UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Departamento de Informática y Sistemas

TESIS DOCTORAL

CoolBOT: un Marco de Programación Orientado a Componentes para Robótica

(CoolBOT: a Component-Oriented Programming Framework for Robotics)

Antonio Carlos Domínguez Brito

Julio 2003

UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA

Departamento de Informática y Sistemas

Tesis Titulada CoolBOT: un Marco de Programación Orientado a Componentes para Robótica, que presenta D. Antonio Carlos Domínguez Brito, realizada bajo la dirección del Doctor D. Jorge Cabrera Gámez y la codirección del Doctor D. Francisco Mario Hernández Tejera.

Las Palmas de Gran Canaria, Julio de 2003

El director

Jorge Cabrera Gámez

El codirector

Francisco Mario Hernández Tejera

El doctorando

Antonio Carlos Domínguez Brito

A mis padres, Carmen Rosa y Vicente;
a mis hermanos, Vicente Manuel e Inma Pino;
y a mi abuelo, Antonio.

Agradecimientos

Escribir los agradecimientos de un trabajo como el que se presenta en este documento no es una labor trivial. Uno teme que alguien pueda olvidársele, puesto que hasta la más pequeña ayuda ha sido valiosa.

Lo primero que he de agradecer es el hecho de haber sido acogido como Becario de Investigación de la Universidad de Las Palmas de Gran Canaria bajo la dirección del doctor Francisco Mario Hernández Tejera en el seno de un grupo de investigación repleto de gente, proyectos e iniciativa.

A lo largo de la realización de una tesis, uno se encuentra muchas veces desorientado y perdido, incluso llegando a la duda metódica del "todo lo hecho está mal". Agradezco a mi director de tesis, el doctor Jorge Cabrera Gámez, el haberme sacado múltiples veces de infinitos atolladeros en los que me empecinaba en permanecer. También debo agradecer todas sus profundamente argumentadas críticas, no hay nada que más ayude. Tampoco me olvido de la infinidad de tés sin azúcar que me he tenido que tomar en su despacho. Eso también lo agradezco.

También, a mi codirector, al doctor Francisco Mario Hernández Tejera tengo que agradecer su confianza, su promoción constante de nuevas ideas y sus siempre constructivas críticas. No me olvido tampoco de su increíble capacidad para contar historias. Aún recuerdo como nos ilustró acerca de la memoria de los peces con su imitación de un pez en una pecera. Memorable.

Debo dar también gracias al resto de compañeros del grupo de investigación,

especialmente a Modesto y Daniel por su ayuda en la realización del resumen en español.

No puedo olvidarme aquí de "mis" usuarios. Gracias Claudio y Jose Luis por ser las primeras "cobayas" de CoolBOT, además de mi mismo. Espero que pronto aumente este club.

Esta tesis nunca hubiera sido posible sin la existencia del Programa de Becas de Investigación y de Formación de Profesorado de la Universidad de Las Palmas de Gran Canaria del cual he sido beneficiario durante 3 años. Por otro lado, este trabajo también ha sido financiado en parte, a través del proyecto de investigación con referencia **1FD1997-1580-C02-02** de la Unión Europea y de la Dirección General de Enseñanza Superior, y a través del proyecto de investigación con referencia **PI/1999/153** del Gobierno de Canarias. Es preciso aquí dar las gracías a dichas instituciones por el apoyo que dan a la investigación en general, y en particular, por el apoyo que han brindado a este trabajo.

Finalmente, y para terminar, el más importante de todos los agradecimientos. Aquí tengo que dar las gracias a mis padres, el ejemplo, infinito apoyo y confianza que me han dado en lo que llevo de vida tienen mucho que ver con este trabajo.

Muchas gracias a todos.

