // Send the joystick data to the Event Channel
        send_event(TYPE_JOYSTICK_SENSOR, "Joystick",++eventId, any);
}


void RMA_Motion_Command_Sensor_i::terminate(){
        // Send values corresponding to the neutral position of the Joystick
        m_MCseq.length(3);
        m_MCseq[0] = 32600;
        m_MCseq[1] = 32600;
        m_MCseq[2] = 32600;
        CORBA::Any any;
        any <<= m_MCseq;
        send_event(TYPE_JOYSTICK_SENSOR,"Joystick", ++eventId, any);

};
```

4 Communication models

4.1 Synchronous and Asynchronous communication

The communication between components is based on two different mechanisms: a synchronous communication mechanism for all management operations and an event based mechanism for exchanging application data. Event based communication has numerous advantages. It decouples data producers from data consumers; it is no longer necessary to know who will use the data and it does not matter how many clients want to receive data produced by a server. Furthermore, it reverses the data communication scheme. In classical client-server applications, the client has to constantly poll the server to check if new data is available. In a supplier—consumer mechanism, data is sent when needed. This is advantageous in control applications, where data is produced by sensors and forwarded to processors.

For operations that are rarely called and are not periodical, classical synchronous calls are lighter and simpler to use. That is why this mechanism is used in CoRoBA for management operations. Synchronous calls are also used internally for utility operations like locating and registering with CORBA Services, locating other components, creating and connecting to Event Channels, etc.

The Figure 12 shows where the data exchange happens in the component architecture. The main thread receives the management events that are used to control the service (*Svc*) thread. Events are exchanged between *Svc* threads of components.

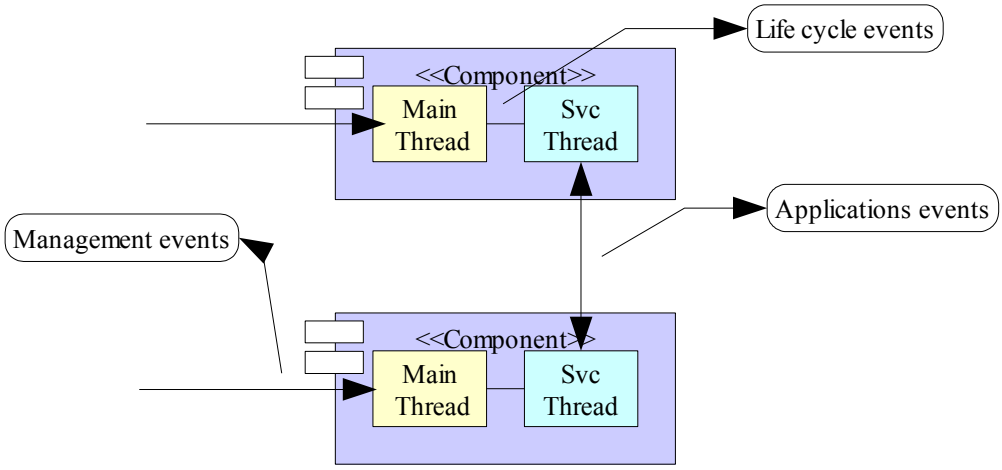


Figure 12. Component architecture

4.2 Remote management of components

As explained in section 2.2, the Service Interface uses synchronous communication and provides operations for remotely managing components. A special component called *Component Remote Control* (CRC) has been developed in order to facilitate the remote management of the live and run-cycle of the components. The data is transported between the component by the ORB mechanism as illustrated in Figure 13 and explained in Chapter 3.

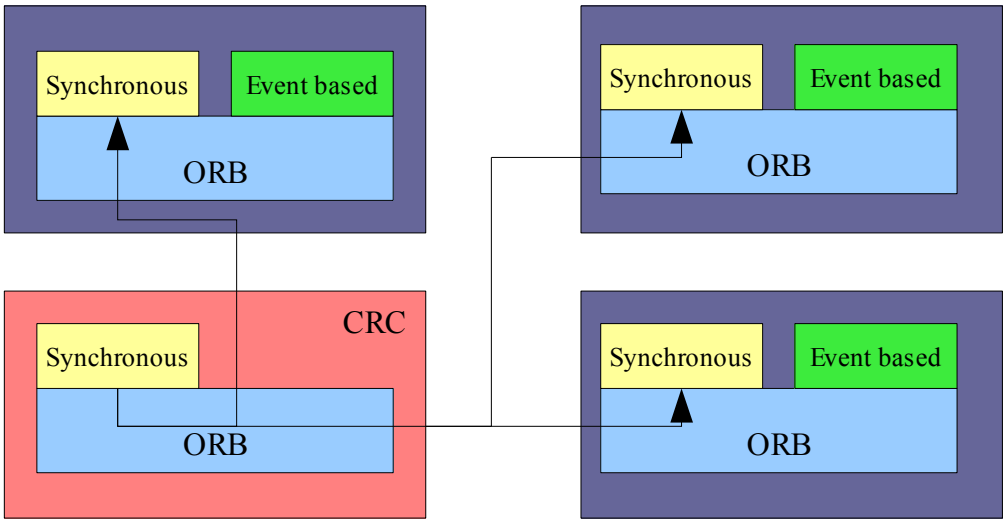


Figure 13. Service interface uses synchronous communication for management operations

The CRC component invokes operations defined by the Service Interface (see Figure 2). As these operations are implemented in a parent class that all CoRoBA components are derived from (RMA_Service_i), all newly created components deriving from this class can automatically be managed without modifying the application.

4.3 Event based communication

4.3.1 Principle

The basic communication model proposed by CORBA is a synchronous one. However, blocking calls are not the most appropriate way to communicate in control software and the Data Bus Design Pattern (Chapter III) provides a better solution. Asynchronous communication in CoRoBA is based on the Event Service and its extended and improved version, the Notification Service that implements this pattern. Components exchange data by pushing Events through Event Channels (Figure 14). These can be seen as pipes connecting suppliers and consumers of Events. Obviously several producers and consumers can share an Event Channel and several Event Channels can be used in an application in order to rationalise the communication efficiency. Asynchronous communication is one aspect of the requirement **R6** expressed in the Chapter II.

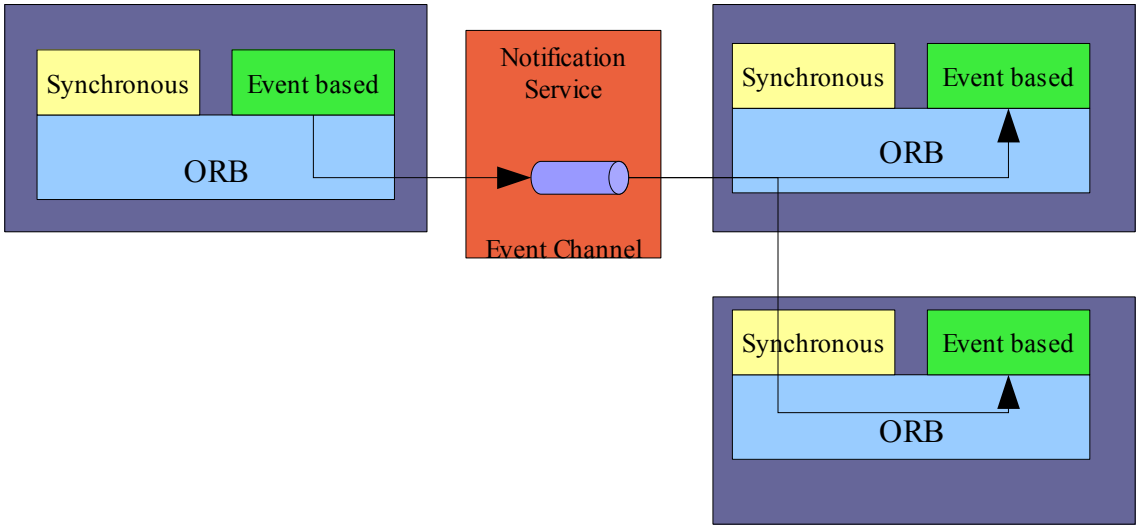


Figure 14. Event based communication between components

The event based communication mechanism is used a little bit differently by the different component types. Sensors produce Events by retrieving data from physical systems they are linked to and injecting it into the network (Figure 15). For instance, in a mobile robotic application Sensor components read data from navigation and environment perception systems.

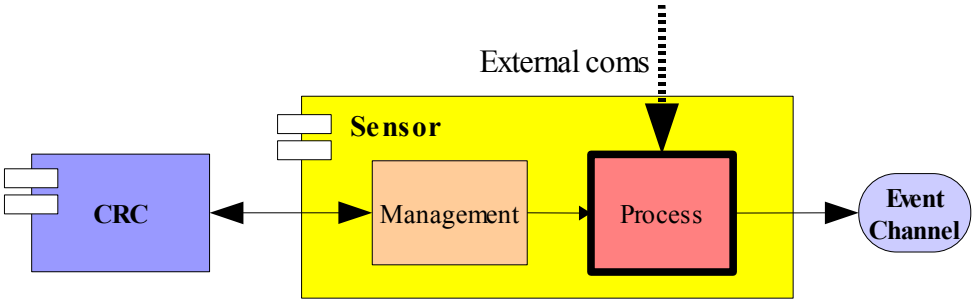


Figure 15. Sensor communication structure

An Actuator component is an Event consumer. It receives Events from one Processor component and adapts the data to the physical device it is connected to (Figure 16). It is important to mention that they only receive Events having the properties they have specified during a registration phase.

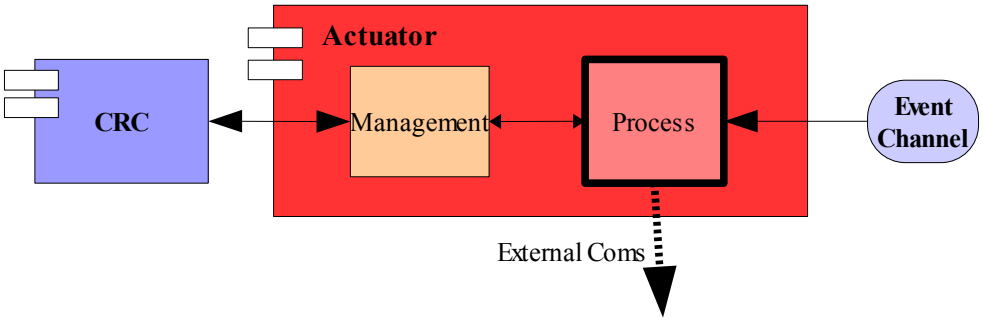


Figure 16. Actuator communication structure

Processors are at the same time consumers and suppliers of events. They perform computations by exploiting data they receive from Sensors or others Processors and produce new data that is sent as output Events (Figure 17). These components are the ones that actually process the data and generates new events, instead of merely translating it into another medium.

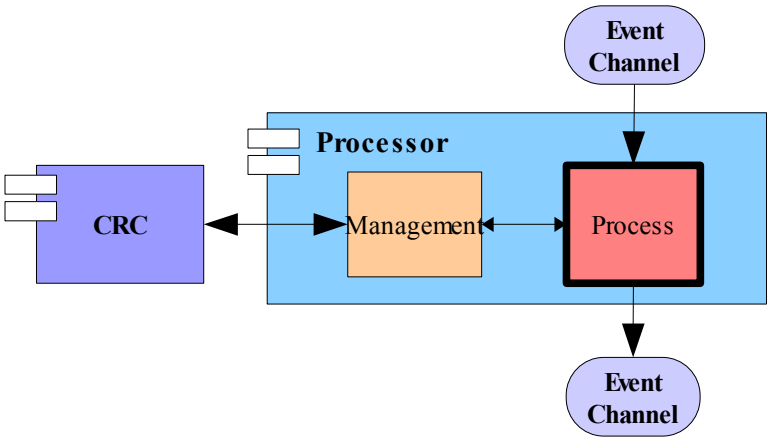


Figure 17. Processor communication structure

4.3.2 Structure of Events

Events have a standard structure that can basically be divided in identification and application fields. Each field contains an identifier and the corresponding value. The identification part is called Header and is itself divided in Fixed Header, which is present in all CORBA Events, and Variable Header, which is specific to the application. The data is composed of Filterable Data and the Remainder of Body (non filterable data) (Table 1).

Table 1: Event Structure

		Fields
Header	Fixed Header	Domain
		Type
		Name
	Variable Header	....
Data	Filterable data	....
	Remainder of body	....

In order to be able to distinguish two events of the same type generated at different time, a time stamp stored in the Variable Header field is used. Managing a global clock in a distributed application is not trivial and it is generally easier to associate each event with an identifier that is incremented for each new emitted event. On the other hand, the NotificationService guarantees that each Event is only sent once to each Consumer.

4.3.3 Definition of Events

While the contains of events is directly defined in applications, user data structures must be defined in IDL in order to be correctly encoded and decoded by CORBA libraries. Several formats, some specific to CORBA, are available for defining data structures (structure, arrays, sequence, ...).
Defining event fields (domain, type, ...) directly in applications is not a limitation because in order to write the functional code of components, a developer has obviously to know which data is needed to perform the work.

In CoRoBA, the Domain field is used to group components logically (for example all components involved in the control of a single robot) while the Type defines the type of Event, that is what kind of information it contains (*MotionCommand*, *LaserData*,...). The Name field contains the name of the component. It is possible to identify the origin of an event with the Domain and the Name fields. For instance, this is necessary if we have two identical sensors in the network or in the case where different processors would produce the same kind of events but with different algorithms or also for comparing results of redundant components.

4.3.4 Transmission of Events

Event Channels
An Event Channel is an abstract concept that is actually implemented by the *NotificationService*. Each supplier or consumer connects to an Event Channel and receives a reference to a proxy that serves as the communication endpoint for the Event Channel (Figure 18). Each event received by the Event Channel is forwarded to all Consumers that want to receive this type of Event. This is not a real broadcasting because it is based on a TCP/IP connection and consequently the server has

to successively send the same data to all Consumers connected to the Event Channel.

Creation and management of Event Channels
In order to create an Event Channel, an object has to contact the *NotificationService* via the *NotificationFactory* object. This object proposes operations for creating and managing Event Channels. All available operations on Event Channels are presented in appendix C.

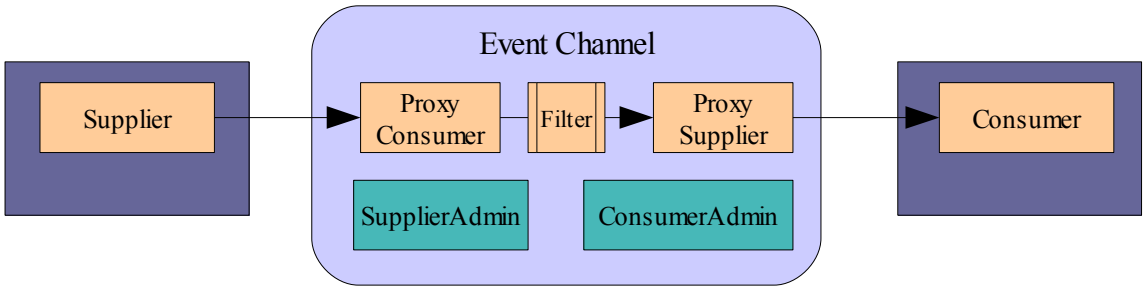


Figure 18. Structure of a CORBA Event Channel

Emission of events
Events Suppliers (Sensors and Processors) send events to Event Channels. Each Sensor is associated with a supplier object of type *RMA_StructuredPushSupplier Object* that is responsible for sending data to the Event Channel. The Figure 19 shows the inheritance diagram of this class.

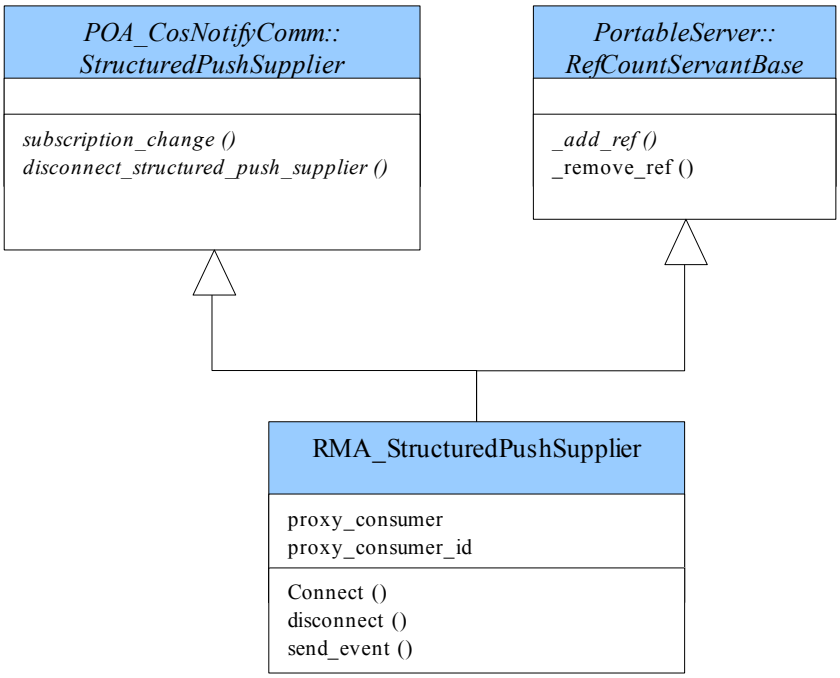


Figure 19. RMA_StructuredPushSupplier class inheritance diagram

The method *send_event* is responsible for pushing events to the Event Channel. It is a method that is called by the method *process* in the *svc* loop (see Figure 6). This method is called when the *trigger* operation is invoked (TRIGGER mode – see section 6) or at each iteration of the loop of the *svc* method that runs in a separate thread (PERIODIC mode). This case is illustrated by the Figure 20. The event is then forwarded to the Event Channel via the Proxy Consumer (*StructureProxyPushConsumer_var*).

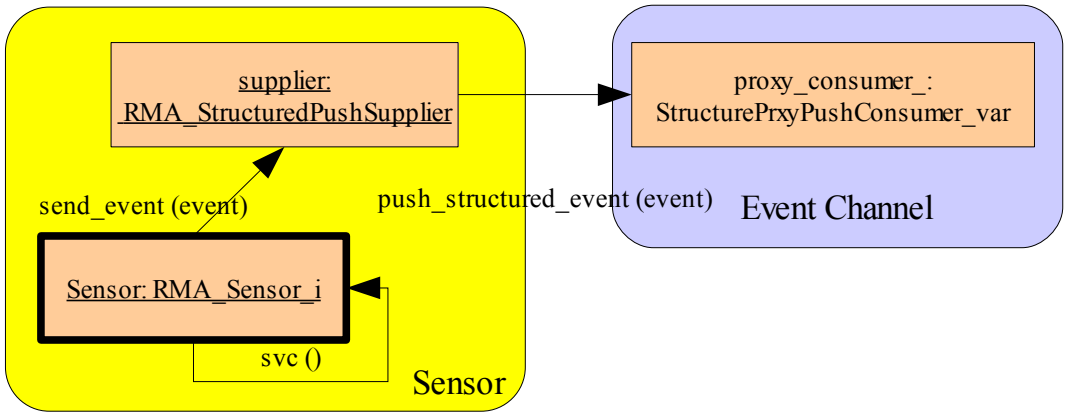


Figure 20. Object interaction diagram for the event emission

Event registration
Each Event Channel is associated with two Administration interfaces (One for the consumers and one for the suppliers) that contain operations allowing to configure it. Consumers can select which events they want to receive by registering with the Administration Interface of the Event Channel and specifying the data that Header fields (Domain, Type) must contain. This is an improvement in comparison with the original CORBA Event Service where all incoming events were forwarded to all connected consumers. In CoRoBA Processors and Actuators contain a consumer object. Sensors have only a supplier object while Processors have both because they receive events from Sensors and send Event to Actuators.

Reception of Events
Events are transferred by the transport mechanism of CORBA to the component where they are identified (events of different types can be received) and stored in member variables of the correct type for further processing. This transfer happens between different thread and is thus protected by mutexes. It is further explained below in the case of Actuators.

An event is sent by the proxy_supplier of the Event Channel to the consumer object in the Actuator component. From there, it is transferred to the RMA_Actuator_i object that stores the event in its notification member (Figure 21). The data is actually extracted from this variable and sent to the physical actuator by the method *process*. As these actions happen in different threads (events are received in the ORB thread and pass to the *svc* thread) synchronization mechanisms are required.

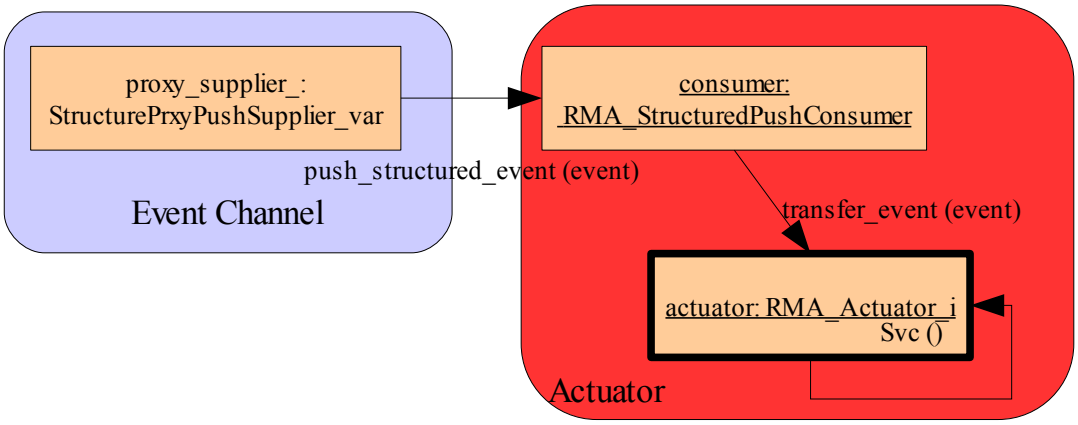


Figure 21. Object interaction diagram for the event reception process

Threads are associated with a stereotype of objects, called "active" objects. Generally an active object is created for each thread. The "passive" objects are then added to the "active" objects via the composition relation. The role of the "active" object is to run when appropriate and call or delegate actions to the passive objects that it owns. The passive objects execute in the thread of their active owner. Threads are usually not independent and must therefore coordinate, synchronise and share information. Concurrency patterns like the **Message Queuing Pattern** presented in Chapter III, provide solution to avoid corruption and erroneous computation when information is shared [DOUG03].

In the current implementation, we consider that Events produced by components are disposable in the sense that we may lose some of them without jeopardising the system stability. Only the most recent event of each type is actually stored and is overwritten by a new incoming one. Consequently, queues are 1 data length buffers for each type of events received by a component. It could be possible to extend the length of the buffers if several consecutive events are required by the algorithms implemented by the components.

Transfer of Events

A processor component combines the characteristics of an actuator and of a sensor. It receives data as events via the consumer object, transfers it to the RMA_Processor_i instance via the transfer_event method where it is stored in the notification member variable. The data is exploited in the *process* method and the result is passed to the supplier object that sends the new event to the proxy_consumer object of the output Event Channel. This process is illustrated by Figure 22.

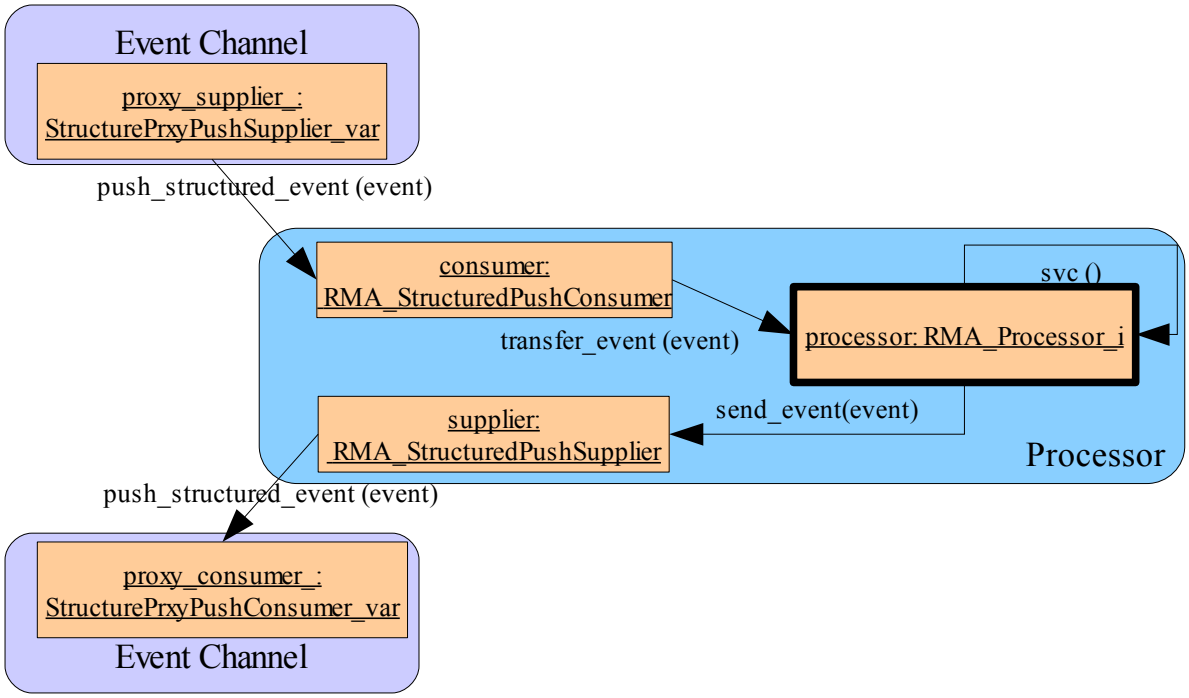


Figure 22. Object interaction diagram for the event processing

5 Running modes

Three different running modes have been defined for the transmission of events: PERIODIC, SYNCHRO and TRIGGER.

- PERIODIC means that components produce events at regular time intervals.
- In SYNCHRO mode, new output events are produced by the component when an event is received.
- In TRIGGER mode, an external component must invoke the *trigger* method that will itself call the processing method that produces output events.

These different modes have been defined and implemented in order to be as exhaustive as possible. Each mode is actually useful in a different context.

The table below summaries what it happens for each component type in the different modes.

Table 2: Actions performed in the 3 mode

	Sensor	**Processor**	**Actuator**
SYNCHRO	Push an event each time new sensor values are available	Process data and push events each time a new event is received	Sends data to an external system each time a new event is received
PERIODIC	Reads the sensor values and push events at periodic intervals	Process data and pushes event at periodic intervals	Sends data periodically to an external system
TRIGGER	Reads sensor values and push events when externally triggered	Process data and push events when externally triggered	Sends data to an external system when externally triggered

5.1 Sensors

Sensors are generally proactive components that needs to read data from external devices. Periodic and Trigger mode are the two modes that are generally used by such components.

5.2 Processors

Processors can be used in all of the three modes. In the Synchro mode, processors produce new data when an input event is received. When a component receives data from different sources, a synchronisation problem can occur. As events arrive asynchronously, different reactions are possible:

- New data is produced each time an event is received.
- New data is produced when all data have been updated.
- A local or a global clock is used to trigger event production.

These different situations are illustrated by Figure 23 where 3 different types of events (A, B and C) are received by a Processor. Output events are represented for the three possible aforementioned reactions (denoted Each, All and Clock). In the first case, an event is produced each time the component receives a new event, no matter its type. In the second case, an output event is sent once a new event of each type has been received. In the third case outputs events are sent at regular intervals.
Output events are labelled according to the number of input events they correspond. Ex: the output event **A1B1C1** has been produced with the data coming from input events A1, B1 and C1. We see that output events are constituted by different combinations of input events in function of the chosen reaction.

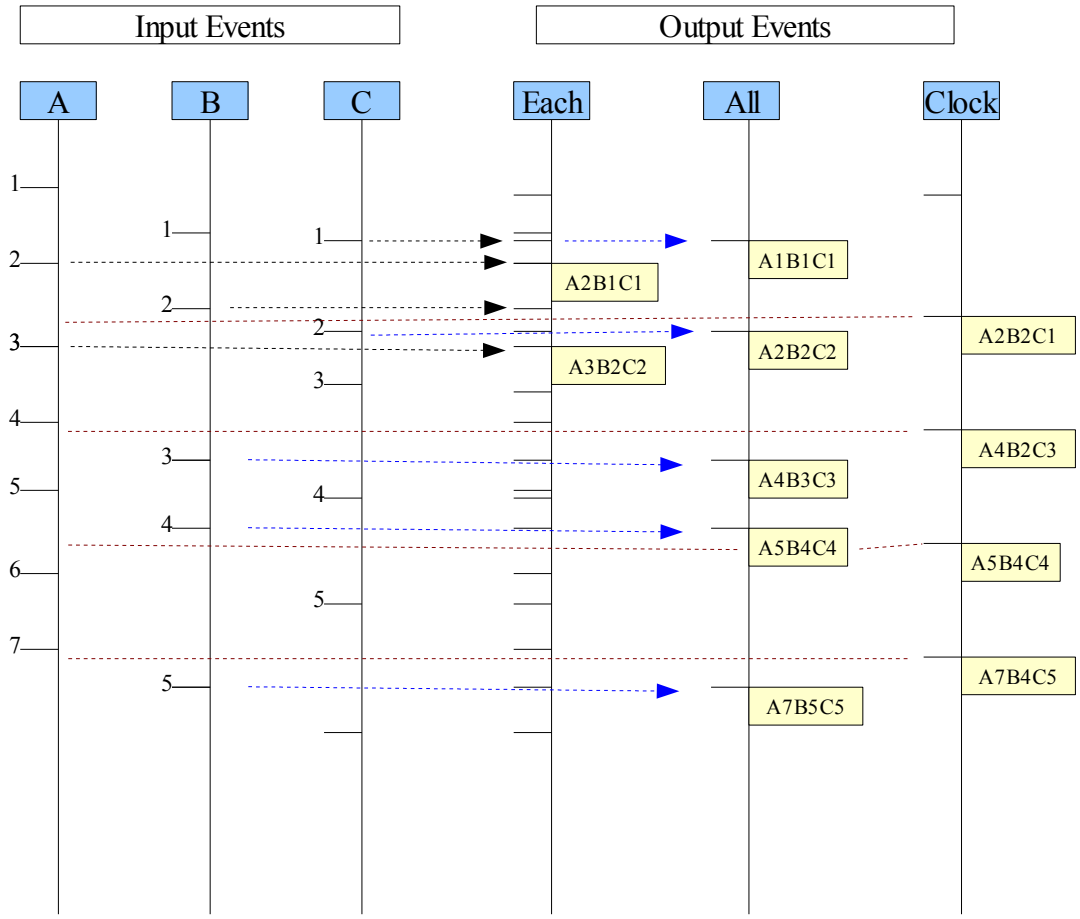


Figure 23: Output events produced as a function of the reaction strategy.

Rem. In the Figure 23 not all output events have been labelled.

The first solution could lead to a large amount of data if the period of received events is short or many different events are received. The advantage is that all incoming events are processed. This works well if the processing period is shorter than the mean period of events arrival.

The second case works well when all events have similar production periods. If events of one type are not periodic, some events of another type could be ignored because we wait that new occurrences of all events have been received before processing them and transmitting an output event. This situation is due to the fact that the data processing is combined with the event transmission. In order to avoid this situation, the data processing should be decoupled from the event transmission. This is not the case in the current implementation.
The choice between the two reactions is left to the programmer. They correspond to the SYNCHRO mode. A processor receiving many events (of the same of different type) could be overwhelmed with the quantity of data. One solution is to decrease the emission frequency of the incoming events or to select the PERIODIC mode. The period must off course be so that the data processing can complete within the selected period.

The third case corresponds to the PERIODIC mode in the case of a local clock or to the TRIGGER mode if we use a global clock.

5.3 Actuators

Actuator components can also work in all modes. The reason for selecting the PERIODIC mode would be to provide the external device with data at regular intervals so it can use it as a security check, stopping the machine when the communication is broken. If this function is not used, the SYNCHRO mode generally consumes less resources and is a good choice.

By default all services are in the PERIODIC mode with a period of 50 msec. By calling the *set_mode* operation, the service manager component (CRC) can modify the period and the operation mode. The operation mode can only be changed when the component is in *idle* state (Figure 7) otherwise a user defined CORBA exception is thrown. An exception is also thrown if the service manager component tries to set the period and the mode is not PERIODIC or if an unknown mode is sent to a component. If the component is not in the TRIGGER mode, invoking the CORBA *trigger* operation also results in an exception.

6 Monitoring and logging

6.1 Monitoring

It is possible to check, add or delete components registered with the NameService with some utility program coming with CORBA implementations, as for instance the application *nmg* that is written in Java and comes with JacORB. The CRC application can also be used to check if components are still alive. This does not however imply that the component is still working correctly. As a general rule, and in order to improve reliability, each component has to check the validity of the data it receives.

In case of problem with one component other components are not affected but off course the application will not work any more. The user can then stop the faulty component with the CRC. If it is a hardware problem the component on the faulty computer could be restarted on another machine.

Detecting failures in asynchronous communications is difficult to implement. However, time-outs can be used to detect communication breakdowns or problems with other components if events are received at periodical intervals.

6.2 Logging

The logging mechanism in CoRoBA relies on the *Telecom Logging Service* (TLS) of CORBA. A central Logging server manages as many logging databases as required to store the information sent by the components. Each component creates its own logging facility through the *LoggingFactory* that is registered with the *NameService*. The logging facility can be used for debugging or for off-line processing of data (visualisation, learning, ...).

There exist two types of Log, *BasicLog* and *NotifyLog*. *BasicLog* allows “event unaware” clients to access a log directly without any knowledge of events. *NotifyLog* is a consumer and a producer of events. It supports filtering on incoming, logged an outgoing events (Figure 24).