Contents

List of Figures

List of Tables

Resumen

Programar software para sistemas robóticos con el objeto de construir sistemas que funcionen y se desenvuelvan adecuadamente según sus especificaciones de diseño, continua siendo una tarea que precisa un importante esfuerzo de desarrollo. Actualmente, no hay paradigmas de programación claros para este tipo de sistemas, y las técnicas de programación que son de uso común hoy, no son adecuadas para tratar la complejidad asociada con ellos. El trabajo presentado en este documento describe una herramienta de programación, un *framework* o *marco*, que ha de considerarse como un primer paso para idear herramientas para manejar esta complejidad. En este framework, el software que controla un sistema se ve como una red dinámica de unidades de ejecución interconectadas a través de caminos de datos. Cada una de estas unidades de ejecución o *componente* es un *autómata de puertos* que proporciona una funcionalidad dada oculta tras una interfase externa, que especifica qué datos se cosumen y cuáles se producen. Los componentes, una vez definidos y construidos, se pueden instanciar, integrar y utilizar en múltiples sistemas. El framewok proporciona la infraestructura necesaria para dar soporte a este concepto de componentes y la intercomunicación entre ellos mediante caminos de datos (*conexiones de puertos*) que pueden establecerse y desestablecerse dinámicamente. Además, y considerando que cuanto más robustos sean los componentes que conforman un sistema, más robusto será el sistema, el framework proporciona la infraestructura necesaria para controlar y monitorizar los componentes que integran un sistema en cualquier momento.

Abstract

Programming software for controlling robotic systems in order to built working systems that perform adequately according to their design requirements remains being a task that requires an important development effort. Currently, there are no clear programming paradigms for programming robotic systems, and the programming techniques which are of common use today are not adequate to deal with the complexity associated with these systems. The work presented in this document describes a programming tool, concretely a *framework*, that must be considered as a first step to devise a tool for dealing with the complexity present in robotics systems. In this framework the software that controls a system is viewed as a dynamic network of units of execution inter-connected by means of data paths. Each one of these units of execution, called a *component*, is a *port automaton* which provides a given functionality, hidden behind an external interface specifying clearly which data it needs and which data it produces. Components, once defined and built, may be instantiated, integrated and used as many times as needed in other systems. The framework provides the infrastructure necessary to support this concept for components and the inter communication between them by means of data paths (*port connections*) which can be established and de-established dynamically. Moreover, and considering that the more robust components that conform a system are, the more robust the system is, the framework provides the necessary infrastructure to control and monitor the components than integrate a system at any given instant of time.

Chapter 1

Introduction

Building software for robotic systems is a very complex and difficult task. The work presented in this document is aimed to reduce the effort needed to program this kind of systems. In this first chapter we will illustrate the problem and the motivations that have brought us to invest efforts in this work, equally, we will briefly outline which aspects we consider are its main contributions.

1.1 Introduction

Programming software for controlling robotic systems in order to built working systems that perform adequately according to their design requirements remains being a task that requires an important development effort. Intrinsically, robotic systems are complex because, in general, they share several important sources of complexity that complicates their programming.

Usually, in this kind of systems, there are involved several and different types of sensors and effectors, each one with their own features and particularities, and needing a different APIs (**A**pplication **P**rogramming **I**nterface) for their programming.

Additionally, even the simplest systems involve multiple computers hosting different operating systems that demand distinct APIs and programming tools, and even different models of computation (embedded, general-purpose, real-time, etc.). Software is not always portable between different operating systems, specially if non-general-purpose operating systems are utilized.

Another source of complexity is the existence of several data links between the units of computation that conform a robotic system: RS-232 links, USB links, TCP/IP links, dedicated buses (I^2C, CAN, ...), infrared links, etc. It is normal that all of them use distinct protocols and APIs, and that impose different bandwidths for communications.

Concurrency and parallelism between processes and threads running in the same or different machines constitute another source of problems to which it is necessary to

put much attention. Here synchronization and the interactions between the units of execution that conform the system become a specially complex issue.

As to human resources, there is another source of complexity. Typically during the process of developing and building robotic systems there are multiple persons taking part into, usually each one having different roles (control engineers, computer-vision scientists, artificial-intelligence scientists, etc.), and, each of them possibly using different programming paradigms and methodologies. Evidently, this problem scales up when the persons involved work in different laboratories.