The Logging Service stores the received information in a standard format defined in the TLS specifications (Table 3).

Table 3: Log Record Format

Field	Description
id	Unique number assigned to the record by the log
time	Time stamp indicating the time the event is logged
attr_list	User defined name/value pairs
info	The event stored in a CORBA any[17]

All events passing via the Event Channel is recorded. A client can also log information that is not event based. The client can query the Log to extract recorded data.

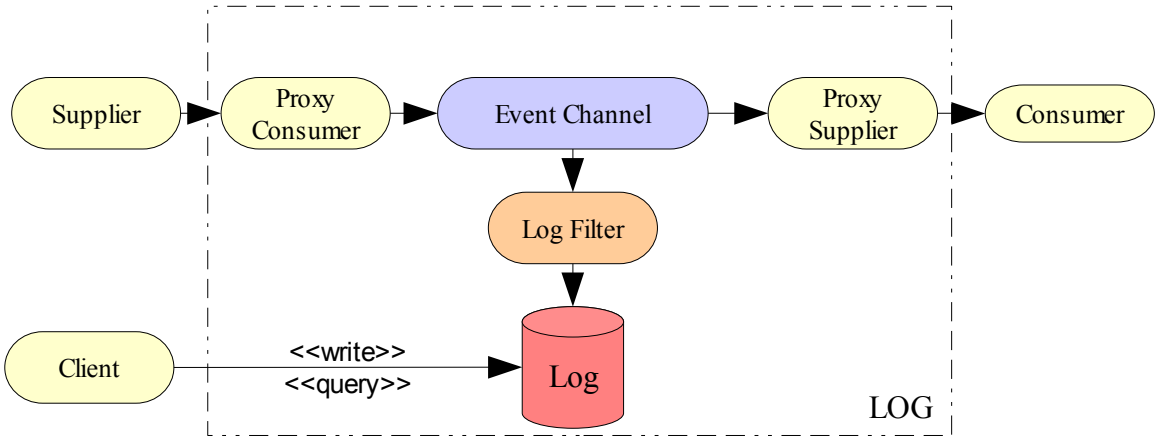


Figure 24. Log structure

Each component is associated with its own log. All events are logged after they have been sent to the event channel. In order to trace events along the components network, some extra information are added to the events.
We have seen that each event contains a header and a data section (Table 1). Besides a fixed header defined by the CORBA specifications, the developer can add arbitrary name-value pairs to the header.
As this header has a variable length, it can be extended each time the event is processed by a component. At the end we obtain an event containing all the processing chain information.
This capability is used in order to add extra information to the events when they moved from component to component. At each step an Id generated by the component is added to the event as well as a time stamp corresponding to the emission time (Table 4). The use of this mechanism will be illustrated in Chapter VI.

17 *any* is a type that can contain any legal IDL type at runtime.

Table 4: Extended Event Format

Header	Fixed Header	Domain	
		Type	
		Name	
	Variable Header (1)	"Id"	Id value
		"TimeStamp"	timestamp
	Variable Header (2)	"Id"	Id value
		"TimeStamp"	timestamp
	...		
Data	Filterable data	....	
	Remainder of body	....	

Actuators also log information but in a slightly different form. They replace the time stamps by the time intervals between the emission an reception of events.

A utility program has been developed to retrieve and store in files the logged information.

7 Location of Components

7.1 Interoperable Name Service

The CORBA specification describes the Interoperable Name Service that contains several procedure for locating CORBA objects. Interoperable Object References (IOR) can be passed on the command line or read from a file but in order to implement a true and versatile distributed system, the NameService is the best choice.

The CORBA NameService provides:

- An implementation of the Object Management Group (OMG) Interoperable Name Service (INS) specification.
- Application programming interfaces (APIs) for mapping object references into an hierarchical naming structure (referred to as a namespace).
- Commands for displaying bindings and for binding and unbinding naming context objects and application objects into the namespace.

The Name Service object provides access to a CORBA Name Service which allows CORBA server applications to advertise object references using logical names. CORBA client applications can then locate an object by asking the CORBA Name Service to look up the name. When a component registers with the NameService, its identity and location, which are encoded in the IOR, are automatically registered in a database that has a graph structure (Figure 25). One component can consequently easily discover and use services offered by other components. What is important to note is that the location is not known in advance by other components but discovered at run time. It is thus possible to transparently move components participating in an application. This is one of the big advantage of using CORBA, the migration of components is transparent to the other ones.

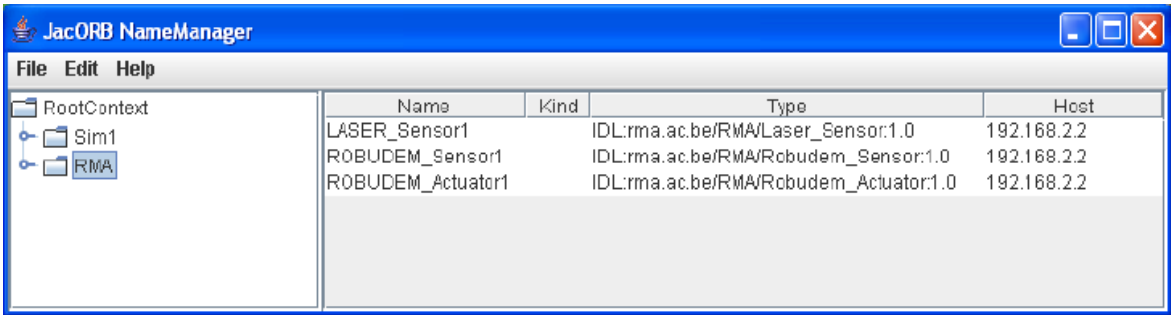


Figure 25. Components registered with the NameService

7.2 Locating Services

The Figure 26 illustrates a typical utilisation sequence of the NameService. The NameService has to be started first, then the server and finally the client. The operations required are numbered and must all complete before a client can invoke an operation on a server.

The server has first to locate (1) the NameService by calling the method *resolve_Initial_Reference("NameService").* This initial location is done with a bootstrapping method,; the location of the NameService can be passed on the command line or read from a file or by using a multicast call or, contacting a web server.

Secondly, the server binds (2) the name of the object with the IOR in the database of the NameService. After that the server waits for incoming calls.

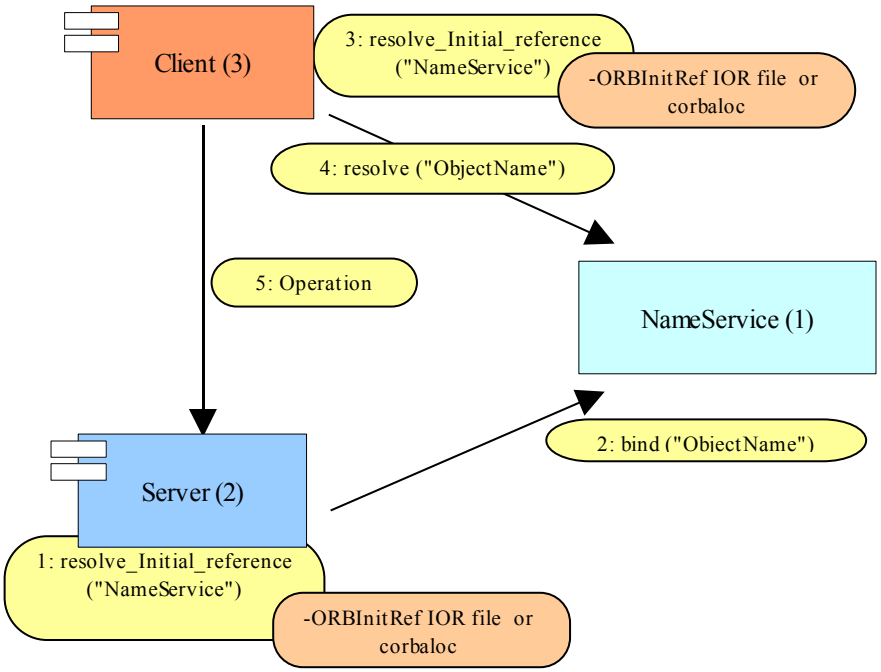


Figure 26. Using The NameService

After having located the NameService (3), the client has to retrieve (4) the IOR of the CORBA object he wants to use by using the *resolve* method with as parameter the object name (that must be known by the client). Once the client has received the IOR and verified that it corresponds to the object interface he wants to use (this operation is called "narrowing"), it can invoke operations (5) on the distant object implemented by the server.

In the CoRoBA Event based communication (Figure 27), The NotificationService has first to register with the NameService (1). When components are started they locate the NameService (2) in order to get the reference of the NotificationService and resolve it (3). Once this last operation is completed, components communicate with the NotificationService and never directly with other components.

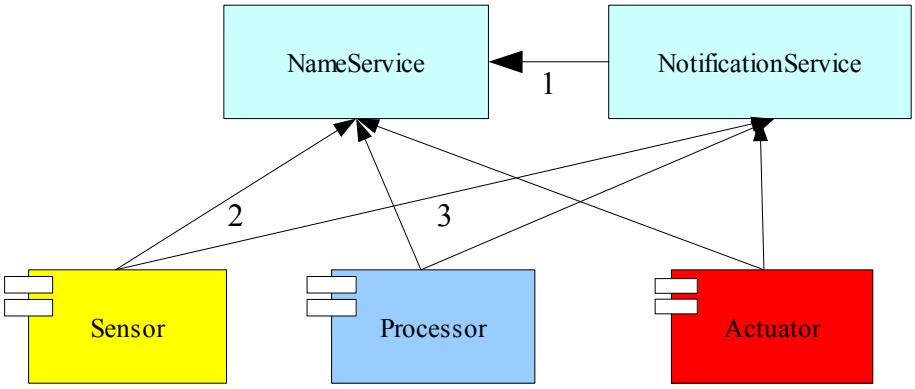


Figure 27. NameService and NotificationService location phase

The Telecom Logging Service is located the same way.

8 Objects creation and initialization

The general structure of an application with the creation of a typical Processor component by a main program is briefly presented. The *main* function of each component creates a concrete factory object derived from the abstract *RMA_Processor_Factory*. This factory creates and returns a pointer to a concrete implementation object (*RMA_Processor_i*). This pointer is then passed as argument to the *RMA_Processor_Object* constructor. Such a class exists for each component category. Only the *RMA_Processor_Object* is presented hereafter.

RMA_Processor_Object is a utility class that is used by all Processor components and whose purpose is to create the CORBA utility objects (ORB, POA, policies) and to implement utility functions, namely NameService, NotificationService and LoggingService location and narrowing. Specialized versions of this class exist for Sensor and Actuator components.
The *register_event* method is then called for each event the component wants to receive. The *create*_consumer method creates the consumer objects. Finally, *run* simply calls the *run* method of the ORB object.
The sequence of operations executed by this main object is described below and illustrated by Figure 28.

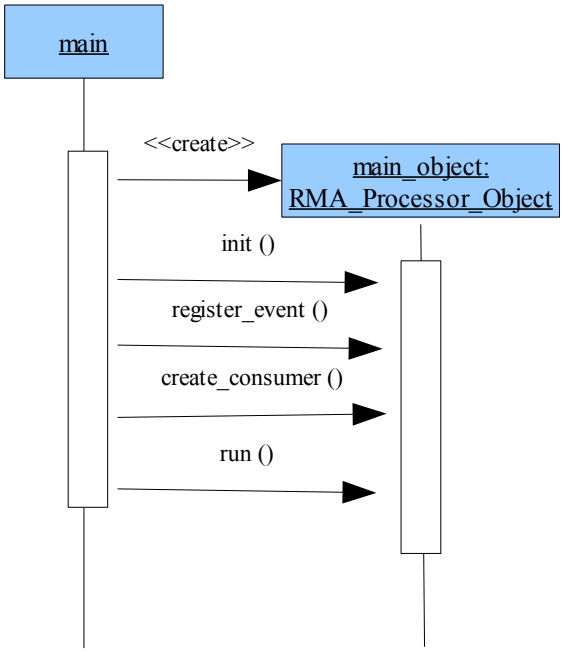


Figure 28. Object creation and initialization sequence for the main process

9 Summary

The proposed model is a fairly coarse-grained partitioning one as defined in section 1.2 of Chapter III. It is decomposed into a set of subsystems providing well-defined interfaces. Interaction between the components is obvious thanks to the use of the Interface Definition Language. The distributed system is partitioned so as to support evolution. There is also a clear separation of the component functionality thanks to the Sensor-Processor-Actuator decomposition model.

The principles stated in Chapter III have been respected:

- Interfaces are cohesive, they support a single concept (sensor, processor or actuator).
- The only coupling between interfaces are limited to the exchanged data.
- Exceptions have been defined.
- Polymorphism has been extensively used in interface and class inheritance.

The three abstract implementation interfaces could be fused to form a single interface whose functionality could be selected at run-time. However, the existence of the three interfaces is not only justified by semantics reasons but serves to identify components.

We show in this chapter that the implementation of the Design Patterns helps fulfilling the requirements listed in Chapter II. The use of Design Patterns facilitates the developments and limits the amount of code programmers have to write when implementing components with this framework.

Chapter V Simulation

1 Introduction

Having a simulator offers many advantages. First of all it is tremendously cheaper than real robots and sensors, particularly when experimenting with multi robots systems. It allows focusing on intelligence and control and disposing of other, less interesting problems. It makes possible reducing the development time by trying different scenarios and algorithms before experimenting them in a real environment. A simulator also increases safety when developing and testing new control applications. Developing a simulator can be easier or harder than building (or buying) hardware. For instance, simulating a high fidelity stereo vision system would require a lot of work and could cost much more money than buying equivalent hardware.

As no free simulator with CORBA interface was available, the only solution has been to implement a new one. While realistic models have been used to represent mobile objects, the simulator has not been developed as an engineering software but more as a tool for testing and validating the proposed CoRoBA framework.

Simulating a physical process can be typically decomposed in three steps: modelling, resolution of equations and visualization of the results. These steps can be mixed in a global application or implemented in different programs.

Different approaches are possible for the modelling step. In some implementations the user has to write equations representing the dynamical behaviour of the simulated system or to draw a 3D model including physical properties, this model being also used for the visualization of the results. In the former solution a separate 3D model has to be provided and the visualization is generally handled by a separate animation application. Commercial software's are generic tools and must consequently be versatile and provide easy to use interfaces for model creation and results visualization.
For instance, in the commercial software "Universal Mechanism"[18] (UM) the user draws the model and defines the constraints in a program called "UM Input" while the simulation and the visualization are provided by the "UM simulation" application. UM must be combined with Matlab® if the multi-body simulation has to be embedded in a global control scheme.
In the simulation library EasyDyn [VERL05] the user has to provide the position equations and the applied forces. Accompanying tools automate the creation of the motion equations and generate a C++ program that the user has to complete with additional control equations. Results are saved in files that can be read by third party applications (GNUPlot, EasyAnim, ...).
The Open Design Engine (ODE)[19] is a library that proposes a mixed approach. The user writes a program (in C++ or Python) that describes the simulated system by using objects provided by the library (world, nodes, joints, forces, torques, ...). This library also provides methods for resolving the implicitly generated equations. The visualization part is the responsibility of the developer who has to use third party libraries like Opengl.

18 Http://www.umlab.ru
19 Http://www.ode.org

Simulators can be divided in off-line and on-line simulators. Off-line simulators compute motion of objects at their own pace and produce data that can be visualized as a movie once the simulation is completed. The aforementioned examples enter in the off-line simulation category. On-line simulators are interactive; the motion of the objects can be modified in real-time by control algorithms or by a user via a GUI or a joystick. The motion of the controlled object are visualized in real-time in 2D or 3D.

The Java based simulator, called MoRoS3D, that has been developed in this work enters in the on-line category. Control commands and environmental conditions can be changed interactively. Furthermore, it runs in real time using any available communication systems and replaces the real hardware in the application control loop in order to test the control components (Processors).

The utilisation philosophy is to develop and tune control algorithms in simulation and to simply replace simulated by real components once satisfying results have been reached, no further modification of the Processor components being required. In Figure 1, the concept of integrating MoRoS3D in the CoRoBA framework is shown. Sensor and Actuator components developed with CoRoBA can be seen as interface components that have to be specific for the simulator or the hardware they are linked to.

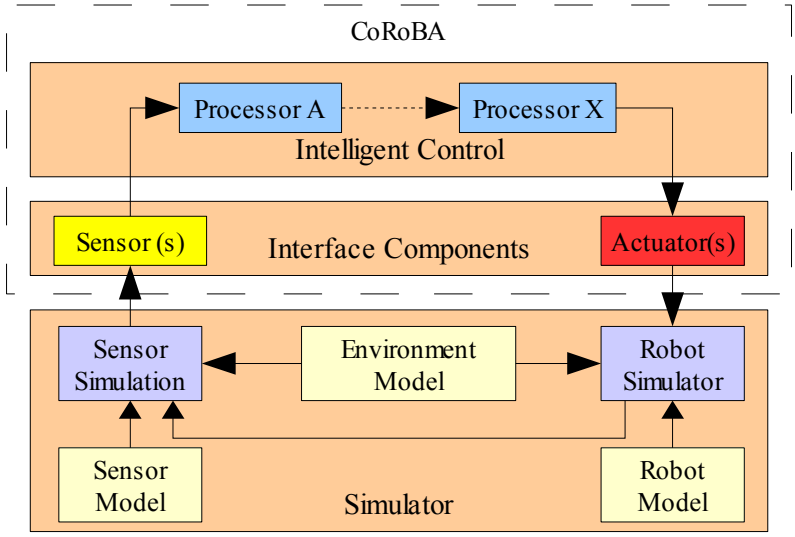


Figure 1. Simulator and CoRoBA integration

The block named "Intelligent Control" on top of Figure contains Processors. This part does not care if real or simulated hardware is used. The Processor components are the key-stone of the control architecture and exhibit the largest potential of reuse between applications involving different robots while Sensors and Actuators, that serve as interfaces or translators between the software and external modules, are specific to these devices. The more abstract a Processor is, the greater the possibility of reusing it without any modification.

The middle block corresponds to interface components that make the link between the Processors and the simulated world. Sensor and Actuator components implement the same interfaces as those implemented by components linked to physical systems, allowing to instantaneously switch between simulation and reality.

The last block represents the simulator. It is constituted by different elements that are described hereafter. First of all it contains models of the physical elements. The robot model deals with the geometric, kinematic and dynamic aspects of the robot. The sensor model encodes information about the sensors like the radiation model, the minimum and maximum distances, the precision, etc. The environment model contains the 3D geometrical representation of the environment. The robot simulator is responsible for the realistic motion of the robot and takes care of the collision with fixed and moving obstacles like other robots. It receives motion commands from Actuator components.

The simulated sensors produce measurement data that are injected in the application control loop by the Sensor components as explained in the section 4.3.4 of Chapter IV. The data is forwarded to Processor components where they are exploited to finally produce motion commands that are sent to the Actuator Components. These Actuator Components adapt and send this information to the robot objects. The sensors affect the vehicles motion through Intelligent Control and vehicles motion affect sensors through the Simulator taking into account the model of the environment.

After this introduction, the next section presents the simulator in detail. Section 3 describes applications that have been developed with this simulator.

2. Simulator Overview

2.1 Functionality

For the user, the visible output of the simulator is a synthetic image. Actually, it is not only an image but it is also a model that is built with algorithms based on physical laws and using well defined data structures.

The simulator provides the following functionalities:

- Real-time simulation of multiple robots concurrently
- 3D real-time visualization of the simulation
- User interaction through a GUI
- Dynamic control of mobile robots
- Detection of and appropriate reaction to collisions between mobile and fixed objects
- Simulation of position and distance sensors
- Integration with the CoRoBA framework

The simulation process is divided in two main steps: the modelling of 3D scenes and robots by a human and the utilisation of the modelled objects in the simulator. These two steps are explained in the following sections and illustrated by the diagram of Figure 2.

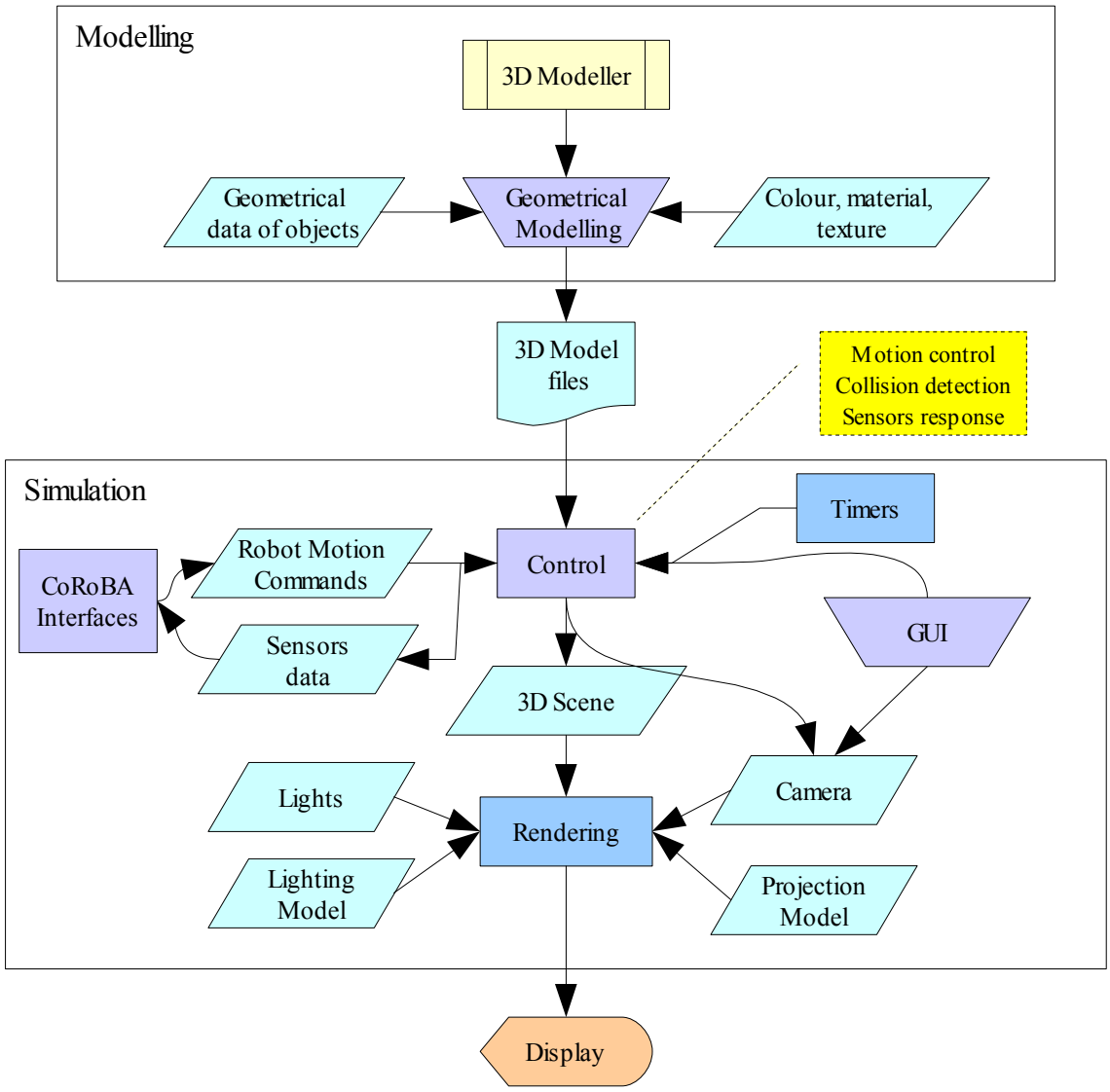


Figure 2. Simulation process

2.2 Scene modelling

Starting from a real or an hypothetical robot, the creator uses a 3D drawing program to generate a virtual model. Other information like colours, material and texture can be applied to the objects to improve the realism. Real or imaginary environments (terrain and obstacles) are created separately from the robots.

There exist different models for representing 3D objects: wireframe, surfaces or solids. In our case a surface representation is created with a surface modeller application, Wings3D[20]. The model is formed by geometrical primitives that can be transformed to obtain the desired shape. It is possible to manipulate the whole primitives, the surfaces, the edges and the points constituting the

20 Http://www.wings3d.com

model. Besides classical transformation operations like rotation, translation, scaling,... more advanced operations are available like extrude, bevel, twist, torque, etc.

Wings3D uses its own format for internal and external representation but models can also be exported in other popular formats. The exported model is encoded in VRML (Virtual Reality Modelling Language). VRML is a language that has been developed for describing 3D virtual environments for Web based applications. The 3D objects are organised as a tree and are described with low level VRML nodes. In Table 1 a VRML example of a scene representation is given. Each object is represented by a named *Transform* node that has a single child. This child is defined by its *Shape* containing appearance and geometry information. The appearance defines the used material, the colour, transparency and shininess properties of the object. The geometry is represented by an *IndexedFaceSet* node that contains coordinates of vertices. Faces are defined by references to these coordinates in a *coordIndex* node. The same way, colours can be defined for each vertex.

2.3 Simulation process

The VRML file is read by the application and transformed by the Java3D [WALS02] import library into a Java3D scene graph and inserted in the global 3D scene.

The process flow (control-rendering-display) represented in Figure 2 continuously runs until the application finishes. It is briefly described here; an extended presentation is given in section 9. The control process updates the Scene (section 4), controls the motion of the robots (section 5), performs the collision detection and response (section 6) and finally computes the output of position and distance sensors (section 7). The Control also receives motion commands for the robots and sends sensors' data via the CoRoBA interfaces (section 8). It can also control the camera motion in automatic tracking mode. The GUI (section 3) lets the user chose the camera mode and position and gives the possibility to position the robot in the virtual world.

The execution of the control process is triggered by timer events. As each robot and sensor is represented by separate objects, the events are propagated to all of them. This means that all motion and measurements are synchronized.

Once all transformations of the 3D scene have been performed, the scene is rendered by the Java3D rendering engine. This engine uses different information in order to produce an image that can be displayed on the screen:

- The lights present in the scene (ambient, directional, ...).
- The lightning model. Here a Gouraud shading is used for calculating the illumination of the scene.
- The point of view given by the camera position and other viewing information (field of view, near and far clipping distances, ...).
- The projection model, which is a perspective projection in our case.

The rendering engine of Java3D can use the DirectX or OpenGL libraries.

Table 1: VRML scene representation