Finally, robotic systems are aimed at and devised to carry out complex tasks with a wide range of requirements: soft and real time constraints, physical resources, responsiveness, robustness, autonomy, etc. It is not strange that the orchestration of all this complexity coming from such a variety of sources makes the construction of this type of systems a challenge.

In spite of the success in different application fields of multiple robotic systems (robot-assisted surgery, agriculture robots, entertainment robots, autonomous museum robots, highway autonomous driving, space robotics, active vision systems, etc.), the problem of dealing with the complexity inherent in programming robotic systems remains. Moreover, this complexity is increasing because there are high demands for systems having more complex functionalities, performing more complicated tasks and having more "intelligent" behaviors.

At the present moment, there are no clear programming paradigms for programming robotic systems, and the programming techniques which are of common use today are not adequate to deal with the complexity associated with these systems. One aspect where this complexity clearly comes up is software integration. In a given system normally it is necessary to integrate a wide variety of software: software dealing with hardware (sensors, effectors, other hardware), software done by other people, software which is not very portable because it is specific to a particular operating system or machine (or both), software done in distinct programming languages, etc. Traditionally software integration has been an underestimated problem in robotics, and frequently it is a question to which it is necessary to invest much more effort than considered initially. Some other authors [Kortenkamp and Schultz, 1999] have identified already this problem. Nowadays it is not only necessary to develop complex algorithms, but also to integrate complex systems that really perform adequately to its own capacities.

Software system integration is a task demanding so many resources that only a few research groups can afford it. It seems evident that fostering cooperation and code reuse between different research groups would be the more convenient solution, but in practise, it has been very rare to see research groups "importing" architectures or systems that has been developed by others. In fact, reuse and recycling of code across laboratories is difficult and nowadays not very common. It is clear that robotics needs to develop an experimental methodology that promotes the reproduction and integration of results and software between different research groups. There are multiple reasons for this situation. In general, the approaches originated by distinct groups have not been designed to be integrated together, and usually, the software for control

robotic systems is not easy-to-use software. Its use and learning is not trivial, and getting to a level of expertise high enough to have productive results takes no little time. All that drives frequently to develop home-made software fitting the specific necessities of each group. On the other side, adopting approaches coming from other laboratories could mean to abandon own ideas. Other authors [Coste-Maniere and Simmons, 2000] have made already similar considerations identifying the building of software architectures as the way the robotics community has mainly chosen to address the problem. In fact, multiple research groups are currently working on the construction and definition of "the software architecture" where everybody could integrate its results. However, it is not clear that imposing "an architecture" should be the way to follow. In fact, another authors [Fleury et al., 1997] [Schlegel and Wörz, 1999a] are working on more generic programming tools like frameworks, which are neutral in terms of control and system architecture, we think it is in this last group where the work presented in this document should be situated.

Thus, this thesis describes a programming tool, concretely a framework, that must be considered as a first step to devise a tool for dealing with the complexity present in robotics systems, and mainly with software integration. In this framework the software that controls a system is viewed as a dynamic network of units of execution inter-connected by means of data paths. Each one of these units of execution is called a *component*. Each component has a clear functionality and a well established external interface specifying which data it needs and which data it produces. Components, once defined and built, may be instantiated, integrated and used as many times as needed in other systems. The framework provides the infrastructure necessary to support this concept for components and the inter-communications between them by means of data paths which can be established and de-established dynamically. In addition, and considering that the more robust the components that conform a system are, the more robust the system is, the framework provides the necessary infrastructure to control and monitor the components than integrate a system at any given instant of time.

In the remaining sections of this chapter we will present more in detail the motivations that drove us to develop this work, which technical challenges need to be faced, our main contributions, and, finally, a brief outline of the rest of the document.