```
DEF cylinder1 Transform {
 children [
  Shape {
   appearance Appearance {
    material DEF default Material {
     diffuseColor 1.0 1.0 1.0
     emissiveColor 0.0 0.0 0.0
     specularColor 1.0 1.0 1.0
     ambientIntensity 1.0
     transparency 0.0
     shininess 1.0
    }
   }
   geometry IndexedFaceSet {
    colorPerVertex TRUE
    coord Coordinate { point [
     0.340617 0.403290 5.25329e-2,
     ...
     0.294284 0.403290 -4.53464e-2]
    }
    coordIndex [
     0, 3, 2, 1, -1,
     ...
     1, 5, 4, 0, -1
    ]
    color Color { color [
     1.00000 1.00000 1.00000]
    }
    colorIndex [
     0, 0, 0, 0, -1,
     ...
     0, 0, 0, 0, -1
    ]
   }
  }
 ]
}
```

3 Graphical User Interface

MoRoS3D allows to place a robot in a 3D environment and to let it interact with that environment in a manner similar to robots situated in the real world. Although the user visualizes the entire surroundings of the robot, the robot software only ”sees” the information it collects through its sensors, just like a real robot would do.

As can be seen in Figure 3, the main part of the Graphical User Interface (GUI) is off course devoted to the 3D view. On the right of the GUI lie several widgets for managing cameras, robots' position and trajectory plots. The user can choose several viewpoints corresponding to virtual cameras in the 3D scene. There are also two mobile cameras, one on board (button BOARD) and one at the vertical of the robot that points downward (button TRACK). With the NEXT button the user jumps from robot to robot when in tracking or on-board mode.

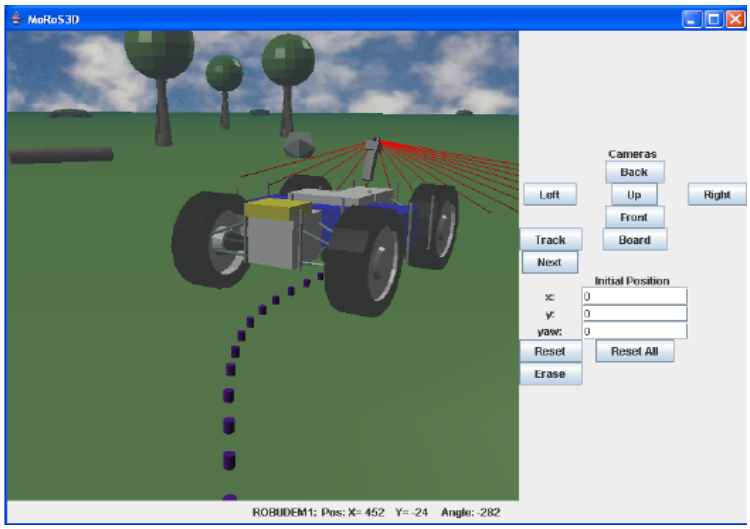


Figure 3. GUI of the simulator

The user can also specify the robots' location and reset one or all robots in a single operation. There is also a button to erase the trajectory plots left behind moving robots. Under the 3D view, name, position and orientation of the selected robot are displayed.

4 Scene Graph

Many free and open-source toolkits are available for building 3D applications[21]. However, most of them focus on visual aspects and few offer high level facilities for managing scenes. This is one reason justifying the use of Java3D for the development of the simulator. The following subsection describes the fundamentals of the Java3D scene model. The simulator scene graph and class structure are presented in section 4.2. Section 4.3 presents the Behaviour mechanism of Java3D and how it is used for synchronising robot motion and sensor detection. The last section deals with collision detection and explains why the built-in Java3D collision detection mechanism is not appropriate for realistic motion simulation.

4.1 Java3D scene model

Java3D is a full-featured API for interactive 3D graphics which manages the display of the scene described basically by a high-level scene graph programming model. Scene graphs are treelike data structures used to store, organize and render 3D scene information [WALS02]. They are made up of objects called nodes, which represent objects to be displayed, aspects of the virtual world or group of nodes.

SimpleUniverse

Java3D provides a utility class called *SimpleUniverse* (blue components in Figure 4) that manages the *VirtualUniverse* and Locale objects holding the virtual world. The *ViewPlatform* (VP) is where the viewer is located in the world, it represents the viewpoint. Changing the transformation matrix (see below) for the *ViewPlatform* moves the viewpoint. The *View* object tells how to turn what the viewer sees into a 2D picture. The *Canvas3D* tells where to draw the 2D picture on the

21 More than 230 engines are recorded in the database of the site http://www.devmaster.net

computer screen.
The content branch (yellow components in Figure 4) containing the nodes of virtual world is attached to the Locale.

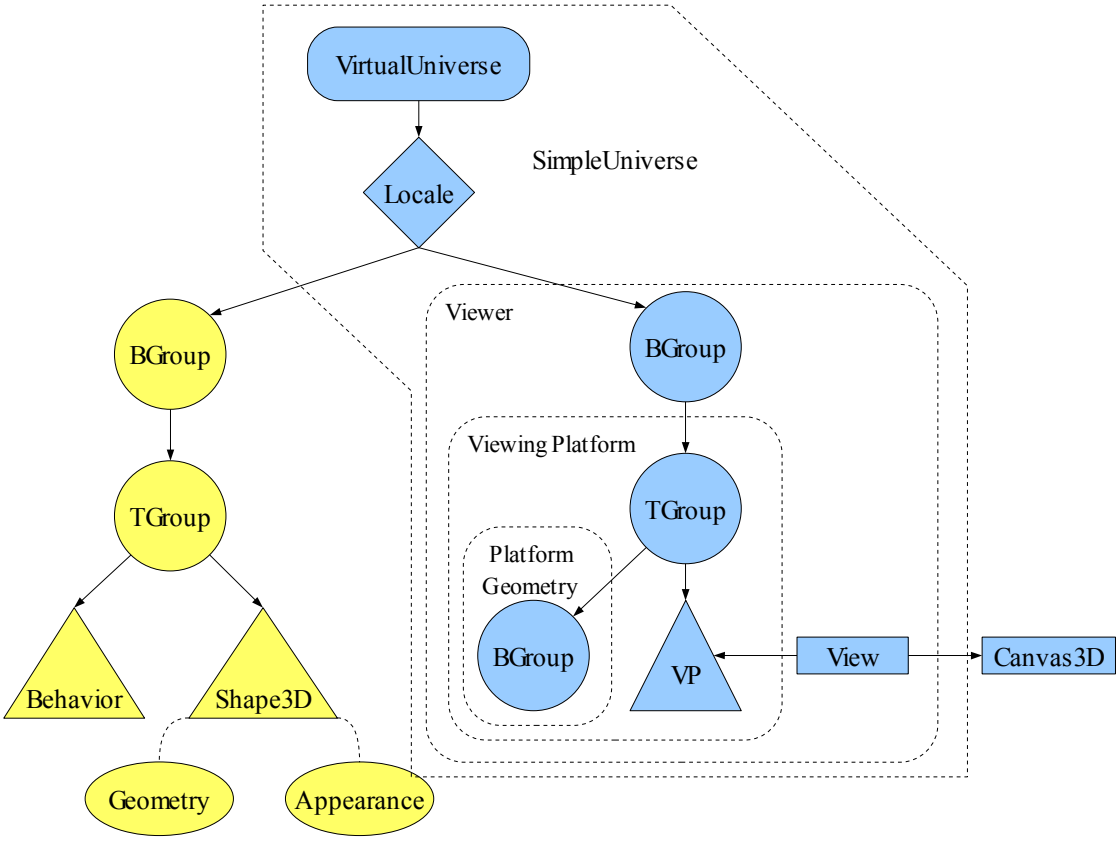


Figure 4. Typical scene graph diagram

Nodes and NodeComponents

Nodes and NodeComponents are the basic elements of the scene graphs. Nodes can be divided into the following basic categories:

- Shape nodes, which represent 3D objects in the world.
- Environment nodes, which represent characteristics of the world such as light, fog, sounds, etc.
- Group nodes, which organise the scene graph.
- The *ViewPlatform*, which is a place where a viewer can look at the world.

Group is the base class for a number of classes that position, orient and control scene graph objects in the virtual universe. The two subclasses used in MoRoS3D are *BranchGroup* and *TransformGroup*. *BranchGroup* holds sub-graphs that can be added and removed while the scene is being displayed. *TransformGroup* changes the transformation of its children, giving them a different position, orientation and size.

By default, each object in a Java3D scene is initially stationary and remains at its starting location unless code specifies otherwise. A *TransformGroup* is associated with a *Transform3D* structure that

corresponds to a 4x4 transformation matrix. A single *Transform3D* object can represent a translation, a rotation, a scaling or a combination of the three. A transformation turns the X,Y and Z coordinates of a point into a new set of coordinates:
This relations can be expressed with 4x4 matrices, where $[x\ y\ z\ 1]^t$ are the original and $[x'\ y'\ z'\ 1]^t$ the transformed coordinates:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

There are many methods to create and modify *Transform3D* objects. These include methods to make a *Transform3D* have a translation, scale or rotation. When a *TransformGroup* is the child of another *TransformGroup*, the effects of their Transform3D objects are multiplied so that all the children of the child *TransformGroup* are affected by both sets of transforms.

NodeComponents are nodes that hold properties or data. *Shape* nodes are NodeComponents that consist of two properties: the geometry, which specifies the 3D coordinates and the appearance, which specifies the colour and other properties of the shape.

Java 3D offers several ways for defining how an object looks like: geometry nodes can be created in the program or by loading files. The most basic way is to work with geometrical shapes, add them together and reshape them to create a complex object. Another one is to import a modelled object from an external file. VRML is one of the format supported by Java 3D. As explained in the overview section, objects are manually drawn with an external program and exported in VRML format. While the full VRML specifications are not supported by the Java3D loader, it is rich enough for building realistic 3D objects and scenes. Colours, textures, transparency effects, sounds... can be used to improve visual appearance of the rendered objects. Besides geometry features the Java3D API supports behaviours that allow programming logic (animations, reaction to events, ...) to be embedded into a scene graph.

4.2 Class hierarchy and Scene graph of MoRoS3D

The scene organization and some essential high level classes are presented in Figure 5. Objects constituting the 3D tree structures are split in different classes that do not necessarily derive from Java3D classes. The scene graph structure (light links) is independent of the class organization (bold links).

At the top level we find an instance (frame) of the MoRos3D class that represents the main object of the application. This object of type *Jframe* includes two instances of classes derived from *Jpanel*, namely *InfoPanel* and *View3DPanel*. *InfoPanel* is used to display text information at the bottom of the windows while *View3DPanel* is the main display panel where the 3D scene and the camera control widgets are drawn. The *View3DPanel* class contains instances of other classes. It also reads and decodes the command line to extract command parameters.

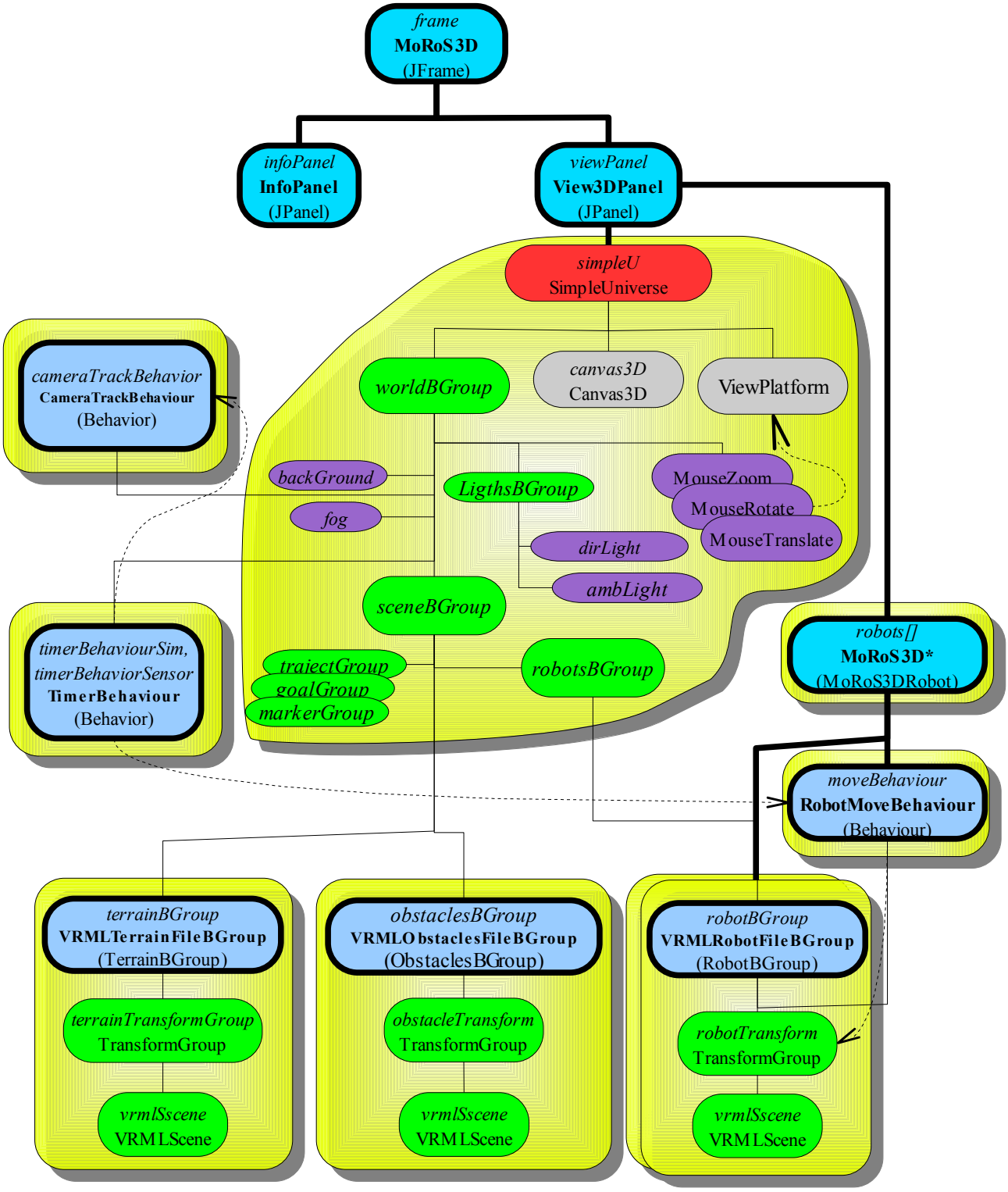


Figure 5. Classes hierarchy and 3D scene graph

SimpleUniverse is a Java3D utility class that manages low level functionality as for instance 3D to 2D mapping. The *SimpleUniverse* renders the image in a *3DCanvas*, which is a drawing widget added to the *View3DPanel*. The *ViewPlatform* is a member of the *SimpleUniverse* used to transform the viewpoint with the mouse via predefined behaviours (*MouseZoom*, *MouseRotate*, *MouseTranslate*).

The *worldBGroup* (*BranchGroup*) contains environmental node such as lights, fog and background and the *sceneBgroup* (*BranchGroup*).

The objects of the 3D world have been divided in three groups: the terrain, the obstacles and the robots. This separation provides flexibility in the composition of the scene. The *sceneBGroup* therefore contains the *terrainBGroup*, the *obstacleBGroup* and the *robotBGroup*. It also contains three *BranchGroup's* used for marking the trajectory followed by the robot, the trajectory that it should follow and the goal he has to reach.

The MoRoS3D* (on the right of Figure 5) correspond to concrete container classes for the different robots and inherit from the *MoRoS3DRobot* class. This class contains objects that specify the geometry of the robot (*RobotBGroup*), the robot motion laws (*RobotMoveBehavior* class) and the sensors associated with the robot (*CircSensorBGroup*). The *RobotBGroup* instance is initialized in derived classes (for instance *MoRoS3DRobudem*) with a derived class constructor, namely *VRMLRobotFileBGroup* (Figure 6).

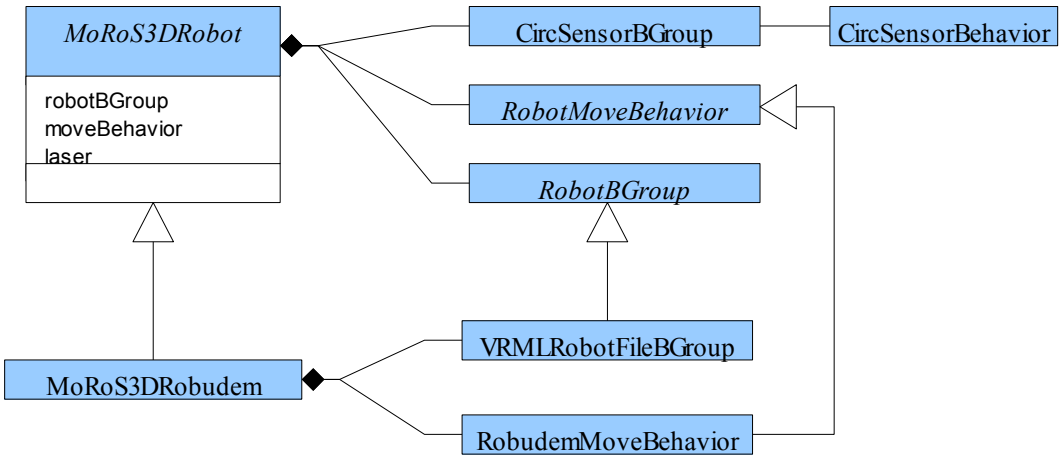


Figure 6. Main classes inheritance diagram

The *RobotMoveBehavior* instance is initialized with a *NomadMoveBehavior* or a *RobudemMoveBehavior* with respect to the instantiated *MoRoS3D** class. The sensors are defined in these derived classes.

The *TerrainBGroup* is a generic class for creating terrains that inherits from the Java3D *BranchGroup* class. The *VRMLTerrainFileBGroup* class derives from *TerrainBGroup* and imports 3D terrain objects from VRML files[22]. The Java3D API allows to import objects from other file formats but the VRML importer seems to be the most robust. It could also be possible to create 3D terrains from geometry entities (polyhedra, fractals, ...) or elevation grids.

22 While all objects are 3D, the ground is actually flat in the current implementation.

Similarly the *VRMLObstaclesFileBGroup* inherits basic functionality from the base class *ObstaclesBGroup* and *VRMLRobotFileBGroup* from the *RobotBGroup* class. The same comments as for the *VRMLTerrainFileBGroup* applies for these two classes.

4.3 Behaviours and events

The physics engine in Moros3D has been implemented using what is called a behaviour. A behaviour is a piece of code that can manipulate the transform at the top of a group of objects. Behaviours are nodes that make changes to the scene graph in response to events, such as user input or the passing of time. A behaviour indicates interest in a set of events, called the behaviour's W*akeUp* criterion. When an event occurs that matches the criterion, Java3D calls a method on the behaviour to process the event, making changes to the scene graph or performing any required task. Besides predefined behaviours, Java3D allows the programmer to set up its own custom behaviours. One of the advantages of using behaviours is that the performed operations are synchronized with the rendering engine, ensuring that all the computations are done before a new frame is displayed. Nothing new will be rendered before the handling methods of the behaviours have finished.

Custom behaviours inherit from the class *Behavior* and specify its action by implementing the methods *initialize* and *processStimulus*. The method *initialize* is called when the behaviour is first made live while the method *processStimulus* is called when there is an event for the behaviour to process. The *wakeupOn (WakeupCondition criterion)* method is called by both methods to indicate which events should wake up the behaviour.

In MoRoS3D, a hierarchical structure has been defined in order to synchronize the behaviours (represented by dash lines in Figure 5). At the top level we have instances of the custom TimerBehavior class that generates time events, typically every 50 msec but different periods can be used for the motion simulation and the sensors. These time events are received by behaviours embedded in each robot, sensor and tracking cameras (Figure 7).

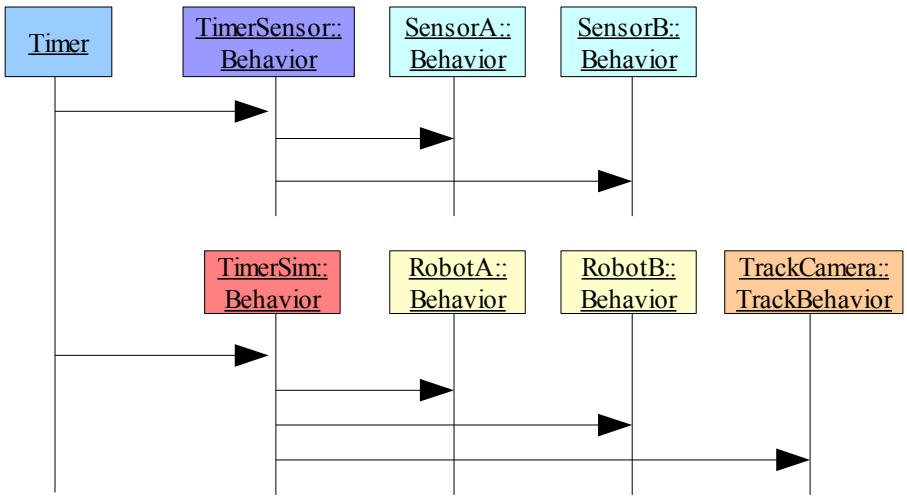


Figure 7. Time events propagation

The robot motion control and the movement of the tracking cameras have to be synchronized

otherwise it generates discontinuities in the visualization process (the camera springs). Each sensor can be controlled by a different timer in order to reproduce the measurement rate of real sensors. For example a US sensors produce data every 50ms or a laser line sensor every 200 ms, etc. At the reception of the time event, the *processStimulus* method of each of them is called and the appropriate actions executed: computation of the robot's motion and detection of collision, measurement of distances for sensors, modification of the position and orientation of the camera. If the same value is used for all timers, all modifications to the 3D scene happen at the same time.

5 Robot models

In order to model robot motion with a scene graph, the shapes must be placed under a transform group with the transformation being modified at each frame. This needs to be coded by the programmer because the behaviours built into VRML/Java3D are not appropriate as they require start and end points which are generally not known in advance.

5.1 Nomad

Geometry

The geometry of robots is determined by their shape and dimensions. The 3D models have been drawn with a 3D modelling application (Wings3D) and exported in the VRML format. The Nomad (Figure 8) has a simple geometry and only visible parts have been modelled.

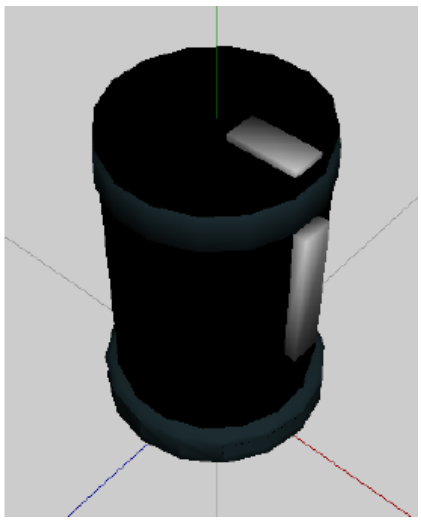


Figure 8. 3D model of the Nomad

Reference frames

The coordinates systems are represented in Figure 9 with:

- the World coordinate system on the left
- the Object coordinate system on the right

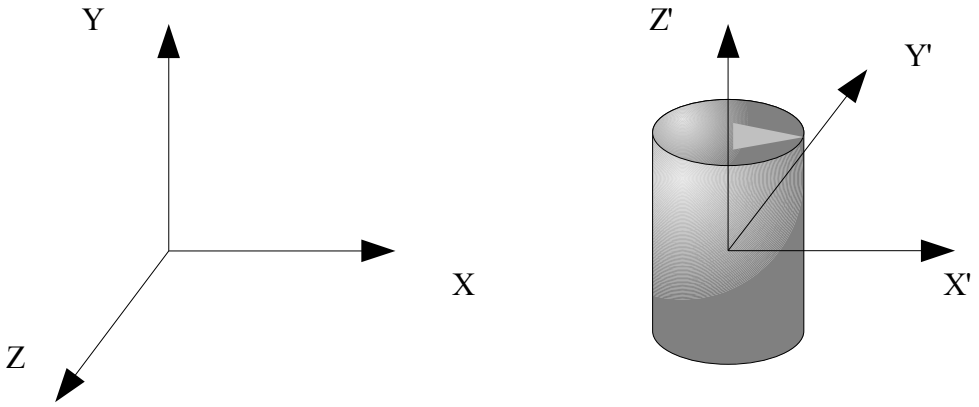


Figure 9. World and robot reference frames

The transformation between the two coordinate systems is given by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

Kinematics

The Nomad is actuated by a synchronous mechanism, each wheel is capable of being driven and steered. The three steered wheels are arranged as vertices of an equilateral triangle and all the wheels turn and drive in unison. The translation speed is v(t) and the steering speed is w(t) (Fig. 10) Actually the real Nomad has a third degree of freedom, the turret can turn independently of the base but this mechanism has not been implemented in the model.

The Euler algorithm [JAME85] is used as integration method to obtain the position from the velocity. The standard Euler integration method requires a single forcing function evaluation, and produces a first order accurate solution. That algorithm assumes that you have a value of a variable, x(t), a state equation that allows you to compute dx/dt, and that you want to compute x(t + Δt). The algorithm for a single variable is simple:

$$x(t+\Delta t) = x(t) + \Delta t \frac{dx}{dt}$$

where dx/dt is the speed of the robot that is obtained from the dynamical equation. This is actually the continuous form of the algorithm. When implemented in a computer, the discrete version has to be used as explained below.

The algorithm is applied repetitively to compute a solution for the state at equally spaced intervals of time. The Euler method is known for accumulating errors at each integration steps. We neglect these errors here as we are more interested by global behaviours and environment interaction than by exact trajectories.

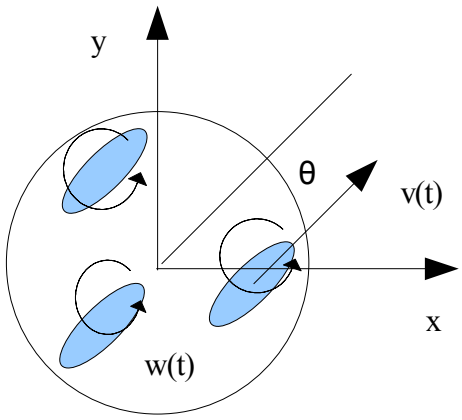


Figure 10. Kinematic model of the Nomad

The state vector is defined as:

$$\begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}$$

The motion of the robot in the global frame is described by the following differential equations:

$$\begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} v(t)\cos\theta(t) \\ v(t)\sin\theta(t) \\ w(t) \end{bmatrix} \quad (1)$$

that we replace by the finite difference approximation:

$$\begin{bmatrix} x(t+h)-x(t) \\ y(t+h)-y(t) \\ \theta(t+h)-\theta(t) \end{bmatrix} = \begin{bmatrix} v(t)\cos\theta(t) \\ v(t)\sin\theta(t) \\ w(t) \end{bmatrix} \cdot h \quad (2)$$

and the discrete formulation can be written as:

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} + \begin{bmatrix} v_k\cos\theta_k \\ v_k\sin\theta_k \\ w_k \end{bmatrix} \cdot h \quad (3)$$

The Java3D API proposes classes to represent homogeneous transformations and to perform classical matrix operations. At each position of the robot we can associate a reference system that can be represented by a 4x4 transformation matrix:

$$cur = \begin{bmatrix} \cos\theta_k & -\sin\theta_k & 0 & x_k \\ \sin\theta_k & \cos\theta_k & 0 & y_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

The next position of the robot results from an elementary translation and rotation that is expressed by the following matrices:

$$trans = \begin{bmatrix} 1 & 0 & 0 & v_k h \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$rot = \begin{bmatrix} \cos\Delta\theta_k & -\sin\Delta\theta_k & 0 & 0 \\ \sin\Delta\theta_k & \cos\Delta\theta_k & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

where

$$\Delta\theta_k = w_k h \tag{7}$$

The matrix product *trans* * *rot* gives the elementary motion:

$$\begin{bmatrix} \cos\Delta\theta_k & -\sin\Delta\theta_k & 0 & v_k h \\ \sin\Delta\theta_k & \cos\Delta\theta_k & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

and the transformation matrix is obtained by multiplying this matrix (4) with the matrix (8):

$$\begin{bmatrix} \cos(\theta_k+\Delta\theta_k) & -\sin(\theta_k+\Delta\theta_k) & 0 & x_k+v_k h\cos\theta_k \\ \sin(\theta_k+\Delta\theta_k) & \cos(\theta_k+\Delta\theta_k) & 0 & y_k+v_k h\sin\theta_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

and, as

$$\theta_{k+1} = \theta_k + \Delta\theta_k \tag{10}$$

we finally obtain the following transformation matrix:

$$\begin{bmatrix} \cos\theta_{k+1} & -\sin\theta_{k+1} & 0 & x_k + v_k h\cos\theta_k \\ \sin\theta_{k+1} & \cos\theta_{k+1} & 0 & y_k + v_k h\sin\theta_k \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11)$$

that contains a rotation matrix characterized by an angle θ giving the orientation and a translation matrix giving the new position of the reference frame associated with the robot. This expression is coherent with equation (3).

Dynamics

Determining the real dynamic behaviour of such a robot is not a trivial task. As the speed of the real robot is controlled by a proportional controller we can represent the system by the model in Figure 11, with m the mass of the robot and Kp_t the proportional gain of the controller. It corresponds to the closed control loop for the translational speed, where v_c is the command speed.

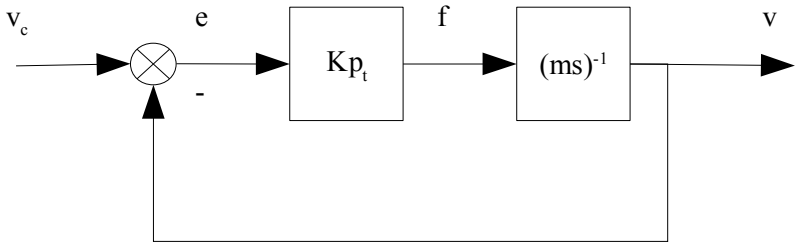


Figure 11. Proportional control loop

The system equation in the Laplace domain is:

$$V = (V_c - V)\frac{K_p}{ms} \quad (12)$$

or

$$VS = (V_c - V)\frac{K_p}{m} \quad (13)$$

and in the time domain:

$$\dot{v} = (v_c - v).\frac{1}{\tau_t} \quad (14)$$

With τ_t equals to m/K_p

The steering speed and the translation speeds are consequently updated according to the following finite difference equations:

$$v_{k+1} = v_k + (v_c - v_k)h/\tau_t$$
$$w_{k+1} = w_k + (w_c - w_k)h/\tau_s \quad (15)$$

Where τ_s and τ_t are the estimated time constants of the system and v_c and w_c are the command speeds. These constants have been adjusted for the typical dynamic behaviour of the Nomad taking into account the usual values used for maximum accelerations (These values can be changed by calling the appropriate function of the Nomad API).

5.2 Robudem

Geometry

The geometry of the Robudem is more complex than the one of the Nomad. The Robudem has four wheels actuated individually by electrical motors. The two axles are steerable and actuated by two linear electrical motors via an Ackerman mechanism. The following figures show the real robot and its 3D model.

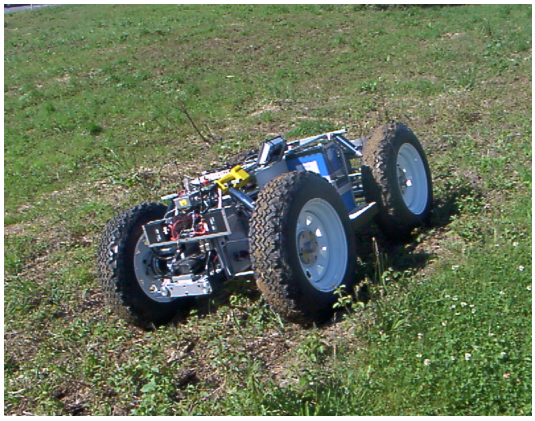


Figure 12 a. Picture of the real Robudem

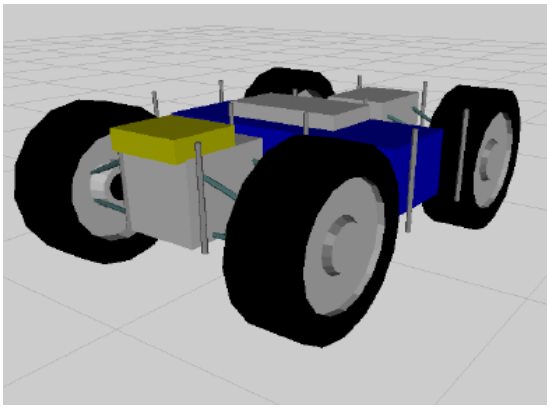


Figure 12 b. 3D model of the Robudem

Kinematics

The trajectory control of the real Robudem is based on two parameters: the instant desired speed v and the instant desired steering lock α. Indeed, at each time, the vehicle trajectory can be expressed with those two values that are given by the user through a joystick interface or by a control program. Three modes can be used to control the motion of the real Robudem (Figure 13).
In the simulator only the "single drive" mode has been implemented as it is the most used one on the real robot. Other modes are more difficult to control and are less efficient because of the larger friction of the wheels with the ground. In the dual mode, the centre of rotation is different for the front and rear axles, and in the park mode, the wheels are not parallel.
Supposing a perfect Ackermann steering mechanism for the front axle results in the instantaneous centre of rotation lying on the axis of the rear axle. In this case we can use a bicycle model for representing the kinematics of the vehicle (Figure 14): the four wheels are replaced by two wheels located in the middle of the vehicle.

Single drive mode: only the front axle is controlled during motion and the rear axle is fixed

Dual drive mode: the remote software controls the front and rear axles during the motion

Park mode: Both axles are steered in the same direction

Figure 13. Control modes of the Robudem

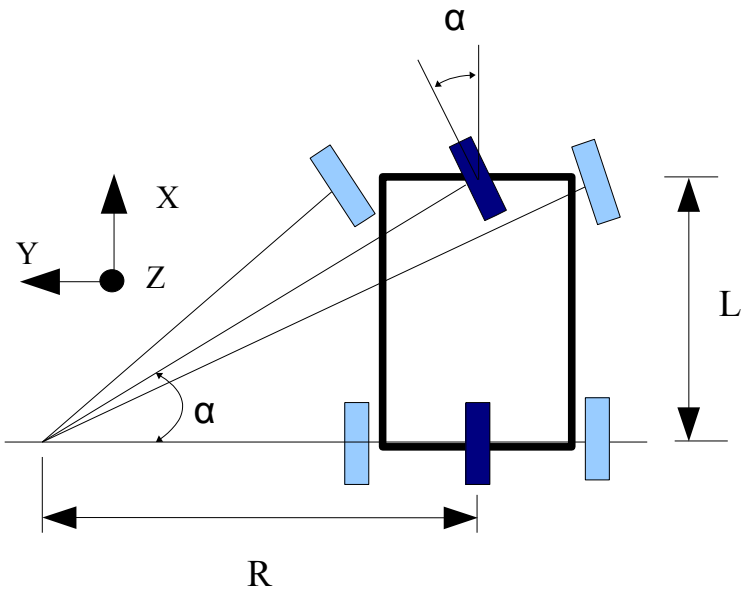


Figure 14. Kinematic model of the Robudem

Let the angular velocity vector along the body z axis be $\dot{\theta}$. Using the bicycle model approximation, the radius of curvature R and the steer angle α are related by the wheelbase L. By definition of the curvature:

$$\frac{d\theta}{ds} = \frac{1}{R} = \frac{\tan\alpha}{L} \tag{16}$$

The rotation rate is obtained from the speed v as:

$$\dot{\theta} = \frac{d\theta}{ds}\frac{ds}{dt} = \frac{1}{R}\ v = v\frac{\tan\alpha}{L} \tag{17}$$

The finite equation deriving from this equation is:

$$\Delta\theta_k = v_k h \frac{\tan\alpha_k}{L} \tag{18}$$

Once the incremental angle has been obtained, The model of the equation 7 to 11 can be used.

Dynamics

Simulating the dynamic behaviour of Robudem is based on the same model as for the Nomad (Equation 15). Off course the time constants have been adapted to reflect the dynamics of this robot. Another particularity of the real controller that has been taken into account is the following: when the user suddenly puts the joystick in neutral position, the controller immediately stops the robot while when he pulls it gently, the speed is reduced by applying a linear profile.

6 Collision detection and response

6.1 Problem

The previous section has presented the motion control of robots in open environment, that is without any obstacles. Off course in any realistic application robots have to cope with static and dynamic obstacles. In the developed simulator, dynamics obstacles are other mobile robots while the environment is static. It this then necessary to be able to detect and to react to collisions. Moving autonomously implies detecting and avoiding obstacles. One of the basic requirement of the simulator is consequently to provide collision detection to detect when the control algorithm fails and the robot collides with the environment or with other robots and to provide adapted response.

Java3D provides classes for detecting collisions between objects. However, this API works asynchronously and does not offer any guarantee when the detected collision will be reported, what happens generally after the object has entered into another one. This is not an appropriate mechanism and therefore a collision detection algorithm exploiting Java3D Behaviours has been implemented.

6.2 Collision detection

Collision detection is more a geometric problem than a physical one. To make sure that any area of space cannot be occupied by more than one object, collision detection based on the geometry of the objects is required [ERIC04].

For any realistic environment and even if simplified shapes are used, the collision detection needs a lot of mathematical operations. So the big issue with collision detection is the number of tests

that have to be made and therefore the CPU resources used. For example if we have n objects then the first object could collide with *n-1* objects (since we don't check if an object has collided with itself), the second object could collide with *n-2* additional objects not counting the possible collisions we have already counted. If we keep going like this the number of possible collisions between objects is:

$$(n-1)+(n-2)+(n-3)+...+1$$

That is equal to:

$$\frac{n(n-1)}{2}$$

In reality each object is composed of hundreds of triangles and the collision detection would required too much time if it had to be performed for any triangle. During the simulation we need to check for collisions at every frame therefore it is important that collision detection be very efficient. We therefore need to apply a method to speed up the computation. Hopefully, there exist different optimization methods for reducing the amount of operations. For instance, bounding volumes can be used to reject non intersecting objects.

In the scene of Figure 15, objects are surrounded by an Axis Aligned Bounding Box (AABB) . If any of the boundaries overlap then the shapes may, or may not, overlap and further tests are required, if the boundaries do not overlap then the shapes have not collided. This allows to eliminate some of the CPU intensive tests for checking overlap of complex shapes.

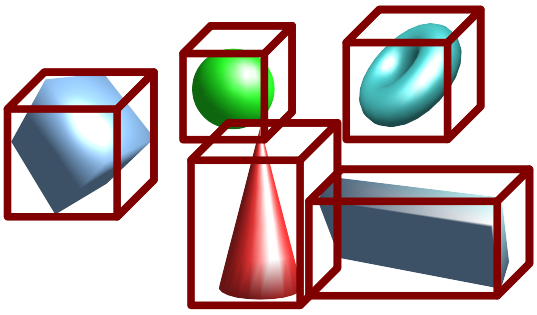


Figure 15. Bounding boxes

It is indeed very easy to test boxes for overlap, provided they are all oriented in the same direction, we just need to compare their minimum and maximum x,y and z coordinates.

For instance if box 'A' is defined by AxMin, AxMax, AyMin, AyMax, AzMin, and AzMax. and box 'B' is defined by BxMin, BxMax, ByMin, ByMax, BzMin, and BzMax.
Then the boxes overlap if all the following conditions are true:

$$AxMin < BxMax \wedge AxMax > BxMin$$
$$AyMin < ByMax \wedge AyMax > ByMin$$
$$AzMin < BzMax \wedge AzMax > BzMin$$

However this only applies if the bounding boxes are axis aligned. If the bounding boxes were

defined in the local coordinates and one of the boxes were under a transform group with a rotation then we would have to either:

- use an algorithm to detect the intersection of arbitrary Oriented Bounding Boxes (OBB), in absolute coordinates, which would be much more complex.
- or recalculate the AABB at every frame in axis aligned absolute coordinates and doing things at every frame is a big overhead.

If a single box around the object does not give accurate enough collision detection for the shape then it is possible to use multiple boxes in a hierarchical way to more accurately match the shape of an irregular object.

A simpler technique consists in defining a bounding spherical envelop around all objects and to calculate distances between the centres of the bounding spheres (Figure 16). This method transforms a complex 3D problem in a simple distance calculation.

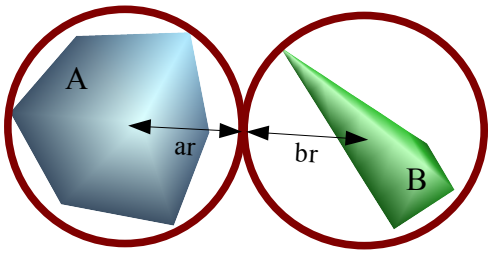


Figure 16. Bounding spheres

It is very easy to detect if bounding spheres overlap, for instance:

- Object A has centre at ax, ay, az and radius ar
- Object B has centre at bx, by, bz and radius br

Then the bounding spheres intersect if:

$$(ax-bx)^2+(ay-by)^2+(az-bz)^2<(ar+br)^2$$

The advantage of this method is that it is independent of orientation. So this does not have the problem mentioned for bounding boxes where the axes need to be aligned. The disadvantage with bounding spheres is that it may not fit a long thin object very well, there will be some false detection of collisions, but in that case we can use a secondary check to test the boundary more carefully.

Using bounding volumes reduce the computing cost by eliminating objects that do not collide but it may not be good enough to rely on the bounding box or sphere alone especially if the objects are complex shapes. However they can at least filter out those objects that do not overlap. Another reason that we cant rely on bounding rectangle or sphere alone is that in order to go on to the next stage of working out the collision response we also need to know the points of impact.

If we want to test for collision of meshes, made up from triangles, and we want to check for

collisions accurately, using all the information from the geometry, we may need to test each triangle on object 'A' with each triangle on object 'B' for intersection. Currently the most efficient test is the algorithm of Moller [MOLL97] that is explained below.

We first determine the equation of the planes containing the triangles and work out the intersection line for the two planes (Figure17). Intervals Ia and Ib are are computed. If Ia ∩ Ib ≠ ϕ than the two triangles intersect.

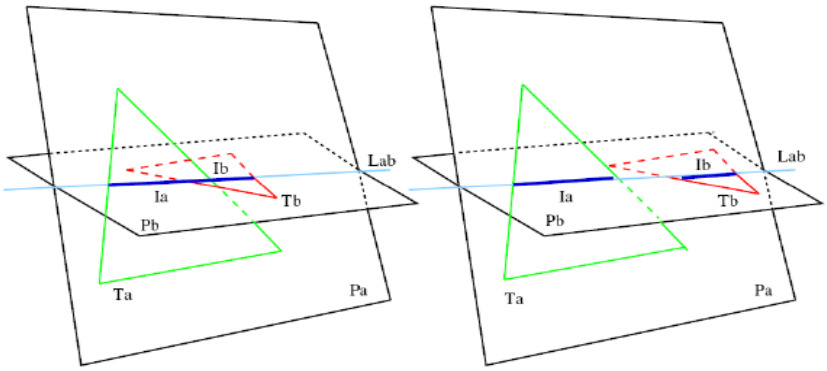


Figure 17 Testing a possible intersection between two triangles

6.3 Implementation

Java3D offers methods for calculating intersection between bounding boxes of objects. However as explained in section 6.2, the automatically computed bounding boxes have always their axes parallel to the global reference frame. This gives an unrealistic representation for any real object.
Therefore the method used in the simulator is a compromise between the two approaches presented above. It consists in replacing mobile robots by a good approximation and checking for collision with the real geometry of other objects. The Nomad is for instance simply replaced by a cylinder. The Robudem has basically a box shape and it is more suited to define its contour manually by specifying a bounding box defined by six orthogonal plans what in the Java3D jargon is called a "Polytope".

In order to render realistic collision occurrences we must be able to predict these events before they effectively happen in order to avoid that a robot enters into an object. Knowing the actual speeds we compute for each frame the pose of the robot by applying equations 3 and 15. With this prediction we check if a collision occurs with any fixed or mobile obstacles.
Once we have detected a collision between two objects, we can compute the reaction by using physical laws and by considering for each object the velocity, the mass, the centre of mass, the inertia, ... As in targeted applications real robots are moving slowly we do not need complex collision response because most of the time robots are simply blocked when they move into an obstacle. So in case of collision we stop the robot by disregarding the last transformation. This algorithm is illustrated by Figure 18.

This collision detection is implemented in MoRoS3D by using Java3D utility classes originally

developed for picking[23] actions. These classes have methods for defining the bounding shapes (*sphere, cylinder, box)* and test for collision with object geometry. If the result is not null it means we collided with something and we stop the robot as explained in the previous paragraph.

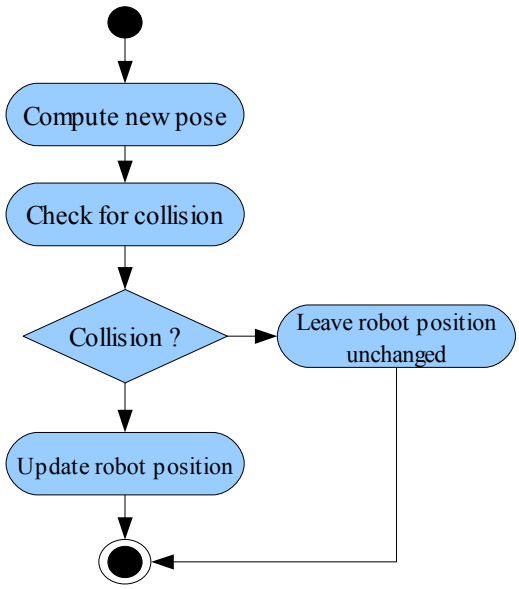


Figure 18. Position update algorithm

7 Sensor modelling

7.1 Perception Sensors

Two kinds of sensors are necessary for developing intelligent control applications in mobile robotics: position and environment perception sensors.

Global position sensors can be easily implemented within the simulator because we perfectly know the position and orientation of the robot and of all its components. Relative position sensors and low level encoder signals can also be derived from this global position knowledge.
A mobile robot can only act intelligently if it perceives its environment. Distance sensors are mandatory for seeing what stands around the robot. Three models of such sensors have been implemented in the simulator, namely laser, infra-red and ultrasonic sensors. To implement the measurement process we have used Java3D's picking routines. The idea is to cast a ray into the space around the robot. This ray has a length equal to the maximum distance the sensor can measure.

23 Picking is the process of selecting shapes in the 3D virtual world using the 2D coordinates of the mouse on the Canvas3D

7.2 Linear Sensors

IR and Laser signals can be simulated with linear beams. The measurement process is based on collision detection between a segment of line with *startpoint* (s) the current position of the sensor and *endpoint* (e) the maximum measurable distance (Figure 19).

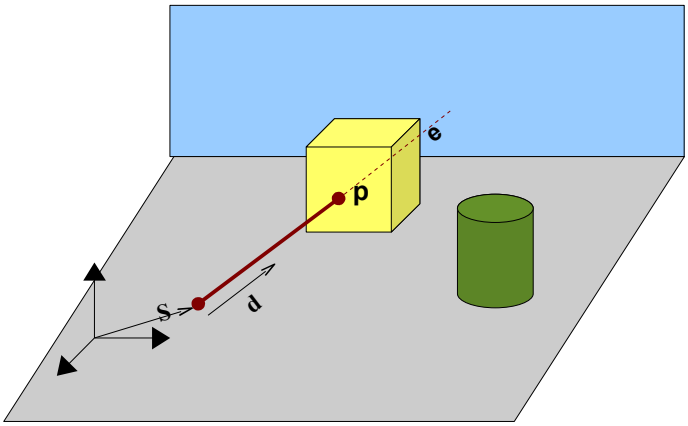


Figure 19. Schematic view of a distance measurement operation

The collision detection requires the following operations to be performed (Figure 20):

- To determine if the endpoint (e) crosses the plane of any triangle.
- To find out where exactly the intersection (p) is on that plane and determine if that point of intersection is actually within the boundaries of the triangle.

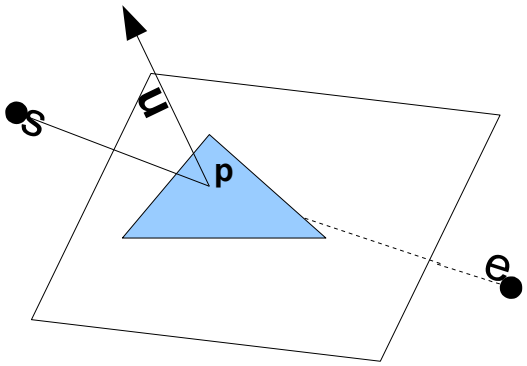


Figure 20. Intersecting with a triangle

This process has been implemented with the Java3D *Picktool* class that is the base class for picking operations. Two useful methods for simulating a linear sensor are *setShapeRay* that takes two arguments, *startpoint and direction* and *pickClosest.* The first method allows to define the location and the direction of the virtual laser beam while the second returns the closest object that intersects with the ray. From this object it is now possible to get the distance (d) from the origin (s) of the picking ray and the closest intersection point (p) (Figure 19).

7.3 Ultrasonic sensors

Ultrasonic sensors have a radiation pattern (Figure 21a) that is generally modelled by a cone (Figure 21b). Complex reflections phenomena's can also happen: specular reflection, multiple paths, cross detection between adjacent sensors,... Only the simple reflection case is considered in this simulator. The value returned by the sensor is the distance between the top of the cone and the closest intersection point between the cone and any triangle of the geometry.

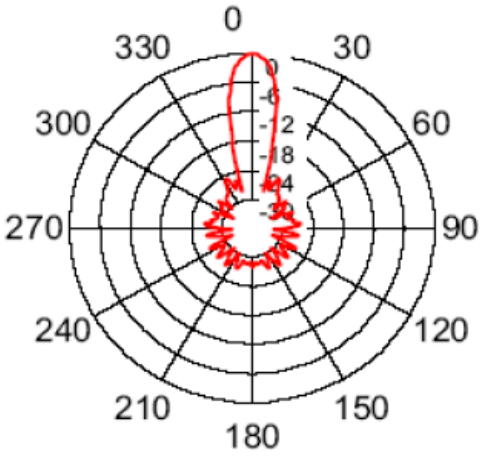


Figure 21a. Typical emission pattern of US sensor

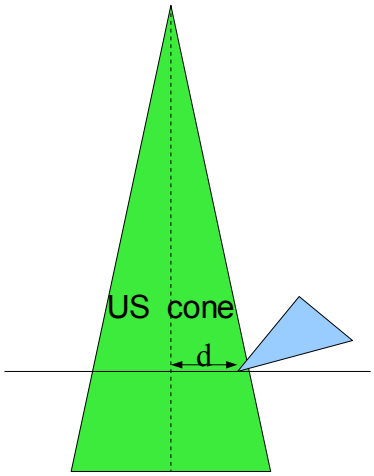


Figure 21b. 2D view of the US cone intersection with a triangle

7.4 Array of sensors

The classes *LaserCircSensorBGroup* and *USCircSensorBGroup* allow to create several sensors and to geometrically arrange and position them in function of different parameters passed to the constructors. They can form a ring (Figure 22a and c) or an arch or in the case of the Laser be grouped at the same location to simulate a 2D laser range finder (Figure 22 b). It is also possible to define the height, the distance from the reference point and the tilt angle of the sensor group.

These classes contain Transform3D instances that store the position and orientation of the sensors, an instance of the *LaserBehavior* or *USBehavior* classes that perform the distance measurement for each individual sensor and geometry information for representing the sensors' housings and beams. The display is updated after each measurement.

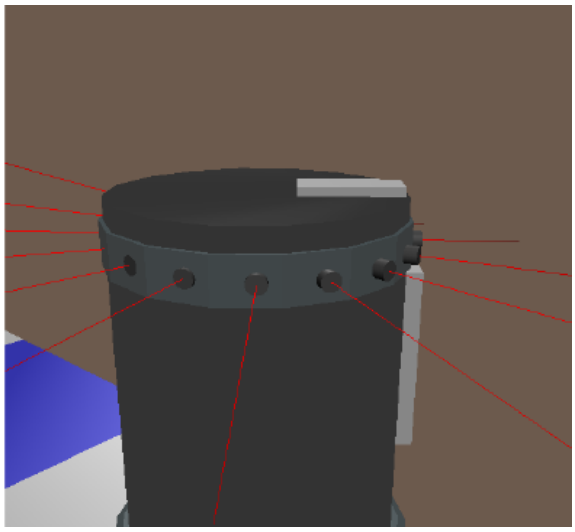


Figure 22 a. A ring of laser sensors

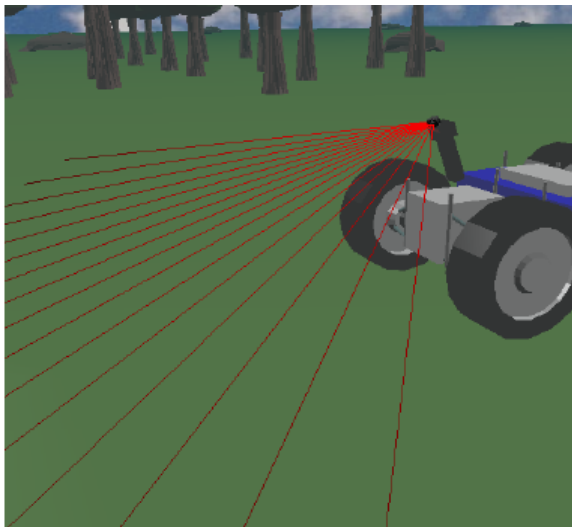


Figure 22 b. A simulated laser range finder

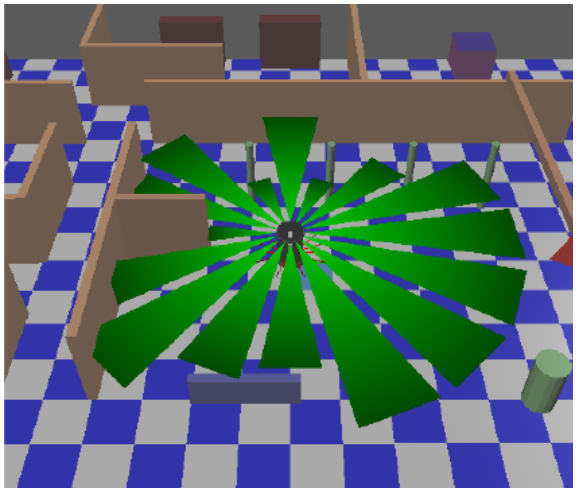


Figure 22 c. Ultrasonic sensors

The range of the different sensors can be adjusted in the program. Typical values are:

- Laser: 0,20 to 10 m
- US: 0,3 to 7 m
- IR: 0,1 to 0,6 m

8 Integration with CoRoBA

8.1 Communication

As the simulator has been designed to provide CORBA interfaces for all robots and sensors, its integration in the framework is straightforward. Two possibilities exist for a CoRoBA component to communicate with the simulator: via CORBA synchronous call or via events.

In the first case Sensor and Actuator components communicate with the Simulator via synchronous calls using operations defined for every sensor and robot (Figure 23 left).

The second possibility is to use event based communication what means that servants also have to

implement the *StructruredPushConsumer* or *StructuredPushSupplier* interfaces. In this case the Simulator directly acts as a Sensor or an Actuator (Figure 23 right). The implementation is more complicated because we must locate the NotificationService, connect to Event Channels, etc. The advantage is that we don't need Sensor and Actuator components.

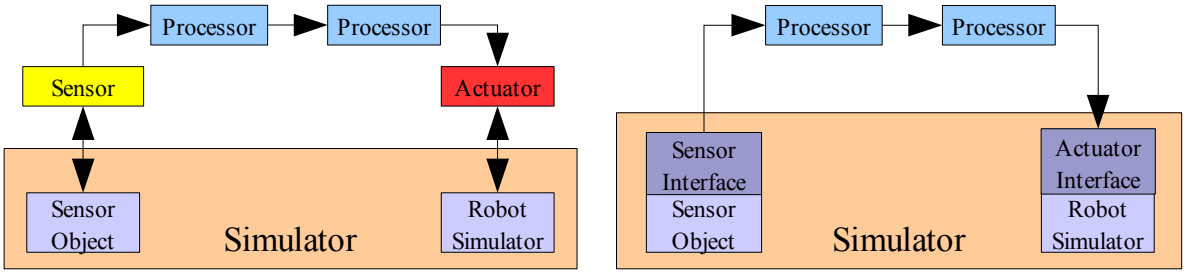


Figure 23. Synchronous (left) and Event based (right) communication between the simulator and CoRoBA components

The first approach has been privileged for implementing applications because we keep the same structure in simulation as in real applications for which we necessarily have Sensor and Actuator components.

As the SUN implementation of CORBA does not support Notification Events, another ORB has been used. JacORB[24] is a free Java ORB that comes with full source code, a couple of CORBA Object Service implementations, and a number of example programs. JacORB implements the Notification specifications and works perfectly with TAO.

8.2 Interfaces

Robots

A base interface has been defined for a generic robot as well as derived interfaces for the different simulated robots. The interface Robot provides generic operations for placing the robots in the simulated environment without taking account of any control. The derived interfaces (Figure 24) provide operations that have the same signature as operations of real Actuator components. For the Nomad, the operation is *vm* that takes as parameters three integers corresponding to the translation speed, the rotation speed of the wheels and the rotation speed of the turret.

CORBA interfaces are mapped to JAVA interfaces that must be implemented by servant object classes. The details of this mechanism is described for instance in [Li00]. The constructor of the class *NomadImpl* receives as parameter a reference to the *MoRoS3DNomad* object that is copied into a member variable of this type. When an object invokes the CORBA operation *vm* implemented by this class, to modify for instance the speed of the robot, the corresponding Java method *vm* of the *MoRoS3DNomad* class is called, resulting in the adaptation of the speed in the simulator.

For each robot model there are an interface and a class implementing this interface.

24 Available at http://www.jacorb.org

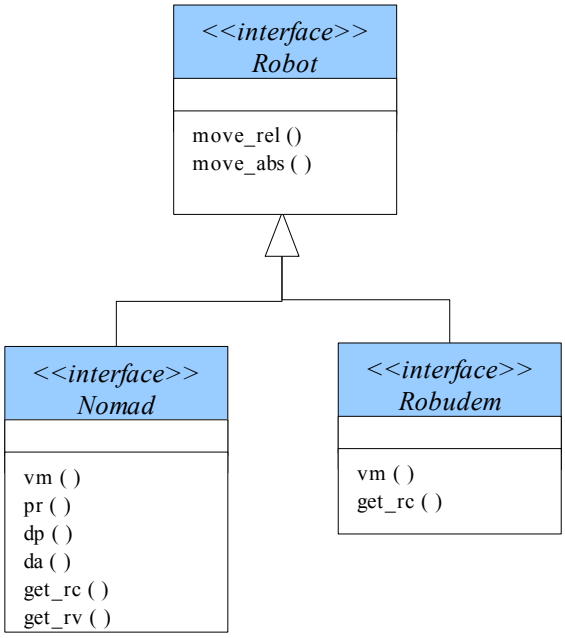


Figure 24. Robots Interface inheritance diagram

Laser interface

The Laser interface gives the possibility to remotely get the data of the simulated laser. The data type and the operation are similar to those defined for the CoRoBA sensor component and are:

```
typedef long Laser_data;
typedef sequence<Laser_data> Laser_Seq;

interface Laser {
        Laser_Seq get_data();
};
```

Defining the data vector as a sequence increases the flexibility of the simulated sensors because this length is defined by the implementation and discovered at run-time by components invoking the *get_data* operation and testing the sequence length. The *LaserImpl* class implements this interface and its *get_data* method returns the distance measured by the simulated sensor. The Infra-red and Ultrasonic interfaces have a similar structure.

Trajectory and Goal interfaces

The aim of these two interfaces is to provide remote access to the display capability of the simulator (Figure 25). MoRoS3D is able to show objects representing goals the robot has to reach and trajectories it has to follow. The CORBA interfaces for the Goal and the Trajectory functionality are listed below.

```
struct traject_pts{
        double x;
        double y;
        double teta;
};
```

```
typedef sequence <traject_pts> trajectory_Seq;

interface Trajectory {
        void display(in trajectory_Seq traject);
        void append(in trajectory_Seq traject);
        void delete ();
};

interface Goal {
        void display(in traject_pts goal);
};
```

They are respectively implemented by the *GoalImpl* and *TrajectoryImpl* classes. Their use will be illustrated in the following chapter.

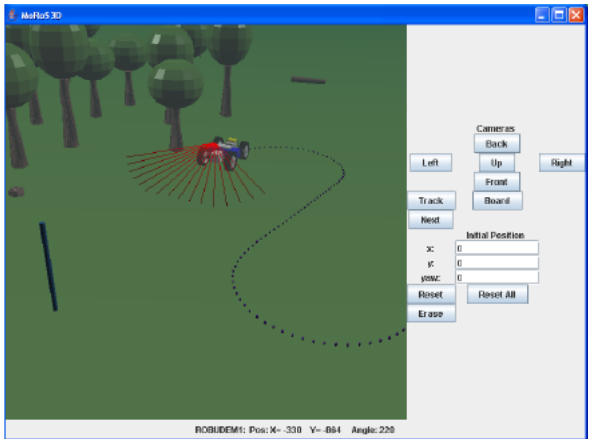


Figure 25a: The goal is represented by a blue cylinder

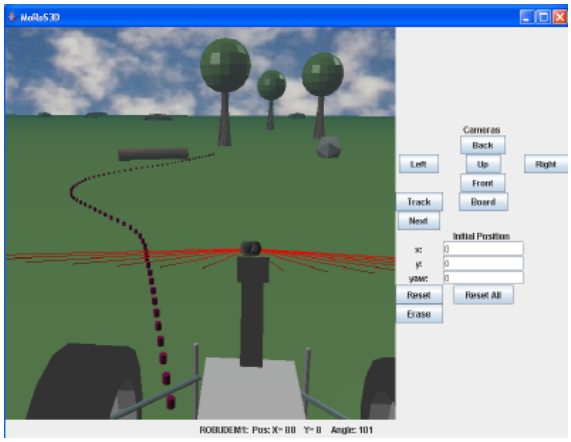


Figure 25b: The trajectory the robot must follow

8.3 Registration

All CORBA objects of the simulation register with the NameService and each simulator uses its own name context (Sim1, Sim2, ...) allowing multiple instances to run simultaneously (Figure 26).

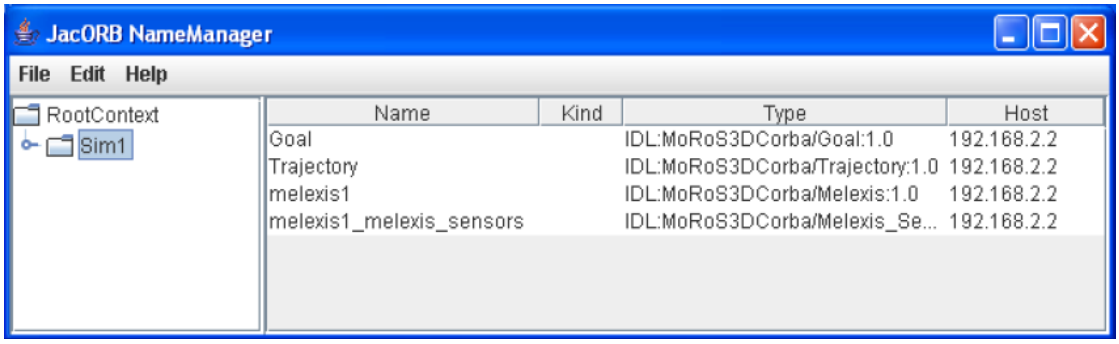


Figure 26. NameService directory tree

Sensor and Actuator components obtain the IOR of the simulator objects by contacting the NameService.

9 Simulation engine

9.1 Control Engine

The algorithm implemented by the control engine is represented in Figure 27. The algorithm is identical for all robots, it contains all operations required for computing the motion of a robot and to detect and react to collisions. These operations are performed in the *processStimulus* methods that are called at regular time intervals by time events.

Each block of the diagram is commented below with references to the preceding sections:

- As timer values are not perfectly constant, the time (Δt) elapsed between two consecutive executions of the method is measured. We obtain so a more regular motion.
- The command speeds (Vt, Vs) are copied from the robot object (*MoRoS3DNomad*). These variables are modified via the invocation to the *vm* operation (implemented by the *NomadImpl* class see section 8.2) by the CoRoBA *Nomad_Actuator* component.
- The real speeds of the robot are computed with the dynamical model of the robot (section 5 – dynamics).
- The kinematic model (section 5 - kinematics) allows now to compute the Transformation matrix (*Tnew*) of the position the robot should occupy. But before really modifying the position we must check that no collision actually occurs between the robot and other robots or with fixed obstacles.
- The first operation consists in computing the characteristics of the bounding cylinder for the new coordinates of the robot.
- With this information we can now check for collision by using methods presented in section 6.
- If no collision is detected then the Transformation matrix of the robot (*robotTransform* – see figure 5) is updated with the values of the previously computed *Tnew*. This value is used by the Java3D rendering engine after the execution of all the Behaviours' *processStimulus* methods.
- In case of collision, the robot is stopped, that is its Transformation matrix is left unmodified and the current speeds are set to zero.
- The last operation concerns the internal timing mechanism. It resets the *WakeUp* criterion so that the *processStimulus* will be executed at the next time event (see section 4.3).

9.2 Sensor Engine

Figure 28 illustrates the algorithm for the laser sensor. After having obtained the Transformation matrix of an individual laser beam, the *startpoint* and the *direction* are computed and used by the collision detection function. If the laser beam hits an object, the distance is obtained via the reference to this object.

The minimum distance is computed for this sensor and used to update the geometrical representation of the beam. These operations are repeated for all sensors. After the loops have been executed, the distance array contains the measured distances.

These data can be retrieved by an external Sensor component by invoking the operation *get_data()* on the instance of the *LaserImpl* class.

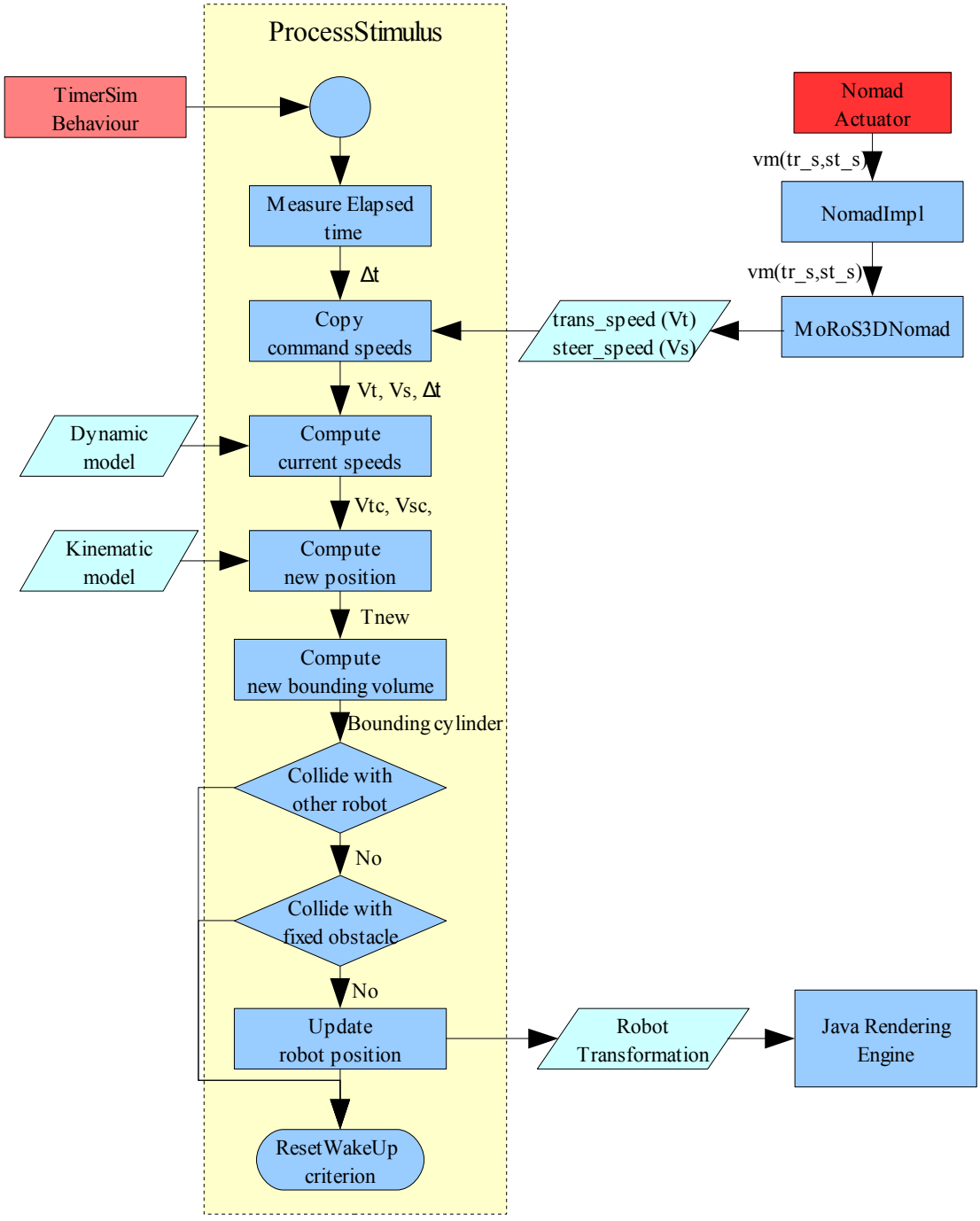


Figure 27. Motion Control algorithm of the simulator

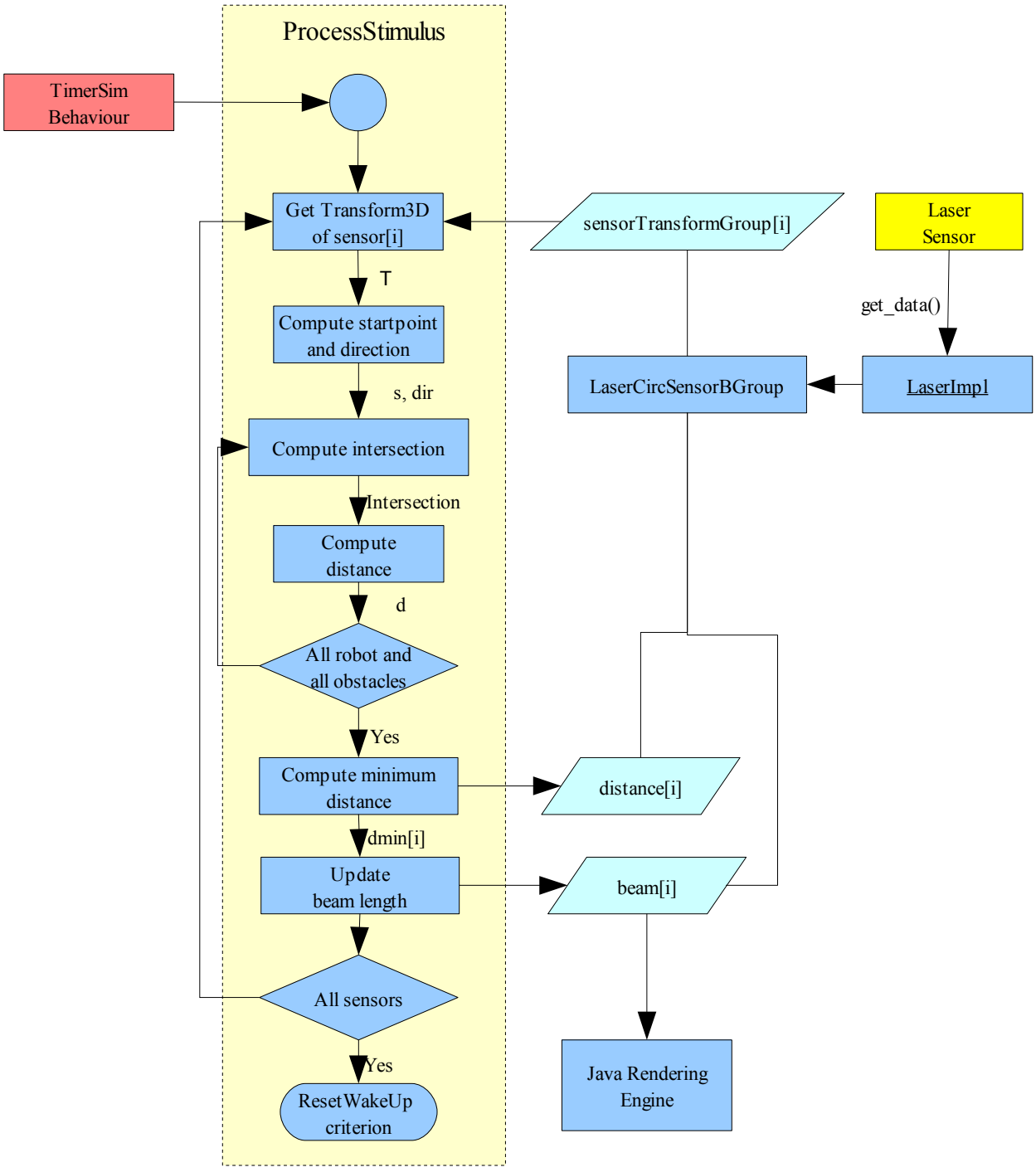


Figure 28. Sensor control algorithm of the simulator

10 Summary

This chapter presented MoRoS3D, a multi-robot and sensor simulation application that simulates 3D environment for developing mobile robots applications. This simulator is versatile enough to simulate different types of robots.

The 3D scene modelling and rendering is based on Java3D and is therefore platform independent. It is extensible and users can easily change the environment (terrain and obstacles) as they are passed as command line parameters, they do not need to recompile the application.
Finally, it integrates seamlessly into the CoRoBA framework thanks to the CORBA middleware.

Despite this tight integration, the simulator can also be used independently of the framework and control applications can be written in any language supporting CORBA interfaces.

Keeping the control algorithms out of the simulator has the advantage that an application developer does not need to deal with Java3D programming.

Chapter VI Validation and Evaluation

1 Introduction

This chapter is devoted to the validation and evaluation of the framework. Validation relates the developed software to the requirements while evaluation aims at making a qualitative and quantitative judgement over the performances of the framework.

In the first part of this chapter, we relate the requirements presented in the first two chapters with the actual framework implementation:

- We compare the characteristics of CoRoBA with the definition of a framework.
- We review the list of requirements that have been met and we propose solutions to implement requirements that have not been addressed in this work.

The second part of this chapter presents applications that have been developed to validate the functionality of the framework and the simulator presented in the two previous chapters.

The last part deals with the evaluation of the framework. In the qualitative evaluation:

- We explain how applications presented in the introduction could be improved by using CoRoBA.
- We compare CoRoBA with other frameworks.

On the other hand, for the qualitative evaluation we have defined and applied evaluation criteria and measures of effectiveness.

2 Theoretical validation

2.1 Framework definition

In the motivation section of the first chapter we pretend that what is required is "a software framework that enables the easy development of distributed applications...". That is what has been developed and presented along the different chapters. We come back to the definition of a framework:

> A **framework** is an integrated collection of classes that collaborate to produce a reusable architecture for a family of related applications. It is a design and an implementation providing one possible solution in a specific problem domain. A framework is a reusable, "semi-complete" application. It provides generic components which generaly need to be customised and extended in function of the application.

For each statement underlined in this definition we show the solution proposed by CoRoBA:

Collection of classes:
The units are programmed in C++ and are off course organised in classes with base classes providing a skeleton implementation for the components.

Reusable architecture:
What is provided is effectively an architecture because it is composed of families of related patterns and components. CoRoBA is based on the Design Patterns presented in Chapter III. Execution units deliver services and are reachable through well-defined interfaces and can consequently be called components.

Family of related applications:
Distributed control applications and sensor networks are family of applications targeted by CoRoBA.

It is a design and an implementation:
Besides the design that has been presented in Chapter IV, an implementation is also proposed. This chapter presents applications that have been developed with the framework.

It is a reusable, semi-complete application. It provides generic components:
Existing components can be directly reused as is, as long as they conform to the required interfaces. The developer has naturally to fill in empty components with code specific to the application. Generic services are available to support any application (Naming Service, Notification Service, Logging Service, Name Manager, remote control component, ...).

2.2 Review of the requirements

R1: Stability and reliability

The skeleton implementation of components has been tested extensively and components have run during several days without suffering from any stability problems or memory leaks. The communication relies on an extensively tested library (TAO).

R2: Modularity

The definition of hierarchical component interfaces and the fact that one component implements a single interface make the framework largely modular. The evaluation of the modularity can be expressed in terms of reuse. It is shown in the section 3 that applications can be built incrementally by reusing components. Actually most of them can be reused as is, as long as they conform to the required interfaces.

R3: Scalability

The TAO CORBA implementation is a high efficient communication library that has been designed to handle thousands of communication at the same time. TAO has been designed carefully using architectural, design, and optimization patterns that substantially improve the efficiency, predictability, and scalability of communication systems. TAO's ORB Core concurrency models are designed to minimize context switching, synchronization, dynamic memory allocation, and data movement.

One possible bottleneck in the current implementation is the Notification Service. Each component has to contact this service in order to connect to an Event Channel. That means that all data transit by this service that redistributes them to the connected clients. If the number of components and the traffic increase too much the Notification Service could become the bottleneck of the application. One solution to reduce this potential limitation would be to use several Notification

Services. This would necessitate to modify the implementation by allowing a component to be able to find and narrow different Notification Services.

R4: Native libraries

As the framework is implemented in C++, it is straightforward to integrate native libraries that are generally developed in C or C++.

R5: High level communication library

The choice of TAO the implementation of the communication has allowed to fulfil this requirement.

R6: Asynchronous communication

The TAO implementation of CORBA proposes an efficient interprocess communication library that allows synchronous and asynchronous communication (Event based communication) thanks to the Event Service and the more advanced Notification Service.

R7: Multi-threaded applications

TAO is based on the ACE library that offers support for easy development of multi-threaded components and their synchronisation.

R8: easy distribution of applications

The NameService defined by the CORBA specifications allows a client to discover and locate CORBA objects at run-time. This capability along with the implementation of a component model provide all the elements to distribute an application over different nodes.

R9: Definition of the application architecture at run-time

The links between the components is provided as command line parameters (Event Channel id., the NameService id., etc.) when components are started. As far as components support the same interface, there are perfectly interchangeable.

R10: Process synchronisation

In the framework the synchronisation issues are actually embedded in the data flow and the component network. Examples are given in section 3 of this Chapter. Another possibility is to use a global time reference. This capability is for instance provided by the CORBA Time Service.

R11: Free choice of the communication model.

An application developer can choose the communication model (synchronous or asynchronous) in function of its needs as this is not imposed by the framework. He can select classical two-ways operations defined in IDL files or an event driven model provided by the Notification system. The first model is privileged by client-server paradigms while the second method promotes decoupling between processes.

R12 and R13: Control is distributed over different users

These requirements are solved by the distribution of components and nothing precludes different users of interacting with the application. Each user can control a robot or a functionality of a robot with its own UI (Joystick, mouse, ...).

R14: The code must be portable

Portability is assured by using only standard C++ libraries. TAO is a very portable library that runs on more than 20 different platforms/compilers.

R15: Different programming languages can be mixed in an application

The framework has been implemented in C++ and the simulator in Java. It could also be possible to use other CORBA compatible languages to develop clients, GUI, simulators...

R16: The framework implementation should not constraint the development of applications' UI's.

The choice of CORBA as communication library lets the developer with free hands for writing UI's. UI components can connect to Event Channels and send or receive data from/to other components.

R17: Robot control GUI's must be independent of the robot.

This requirement can be met by sending only abstract motion commands to the robots. Each robot has then to interpret these commands for its own. A Joystick Sensor and a console control command component have been implemented to illustrate this principle.

R18: The GUI's must be platform independent.

GUI's can be written in platform independent languages for which there exists a CORBA mapping (C, C++, Java, Python, ...). The GUI's can be made platform independent if a multi-platform language (Java, Python, ...) or a multi-platform API is used (wxWidgets, QT, ...).
No specific GUI has been developed in the work.

R19: The GUI's must display and save data in various format.

This requirement has not been addressed in this work.

R20: The GUI must be independent of the application and must adapt itself to the application capabilities

This requirement has not been addressed in this work.

R21: It must be possible to select a monitored service at run-time.

This requirement has not been addressed in this work.

R22: Tools are required to facilitate the development of new components.

Template sand wizards are usual tools that are provided to facilitate application development. There is no such tool for CoRoBA at the moment. However, as all components have a similar structure , it is very easy to start from an example, to clone and modify it, in order to develop a

new component or to alter its original behaviour.

R23: A simulator is required in order to perform extensive tests of components

The 3D simulator that has been presented in Chapter V has been developed in order to test components.

R24: Tools must be provided to facilitate the integration of existing components into applications.

What is needed to fulfil this requirement are editors to graphically connect components and graphical applications to manage the life and run-cycle of component. In the current version, only a command line application (CRC- see Chapter IV) is provided to manage components.

R25: The command and visualization data flow must be separated from each other.

By defining three classes of components, CoRoBA provides a clear separation of functionality. The components are generally connected together to form a closed control loop. The command and visualisation data flow transit by different Event Channel and are consequently not tightly coupled.

R26: Motion commands must be independent of the robot

This requirement can be met by sending only abstract motion commands to the robots. Each robot has then to interpret these commands for its own. The demonstration applications described in the next section illustrate this principle.

R27: In order to coordinate control actions, communication between operators may be required.

This requirement was out-of-focus of this thesis.

R28: High quality documentation of the system design, implementation, development and use

This text fulfils the first two aspects of this requirement. The Doxygen application can be used to automatically generate documentation about implementation. A tutorial document about the usage of CORBA communication models with TAO and JacORB has also been written [COLO06a]. Another report describes the software required to develop applications with CORBA as well as its installation[COLO06b].

R29: Monitoring tools

Monitoring provides runtime information that is dealt with at run time and that captures the state of the system as a whole. The CRC application can be used to configure/check component configuration, the JacORB *namemanager* application can be used to read the contains of the name service and allows knowing if a component is running, which interface its objects implement and on which machine it is.

R30: Logging tools

Logging capabilities is based on the Logging Service. Each component and Event Channel can log information containing the data transferred and time stamps. This makes possible to trace message in order to analyse data processing and transmission, to post-process data off-line or to

debug applications. This tools have been used to display information transferred in demonstration applications presented in section 3.

From this review we note that most of the requirements that have not been addressed in this work are related to the GUI and the security. CoRoBA does not impose any GUI for data visualization or to let the user enter information or commands. Actually the framework implementation does not restrict the developer in its choice. A platform specific (MFC on windows) or a platform independent widget API like wxWidget or QT[25] could be used for developing GUI's. Components can also be developed in platform independent languages like Java or Python, taking profit of their graphical libraries.
Security aspects have been presented in Chapter III. The CORBA Security Service provides security for applications and users and can be added to an ORB in a non-intrusive manner because it is implemented with interceptors [SCMI00]. Existing distributed applications that make use of a normal CORBA ORB can thus run without alteration using a secure ORB. The Secure Socket Layer can also be easily integrated in any CORBA application.

3 Validation through applications

This section begins with a discussion over components integration and continues with typical control applications operating in simulation. Components involved in each application are explained in detail and reuse of components is emphasized throughout this chapter. Finally an on-going development that consists in porting an existing teleoperation application to the framework is presented. Multi-robots, distributed simulation and real robots are also covered in this chapter.

3.1 Components integration

In control applications we mainly integrate existing applications and libraries. When integrating libraries, there are two important points that should be considered: if the library is not written in object-oriented languages, it is necessary to evaluate the degree of coupling between the functions and if the library is not thread-safe, integrity will need to be ensured by providing locks. Integration with an existing application is generally a hard work unless some form of communication mechanism is available. Sometimes it is simpler to rewrite the application than trying to integrate it into the distributed architecture. Developing with a framework facilitates the integration and the reuse of components as it will be demonstrated in this chapter.

As presented in the previous chapter, the components composing the CoRoBA framework are divided in Sensors, Processors and Actuators that form a chain along which information is transferred (Figure 1). Like in classic control schemes, the data flow is unidirectional.

We give now examples of the three component categories, Sensors, Processors and Actuators whose architecture has been presented in Chapter V.

25 QT is a GUI software toolkit developed by Troll Tech and available at http://www.trolltech.com

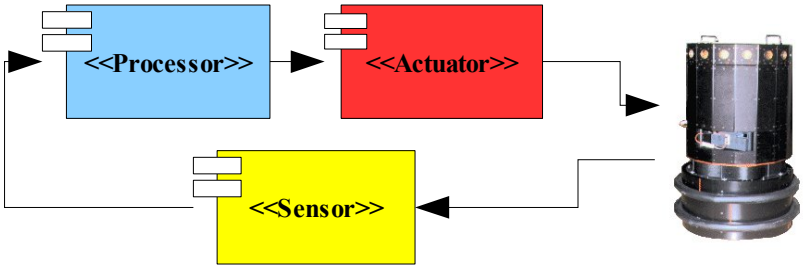


Figure 1. Closed-loop chain of Sensor, Processor and Actuator components

3.1.1 Sensors

There exist numerous useful sensors that are used in robotic applications. They can be grouped in different categories:

- Motion and attitude (encoders, gyroscope, angle, tilt, accelerometer, ...)
- Global Position sensors (GPS, Compass, triangulation...)
- Distance sensor (US, laser, radar, ranging)
- Vision sensors (video, IR, UV)
- Users' input (Joysticks, mouse, Haptic interfaces, ...)
- Other sensors: temperature, pressure, force, microphones, gas, light, humidity, tactile, switches...

In order to validate the proposed framework different sensor components have been implemented. Some are adapters to real sensors while other are linked to virtual sensors in relation with the simulator presented in Chapter V. These components are listed in the table below and covered in the next sections.

Table 1: Sensors integrated in the framework

Category	Type	Robot	Simulation/Real
Motion	Wheel encoders	Nomad, Robudem	Simulation and Real
Global position	GPS	-	Simulation
	Dead reckoning	Nomad	Real
Distance	Laser and Laser line scanner	-	Simulation
	Ultrasonic	-	Simulation
	Ultrasonic, Infra-red and bumpers	Nomad	Real
User input	Joystick	-	Real

A GPS and a Laser sensor have been recently acquired and will be integrated in the framework.

3.1.2 Actuators

This work targets the integration of multi-robotic systems and consequently most of the integrated actuators are actually robots. Other kind of actuators that could also be integrated within the

control framework are for instance motors, solenoids, speakers, grippers, linear actuators, ...
Two simulated and real robots actuator components have been implemented: the Nomad and the Robudem
However, Actuator components are not limited to mechanical systems but also abstract all components that produce output data. Displaying images coming from a camera (with or without pre-processing) or updating data information in a GUI or in the simulator are examples of operations that can be done with Actuators.

3.1.3 Processors

Processors are the keystone of any control architecture. While Sensors provide data to the Processors and Actuators forward data to output systems, Processors are the intelligent part of the network. Processors are intended to be reused with different robots while Actuators and Sensors can be reused in different applications involving the same hardware. The more abstract the function of the Processors the greater the possibility that a component can be reused without being modified. Thanks to the modular architecture of CoRoBA, switching between simulation and reality only requires changing Sensor and Actuator components, while Processor components stay unchanged.

In order to demonstrate the capabilities of the framework processors have been developed for the following applications

- Shared control and obstacle avoidance applied to the Nomad.
- Goal navigation in an obstacle free environment applied to the Robudem.
- Behaviour based Fuzzy Logic navigation applied to the Nomad.

Other kind of processors that could be implemented are for instance:

- Control algorithms: PID, Fuzzy logic, kinematic inversion,...
- Path planning, trajectory computation, ...
- Data processing (vision, ...), filtering, ...

3.2 Control Applications

This section presents applications that have been developed with CoRoBA and tested with the simulator. These applications are representative of typical robotic applications and implement two different modes of supervised and autonomous control as described in Section 3.6 of Chapter II. They have been developed to explain how to use the framework and to prove its reliability. To conclude this section, multi-robot applications and distributed simulation are discussed.

3.2.1 Nomad Shared control

Description

This application demonstrates how to use CoRoBA for implementing shared autonomy It uses two Sensor components: a Joystick component and a Sim_Laser component that reads laser distance data, one Processor, named Avoid and a Sim_Nomad Actuator. Figure 2 represents the application structure, the relation between the components and the information that they exchange.
The user gives general motion commands [x y] with the joystick and the Processor combines these commands with the distances measured by the Laser [d1 d2 d3...] in order to avoid collision. It produces speed commands [Vt Vs] that are sent to the Sim_Nomad Actuator. This component is in charge of sending the speed commands to the robot in the simulator.

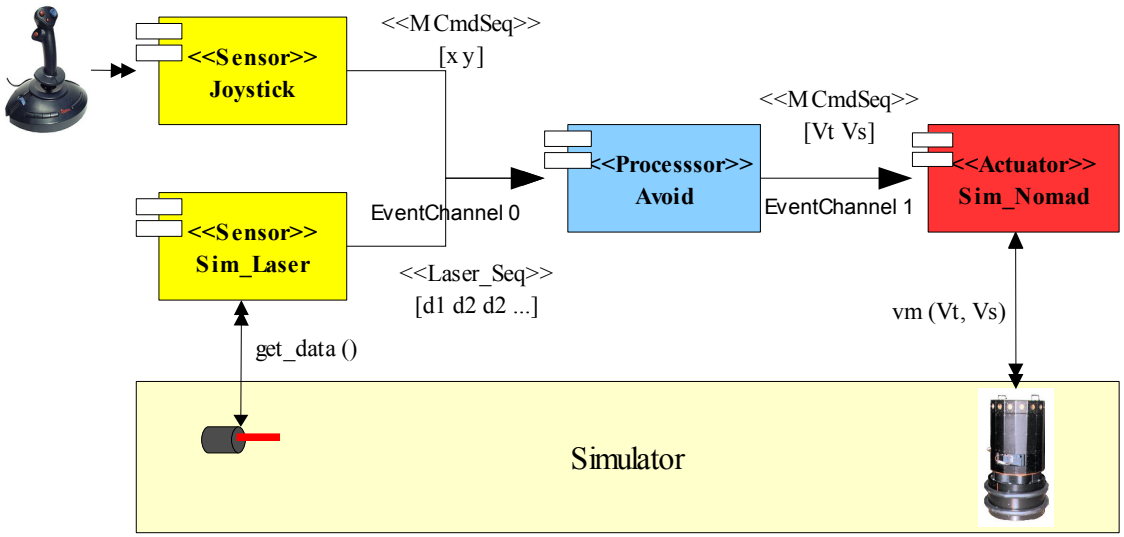


Figure 2. Shared control application structure

The screen capture in Figure 3 shows the simulator GUI and the console outputs of the different components. The yellow windows are Sensor components (a third sensor is used for recording the robot kinematics data but it does not play any function in the actual application), the blue one corresponds to the Processor and the red one to the Actuator.

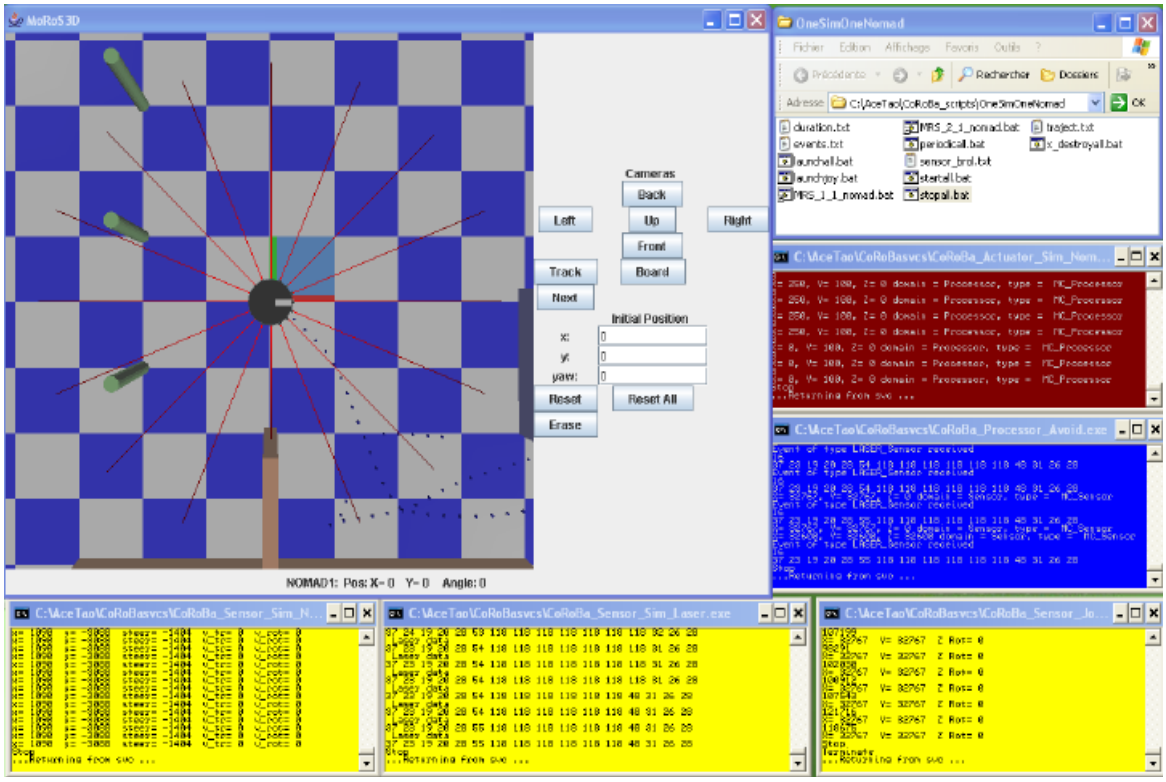


Figure 3 Screen output of the application

To perform the simulation, a number of scripts are launched to ensure the communication between the different programs, the robot and the environment in which it navigates.
First, a script named ”ns” (or NameService) is launched to create the process and to name the computer. The second script to launch is ”not” (or NotificationService). This script launches the Event communication server. The simulator is then launched by running a script ”Nomad_1_1”. This script launches the graphical user interface of the simulator. After that, the script ”launchall” is run. This script launches the different CoRoBA programs. The script ”startall” starts the execution of the different processes. The robot moves according to the control commands sent by the Actuator component. To stop the simulation, the script ”stopall” is run, then ”xtroyall” closes all the programs.

Components

Sensor Joystick

In order to promote re-usability, the idea is to define motion commands that are independent of any robot geometry and kinematics. Each Actuator component will actually have to interpret them for its own.

The general structure of events is defined by the CORBA specifications [CORBA00]. The values of the fixed Header for the Joystick component are listed in Table 2. By default the Domain is set to GLOBAL. Processors and Actuators use the Domain and Type information during the registration with the Event Channel to define the events they want to receive (see section 5.3.4 – Event registration in Chapter IV) .

Table 2: Motion_Command events structure

Header	Fixed Header	Domain	GLOBAL
		Type	MC_Sensor
		Name	Joystick
	Variable Header		-
Data	Filterable data		-
	remainder of body		MCmdSeq

The actual joystick data is contained in the “*remainder of body*” field. Two CORBA types have been defined for representing the data: *MotionCommand* that contains individual values and *McmdSeq* that is a sequence of *MotionCommand* instances. A Sequence is a special CORBA format that can contain any element type and can be bounded or unbounded. The length of the sequence can be determined at run-time. The sequence type offers more run-time and development flexibility in regard to arrays, which are fixed-length [HENN99].

The motion command values range from 0 to 65535 and vary as showed in Figure 4. The component is programmed to control three degrees of freedom but it could be extended as necessary and in function of the axis available on the joystick.

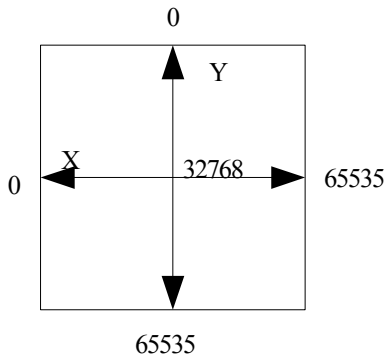


Figure 4. Values generated by the joystick X en Y axis

When this component starts, a separate thread is created that runs the method *svc* .This structure is common to all components (section 4 – Chapter IV). At each iteration, the method *process* is executed: the joystick values are read, transformed to events and sent to the appropriate Event Channel (Figure 1). The details of the event production mechanism can be found in the section 5.3.4 - Production of Events of Chapter IV.

Sensor Sim_Laser

This sensor invokes operations on the Laser object of the simulator to receive distance measurements from the simulated laser sensor as explained in the section 8 of Chapter IV and illustrated by Figure 2.

Specific data structures have been defined for containing the simulated laser distance sensor data. Because the number of sensor can be modified in the simulator, a sequence data type has been used again. The event type is *Laser_Sensor* and the data format is *Laser_Seq*.

The component contacts the NameService to get the IOR of the simulated laser. This process is explained in the section 8 (Location of components) of the Chapter IV. The method *process* invokes the method get_*data* of the Laser object in the simulator, puts the information into a *Laser_Seq* variable and sends the event to the Event Channel 0 (Figure 2).

Processor Avoid

This processor is intended to work with the Nomad or other robots having an equivalent geometry. It is a pure reactive component whose purpose is only to avoid obstacles. We suppose that the environment is perceived through distance sensors that are uniformely distributed around the robot (only the five front sensors are actually used). This component also receives motion commands from another component (here a Joystick Sensor).

In this shared control mode the Processor Avoid has the responsibility of the local navigation. To accomplish this task an obstacle avoidance controller based on command fusion mechanism is included in the Processor. The output from the distance sensor is combined with the input direction from the operator. This combination should be seen as a weighted sum of two vectors:

- the operator's reference command F_t
- the obstacle avoidance feedback F_r generated by the distance sensor signals.

$$F_{res} = F_t + F_r$$

The repulsive force is active when the distance (d) to the obstacle is smaller than *1* meter. It can be seen as the force generated by a spring with a coefficient K being chosen so that the robot can move up to the obstacle with a decreasing speed and stops when it comes in contact with the obstacle.

$$F_r = K * (1 - d)$$

The steering of the robot is aligned with the direction of the resultant vector F_{res} and yields continuous and smooth motion (see Figure 5). In the absence of obstacles, $F_r = 0$, the robot follows the operator's directions. If the robot approaches an obstacle, F_r gradually increase in magnitude and cause a progressive avoidance manoeuvre. This gradual shift in control is completely transparent for the operator. Depending on the configuration of the obstacles the robot slows down, goes back if it is too close of an obstacle or turn right or left.

Input events are of type *MC_Sensor* and *Laser_Sensor*, both are in the *Nomad1* Domain. When events are received by the consumer object via the ORB, they are transferred, to *Processor_avoid_i* class via the transfer_event method where data is extracted from the received event and stored in local member variables. This transfer is protected by mutexes because the data is also accessed asynchronously by the *process* method. More details about this mechanism can be found in the section 5.3.4 – Reception of Events of the Chapter IV. The Type of the output events is *MC_Processor*. The component produces motion command sequences (*McmdSeq)*.

Nomad Actuator

The Nomad Actuator component actually receives events from motion command Sensors like the Joystick component described above or from a navigation Processor like the Processor Avoid used in this application. The formats for the motion command data are obviously the same as the ones defined for the joystick sensor, *MotionCommand* and *McmdSeq*.

As input events can come from two different kind of components, namely Sensors and Processors, this component registers its interest for two different events when subscribing with the *consumer_admin* of the Event Channel (Domain Nomad1, Type *MC_Processor* and *MC_Sensor*). Both events are propagated by the Event Channels and received by the consumer object of the Actuator component (section 5.3.4 – Reception of Events of the Chapter IV).

The origin of the event is determined in the method *process* allowing a different processing in function of the data origin. The length of the command sequence is also tested because it could contain 2 or 3 parameters and a correct data handling is required in both cases.

The generic motion commands are also adapted to the kinematics of the robot according to the following equations:

$$t_{vm} = \frac{-255 * y}{127} + 255$$
$$s_{vm} = \frac{-255 * x}{127} + 255$$

where x and y are the motion command values coming from the Processor that vary from 0 to 255 and t_{vm} and s_{vm} are the desired translation and, base rotation speeds.

After this adaptation the data is sent to the simulated robot (Figure 2) by invoking the *vm* method of the Nomad Interface (see section 8 – Chapter V). The method *terminate* stops the robot by sending null speeds.

Results

Figure 5 shows the robot avoiding the wall while the input command corresponds to a straigh motion.

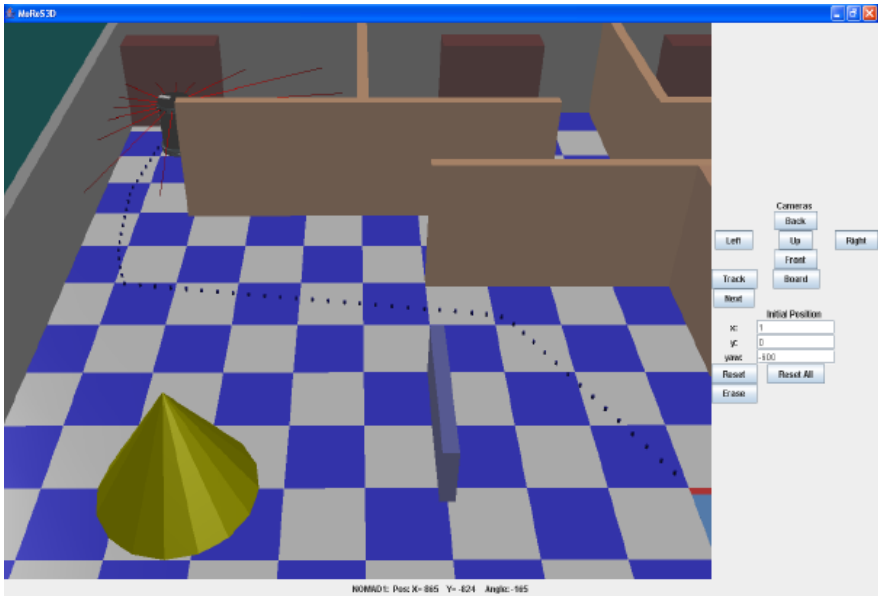


Figure 5. Avoiding manoeuvre

The following graphics show the data produced by the Joystick, the measurements of the 3 front sensor of the Sim_laser and the motion commands produced by the Avoid component. The last graphic depicts the trajectory followed by the robot.

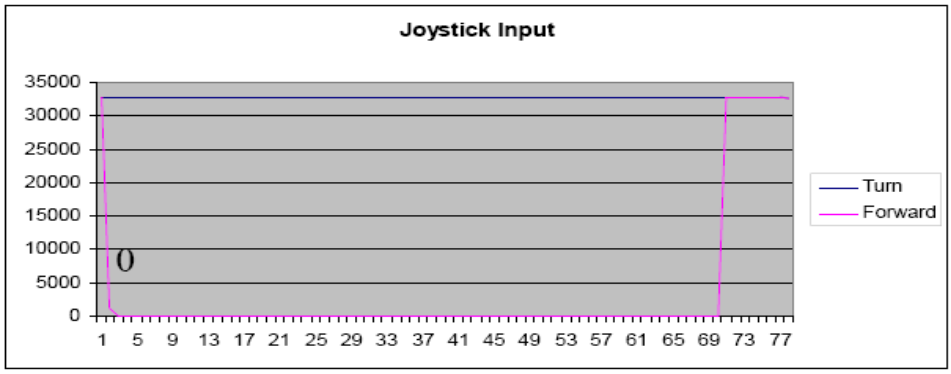


Figure 6a.

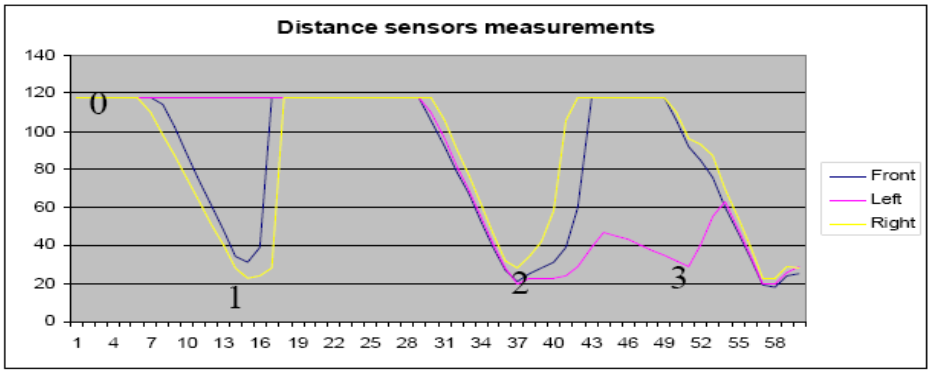


Figure 6b.

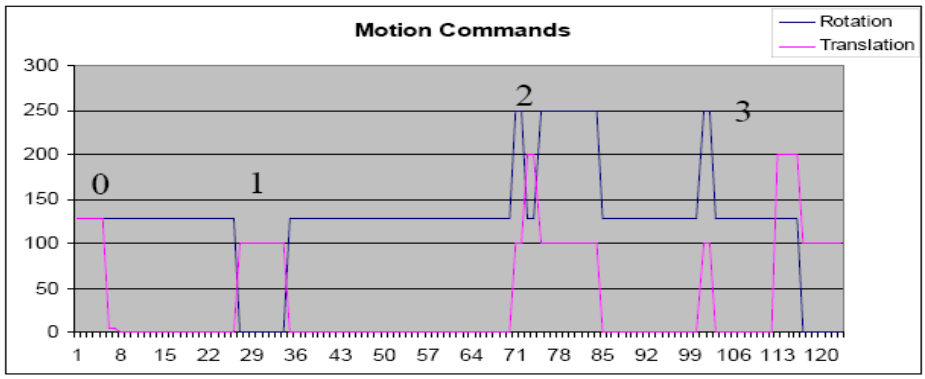


Figure 6c.

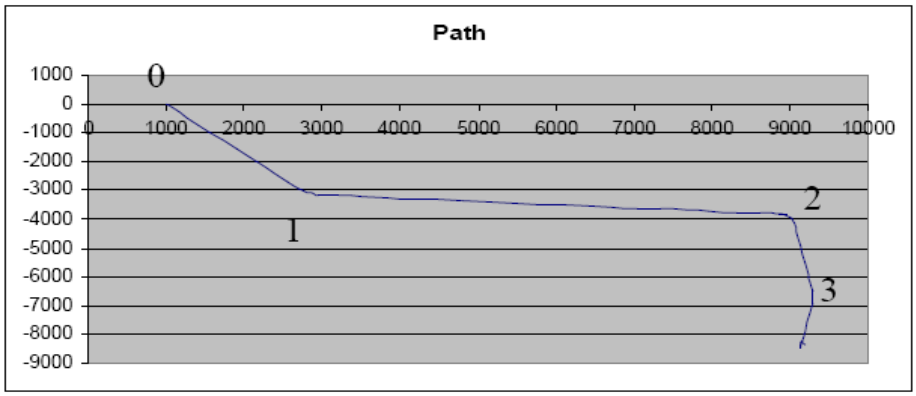


Figure 6d.

The sequence of events for the starting phase is showed in Figure 7. Numbers correspond to the sequence id's as explained below.

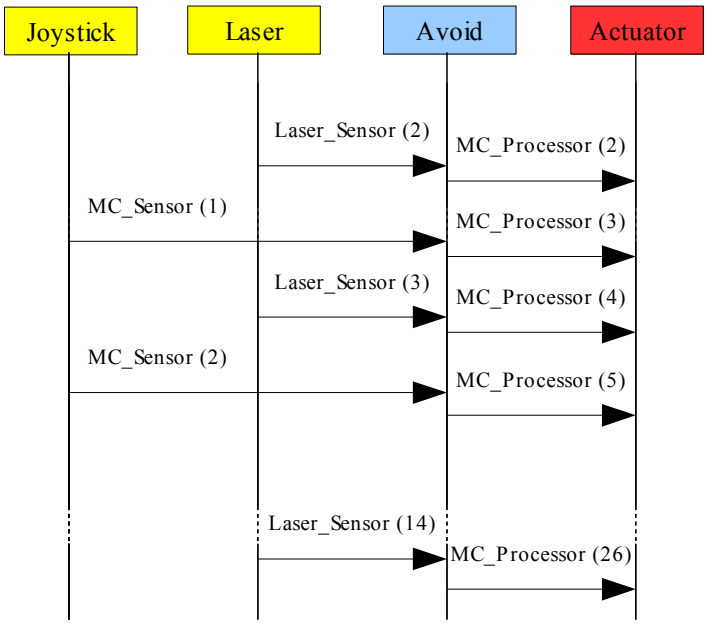


Figure 7. Sequence diagram of events

The name of the components is in italic and the event type is in bold. The elements of the recording have the following meaning:

> Log Id, time of log in milliseconds, Cluster name, **Event Type**, *Component name*,Time of event transmission in seconds and microseconds, event id, data.

Rem. Cyan and Magenta colours correspond to the colours of the curves in the graphics. Green indicates the Events Id.

The joystick becomes active from the second sample. (Marked as 0 in Figure 6a).

> 1, 800236, GLOBAL, **MC_Sensor**, *Joystick*, 1154000795, 850545, 1, 32767, 32767
> 2, 800767, GLOBAL, **MC_Sensor**, *Joystick*, 1154000800, 737572, 2, 32767, 1279
> ....

The Processor reacts to the command input variation (Figure 6c) and the robot begins to move. This is visible on the data below: the motion commands vary from 127, 127 to 127, 4. This sequence corresponds to the upper part of Figure 7.

> ...
> 3, 799816, NOMAD1, **MC_Processor**, *Sensor_Sim_Laser*, 1154000799, 776190, 2, Processor_Avoid, 1154000799, 786204, 2, 127, 127
> 4, 800226, NOMAD1, **MC_Processor**, *Joystick*, 1154000795, 850545, 1, Processor_Avoid, 1154000800, 226838, 3, 127, 127
> 5, 800336, NOMAD1, **MC_Processor**, *Sensor_Sim_Laser*, 1154000800, 326982, 3, *Processor_Avoid*, 1154000800, 326982, 4, 127, 127
> 6, 800747, NOMAD1, **MC_Processor**, *Joystick*, 1154000800, 737572, 2, *Processor_Avoid*, 1154000800, 737572, 5, 127, 4

At the point 1, the robot approaches the wall and the distance sensor values decreases. At the Laser sample 14, the right sensors makes the Avoid Processor reacts. The robot turns to the left (Point 1 on Figure 6). This sequence corresponds to the lower part of Figure 7.

14, 806255, NOMAD1, **Laser_Sensor**, *Sensor_Sim_Laser*, 1154000806, 215449, 14, 34, 118, 118, 118, 118, 118, 118, 118, 118, 118, 92, 89, 69, 38, 29, 28

27, 806245, NOMAD1, **MC_Processor**, *Sensor_Sim_Laser*, 1154000806, 215449, 14, *Processor_Avoid*, 1154000806, 215449, 26, 0, 100

At points 2 and 3 , the left sensor detects the wall and consequently the robot turns to the right.

These results have been obtained with the Sim_Nomad in Periodic mode (100 ms) and the other components in Synchro mode. The data has been recorded with the Log mechanism explained in Chapter IV, section 6.2.

3.2.2 Robudem Autonomous navigation

Description

The purpose of this application is to let the Robudem move autonomously from a given position to succession of goals in an obstacle free environment.
The application comprises the following components:

- Sensor: Sim_Robudem
- Processors: Goal_Provider, Goal_Controller, Goal_Scheduler
- Actuators: Sim_robudem, Goal_Display

The components involved in this application and the transferred data are shown in Figure 8 .

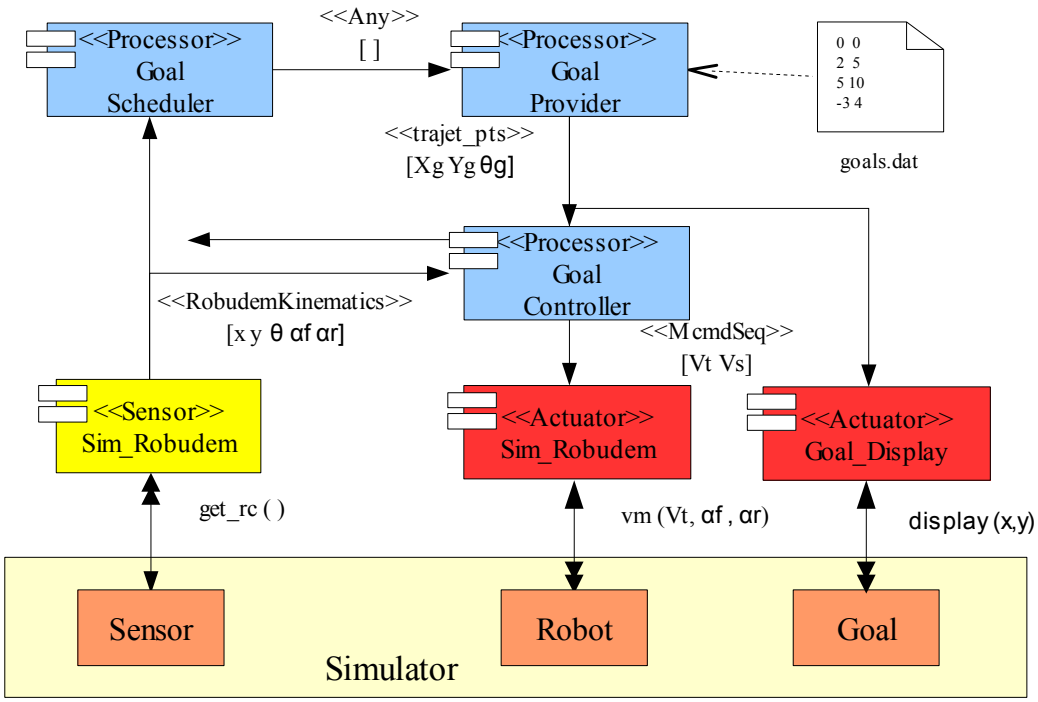


Figure 8. Components and data flow of the Robudem Navigation application

Components

Sensor Sim_Robudem

This component connects to the Robot object of the simulator to read the pose and kinematic data of the Robudem. The event Type is *Robudem_MotionSensor* and the event data is of type *RobudemKinematics*, a specific structure defined to store the kinematic data of the Robudem.
The component implements the inherited virtual methods *process* and *terminate.* The method *process* invokes the method *get_rc* on the Robudem object in the simulator that returns the pose and kinematic data of the robot.

GoalProvider

This component is intended to be used with the GoalController and GoalScheduler components described below. When this component is started, it opens a file, reads a list of points and sends the first goal as an event to the Event Channel it is connected to.
This component may receive two types of events: *GoalScheduler* and *GoalProducer* (Figure 9). Receiving an event of type *GoalScheduler* means that it has to send the next goal. The type *GoalProducer* is reserved for future use where goals could be sent by a goal producer component implementing a path planning algorithm. Output events have the type *Goal*.
The method *transfer_event* (section 5.3.4 - Tranfer of Events of Chapter IV) is overloaded and implements the logic of the goal sequencing. When an event of type *GoalScheduler* is received, the private method *next_goal* is called. This method replaces the current goal with the next one. After that the method *process* formats the data and sends it to the Event Channel.

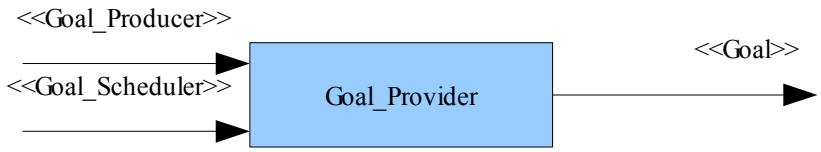


Figure 9. Events received and sent by the Goal_Provider Processor component

GoalScheduler

The role of the *GoalScheduler* is to compare the goal position with the current position of the robot. When the robot has reached the goal an event is issued from the *GoalScheduler* to the *GoalProvider*. The *GoalScheduler* receives two types of events: *Goal* and *Robudem_MotionSensor* and sends events of type *GoalScheduler* (Figure 10). The method *transfer_event* is overloaded because this component receives two kinds of events, namely *Goal* and *Robudem_MotionSensor*. The comparison between the current position and the goal is performed in the method *process*.

Figure 10. Events received and sent by the Goal_Scheduler Processor component

GoalController

The GoalController issues motion commands to the Sim_Robudem component taking into account the current position of the robot and the goal. A fuzzy controller has been implemented for this purpose.

The variable inputs are the distance and the angle between the robot current position and the goal position. A classical Suggeno method is used to obtain the direction and speed of the robot. The method transfer_event is overloaded because this component receives two kinds of events, namely *Goal* and *Robudem_MotionSensor* (Figure 11). The motion commands are generated by the Fuzzy Inference System in the method *process*. The method *terminate* is called when the application terminates; it sends null speeds to the robot to safely stop it.

Figure 11. Events received and sent by the Goal_Controller Processor component

Actuator Sim_Robudem

This component has the same implementation as the Actuator *Sim_Nomad* presented in the section 3.1.2 of this chapter. Motion command transformation equations are similar to those defined for the Nomad. Only the semantic for the motion command parameters is different. The Robudem expects a global translation speed and two directions angles, one for each axle (Actually only the first angle is used for the Single Drive mode as explained in the Robudem Modelling - section 5.2 of Chapter V).

Actuator Goal_Display

This component receives the *Goal* Events from the *Goal_Producer* and invokes the operation *display* of the *Goal* Interface implemented by the class *GoalImpl* in the simulator (This Interface has been described in the section 8 of the Chapter V). Each time a new goal is received, the *Goal_Display* sends this information to the simulator in order to update the display.

Operations

During the execution of the application, the following operations are executed (the steps refer to Figure 12):

- At initialization, the *Goal_Provider* reads a list of goals from a file (goals.dat).
- When the components are started, the first goal position [Xg Yg θg] is sent to the *Goal_Controller* and to the *Goal_Scheduler* (1).
- These components also receive the global position of the robot [x y θ αf αr] from the *Sim_Robudem* Sensor component (2).
- The *Goal_Controller* uses this information to produce steering and driving commands [Vt Vs] in order to reach the goal (3).
- These commands are received by the *Sim_Robudem* Actuator that adapt them to the controlled robot.
- The *Goal_Scheduler* compares the goal position received from the *Goal_Provider* with the

instantaneous position of the robot. When the robot is sufficiently close to the goal, the *Goal_Scheduler* sends an event to the *Goal_Provider* to inform it that it has to provide the next goal (4).

- A new goal is sent to the *Goal_Controller* and *Goal_Scheduler* (5).
- These operations are repeated until the last goal is reached.

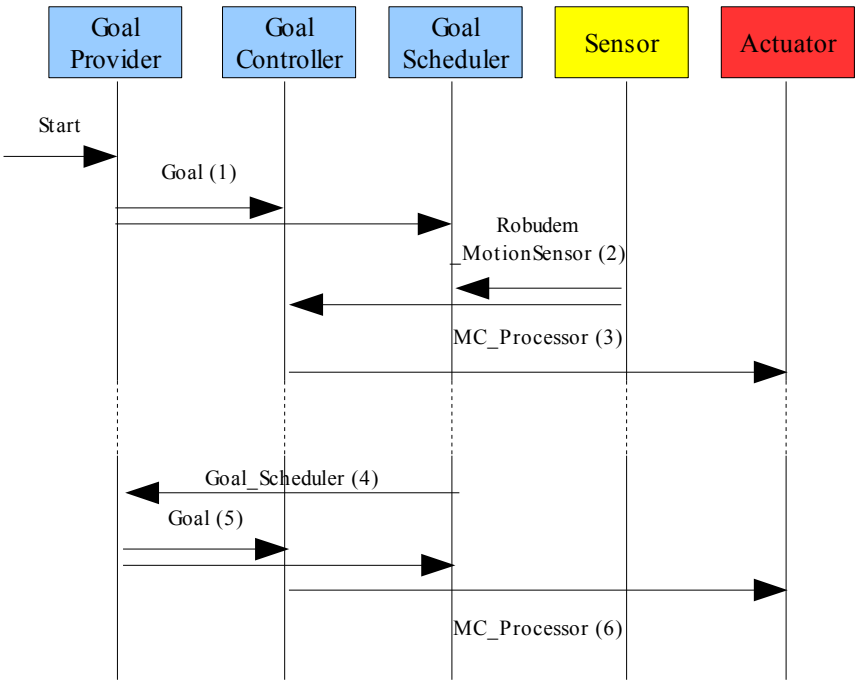


Figure 12. Sequence diagram of events

Events received by the Actuator follow two different paths. The chain can be initiated by an event produced by the Sensor *Sim_Robudem* and processed directly by the *Goal_Controller* (sequence 2 – 3) or pass through the *Goal_Scheduler* and the Goal_Provider before reaching the *Goal_Controller* and finally the Actuator (sequence 2 – 4 – 5 – 6).

The logged data reproduced below illustrates both sequences. The name of the components is in italic and the event type is in bold. Colours indicate the Event Id. The elements of the records have the following meaning:

Log Id, time of log in milliseconds, Cluster name, **Event Type**, *Component name*,Time of event transmission in seconds and microseconds, event id, data.

For the short squence (2 -3):

49, 9886632, ROBUDEM1, **Robudem_MotionSensor**, *Sensor_Sim_Robudem*, 1154009886, 602404, 49, 1480, -197, -79, -1, 0

52, 9886632, ROBUDEM1, **MC_Processor**, *Sensor_Sim_Robudem*, 1154009886, 602404, 49, *Processor_GoalScheduler*, 1154009886, 602404, 52, 129, 123

We can see that the sequence <*Component name,* Time of event transmission in seconds and microseconds, id of event> is repeated for each step of the transmission sequence.
For the long sequence (2 – 4 – 5 - 6):

50, 9887233, ROBUDEM1, **Robudem_MotionSensor,** *Sensor_Sim_Robudem*, 1154009887, 133168, 50, 1482, -198, -80, -5, 0

1, 9887223, ROBUDEM1, **Goal_Scheduler**, *Sensor_Sim_Robudem*, 1154009887, 133168, 50, *Processor_GoalScheduler*, 1154009887, 133168, 1, Nihil

2, 9887143, ROBUDEM1, **Goal**, *Sensor_Sim_Robudem*, 0, 0, 50, *Processor_GoalScheduler*, 0, 0, 1, *Processor_GoalProvider*, 0, 0, 2,, 16.000000, -8.000000, 0.000000

53, 9887213, ROBUDEM1, **MC_Processor**, *Sensor_Sim_Robudem*, 1154009887, 133168, 50, *Processor_GoalScheduler*, 1154009887, 133168, 1, *Processor_GoalProvider*, 1154009887, 133168, 2, *Processor_GoalController*, 1154009887, 183240, 53, 255, 67

The Processor network forms a loop that could lead to problems. When a goal is reached, a **Goal_Scheduler** event is sent to the *Goal_Provider* that sends a new goal to the Goal_Scheduler and the *Goal_Controller*. What happens if an event of type **Robudem_MotionSensor** arrives before the new goal is received? If we don't pay attention this would result in the emission of a second **Goal_Scheduler** event and consequently to the erroneous emission of a new goal.
We can avoid this situation by using flags in the *Goal_Scheduler* component. When a goal has been reached, no new event is sent before the coordinates of the new goal have been received (Event of type **Goal**).

Figure 13 represents a sequence of a typical autonomous navigation of the Robudem where the robot navigates from goal to goal. Each time a goal has been reached, a new one is sent by the Goal_Provider Processor and the display in the simulation GUI is updated via the Goal_Display Actuator.

The sequence has been realised with all Processors and Actuators in SYNCHRO mode while the Sim_Robudem was in PERIODIC mode (see section 5 of the Chapter IV for the definition of the modes). This means that the event transfer sequence is initiated by the sending of the robot position by the Sim_Robudem and the motion command computation is triggered by the reception of this event. Different periods have been tested. As the speed of the robot is small, 1 m/s at the maximum, the period of the components does not need to be too small. Equivalent results have been obtained for periods varying from 50 ms to 1 second.

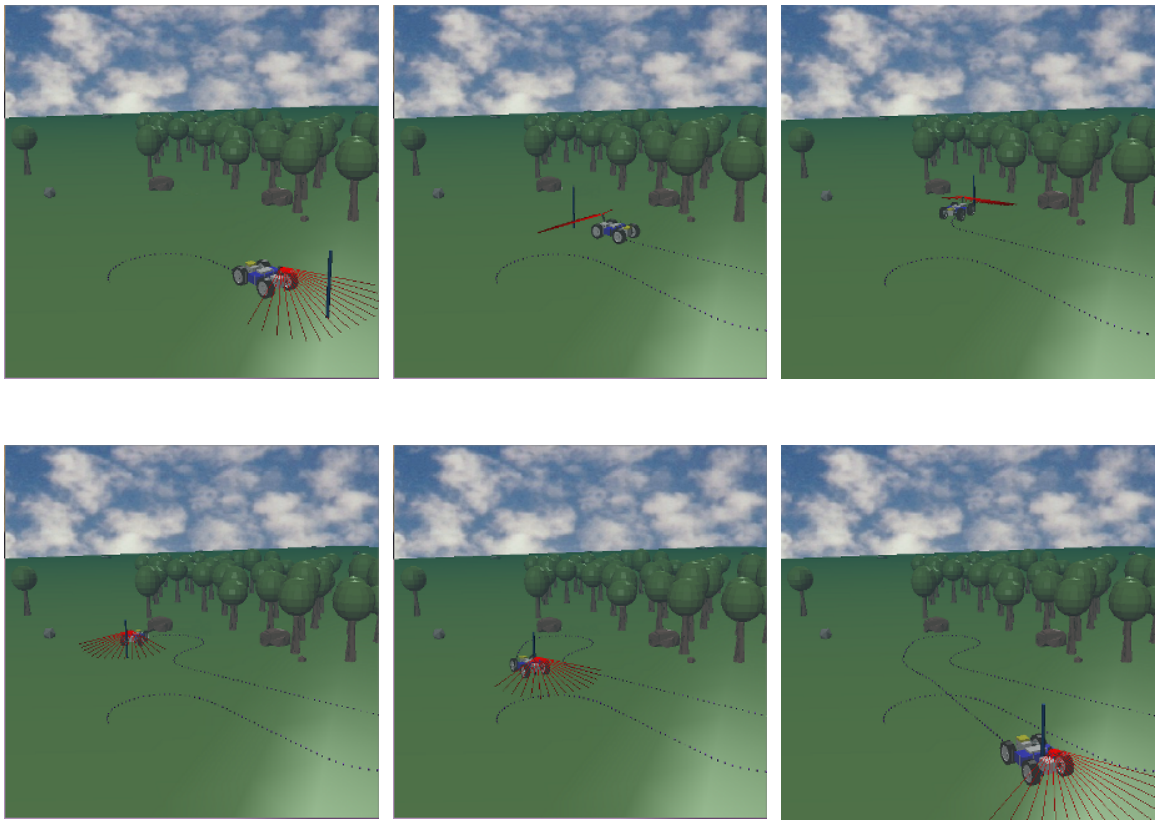


Figure 13. Navigation sequence of the Robudem

If the robot misses a goal the control algorithm tries to drive the robot to the goal again what gives circular trajectories that generally do not allow to reach the point. This situation is detected and in the current version an alert message is displayed on the output console of the component. A better solution would be to report the message to a GUI or to a higher level component that could re-plan intermediate goals.

3.2.3 Nomad Autonomous Navigation

Description

An adaptive fuzzy logic system controls the motion of the Nomad from a start position to an end position such that it reaches its goal position without collisions with obstacles in the workspace. A first approach is based on a simple fuzzy logic system composed of two fuzzy logic controllers: the goal seeking controller and the obstacle avoidance controller. The goal seeking controller tries to find the path to the goal and ignores if it causes collisions, while the obstacle avoidance controller has for mission to avoid obstacles and ignores if it deviates from the goal direction or not. These two behaviours function independently. A command fusion scheme based on a conditioned activation for each controller arbitrates between the two behaviours. As the path obtained by using this fuzzy logic system is not a smooth path, a learning procedure is applied on the fuzzy logic system to optimize the path of the robot.
The following components are involved in this application:

- Sensors: SensorGlueBlock and SensorGoal
- Processor: Pathplanner
- Actuator: Sim_Nomad

The Figure 14 shows the communication architecture of the application.

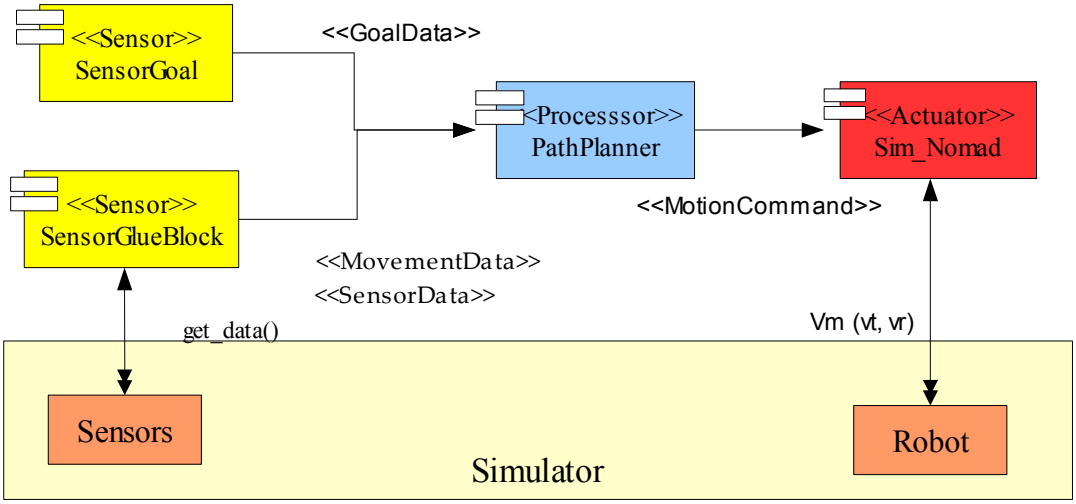


Figure 14. Nomad autonomous application components

Components

SensorGoal

The *SensorGoal* is a simple component with a console input interface that is used to enter the coordinates of the goal the robot has to reach. The coordinates of the goal are relative to the position of the robot. The coordinates are sent by an event of type *Goal_Sensor* and the goal is represented by the *GoalData* structure.

SensorGlueBlock

The *SensorGlueBlock* reads data from the ultrasonic and infra-red sensors as well as the robot position from the simulator and injects them into the communication network. The data of the sensors are sent in separated events. The ultrasonic and infra-red sensor data are sent as sequences of specific structures as depicted by Figure 14.

SimNomad

This Actuator converts the turn angle and the speed received from the Processor *PathPlanner* to motor actions for the simulated robot.

PathPlanner

This component uses sensor and movement data received from *SensorGLueBlock* and the goal data received from *SensorGoalModule* as inputs for the fuzzy logic control of the robot. The system is composed of two fuzzy logic controllers: a Goal Seeking controller and an Obstacle Avoidance controller. The goal direction and the goal distance are calculated using the goal and the robot position received from *SensorGoalModule* and *SensorGlueBlock*, respectively. These variables are

used as inputs for the goal seeking controller. The output of this fuzzy logic controller is a turn angle that causes a deviation in the direction of the target.
Distances to the obstacles detected by the ultrasonic and infra-red sensors are stored in two sequences of 16 components US and IR, respectively. These distances are used as inputs for the obstacle avoidance controller. The output of this fuzzy logic controller is a turn angle to avoid the obstacles on the path of the robot. The two fuzzy logic controllers are implemented using the Matlab fuzzy toolbox. The control command is calculated by the means of a function, called "controle", that returns the output of the Matlab fuzzy toolbox to the C++ code. This command consisting of an angle and a translation speed is then transmitted to the *ActuatorGlueBlock*.

Goalseeking controller

Given the sonar data, the fuzzy controller calculates the turn angle of the robot in order to reach the target. The model of the controller is based on the distance of the robot to the goal and the goal direction. The fuzzy logic controller is of a Sugeno type [SUGG82]. The inputs of the *Goalseeking* controller are the *goal_distance* and the *goal_angle*. The *goal_distance* ranges between 0 and 7 m and the *goal_angle* between −180° and 180°. The output of the controller is the turn angle of the robot, which ranges between −180 ° and 180°.
The first step in defining a fuzzy logic controller is to determine the input and the output variables and map them into linguistic variables linked to fuzzy sets.

The direction of the goal is fuzzified into 11 Gaussian fuzzy sets Back Right (BR), Oblique Back Right (OBR), etc. The membership functions of the first input variable *goal_angle* are depicted in Figure 15.

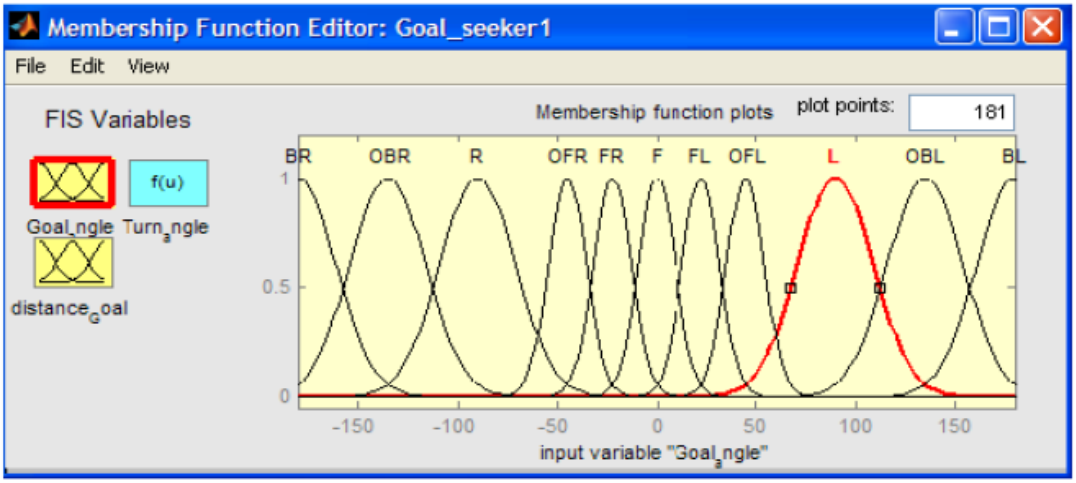


Figure 15. Membership functions for the goal direction

The distance of the goal to the robot is fuzzified into four Gaussian fuzzy sets: VN (Very Near), NR (Near), M (Medium) and FR(Far) (Figure 16).

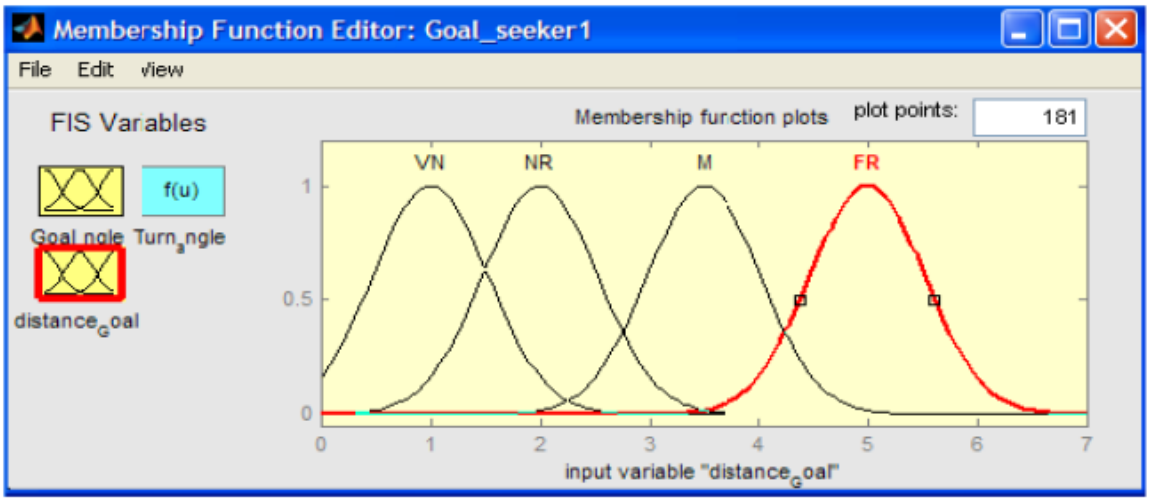


Figure 16. Membership functions for the distance

Since we want our robot to choose a smooth path to its goal, we have opted for a fuzzy distribution of the output (*turn_angle*) such that it covers 360°. Since the model of our fuzzy controller is a Sugeno, the output membership functions are of a constant type. The *turn_angle* of the robot is fuzzified into 11 constant fuzzy sets: Back Right (BR), Oblique Back Right (OBR), Right (R), ... as represented by Figure 17.

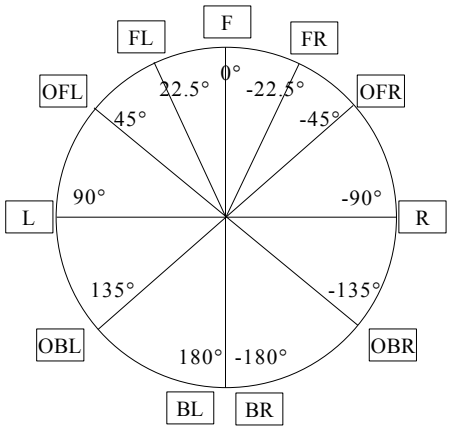


Figure 17. Angle divisions

The two input sets are combined in 44 fuzzy rules. For instance, the first rule is :

If *Goal_Angle* is *Back_Right* and *Goal_Distance* is *Far* then the *turn_angle* is *Oblique_Back_Right*.

For defuzzification we have chosen the weighted average method. The turn angle calculated by the fuzzy controller corresponds to the weighted average of each output of the set of rules stored in the knowledge base of the system. Figure 18 shows a graphical representation of dependency of the turn angle on the goal distance and the goal direction.

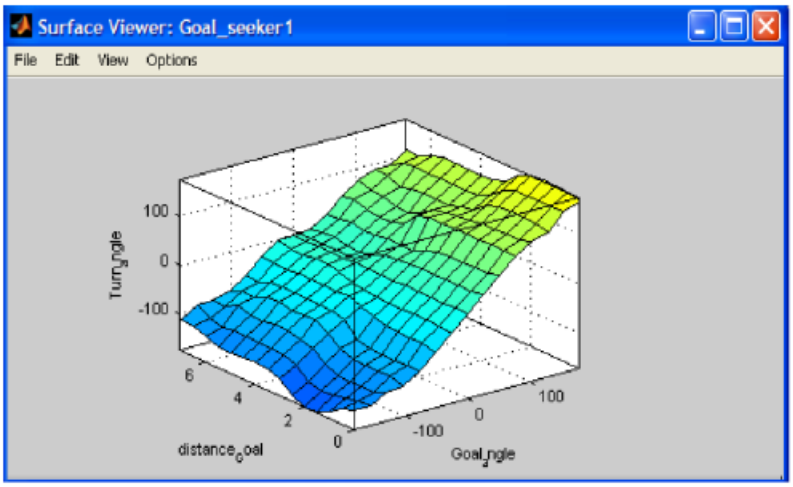


Figure 18. Turn angle as a function of goal distance and goal direction

We can see from this figure that the turn angle increases with the increase of the goal direction. This is logical since the robot has to reach the goal direction to get to its target. We can also see that the dependency of the turn on the goal direction is more important than its dependency on the goal distance. If the distance is very short, then the turn angle increases rapidly with the goal direction and if this distance is long, the dependency of the turn angle on the goal direction is less important.

Obstacle Avoidance Controller

Given the sensor data, the fuzzy controller calculates the turn angle of the robot in order to avoid the obstacles. The model of the controller is based on the distances to the obstacle detected by the sonar and infra-red sensors. The fuzzy logic controller is also of a Sugeno type.

The sensors of the Nomad have been divided in groups:

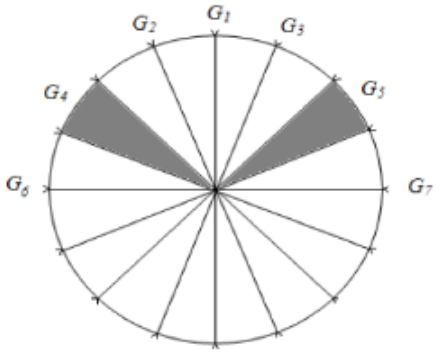


Figure 19. Groups of sensors

For the obstacle avoidance behaviour, we have considered the distance to an obstacle measured by each sensor group and the goal direction as the inputs of the fuzzy controller. This distance is defined as follows for each sensor group G_i:

For i = 1, ..., 5, $d_i = \min\{d_{imax}, \min_j\{\{d_{ij}\}\}$; j = 1, ..,N

where d_{ij} is the distance measured by the j*th* sensor of the sensor group *i*, d_{1max}=4, d_{2max} = d_{3max}=2 and d_{4max} = d_{5max}=1. The output of the controller is the turn angle, which ranges between −180° and 180°.