1.2 Motivation

The work presented in this document has been mainly motivated by practical reasons. We have experienced ourselves that programming robotic systems is a task of enormous complexity that requires an important development effort [Hernández-Tejera et al., 1999] [Cabrera et al., 2000]. Specially we have experienced that, in addition to other problems, an important effort had to be dedicated to integrate the software that finally will constitute the system. There are no clear programming paradigms to program robotic systems, neither are there standard programming tools. In mobile robotics a large research effort has been devoted to architectures [Kortenkamp and Schultz, 1999] [Coste-Maniere and Simmons, 2000]

[Orebäck and Christensen, 2003].

Software integration is not only a problem of robotic systems. In other fields of computer science, as business software, "de facto" standard tools exist to define deployable units of software identified as software components (e.g., ActiveX from Microsoft [Chappell, 1996], JavaBeans for Java from Sun Microsystems [Monson-Haefal, 2001]), and a supplier component software industry even exists. On the contrary, in the robotic field there are not any established standards to address this problem. We consider this concept of deployable software components as a key concept to reduce software integration efforts.

A software component should be something like an electronic component or chip in electronic industry. It is many years that off-the-shelf chips can be bought and deployed in other parts of the world. Each component has a clear functionality and a well established external interface. Furthermore, numerous standard tools exist to design electronic devices based on the composition, assembly and combination of these electronic components. A similar panorama would be desirable for robotics. We consider that a concept of software component analogous to an electronic component would allow using them as pieces of deployable software. Imagine the software of a robotic system that were seen as the integration of multiple software components in the same way that electronic circuits are made from integrating electronic components.

Thus, the construction of a programming tool allowing to program robotic systems by integrating and composing software components was the main objective of the work presented here. This programming tool it is a component-oriented framework called CoolBOT that provides means to define components, to run and connect them and to monitor and control their operation.

1.3 Technical Challenges

There are several technical challenges that must be faced along the elaboration of a work like the one presented in this thesis. The first of them is finding an adequate model of software component. In this model, there should be a clear separation between external interface and internal functionality. The external interface should only express which data a component consumes and which data the component produces. In addition, analogously to electronic components, through this external interface a component should be able to connect to any other component. As to the internal structure, it should have their own units of execution (processes or threads).

Evidently, a programming paradigm based on interconnected software components needs to pay much attention to the mechanisms that will carry out at last term inter component communications through the connections established between components. A clear technical challenge is to choose which mechanisms would be used to support a model of communications based on connections between components. Such mechanisms should be generic enough to allow for any imaginable type of interaction between components. At the same time they would have to be efficient enough in

terms of computational resources to be comparable to the present ad-hoc solutions (shared memories, message queues, etc.). Furthermore, that components were in same machine, or in a different one residing in the same computer network, should be indifferent in terms of component integration and interconnection. Therefore, connecting two components residing in different machines should be as easy as if they were in the same one.

A question comes naturally out from a solution based on integrating software components. Is the whole the joining of its parts?, or, in other words, is a system just the joining of its components?. For us, the answer is no. If we integrate components we need means to observe and control them in order to obtain systems that we can also observe and control. Obviously the design and implementation of the infrastructure necessary to make components and systems able to be observable and controllable is also a technical challenge.

Finally, constructing a software tool that were generic and complete in order to be able to build any possible system is evidently another technical challenge.

1.4 Contributions

This document describes a software tool, concretely, a software framework called CoolBOT which permits to program robotic systems by integrating software components. In particular, CoolBOT allows to program robotic systems as if they were networks of interconnected software components. Software components that have been independently developed and built, and that have their own functionalities. When a component is integrated in a system taking part into a network of components, it executes following its internal own flow or flows of execution (threads), so they are independent entities that act under their own initiative. Additionally, the connections conforming these networks of components constitute data paths between them forming a topology of multiple consumers and producers where those data paths may be dynamically established and de-established. Moreover, connections may be established between components residing in the same or different machines over a computer network. Developers do not have to worry about the mechanism of inter-communication between components, they only have to define the internal functionality of the components, and which external interface they will offer. It is also possible to define compositions of components, in such a way that compositions can be also handled as single components.