The distance measured by each sensor group Gi is fuzzified into three Gaussian fuzzy sets (VN, NR and FR). The range of this distance depends on the sensor group since the front sensors are the most important for a collision-free movement of the robot. The lateral sensors are used only to check if there is an obstacle on the sides of the robot each time it makes a rotation. The membership functions of the distance "di" are depicted in Figure 20.

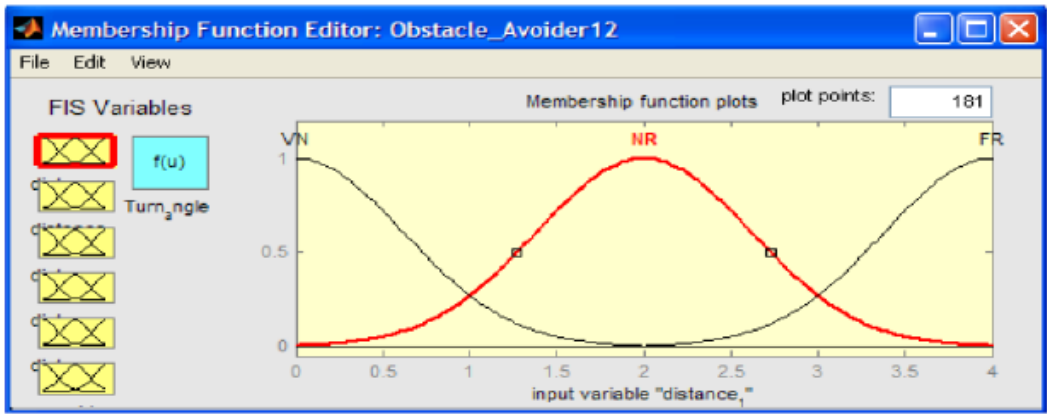


Figure 20. Membership functions for the distance di

The obstacle avoider based only on the sensor distances, does not take into consideration the position of the goal. However, it would be better to avoid the obstacles but also to seek the goal. For doing so, we have also considered the goal direction as an input variable for the obstacle avoider. The goal direction is fuzzified into two trapezoidal membership functions (Figure 21):

- Negative (N): which is equal to 1 for values between [−180° 0°] and 0 for values between [0° 180°]
- Positive (P): which is equal to 0 for values between [−180° 0°] and 1 for values between [0° 180°]

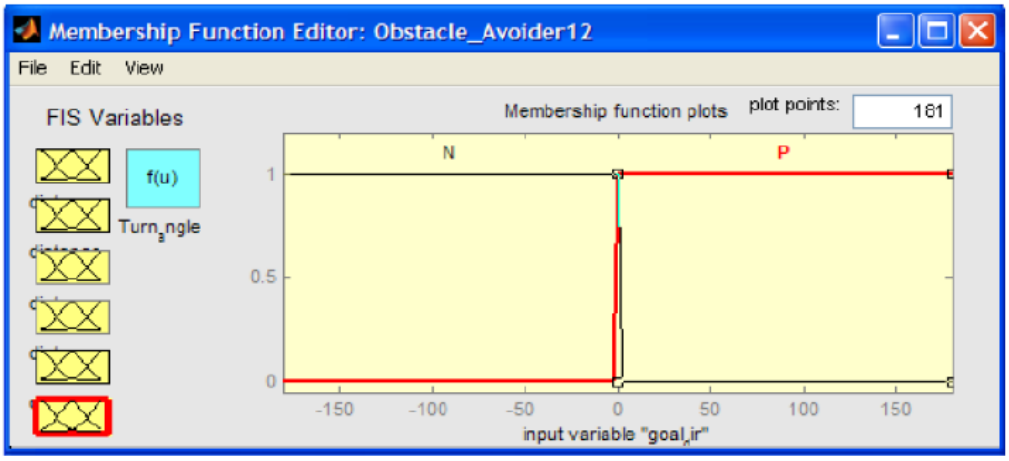


Figure 21.Membership functions for the goal direction

The output of the Fuzzy controller, the turn angle, is fuzzified into 6 constant fuzzy sets:

- Large Negative (MN): The turn angle is −90°.
- Small Negative (SN): The turn angle is −45°.
- Zero (ZE): The turn angle is 0°.
- Small Positive (SP): The turn angle is +45°.
- Large Positive (MP): The turn angle is +90°.
- Turn Back (TB): The turn angle is +180°.

The Fuzzy Rule Base contain 36 rules. The obstacle avoider functions the same way as the human's brain. When an obstacle is detected on the left side of the robot, it avoids it on the right side and vice-versa. But when it has the choice between two directions, it chooses the direction leading to the goal.

The defuzzification mechanism for the obstacle avoider is also the weighted average mechanism. The turn angle is a weighted average of the fuzzy set obtained by the intersection of the different output fuzzy sets of each rule.

Command Fusion

The obstacle avoidance controller and the goal seeking controller work independently. In fact, the goal seeker enables the robot to move towards its goal and neglects if it causes a collision. However, the obstacle avoider avoids obstacles and ignores whether it causes the robot to deviate from its goal direction. When the robot encounters an obstacle, these two behaviours are in conflict. To mediate between them in these cases, a command fusion scheme is needed. For command fusion, we have considered a fusion with arbitration; when the target path is free of obstacles, the goal seeker is called. Otherwise, if an obstacle is detected on the path of the robot, then the obstacle avoider is activated and the goal seeker is ignored.

For doing so, we have considered sensors from groups G1, G2, G3, G4, G5, G6 and G7 (on Figure 19) for testing if they detect an obstacle. However, we have accorded more importance to the front sensors, and the least importance is accorded to the lateral sensors R and L.

In fact, we have chosen a minimum distance to the obstacle, under which the obstacle avoidance behaviour should be activated. This distance depends on the position of the sensor that detects the obstacle.

Based on this command fusion scheme, a motion command is sent to the robot to move to a new position. This motion command is composed of a turn angle, chosen as mentioned above, and a speed calculated as a function of the turn angle. The larger the turn angle of the robot, the slower its speed should be. The relation we have chosen between the turn angle and the speed can be expressed by the following equation:

$$Speed = (1 - 2\,Turnangle)\frac{\Pi}{180} speed_{max}$$

where $speed_{max}$ is the maximum speed.

If this equation gives a negative speed, then we set it at zero. The different trials we have done on the simulator have shown that this relation is reasonable. The speed is not very high such that the robot gets quickly in collision before the obstacle avoider is activated. It is not very slow such that the robot can not move from its position while rotating as well.

Simulation Results

Figure 22 illustrate typical navigations performed using the GoalSeeking controller. The robot starts from 0, 0.

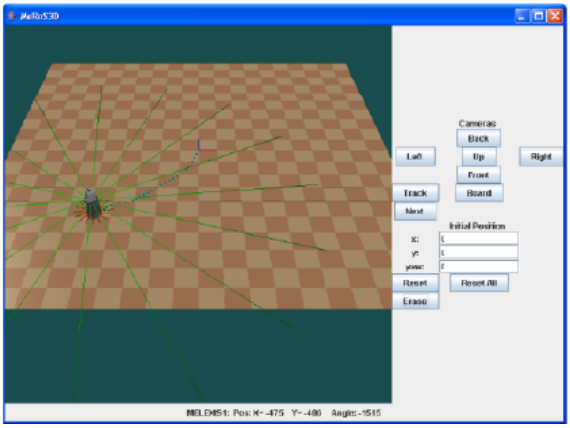


Figure 22a. Goal Position at -5, 5

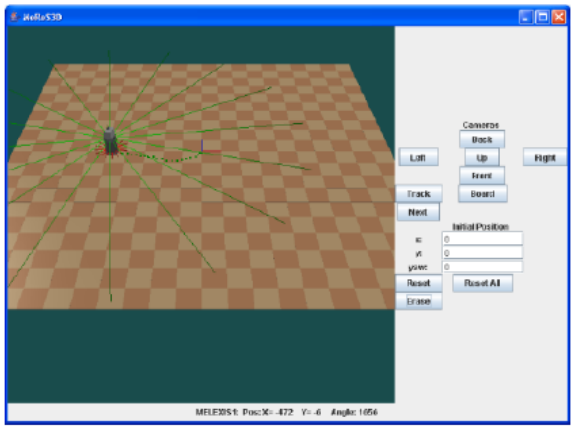


Figure 22b. Goal Position at -5, 0

As the GoalSeeking controller gives correct results, obstacles are added to the workspace of the robot to test the Obstacle Avoidance controller. Figure 23 shows navigations controlled by the combined GoalSeeking and Obstacle Avoidance controllers.

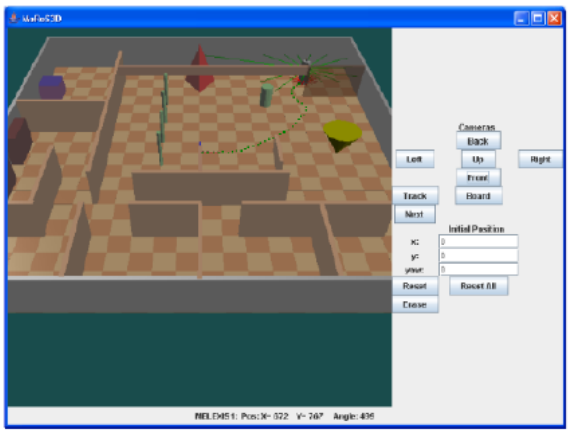


Figure 23a. Goal Position at 7,8

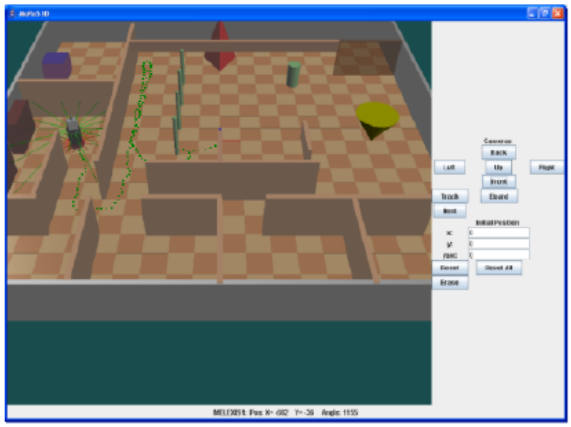


Figure 23b. Goal Position at 9, -8

We can see on Figure 23 that the robot is able to reach the goal even in difficult situations where he has to make u-turns as in the second example.

The Learning approach.

A Simple fuzzy control for the obstacle avoidance behaviour is able to drive an autonomous mobile robot. The used controllers are only based on the expert knowledge, which is not enough

to design a smooth path for the navigation of the robot. The learning approach can cope with this limitation.

In this work, we have applied off-line learning to the obstacle avoidance controller to adjust its parameters. For doing so, we have used the Matlab function ANFIS (Adaptive-Network-based Fuzzy Inference System) [SHIN93]. A predefined path has to be followed by the robot. This path contains a number of intermediate goal positions. Data (d_1 to d_5, goal distance and turn angle) saved during the navigation of the robot to the intermediate targets are used by the ANFIS module to adapt the membership functions and creates a new Fuzzy Inference System (FIS). This process is illustrated by the Figure 24 a,b and c. Figure 24 d shows the evolution of the training error.

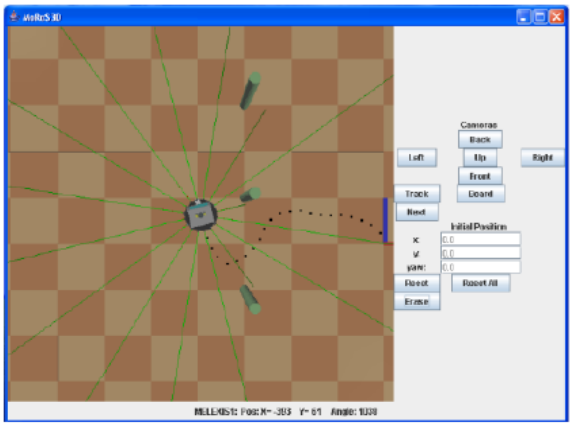


Figure 24a. Before training

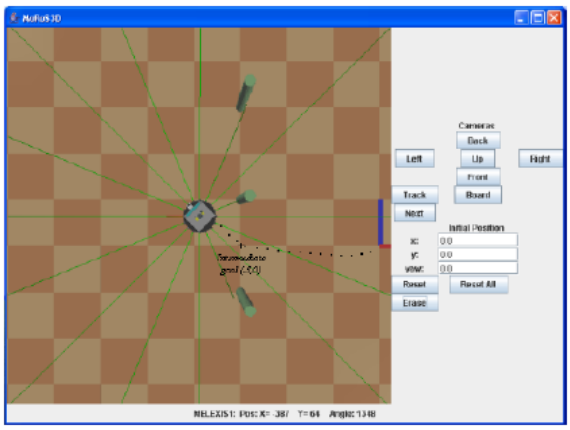


Figure 24b. Intermediate Goal

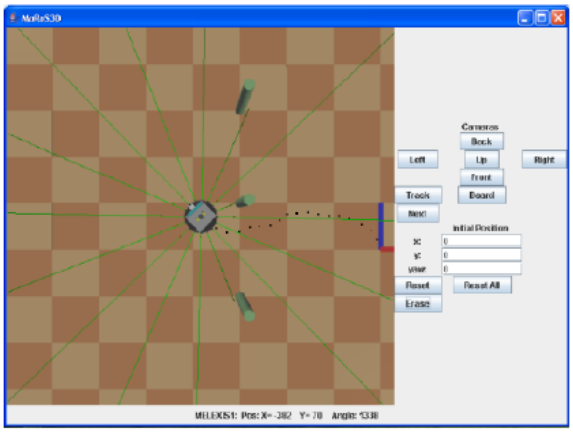


Figure 24c. Path after learning

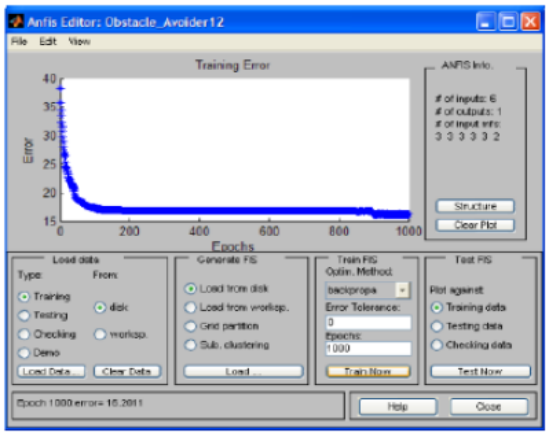


Figure 24d. Training error evolution

The training procedure has improved the path followed by the robot, movements are smoother and the path followed by the robot is shorter than in the first case.

3.2.4 Multi-robot applications

Developing multi-robot applications with CoRoBA is not more complicated than for a single robot, there are only more components involved in the control network. Existing interface components can be directly integrated into any new applications, each robot being associated with the existing Sensor(s) and Actuator components. What needs to be developed are components that implement collaboration and coordinated actions.

If each robot or system co-exists with others without any interaction, there is no special difficulties and each chain can use different Event Channels (Figure 25).

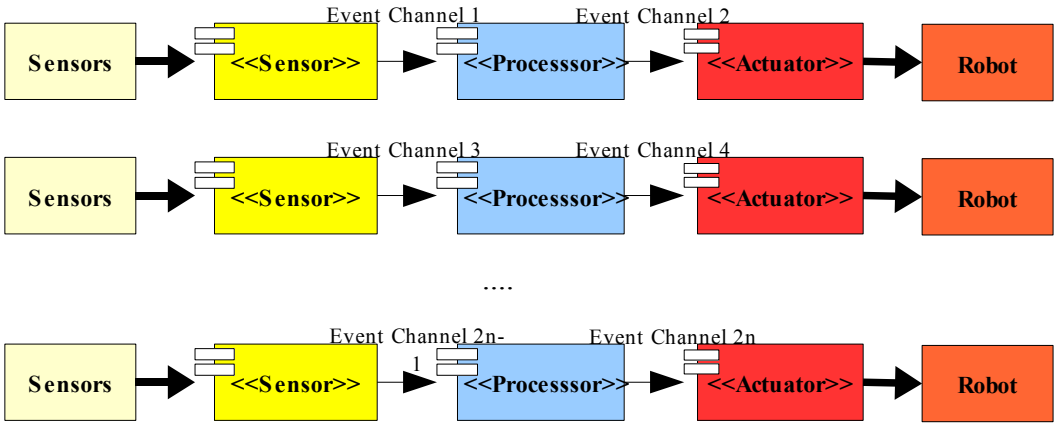


Figure 25. Coexistence of identical systems

Example: Simultaneous Goal Navigation of 2 Robudem. Each robot has its own set of goals it has to reach (Figure 26).

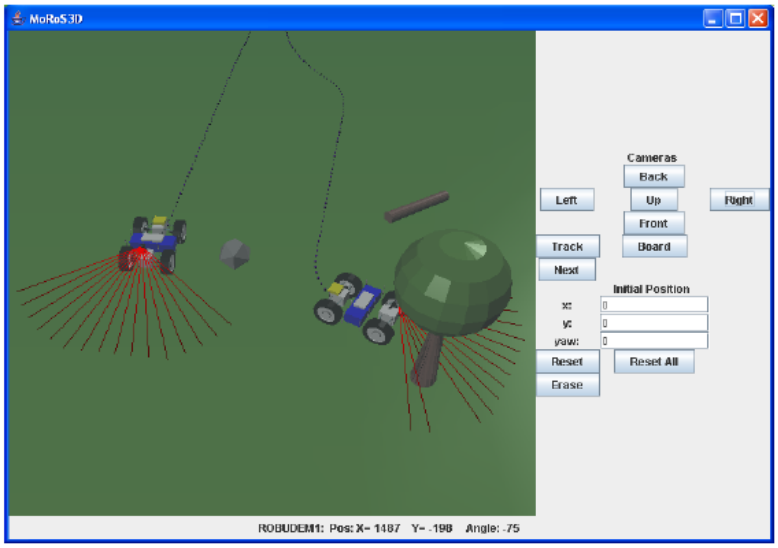


Figure 26. Two independent Robudem

But if we need to share information, as for example if we want to control 2 identical robots with one joystick as both robots use identical components (Figure 27), we must be able to distinguish events coming from the two Sensor components. In order to distinguish events emitted by different running instances of the same component, a unique identifier must be used. We could have stored this information into the variable header or as filterable data. This would have consumed some bandwidth because all events of the same type are received by all the clients. But as we have encoded this information in the domain field of the Header, we can use the registration mechanism of the Event Channel to filter the events. We are consequently able to group components in logical clusters. In order to have the possibility to send events from one sensor to several processors, a special identifier, GLOBAL, is used.
For instance in the case of the shared control application presented in section 2, we can give input commands to 2 robots (or more) with a single joystick, each robot being controlled by its own obstacle avoidance loop.

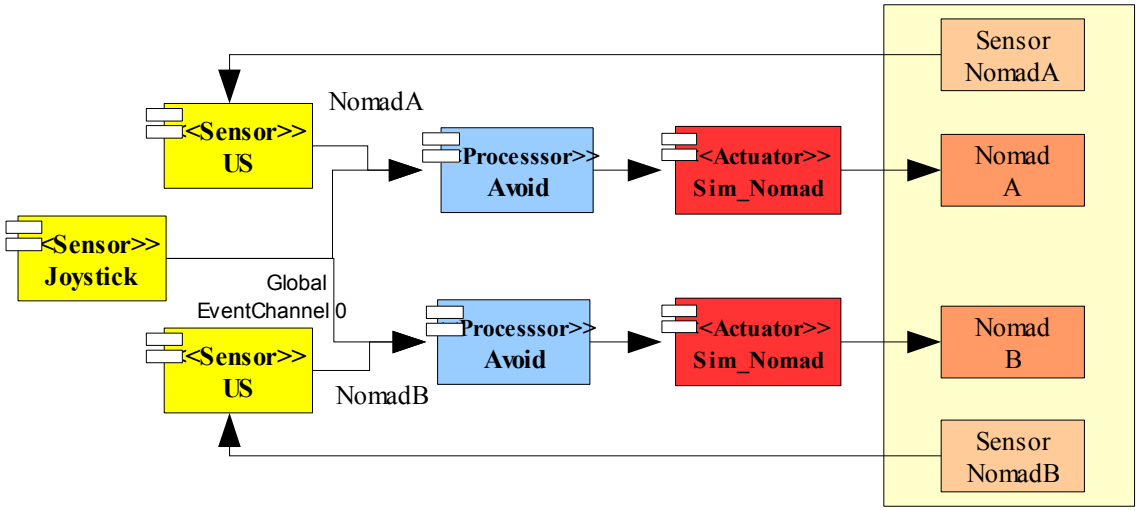


Figure 27. Architecture for the shared control of two robots simultaneously

3.2.5 Distributed simulation

Two possibilities can be considered for the conceptual solution of distributed simulation:
• A centralized server that holds the current state of the distributed virtual world. It receives periodic updates from each robot vehicle dynamic simulation (client), and it broadcasts to the rest of the visualization clients.
• A distributed world where the state of the world is distributed among clients. Hence, every client holds a partial copy of the state of the world and it has to broadcast changes to other clients.

The client/server architecture of the virtual world is known to have limitations due to the fact that the server becomes a bottleneck when the number of robot vehicles (clients) is large. The load is even larger in cases of a large number of visualization clients.

In this work, the second alternative (distributed world) has been selected at the conceptual level. Each instance of the simulator has the same model of the world and the robots have the same posture in all simulators. In the example depicted in Figure 28, *n* robots are simulated. Robots *1..j*

are simulated in simulator 1, robot *j+1..k* in simulator 2 and robots *k+1..n* in simulator 3. The position of non simulated robots are replicated in other simulator instances. Thanks to the event architecture it is straightforward to provide the visualization with data produced by odometry sensors of robots simulated in an other instance of MoRoS3D.
Off course each robot has its own controllers (for obstacle avoidance, path following, location, ...) that are not represented on Figure 28. Additional components should also be developed for inter-robot communication in order for them to collaborate.

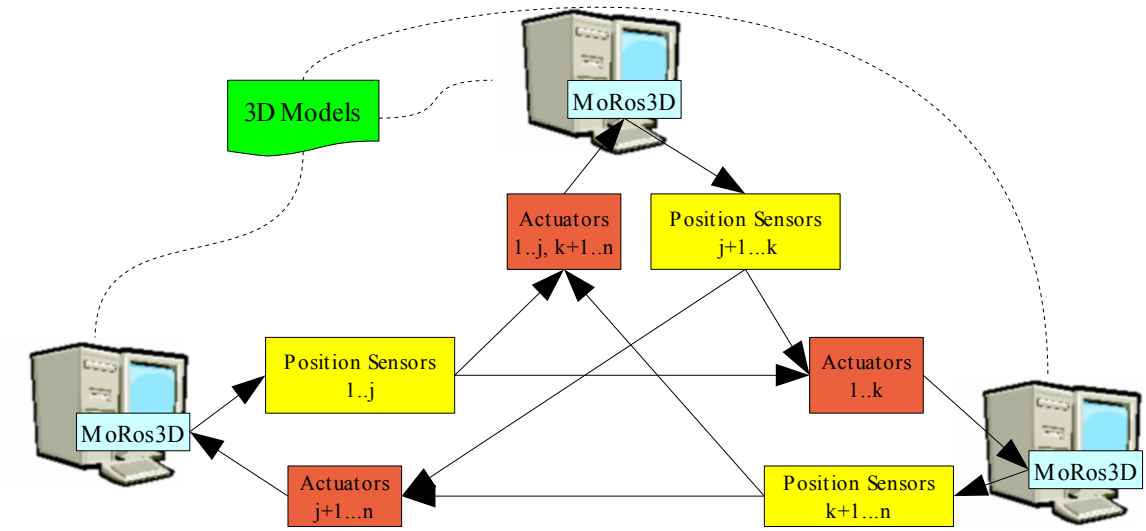


Figure 28. Distributed simulation principle

3.3 Real robots

Some components linked to real systems have also been developed. Sensors, Processors and Actuators interfacing with real robots are described in this section.

3.3.1 Nomad

This Direct Control Pattern makes use of two types of components, namely Sensors and Actuators. Users provide motion commands for instance with a joystick and these commands are sent to the actuator that adapts them to the controlled robot (Figure 29).

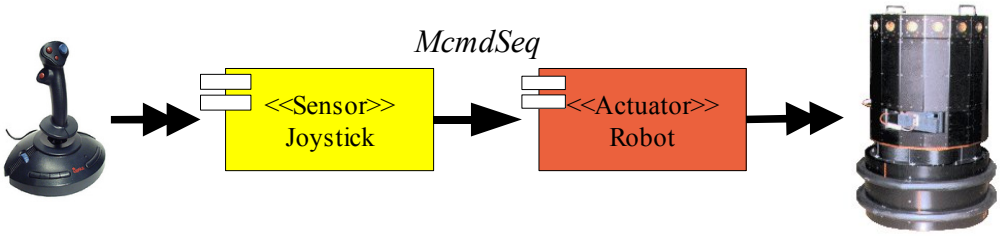


Figure 29. Direct Control components

Sensor

The Joystick Sensor has been presented in section 3.2.1.

Nomad Actuator

The Nomad Actuator actually receives motion commands from motion command Sensors like the Joystick component described above or from a navigation Processor.
The format for the motion command data is obviously the same as the one defined for The Sim_Nomad (section 3.1.2) : *MotionCommand* and *McmdSeq*.
The communication between the Nomad Actuator and the on-board robot controller is realised with sockets. This component actually wraps the original motion command functions of the robot. As the limited power of the on-board computer does not allow to directly run CORBA applications on it, the Nomad Actuator has to run on a separate computer.

As input events can come from two different kind of components, namely Sensors and Processors, this component registers its interest for two different events (*MC_Processor* and *MC_Sensor*) when subscribing with the *consumer_admin* of the Event Channel. Both events are propagated by the Event Channels and received by the consumer object. The origin of the event is determined in the method *process* allowing a different processing in function of the data origin. The length of the command sequence is also tested because the motion command sequence could have 2 or 3 parameters and a correct data handling is required in both cases.

The generic motion command are also adapted to the kinematics of the robot according to the following equations:

$$t_{vm} = \frac{-255 * y}{127} + 255$$
$$s_{vm} = \frac{-255 * x}{127} + 255$$

where x and y are the motion command values that vary from 0 to 255 and t_{vm} and s_{vm} are the desired translation, base rotation and turret rotation speeds of the robot.

The method *terminate* stops the robot by sending it null speeds.

3.3.2 Robudem

The Robudem component actually abstracts the low level communication and single access point to the on-board Robudem control system. A Processor has been chosen because the Robudem server accepts only one connection at a time and consequently this Processor actually groups in a single component the functions of a Sensor and an Actuator. It receives motion commands, forwards them by using sockets to the robot, and receives the robot's kinematic data that it sends as events to the output Event Channel (Figure 30). It is possible to use only the input side of this component, that is, to use it as if it was an Actuator. An interface called *Robudem_ProcessorProxy* has been defined as well as the structure *RobudemProprio* regrouping the robot proprioceptive data.

The Robudem Linux Controller is composed of two parts: a Linux server that manages sockets communication with clients and that communicates via shared memory with the real-time controller running under RT-Linux. The structure of the shared memory has been kept for the data transfer between the RobudemProxy and the Linux Controller. The proprioceptive data comes from the incremental encoders of the four wheels and from the two absolute encoders of the

steering motors (see section 5.2 of the chapter V for an overview of the Robudem). This component has been successfully tested in direct control mode.

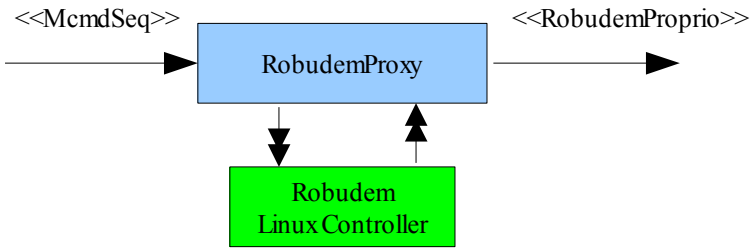


Figure 30. Robudem communication structure

From these two examples we see that Actuators are effectively adapters to real robots. If a library of control functions would be available, it could be directly integrated into the Actuator components. However, when robots can only be controlled via a server using socket connections, this adaptation is unavoidable.

3.4 Telecontrol application

3.4.1 Introduction

The aims of this section is to show how an existing applications based on a centralised blackboard [GEER05] could be reimplemented with the CoRoBA framework and how real applications can benefit from the framework and the simulator. Starting from Processors tuned in simulation, a new application using real robots can be rapidly assembled as far as Sensor and Actuator components for the real robot and sensors exist.

The platform is a Nomad200 that is equipped with a stereo-head. The operator controls the robot with a wheel and a Head-mounted tracker (Figure 31). Besides pure teleoperation, this complex application combines advanced control techniques and proposes the choice between different levels of autonomy. Section 3.4.2 details the application structure and explains how it could be implemented with CoRoBA while section 3.4.3 presents the first results obtained in simulation.

Figure 31. The Nomad with the stereo-vision system on top, the VR control interface and the GUI on the robot

Operation modes

The basic mode of operation for the system is traditional or direct tele-operation, including the creation of feeling of presence. For shared or supervisory autonomy control fixed static responsibilities for human and robot have been selected. The fixed responsibilities are defined in 4 levels of autonomy:

- Tele-operation: The user has full, continuous control of the robot at low level. The robot takes no initiative except perhaps to stop once it recognizes that communications have failed. It does indicate the detection of obstacles in its path to the user, but will not prevent collision. This is the default autonomy level.
- Safe Mode: The user directs the movements of the robot, but the robot takes initiative and has the authority to protect itself. For example, it will stop before it collides with an obstacle, which it detects via multiple US and IR sensors.
- Shared Control: The robot takes the initiative to choose its own path in response to general direction and speed input from the operator. Although the robot handles the low level navigation and obstacle avoidance, the user supplies intermittent input to guide the robot in general directions.
- Full Autonomy: The robot performs global path planning to select its own routes, acquiring no operator input. The goal of the robot can be specified by the operator or by the robot's vision system.

Note that, the change in autonomy level is made dynamically; whenever the operator desires to change the level of autonomy the robot changes its behaviour.

A behaviour is defined here as a representation of a specific sequence of actions aimed at attaining

a given desired objective. Each behaviour comprises a set of fuzzy-logic rules. The navigation strategy used in this application is a reactive navigation. It differs from planned navigation in that, while a mission is assigned or a goal location is known, the robot does not plan its path but rather navigates itself by reacting to its immediate environment in real time. The result of applying iteratively a reactive navigation method is a sequence of motion commands that move the robot from the initial location towards the final location, while avoiding collisions.

In our approach for robot navigation we describe the possible situations by a set of basic rules:

- If no obstacle is detected then use the "Goal seeking behaviour".
- If it's not possible to change direction toward the goal and there is no obstacle in front of the robot, then use the "Go straight ahead behaviour"
- If an obstacle is detected in front of the robot and it's still possible to change direction (to turn), then use the " Obstacle Avoidance behaviour"
- If there is an obstacle in front of the robot and there is no possibility to change the direction, then use " Make U-turn behaviour"

The robot uses a reactive navigation approach by considering the local information from its environment obtained by sonar and infra-red sensors. The adaptation of the navigation strategy to the real robot is done through the fuzzy-logic rules parameters (Membership functions, Fuzzification and Defuzzification process) of the different behaviours.

Goal Seeking Behavior: This controller allows the mobile robot, starting from the actual position, to reach a target point. This operation is realized in an environment where there are no obstacles around the robot. Given the azimuth (φ) and the range to the target (ϱ), a fuzzy controller calculates the turn angle and speed commands to apply to the robot to reach it. The used controller is of zero order Sugeno's type and uses linguistic decision rules of the form:

$$\text{If } (\varrho \text{ is } Ai) \text{ and } (\varphi \text{ is } Bi) \text{ then } (\Delta\theta \text{ is } Ci)$$

Where Ai and Bi are fuzzy sets defined respectively in universes of discourse, and Ci is a constant. The control law of the controller is represented by its output surfaces in Figure 32.

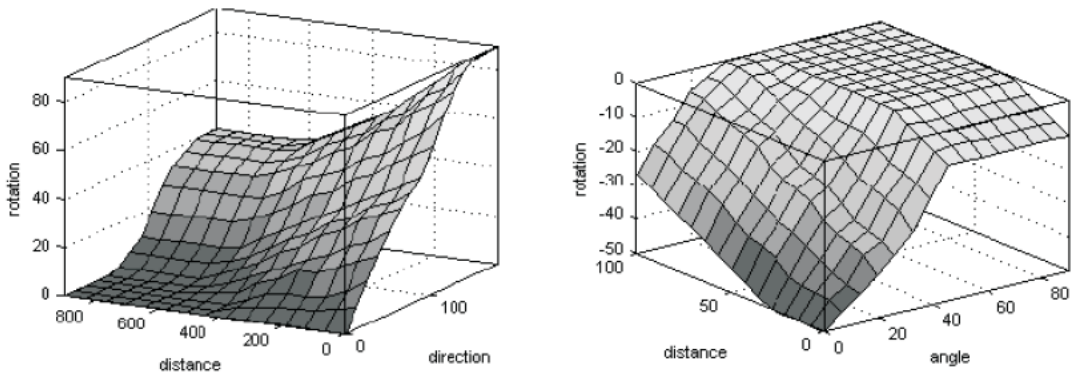


Figure 32. Output surfaces of the Fuzzy controller

Obstacle Avoidance: If an obstacle is detected in front of the robot, the nearest point (of this obstacle) to the robot and making the smallest angle (azimuth) with its axis is marked. A fuzzy controller using the information provided by the sensors is initiated. It considers the polar coordinates in the robot frame of the detected points from the obstacles to estimate the change in angle to apply to the robot to avoid these obstacles. The used controller is a zero order Sugeno's type too. In this controller the change in angle to apply to the robot is more important as the obstacle is closer to the robot and closing its way.

Go Straight Ahead Behaviour: This action is used by the robot if there is an obstacle embarrassing it to go toward its goal but no obstacle is detected in front of it. In this case the robot continues moving with its currents speed and orientation.

Make U-turn Behaviour: The robot uses this action in order to leave some blockage situations like a closed way or a narrow way. When this action is activated, the robot makes a U-turn in its position and moves straight ahead until a rotation at the right or at the left is possible.

Data Fusion and Map Building

For direct teleoperation the building of a map of the robot's local environment is not indispensable. However, the addition of the mentioned levels of autonomy implicates the need for an accurate representation of the local environment of the robot into a internal obstacle data map. The map is constructed combining the US and IR sensory information. In this context sensor fusion can be defined as the process of combining different sets of, or data derived from, sensory data into a map which represents the environment. Although control architectures with strong reactive characteristics, like the one applied here, do not require an environmental model in order to navigate, the enhanced information using sensor fusion can lead to a more intelligent motion planning.

One of the main motivations for implementing the sensor fusion module is the extension of the spatial coverage. As the number and range of sensors on the robot are limited, not the whole environment of the robot can be scanned at a given moment. The usage of an environmental map enables a memory function, which ensures that the information gathered by past measurements does not get lost. By doing so, successive measurements performed by one and the same sensor can be used to reduce the uncertainty on the position of an obstacle as the robot moves. However, as not all measurements are reliable, it can be preferable to delete from the map objects that originate from erroneous measurements. For this reason, each object is characterized by the number of spotting and the number of fusion cycles passed since the last spotting, referred to as the "age" of the object. At the beginning of every fusion cycle, the age of every object on the map is increased by one and then the object is subjected to an elimination test. If the age exceeds a value that depends on the number of spotting, the object is deleted from the map. The dependability itself should be determined experimentally, as it is influenced by the reliability of the sensor measurements as well as the motion speed and cycle time of the sensor fusion module.

Navigation Strategy - Motion Controller

According to the selected level of autonomy, the navigation strategy controller selects the proper robot driving behaviour. For direct teleoperation, this behaviour is straightforward: simply feed the acquired speed and steering commands to the robot's motion controller. In safe mode the

available map is checked for collision danger and if necessary an emergency stop is performed. In shared control mode as well as in autonomous mode the robot has the responsibility of the local navigation. To accomplish this task a path planner is included in the system. The input from the operator can be:

- a final goal if the way and the time to reach this goal are not very important.
- a final goal with a set of desired intermediate passing points.
- or a continuous path until the final goal. In this case, the robot follows the trajectory set up by the user to reach the target point. If an obstacle is detected, the robot uses the obstacle avoidance behaviour to bypass it and retrieve its path afterwards.

In all cases the same path planning controller is used and the user has only to define graphically the target point(s). The output from the obstacle avoidance controller is combined with the input direction from the operator.

3.4.2 Application architecture and components

The architecture of the application as it could be implemented with CoRoBA is presented in Figure 33. The functionality of the components is described below.

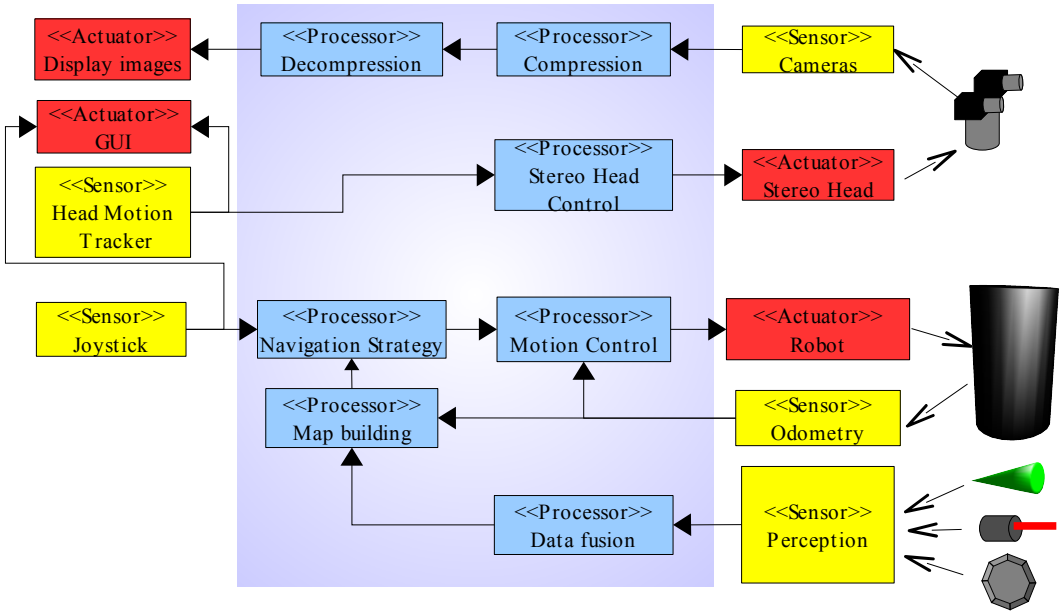


Figure 33. Structure of the advanced telecontrol application

Client sensors: Head Motion Tracker and Joystick

The main task of these modules is the regular update of input commands provided by the operator. By means of two hardware devices the operator controls the robot and the stereo head. The robot is controlled by a joystick, which is interfaced using Direct Input. The stereo head is controlled by movement of the operator's head. A motion tracking device is placed on the head of the operator and registers the rotations of the head made by the operator. The control commands are also transferred through the Event Channel networks to specific Processors, able to handle them in an intelligent way. In all control modes the operator selects a certain goal or location of interest based on the visual feedback information received from the robot. The human in the control loop is fully responsible for this goal selection using his own capabilities for active visual search tasks.

Client Actuators: Display and GUI

Visual feedback to the operator is provided by means of an actuator component, communicating with a head mounted display (HMD). The operator sees a 3D view of the robot's environment, due to the stereo-vision set-up, including the HMD. Being able to look around freely, from a remote location, at a sufficiently high frame rate, due to the image compression, provides the operator with a certain feeling of presence at the remote site. The inputs from the operator registered by two Sensor components, are displayed on a graphical user interface, which is an Actuator component.

Camera Sensors & Actuators

The robotic platform is a Nomad200 equipped with extra mechanical and sensory structures. On top of the Nomad 200 another PC platform (Figure 32) is placed, linked by a coax cabled Ethernet connection. The robot vision module consists at the hardware level of a stereo head, type Biclops, and two miniature CCD Colour Cameras. The head is mounted on the upper PC platform, carrying both cameras (Figure 32).

An Actuator component receive events from the Stereo Head Control Processor and forwards the movement commands to the Biclops system.

A Sensor captures in a synchronized way frames from the left and right cameras by means of a well suited frame grabber. Before being transferred to the remote user and in order to reduce the time needed for the transfer of the images, the captured frames are compressed either using the classical JPEG encoder or a Wavelet based coding technique

Robot Sensors and Actuators

The Nomad actuator has already been described in the preceding section. The Nomad Sensor connects to the real Nomad via sockets. It receives kinematic and pose data and forwards them in events of type *Nomad_MotionSensor*. A structure grouping the kinematic data (*NomadKinematics)* is defined in the IDL file.

```
struct NomadKinematics{
        long x;
        long y;
        long steer_angle;
        long turret_angle;
        long vel_trans;
        long vel_steer;
        long vel_turret;
};
```

Image Compression and decompression Processors

Compression and decompression, aim at the fast transfer of the images over the distributed framework with respect to their quality. The employed Wavelet based coding scheme, i.e. SQuare Partitioning (SQP) [MUNT99] has been developed at the VUB. It allows rate-distortion performances comparable with state-of-the art encoding techniques, allowing lossy-to-lossless reconstruction and resolution scalability. In terms of rate distortion, SQP outperforms JPEG, at considerable compression ratios.

The compressed frames are communicated via the wireless link to the client. The resolution scalability feature of SQP comes in handy when progressively streaming the data, since the decoder at the client site does not need to wait until all the data has arrived, but it may start reconstructing a lower resolution of the image (from the received data) and start processing that image first while waiting to receive the remaining data that would allow to reconstruct the image

at full resolution.

Stereo head processor

This module accurately controls the pan and tilt angle of the stereo head to seamlessly change the viewpoint.

Robot navigation Processors

The joystick commands are fed into the navigation strategy manager. Based on the level of autonomy and the local map of the robot's surroundings, these control commands are adapted. A motion control process communicates these commands to an Actuator component interfacing with the robot hardware. The map is generated based on a data fusion process. On its turn this process receives input from a Sensor component interfacing with the available sensor set-up on the robot. An odometry sensor component keeps track of the robot's motion and updates the robot's position. This information is of importance for the map building and the motion control.

3.4.3 First results

First results of the implementation of this complex application are presented here. Components used in previous application that have been tested in simulation have been reused. The Nomad Sensor and Actuator are already available. By using the on-board camera capability of the simulator, the (mono-vision) teleoperation mode has been tested. The processors of the shared autonomy that has already been described in section 3.2.1 and the autonomous navigation presented in section 3.2.2 can also be reused.

The following Figure represents a model of the Nomad with the additional computer and the stereo-head. It illustrates an obstacle avoidance navigation of the robot in the simulator. Once algorithms have been tuned the components can be directly tested with the real Sensors and Actuators.

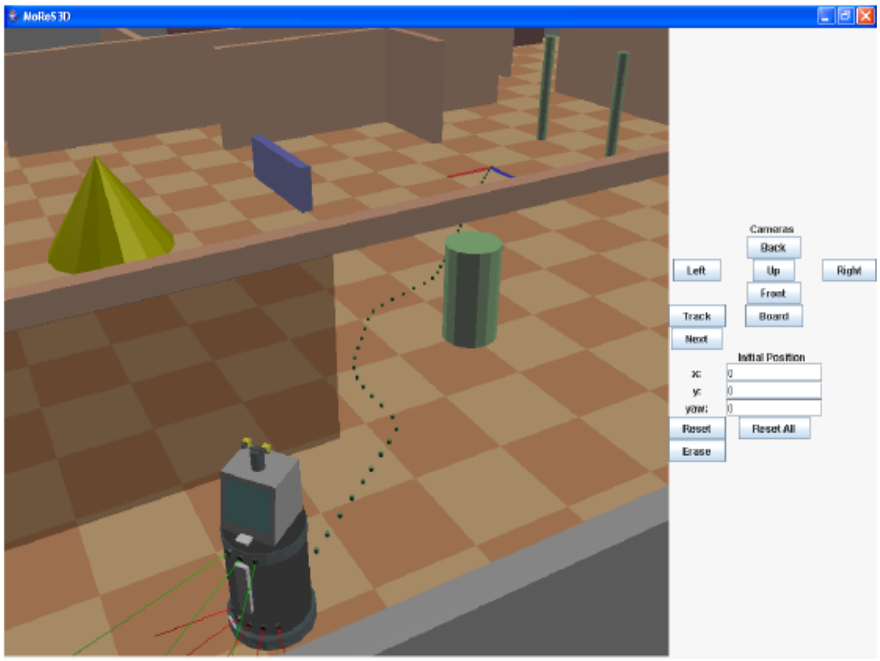


Figure 34. Screen capture of the simulated autonomous navigation application for the Nomad

4 Evaluation

The last part deals with the evaluation of the framework. The qualitative evaluation is based on the following criteria:

- We explain how applications presented in the introduction could be improved by using CoRoBA.
- We compare CoRoBA with other frameworks.

On the other hand, for the qualitative evaluation, we have defined and applied evaluation criteria and measures of effectiveness.

4.1 Improvement of applications

We now show that difficulties and limitations of applications presented in Chapter I could be eliminated by using CoRoBA.

Corode:

- Corode was written for windows: The implementation of CoRoBA is platform independent.
- The acquisition and control was grouped in one thread and mixed with the GUI: CoRoBA provides a clear design that separates the data flow from the control flow. The visualization of data is not mixed with the application logic.

FuzzyNomad:

- The algorithm code was mixed with the robot function calls: with CoRoBA all functions and method calls specific to a given robot are located in Actuator components while the algorithm code is embedded in Processors. This clear decoupling facilitates the recycling of components.

VRNomad:

- Sockets were used for communication: platform independent communication and CORBA objects replace low-level sockets communication.

Vizir:

- The 3D model was based on OpenInventor which is a commercial libray: Java 3D has been used for the 3D visualization part of the simulator.

4.2 Comparison with other frameworks

We can also compare in a more detailed way the CoRoBA implementation with other similar frameworks. By doing this comparison, we intend to show that CoRoBA is able to do what all the other ones can do. Therefore, we compare CoRoBA, MCA, DCA, GenoM and ORCA from the architectural point of view. Similarities and differences between CoRoBA and several frameworks are given below.

MCA

MCA uses MACRO definitions to simplify the coding of interfaces while interfaces definitions in CoRoBA are mapped to code by the CORBA IDL compiler. In this cases, interfaces are mapped to classes organised in independent files that can be easily reused in different projects. On the contrary, MACROS are expended by the preprocessor and does not produce separate and visible code.

Functionality of CoRoBA components is located in the *process* method running in a separate thread. In MCA, each module has to provide a *control* and a *sense* method that are called periodically by a general timer. In CoRoBA, each component can work at a different frequency.
While CoRoBA defines a unidirectional data transfer for each component, MCA uses bidirectional data transfer, one for the bottom-up sensor data flow and one for the top-down control data flow. This can confuse the novice because it allows to mix up different functions in a single module what is not in favour of the modularity.
MCA implements its own communication library based on sockets and shared memory. Disadvantages of this approach have already be largely commented on.
In CoRoBA the user can define its own data structure, which are defined with the IDL. MCA is limited to arrays of double and rely on blackboards for the transfer of other data types. This introduces a non uniform communication scheme.
The communication in CoRoBA is asynchronous and different running modes are available (PERIODIC, SYNCHRO, TRIGGER). MCA has only a periodic mode. MCA uses a polling approach to check if new data is available while CoRoBA relies on the mechanism implemented by the TAO ORB which implements the Reactor Pattern [SCMI00]. One strong point of MCA is the use of Parameters. These are variables that can be modified at run time through a standard GUI.

GenoM

Like in CoRoBA, control and data flows in GenoM are clearly separated. The control flow is made by requests and answers while the data flow between the components relies on a second protocol; data are exported in read-only structures. Clients send a request for a service and get an acknowledge when data is available in what is called "posters". This can be seen as an asynchronous result notification. This working mode can be reproduced in CoRoBA components by using a combination of 2-way calls and user-defined control events. GenoM uses a proprietary communication library based on sockets. Definition of interfaces and data in GenoM relies on standardised servers generated from a synthetic description.

DCA

Controller modules are very similar to the components implemented in CORBA. The core functionality is located in a base class that is inherited from by derived components. The DCA communication is based on a library inspired by ACE. The execution relies on a tree organisation containing supervisors and controllers. The controller contains a process algebra interpreter that organises the execution of the controller modules. This is the main originality of DCA

MIRO

MIRO is very similar to CoRoBA. Both are based on the same CORBA implementation, namely TAO, and they offer equivalent capabilities regarding synchronous and asynchronous communication, this later being based on the Notification Service.
One of the advantages of CoRoBA is the interface hierarchy and the implementation inheritance. The remotely callable management methods provide easy control capabilities while in MIRO to stop services you must send SIGINT or SIGTERM signals. Another limitation of MIRO is the implementation of the Behaviour engine. In [MIRO03], it is mentioned that all behaviours run sequentially in one loop and are not multi-threaded. The arbiter is included in the Behaviour process. In CoRoBA, behaviours can be implemented as separate processes providing more flexibility in the deployment and at run-time.

MARIE

The Marie approach is interesting if existing applications that do not relate to each other have to be integrated. One of the prerequisite is of course to dispose of an API for interacting with those applications.
MARIE's approach, as mentioned in [COTE04] suffers from many drawbacks, namely, overhead, complexity and system resource management. The Mediator Design Pattern [GAMM95] centralises rather than distribute and can rapidly become a bottleneck. CoRoBA on the other hand promotes partitioning and loose coupling by using the Communication Design Patterns presented in Chapter III.

ORCA

The principles of Orca are very close to CoRoBA's philosophy:

- An Orca component is a stand-alone process. A system consists of a set of process which run asynchronously, passing objects to one another.
- Communication is performed using a set of communication patterns, which are abstract policies for how objects are sent, implemented using some transport mechanism.
- Orca does not prescribe anything about how robot architectures should be built. It simply defines components and their interactions, leaving the developer free to connect them in any way he chooses.

In comparison with Orca, CoRoBA defines an architecture that mimics components found in control applications and provides an implementation for these components. This allows to reduce the development time because developers have only to provide the code related to the functionality of the component.

Player/Stage

Recently Player has undergone a complete change in its implementation because of a number of issues that the development community have found with the previous version. Prior to 2.0, Player was a network oriented device server. The Client-Server model was too restrictive, the wire data transformations was not robust nor flexible (limited to integers), the driver API was complicated, only single data and command type was possible for each interface and the only transport protocol was TCP/IP. In [Coll05] Collet proposes a list of requirements for a robot framework that is similar to what we have proposed. The changes have focussed on allowing more flexibility and a simpler message processing system. The implementation has been divided in two main parts: the Player core and the Transport layer.
In Player 2.0 the core system is a queue-based message passing system and a driver also broadcasts data to all subscribed client-queues. Player message structure has also been modified (host, robot, interface, index) and the message namespace has been expanded to two layers.
The transport layer provides two libraries: TCP/IP communication and a platform independent data representation(XDR).
We note that Player 2.0 reimplements solutions that have been available for years in existing communication middlewares like CORBA. Synchronous and asynchronous communication, queues, messages with multi-headers, filters, Naming Service, Interface Repository, Common Data Representation, ... provide equivalent capabilities.
An advantage of the new structure is the possibility to develop monolithic applications, that is, without network communication.
The same possibility exists with CORBA by using co-located objects, that is when client and server are collocated in the same address space. No changes to the source code are necessary in either client and server if we link the server to the client [HENN99].

We conclude this comparison with a citation from [Coll05] "Player now [mid 2005] acts as a distributed framework with servers being able to subscribe to each other to meet the requirements of individual interfaces". It was obviously not the case at the time we started on own development.

From these comparisons we see that different frameworks solve identical problems in different ways and it is very difficult to conclude that one is better than the others, every framework having its strong and weak points.

4.3 Improvement in development time

There are different ways to evaluate the improvement provided by the framework. For example, we can measure how many operations have to be performed in order to generate an empty component, how long it takes to implement a new empty component, how much code has to be manually written, etc. This kind of evaluation takes a lot of time and requires the availability of a large number of programmers in order to obtain reliable results. The only results available are based on a few student developments.

CORBA has a steep learning curve and several months are generally required for understanding and mastering its programming subtleties (especially in C++). In comparison, graduate students were able to develop new components with CoRoBA in a few weeks, what confirms the improvement in development time in comparison with raw CORBA development.

Thanks to the modular architecture, developing a new empty component takes less than 15 minutes. The total time required to implement the functional code depends off course on the contains of the component.

If Sensor and Actuator components are available for real robots and sensors (for example developed in a previous applications), porting an application from simulation is immediate. We only need to start the components making the link with real components instead of the ones used to connect to the simulator. The core of the application is made up of Processor components that do not need any modification.

4.4 Measures of effectiveness

4.4.1 Definitions

The purpose of carrying out different validations is to verify that the proposed concept works in a range of scenarios with different control requirements. Implementing and testing components in concrete applications provide a first opportunity for validating the framework. We must also be sure that the components work as they should and that particularly:

- Order of events are preserved
- All sent events are received
- Operations are executed in the right order

Order of events

Except in simple configurations, it is not possible to certify that all events will be received in the same order they have been sent. If components run on different machines with long transmission

delays and different routes are possible, we cannot guarantee the event ordering. However in typical control applications where the transmission distance is short and the event always follow the same route, the order is preserved.

No lost event

The Quality of Service of the Notification Service certifies that all received events are sent to consumers. At a lower level the TCP/IP mechanism also guarantees that all packets are received.
Furthermore, as all events have an individual Id and a time stamp, it is also possible to add checking code in the components.

Right order execution:

In data flow mechanism, no scheduler is present to control the right order of the operation execution.
It is the defined data path that controls the operation ordering. We must however pay attention to the loops that could appear in the components network. An example of such a situation has been presented in the preceding Chapter.

Besides this qualitative evaluation it is also possible to perform a quantitative evaluation. In order to objectively evaluate the individual components and the framework, measures of effectiveness have to be defined:

- Performance tests: memory, footprint, stability, memory leaks, processing time, data throughput, ...
- Modularity will be expressed in term of percentage of reuse.
- Development time: how long it takes to write a new component, how much code has to be written, how many operations have to be performed.
- Flexibility: when changing the type of a component (from sensor to processor for instance) the number of lines that have to be (re)written.
- Extensibility: adding new services,...
- Interoperability with other control systems.

4.4.2 Performances

Footprint (size of components on disks)

As can be seen in Figure 35, the size of typical components varies from 300 to 600 kB, which is rather small according to the current standards.

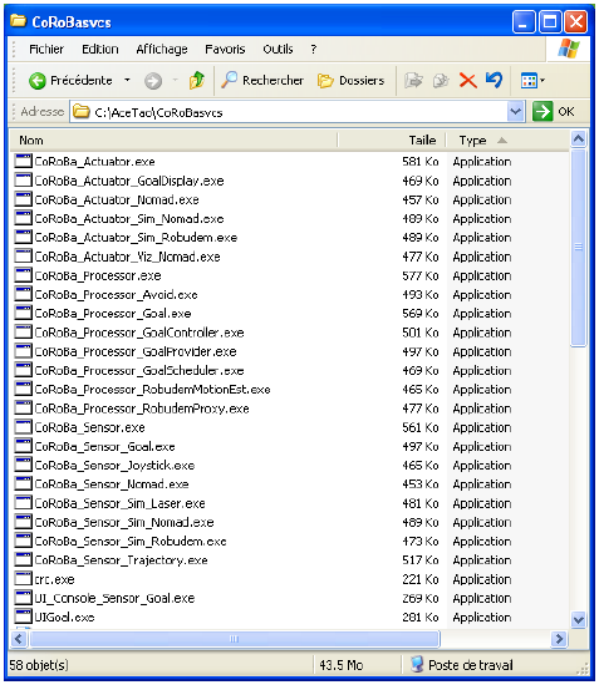


Figure 35. Size of typical CoRoBA components

Memory used by components

Each component needs approximately 10 MB. This is principally due to the CORBA libraries that seems to use a large amount of memory. The available amount of memory will determine how many active components can run concurrently without slowing down. In practice 20 to 50 components will be able to run on the same machine. However, the maximum number of components that can run concurrently also depends on the power of the processor.

Stability and memory leaks

The skeleton implementation of components has been tested extensively and components have run during several days without suffering from any stability problems or memory leaks.

Processing time

Typical components use a small amount of processing power because they are not running continuously but with a fixed period or when new data is available. Furthermore, each component generally performs a limited number of operations because the modularity and the distribution is a design principle of the framework.

If we consider the case of the PERIODIC mode, the required processing power increases when the period decreases (Figure 36). In this test 15 components (2 sensors – 12 processors and 1 actuator) ran concurrently. The simulator and other applications were also running and consumed more or less 25 % of the processing power.

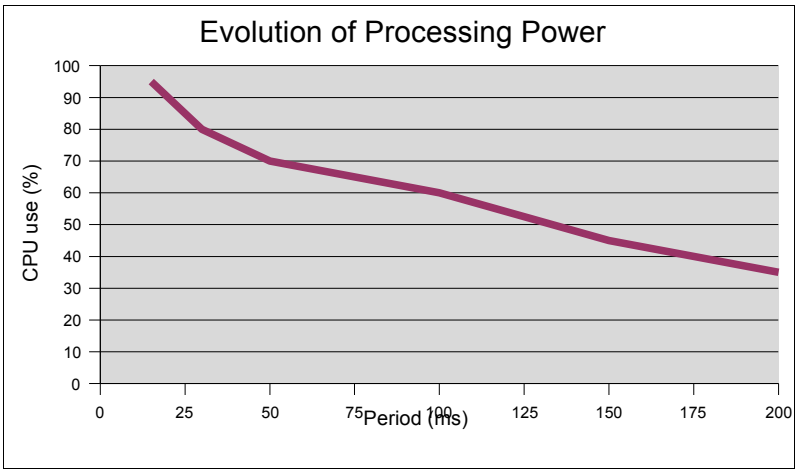


Figure 36. Processing power used by 15 components with the same period

Periodicity

The periodicity variation is function of the operating system the components run on. Linux capabilities in this domain outperform Windows and components requiring a strict respect of the periodicity should run on this OS. Figure 37 shows typical variations of the period for a sensor component. This variation is generally limited to 10% of the period (here 100000 µs) during shorts period of time.

The periodicity is not based on timers but on sleep calls that can be adjusted in function of the processing time needed by each component. Off course the period has to be chosen by the application developer by taking into account the time needed by each component to perform the required calculations, and by the requirements of the global application.

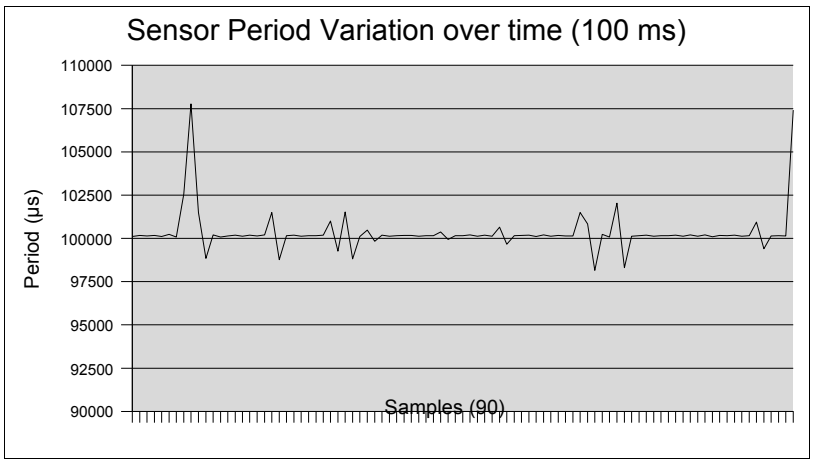


Figure 37. Period variation of a typical sensor component

How to choose the periods of the events ?

We consider a typical navigation application whose components are represented in Figure 38. A component that produces a map of the environment may run slower than an other one that performs obstacle avoidance. It is also influenced by the time required for sending and receiving data and consequently by the amount of data produced by the components.

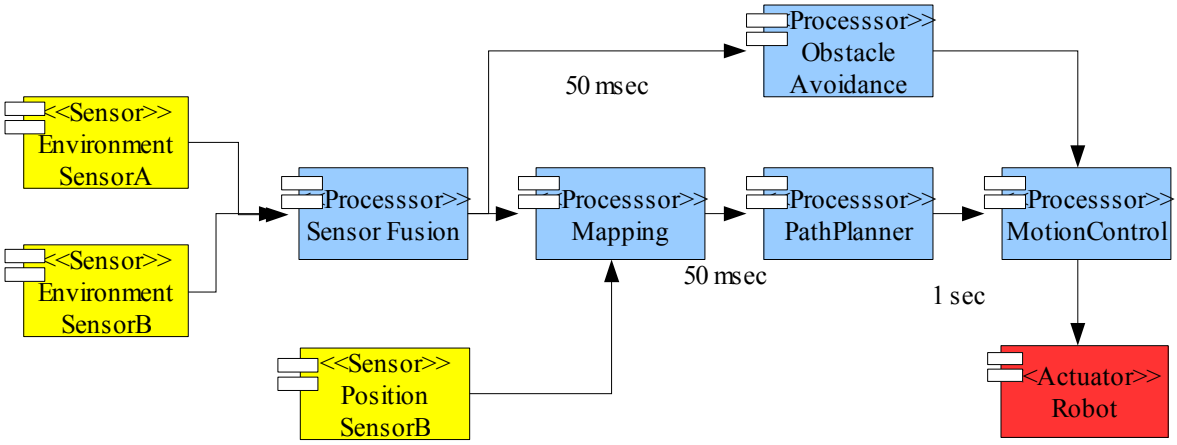


Figure 38. Typical control application

We suppose that the slowest component is the Pathplanner and we consider that it can produce a new path every second. It means that it is not necessary to provide it with a new map at a larger frequency.
On the other way we want to obtain a precise map of the environment and consequently wants to have as much information from the sensors as possible. The same applies for the obstacle avoidance component.
In the current implementation, data processing is linked to event transmission. So if we want to integrate all the sensor measurements, we must process data synchronously and send them at the same frequency to the next component. But if we send 20 times the map per second to the Pathplanner, most of the data will be dropped. It could be advantageous to decouple the data reception and processing from the event transmission. By doing this, all sensor data could be integrated into the map but this would only be sent for instance every second to the Pathplanner.

Data throughput and transmission time

The network performance can be expressed by the following relation:

Message Transmission Time = latency + length / data transfer rate

The latency is the delay between the start of a message's transmission from one process and the beginning of its receipt by another. The latency includes the time taken by the operating system communication services at both ends, the delay in accessing the network and the propagation time before the first bit reaches its destination.
On one hand, the operations added by CORBA increases the latency (that is independent of the message length). On the other hand, the extra information contained in a CORBA frame is quite constant (a few hundreds of bytes) and has therefore a larger influence for small data packets. It comes to a 20 to 30% overhead in comparison with raw socket communication. This result is confirmed by a comparative performance experiment reported in [GILL02] However, with increasing computing power and communication bandwidth, the overhead introduced by CORBA becomes every day less and less significant. As the framework communication is directly based on CORBA, there is no extra overhead from using the framework.

Simulator performances

Concerning the simulator performances, typical figures for 10 robots with 16 laser distance sensors is 80% processor activity (Intel Centrino 735) and a memory usage of 40MB. The scene refresh period in this configuration is 80 ms. (Figure 39)

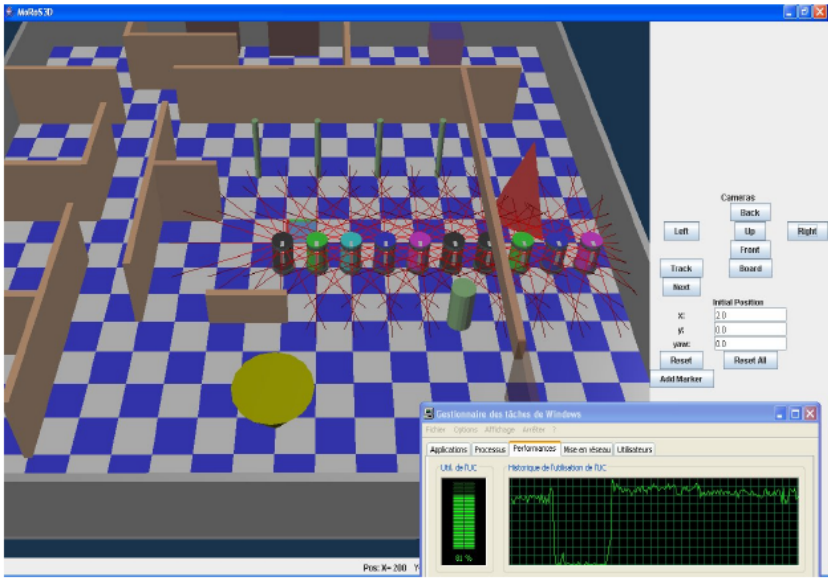


Figure 39. Typical processor load with 10 robots

The executable is stable and does not have any memory leaks. Figure 40 and 41 show two screen shots taken at one hour interval where we can see that memory usage is constant.

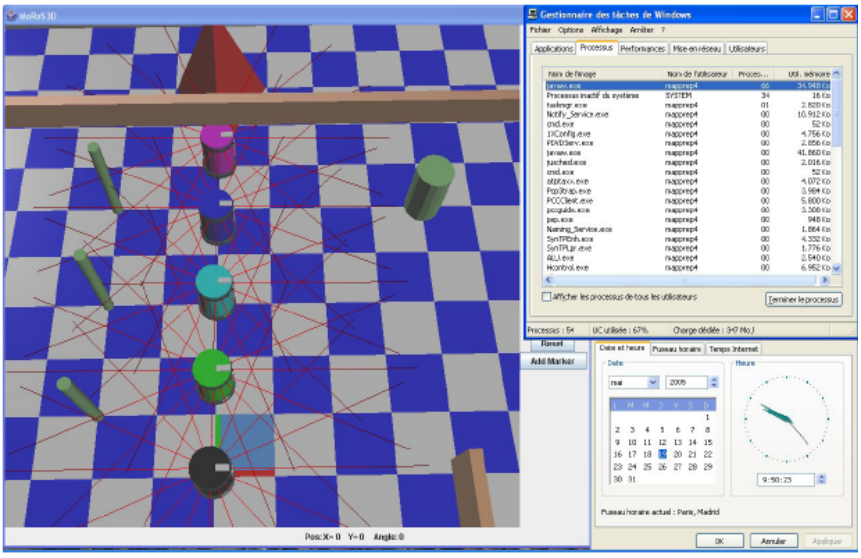


Figure 40. Screen shot taken at 9.50.

Regarding pure performances, Java3D is certainly not the fastest 3D engines but it is not too slow because all 3D operations actually rely on the 3D rendering library (DirectX or OpenGL), only collision detection and motion control algorithms are written in Java.

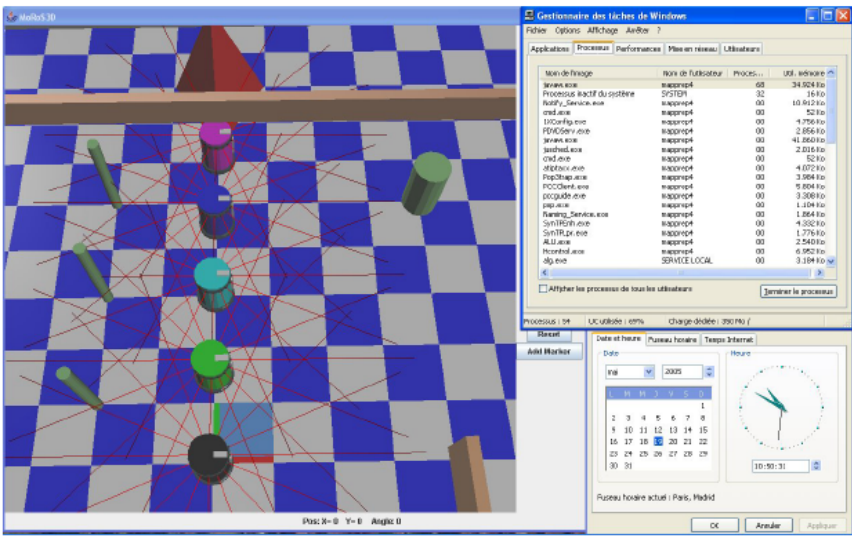


Figure 41. Screen shot taken at 10.50.

Real Time simulation

As explained in Chapter V – section 9 (p118), at each iteration of the simulator engine, the elapsed time is measured. The motion of the robot is based on this time difference and consequently the distance made by the robot corresponds to the reality. The next Figure illustrates a linear motion of a simulated Nomad with a constant speed of 1 m/s. We can see that after 10 seconds (real time) of simulation the robots has moved 10 meters. The coefficient of the line is 0.97, what gives an acceptable mean error of 3%. The measurement has been done with a Nomad_Sensor component and the Logging service.

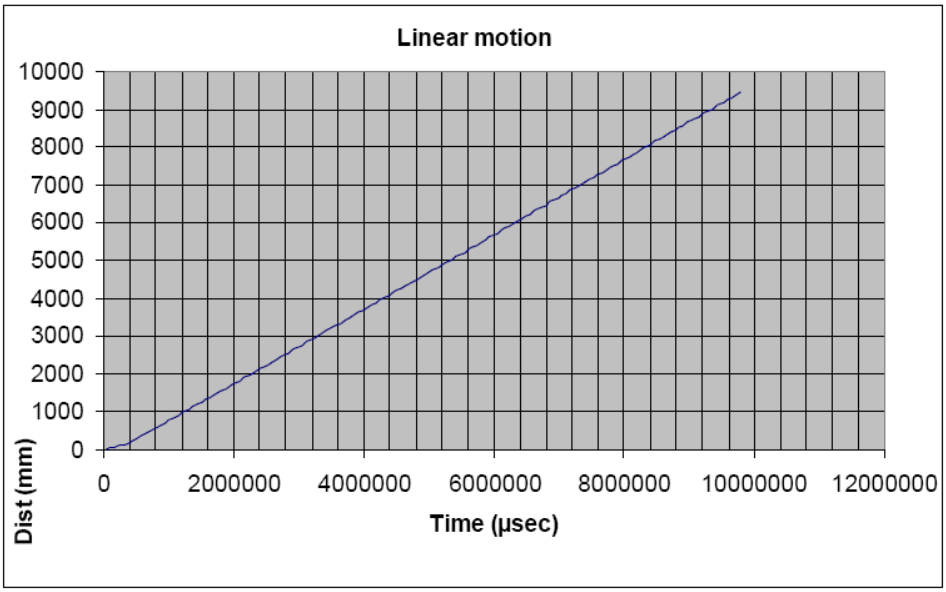


Figure 42. Linear motion of one Nomad with a constant speed of 1m/s

It is however possible to use a scale factor in order to speed the simulation up or down

5 Summary

The choice of an efficient and platform independent library for the implementation of the communication has allowed to fulfil the main requirements in computing and communication. The TAO implementation of CORBA proposes an efficient interprocess communication library that allows synchronous and asynchronous communication, offers support for easy development of multi-threaded components and their synchronisation and runs on different platforms. It also provides the flexibility to make the distribution of an application over multiple nodes easy for the developer and transparent for the user. An application developer can choose the communication model (synchronous or asynchronous) in function of its needs as this is not imposed by the framework. C++ and Java have been used because of the numerous advantages of Object Oriented languages in large projects. Inheritance, method overriding, virtual methods and abstract classes are powerful tools that simplify the development and improve the productivity of the programmer. Portability and modularity are two other requirements that have been met in the implementation of CoRoBA. The component based architecture contributes to the modularity too.

Applications presented in this chapter have demonstrated how existing systems could be integrated and combined with the framework. The data structure are defined in IDL and can be easily changed without requiring any modification to the components architecture. On the other hand, as long as 2 different robots or sensors implement the same interfaces, they can easily be exchanged with each other.

The applications that have been developed are representative of what the framework is good for and allowed us to validate its functionality and modularity. It has been shown that applications can be built incrementally by recycling existing components. Components developed in simple applications can be reused without any modifications in similar or more elaborated ones. Different applications illustrate the Robot Control Patterns presented in Chapter II: direct control, teleoperation, shared and autonomous navigation with Fuzzy logic engines and Behaviours arbitration mechanisms. All these applications demonstrate that CoRoBA provides enough flexibility to develop a large panel of control architectures.

Furthermore, the code the developer has to write is generally limited to a few tens of lines, that is, to the code implementing the application algorithms. All the rest is provided by the framework.

The presented applications also demonstrate the use and usefulness of the simulator that has been presented in the previous chapter.

Chapter VII Conclusion

While the present implementation of CoRoBA already offers much satisfaction, it is certainly possible to improve it in many ways.

One possible bottleneck in the current implementation is the Notification Service. Each component has to contact this service in order to connect to an Event Channel. That means that all data transit by this service that redistributes them to the connected clients. If the number of components and the traffic increase too much the Notification Service could become the bottleneck of the application. One solution to reduce this potential limitation would be to use several Notification Services. This would necessitate to modify the implementation by allowing a component to be able to find and narrow different Notification Services.

Anyone willing develop with CoRoBA will need to know and understand the fundamentals of CORBA. While these are not to difficult, it could be opportune to hide as much as possible the CORBA flavour in order to facilitate the development and to widen the audience.

Tools are also needed for writing management scripts and graphical interfaces for managing the life and run cycle of components.

A robust infrastructure supporting the deployment of the components would be appreciated for large applications.

In the current implementation a component provides a single service and the most simple application requires that several components and processes be deployed. CORBA provides a mechanism called "collocated objects" that allows a process to give access to many objects and thus services. With this mechanism, a server and a client may reside in the same component and consequently communicate directly without using TCP/IP mechanism.

Besides improvements of the framework itself, developing more applications is also required. For example, CoRoDe could be reimplemented with CoRoBA. For this we would have to add the following elements:

- The control of the 3D scanner.
- The data acquisition with the Metal Detector and the visualisation of the acquired data.

MoRoS3D could be used to simulate a mine detection system by adding a mine detection sensor as well as and visualisation and map generation components. It could also be used to simulate risky interventions of mixed robots and humans teams after natural or artificial disaster (explosion in a chemical plan, earthquake, ...)

Other possible improvements of the MoRoS3D simulator are:

- to add more robots and sensors,
- to develop a plug-in mechanism for adding new robots,
- to implement 3D terrain following,
- to use Multi-body dynamics for simulating shocks and friction.

In this thesis the design, implementation and evaluation of a multi-sensor robotic control framework has been performed and typical components used in robotic applications have been developed to validate the framework. Besides the particular comments made along all chapters, there are a number of general conclusions that can be drawn from this work.

The systematic analysis and the decomposition of typical applications into Robotic Control Pattern facilitated the identification of generic requirements. Those were completed by requirements from a computational point of view to finally yield an exhaustive list that was used as guidelines in the design phase.
The architecture of the framework based on suited Design Patterns and object oriented techniques provides a robust implementation without limiting future modifications.

Moreover, the separation of components into Sensor, Processor and Actuator gives the developer a clear view of the functionality of each component. It also facilitates the choice concerning the granularity and the partitioning and promotes the modularity and the distribution of applications.

In order to validate the proposed framework, concrete components have been implemented to build applications according to the Robotic Control Patterns presented in this thesis. It has been shown that complex applications can be built with an incremental approach and that the possibility to reuse previously developed components reduces the development time.

Besides the framework and the components a 3D simulator completes the development solution proposed in this work. But as the simulator is totally independent from CoRoBA, it can be used without it and in this case, robots can be controlled by programs written in any language having a CORBA mapping. On the other way, another simulator with a CORBA interface could be used with the framework.

Off course this is only the beginning of the story and future developments will certainly require to modify and to extend the framework but we are strongly convinced that thanks to the modular architecture this will not be a real challenge.

Bibliography

[ALAMI98] Alami R., Chatila R., Fleury S., M. Ghallab, F. Ingrand, An Architecture for Autonomy, International Journal of Robotics Research, special issue on "Integrated Architectures for Robot Control and Programming", 1998

[ALAMI00] Alami R. & all., Around the Lab in 40 days..., IEEE ICRA conference 2000, San Francisco (USA), pp 88-94

[ARNA00] Arnaud P., Des Moutons et des Robots, Presses Polytechniques et Universitaires Romandes, Collection Meta, ISBN: 2-88074-458-X

[BALE00] Balen H., Distributed Object Architectures with CORBA, Cambridge University Press, Managing Object Technology Series, 2000, ISBN 0-521-65418-1

[BROO85] Brooks R. A., A Robust Layered Control System for a Mobile Robot, AI Memo 864, MIT, 1985

[BRUY02] Bruyninck H. And Soetens P., The OROCOS Project, http://people.mech.kuleuven.be/~psoetens/orocos/doc/orocos-overview.html

[COLL05] Collett Toby H.J., MacDonald Bruce A. and Gerkey Brian P., Player 2.0: Toward a Practical Robot Programming Framework. In Proceedings of the Australasian Conference on Robotics and Automation, Sydney, Australia, December 2005.

[COLO96] Colon E., Baudoin Y., Development and evaluation of distributed control algorithms for the mobile robot Nomad200, Mobile Robot and Automated Vehicle Control Systems, SPIE PHotonics East 1996, Boston

[COLO98] Colon E, Viewing and controlling a mobile robot with common web technologies, Aerosense 98, Orlando, USA

[COLO99] Colon E., Virtual and Augmented Reality Aided Vehicle Control, ISMCR, June 1999, Tokyo, Japan

[COLO02] Colon E., Review of Software Frameworks for Robotics Applications, Technical report, Royal Military Academy, Not published.

[COLO02a] Colon E. & all, An Integrated robotic system for antipersonnel mines detection, Control Engineering Practice 10, 2002, p 1283-1291

[COLO02b] Colon E., Review of Robotic Control and Teleoperation Architectures, Technical report, Royal Military Academy, Not published.

[COLO04] Colon E., Evaluation of CORBA communication models for the development of a robot control framework, HUDEM04, Brussels, June 2004.

[COLO06b] Evaluation of CORBA Communication Models, Colon E., Technical report, Royal Military Academy.

[COLO06a] Installation and configuration of CoRoBA (including MoRoS3D), Colon E., Technical report, Royal Military Academy.

[COUL01] Coulouris G., Dollimore J., Kindberg T., Distributed Systems Concepts and Design,

Addison-Wesley, Pearson Education, Third edition 2001, ISBN 0-201-61918-0

[DOUG03] Douglass B. P., Real-Time Design Patterns, Addison-Wesley, Object Technology Series,2003, ISBN 0-201-69956-7

[COTE04] Côté C., Létourneau D., Michaud, F. Valin, J.-M., Brosseau, Y., Raievsky, C., Lemay, M., Tran, V., Code Reusability Tools for Programming Mobile Robots, Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2004.

[ERIC04] Ericson C., Real-Time Collision Detection, The Morgan Kaufmann Series in Interactive 3-D Technology, 2004, ISBN: 1558607323

[FERB99] Ferber J., Multi-Agent Systems, Addison-Wesley, ISBN 0-201-36048-9

[GAMM95] Gamma E., Helm R., Johnson R and Vlissides J., Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, Professional Computing Series, 1995, ISBN 0-201-63361-2.

[GILL02] Gill, C. & Smart, W. (2002). Middleware for Robots?, In Intelligent Distributed and Embedded Systems, Papers from the 2002 AAAI Spring Symposium, Gaurav S. Sukhatme and Tucker Balch (Ed.), pages 1-5, 2002.

[GOWD00] Gowdy, J. (2000). A Qualitative Comparison of Interprocess Communications Toolkits for Robotics, Internal report CMU-RU-TR-00-16, the Robotics Institute, Carnegie Mellon University, Pittsburgh, PA

[GRAV00] Graves A. R. And Czarnecki C, Design Patterns for Behavior-based Robotics, IEEE Transactions on Systems Man and Cybernetics Part A: Systems and Humans. January 2000, Vol. 30, N°1, pp. 36-41.

[GEER05] Thomas Geerinck, Valentin Enescu, Ioan Alexandru Salomie, Sid Ahmed Berrabah, Kenny Cauwerts, Hichem Sahli: Tele-robots with shared autonomy: tele-presence for high level operability. ICINCO2005, September 2005, Spain, pp 243-250.

[HENN99] Henning M., Vinoski S. Advanced Programming with C++, Addison-Wesley, Professional Computing Series, 1999, ISBN 0-201- 37927-9

[HUST04] Huston D., Johnson J., Syyid U., The ACE Programmers Guide, Addison-Wesley, Pearson education, 2004, ISBN 0-201-69971-0

[JAME85] James, M. L., Smith, G. M., and Wolford, J. C., Applied Numerical Methods for Digital Computation, 3rd. ed., Harper & Row, New York, 1985.

[Li00] Li S., Professional Jini, Wrox Press Inc, 2000, ISBN 1-861003-55-2, pp 90-129.

[MIRO01] Enderle S., H. Utz, S. Sablatnön, S. Simon, G. Kraetzschmar, G. Palm, MIRO: Middleware for autonomous mobile robots, in Proceedings of Telematics Application, pp 149-154, Weingarten, Germany, 2001

[MIRO03] Miro Manual Version 0.9.4 (10 Jan 2006). Available at: smart.informatik.uni-ulm.de/MIRO/miro_manual.pdf

[MOLL97] Moller, T., A fast triangle-triangle intersection test. Journal of Graphics Tools. (1997)

[MUNT99] Munteanu A., Cornelis J., Van der Auwera G. and P. Cristea, Wavelet-based lossless compression scheme with progressive transmission capability, International Journal of Imaging Systems and Technology, Special Issue on Image and Video Coding 10 (1999),

no. 1, 76–85.

[NELI05] Nelissen M., Autonomous Mobile Robot Software Design, Master Thesis, Vrije Universiteit Brussel, September 2005.

[PETE01] Peterson L., D. Austin, H. Christensen, DCA:A Distributed Control Architecture for Robotics, IROS 2001 - IEEE International Conf. on Intelligent Robots and Systems, Hawaï, USA, 29 oct. – 3 nov., 2001

[PETE02] Peterson L., A Framework for Integration of Processes in Autonomous Systems, Doctoral Dissertation, Kungl Tekniska Högskolan, Stockolm 2002

[PROS99] Prosise J., Programming Windows with MFC, Microsoft Press, 1999, ISBN 1572316950

[SCMI00] Schmidt D., Stal M., Rohnert H. and Buschmann F., Pattern-oriented Software Architecture: Patterns for concurrent and Networked Objects, Volume 2. Wiley & Sons, New York, 2000

[SHER92] Sheridan T. B., Telerobotics, Automation and HumanSupervisory Control, The MIT Press

[SHIN93] Jyh-Shing, Jang R., ANFIS: Adaptive-Network-Based Fuzzy Inference System, 1993, IEEE Transactions on Systems, Man and CyberneticsPart B: Cybernetics, Vol. 23, NO. 3, 665-685.

[SMAR05] Smart J., Hock K. and Csomor S., Cross-Platform GUI Programming with wxWidgets, Prentice Hal, 2005, ISBN 0131473816Press, Cambridge, ISBN 0-262-19316-7

[SUGG82] Sugeno M and Takagi T., Derivation of fuzzy control rules from human operator's control actions, 1983, Proc. of the IFAC symposium on Fuzzy Information, Knowledge Representation and Decision Analysis, 55-60.

[TOBY05] Toby H.J. Collett, Bruce A. MacDonald, and Brian P. Gerkey, Player 2.0: Toward a Practical Robot Programming Framework. In Proceedings of the Australasian Conference on Robotics and Automation, Sydney, Australia, December 2005.

[TREPA03] Trépanier F.-E. and MacDonald B., Graphical simulation and visualisation tool for a distributed robot programming environment. In Proceedings of the Australasian Conference on Robotics and Automation, CSIRO, Brisbane, Australia, December 1-3 2003

[VERL05] Verlinden O., Kouroussis G.,Conti C., EasyDyn: A framework based on free symbolic and numerical tools for teaching multibody systems, Proceedings of the Multibody Dynamics 2005, ECCOMAS

[VINO02] Vinoski S., Where is Middleware?, IEEE Internet Computing magazine, March-April 2002, pp 83-86

[WALS02] Walsh A. & Gehringer D, Java3D API Jump-Start, Prentice Hall, ISBN 0-13-034076-6

Publications

Journals

- CoRoBa, a multi mobile robot control and simulation framework, Colon E., Shali H., Baudoin Y., Special Issue on "Software Development and Integration in Robotics" of the International Journal on Advanced Robotics, pp 73-78,Volume 3, Number 1, March 2006.

Conferences

- Development and evaluation of distributed control algorithms for the mobile robot Nomad200, Colon E., Baudoin Y., Mobile Robot and Automated Vehicle Control Systems, SPIE Photonics East 1996, Boston
- Viewing and controlling a mobile robot with common web technologies, Colon E, Aerosense 98, Orlando, USA
- Virtual and Augmented Reality Aided Vehicle Control,Colon E., ISMCR workshop, June 1999, Tokyo, Japan
- Software modularity for mobile robot applications, Colon E., Sahli H., CLAWAR2003, September 2003, Catania, Italy, p 417 - 424
- Evaluation of CORBA communication models for the development of a robot control framework, Colon E., HUDEM04, Brussels, June 2004.
- CoRoBA, an Open Framework for Multi-Sensor Robotic Systems Integration, Colon E., Sahli H., CIRA2005, June 2005, Helsinki, Finland
- Distributed Control of Robots with CORBA, Colon E., Sahli H., Baudoin Y., ISMCR05, 8-10 November 2005, Brussels, Belgium
- Development of a control architecture for the ROBUDEM outdoor mobile robot platform, Daniela Doroftei, Eric Colon, Yvan Baudoin, IARP Workshop RISE, Brussels, June 2006
- Tele-robotics:Distributed Training-oriented Navigation Framework,Thomas Geerinck, Eric Colon, Sid Ahmed Berrabah, Kenny Cauwerts,Hanene Bahri, Hichem Sahli, IARP Workshop RISE, Brussels, June 2006
- MoRoS3D, a multi mobile robot 3D simulator, Eric Colon, Hichem Sahli, Yvan Baudoin, Introductory Workshop to the CLAWAR 2006 conference, Brussels, 11 September 2006.
- A modular control architecture for semi-autonomous navigation, Daniela Doroftei, Eric Colon, Yvan Baudoin, CLAWAR 2006 conference, Brussels, 11 September 2006.

Technical reports

- Review of Software Frameworks for Robotics Applications, Colon E.,Technical report, Royal Military Academy.
- Review of Robotic Control and Teleoperation Architectures, Colon E., Technical report, Royal Military Academy.
- Installation and configuration of CoRoBA (including MoRoS3D), Colon E., Technical report, Royal Military Academy.
- Evaluation of CORBA Communication Models, Colon E., Technical report, Royal Military Academy.

Appendices

Appendix A: Unified Modelling Language Notation

UML defines diagrams that are suited for the software development in the specification, design and implementation phases.

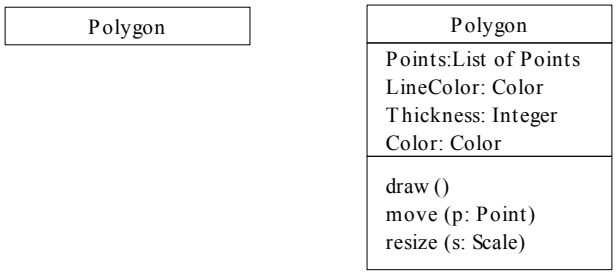
Structural Modelling

Class diagrams are certainly the most important diagrams. They offer a static view of the system by representing classes and relations between them. We distinguish class diagrams that describe the general model of the system and objects diagrams that represent particular instances of these classes.

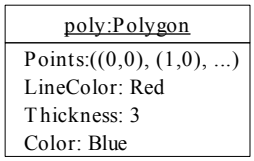
A class diagram contains three parts:

- The name of the class (abstract class names are written in *Italic*)
- The list of attributes
- The list of operations

The last two parts can be omitted.

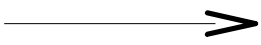


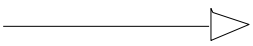
An object is depicted by a rectangle containing two parts:



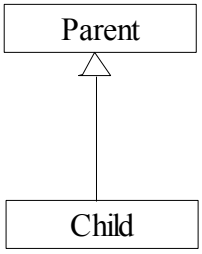
Classes generally collaborate to build an application. There exists several types of relationships between classes:

Dependency 

Association 

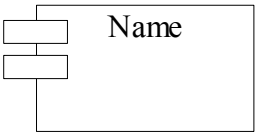
Generalization 

Classes inheritance is represented by a generalization relationship:

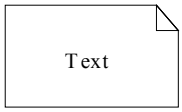


Other building blocks of UML are:

Component:



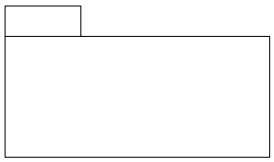
Note:



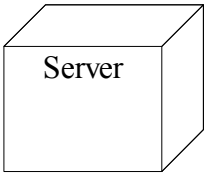
Interface:

IService

Package:



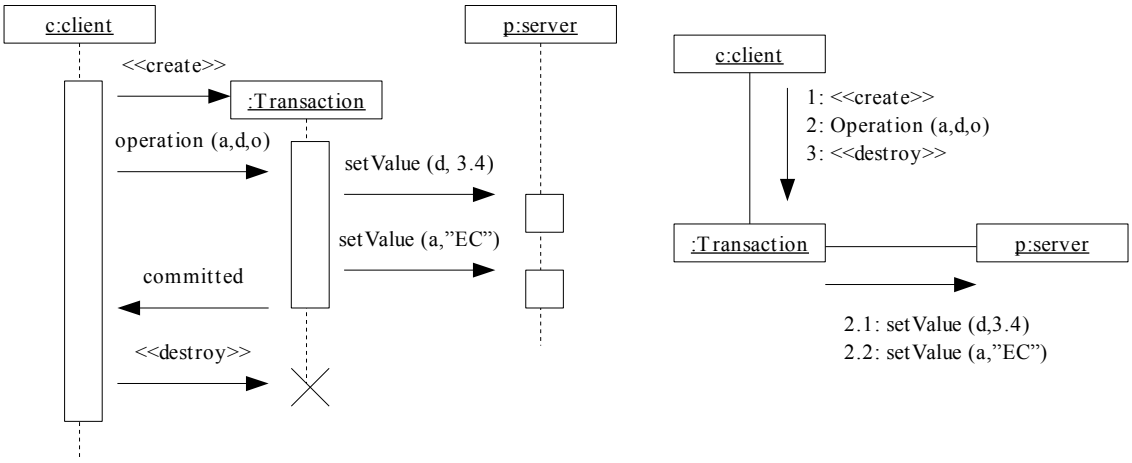
Node (Computer):



Active Class:

Behavioural Modelling

Sequence diagrams and collaboration diagrams are two of the five diagrams used in the UML for modelling the dynamic aspects of systems. A sequence diagram (left) is an interaction diagram that emphasizes the time ordering of messages while a collaboration diagram (right) is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages.



Appendix B: Service Interface

The commented file RMA_Service.idl is listed hereafter.

```
1       #ifndef RMA_IDL
2       #define RMA_IDL
        ...
46      #endif
```

These instructions are a classical method for avoiding multiple includes of the same file in an other one.

```
4       #pragma prefix "rma.ac.be"
```

Every type in a specification is assigned a repository identifier (ID) by the IDL compiler. The ID's are stored in the Interface Repository and allow run-time access to IDL definitions. These ID's are formed by the different scope names in the IDL file. The prefix **pragma** permits to add a unique identifier to a repository ID to avoid name clashes.

The ID's for the module and the Service interface are :
IDL:rma.ac.be:RMA:1.0
IDL:rma.ac.be:RMA/Service:1.0

```
6       module RMA {
...
45      };
```

IDL uses the **module** construct to create namespaces. Modules are mapped to namespaces in C++ and packages in Java.

```
8       interface Service {
...
44  };
```

Instruction on line 8 defines a new **interface** called Service. Interfaces are mapped to abstract classes in C++ and Interfaces in Java. The difference with C++ classes is that interfaces don't distinguish public, protected or private areas and don't have member variables.

```
10      typedef unsigned long Msec;
```

A new type is defined for transferring time values in milliseconds.

```
12      enum SvcMode { TRIGGER_MODE, PERIODIC_MODE, SYNCHRO_MODE};
13      enum SvcType { SERVICE_TYPE, SENSOR_TYPE, PROCESSOR_TYPE,
                ACTUATOR_TYPE };
```

Three modes have been defined for event-based communication: Synchronous, Periodic and External trigger. An enum type is defined for generic services and for each component type (see hereafter). The

way data is handled and how components act and react depends on the selected mode and on the component type.

```
15      struct SvcInfo {
16              SvcType type;
17              string author;
18              string version;
19      }
41       SvcInfo get_info();
```

The SvcInfo data structure stores information about the service. This information can be queried by other services by invoking the *get_info* operation.

```
21      exception CannotStart { string msg_error; };
22      exception UnknownMode { string available_modes };
23      exception BadMode ( SvcMode requested_mode, string permitted_mode };
24      exception NotRunning {};
25      exception AlreadyRunning {};
```

These user exceptions are thrown by the operations listed below.

```
27      SvcMode get_mode();
28      void set_mode (in SvcMode mode) raises (UnknownMode);
```

These operations allow to set and get the service mode. An exception is raised by the *set_mode* operation if the mode is not known by the service.

```
30      Msec get_period();
31      void set_period (in Msec period) raises (BadMode);
32      Msec get_duration () raises (NotRunning);
```

The *set_period* sets the desired execution period for the PERIODIC mode. An exception is thrown by the *set_period* if one attempts to change the period if the service is not in this mode. The *get_period* retrieves this value. The *get_duration* gets the real duration of a loop execution. This allows to adjust a timeout to reach the desired period duration. The *get_duration* raises an exception if the service is not currently running when invoked.

```
34      void trigger () raises (BadMode);
```

A service uses this operation to fire the data processing in the TRIGGER mode. If not in this mode, an exception is thrown.

```
36      void start() raises (CannotStart, AlreadyRunning);
37      void stop() raises (NotRunning);
38      void pause() raises (NotRunning);
39      void wakeup() raises (NotRunning);
```

Operations defined at lines 36 to 39 are used to manage the components run cycle. This is further explained in section 2.

```
43      oneway void destroy();
```

The destroy operation allows to close the service remotely.

Appendix C :NotificationService Operations