We think the work presented in this document makes some interesting contributions that we enumerate more in detail as follows:

- **Uniformity**: The framework defines software components as units of functionality having an uniform external interface and an uniform internal structure. This uniformity makes components externally observable and controllable, and treatable in an uniform and consistent way. Furthermore, the uniformity the framework imposes makes them also integrable with others in order to built more complex systems.

- **Deployment, Reuse and Recycling**: The framework permits to program units of deployable software that are easily integrable wherever they are needed. This would allow to reduce integration efforts when programming robotic systems, since components from other projects, or other labs could be integrated and reused without much cost.

- **Visibility**: The framework provides means to externally observe and control the operation of a component through its external interface, whether individually or integrated with multiple components. We consider this is a key feature to build development tools as debuggers and profilers for components and systems.

- **Inter Component Communications**: The way components interact in CoolBOT is carried out through data paths established between components. Data along these data paths are asynchronously transported by the framework by means of a set of inter component communications mechanisms that make irrelevant for developers to worry about how the data are sent and received. They only have to indicate the format of those data, and what is sent and received, even when components do not reside in the same computer.

- **Multithreading**: Components in CoolBOT are active entities that have their own flow of execution, so they have internally at least a unit of execution (a thread), but components might make use of multiple threads if necessary, depending on their particular functionality and design. The framework allows defining such threads inside components and modelling its interactions using the same mechanisms that are used for inter-communicating components, therefore, developers do not have to worry about synchronization and race conditions between components and the threads they use internally.

- **A Model for Exception Handling**: The framework provides a model for exception handling. CoolBOT promotes an uniform approach to handling faulty situations, establishing a local level of exception handling inside individual components and an external level corresponding to exception originated in a composition of components.

- **Generality**: We consider the framework is generic enough not only to fit the requirements of the robotic systems domain, but also for other computer science domains like, for instance, multi-agent systems.

- **Asynchronous Model of Execution**: In general, CoolBOT favors a programming methodology that fosters concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing. We think these are features characteristic of robotic systems. This is a framework where all these concepts have been integrated and put together.

- **Control vs. Functionality**: CoolBOT provides means to separate control and functionality when programming systems. This separation favors independent development of components, and fosters software integration.

- **Operating System Support**: Currently, the framework is supported in the two most used operating systems: the Windows family of operating systems (Windows NT, 98, 2000 and XP), and GNU/Linux. Additionally, although the framework it is not real-time, it can keep soft real-time requirements, and in fact, it offers some mechanisms and resources which usually are characteristic of real time operating systems (timers, watchdogs, etc.)

The work presented in this document constitute a programming tool we think fits a taxonomy of problems (software integration, distributed computing, multithreading, inter process communication, synchronization, etc.) that usually should be faced when programming robotic systems. At last term, it provides with a programming model of asynchronous communications and computation which constitute in our opinion fundamental elements in any tool for programming robotic systems.

1.5 Outline of the Document

This document has been organized in five chapters and one appendix:

- **Chapter 1**: This is this introductory chapter.
- **Chapter 2**: This chapter presents initially some terminology that will be used along the rest of the document. Then a review of related research will be given in order to have a vision of the state-of-art. Finally, a brief introduction of CoolBOT, and an enumeration of the principles that have driven its design and implementation will be explained.
- **Chapter 3**: This is the main chapter were the whole framework will be explained in detail. From general concepts and abstractions to some implementation decisions.
- **Chapter 4**: In this chapter, some examples of using CoolBOT will be presented in order to illustrate some features of the framework we consider significant. At the same time these examples will be used as a proof-of-concept.
- **Chapter 5**: This last chapter is a recompilation of the results and conclusions we think we have achieved with the work presented in this thesis.
- **Appendix A**: Finally, this is an appendix, commenting the coding rules we followed when implementing CoolBOT, that can be of interest for anyone willing a better understanding of CoolBOT's code.

Chapter 2

Review of Related Research

How to programming robotic systems in order to built better and more capable systems and more easily, is a problem to which has already been dedicated research efforts in other laboratories and institutions. In this chapter we will present the related research we consider closer to the work presented in this document. But first some terminology will be introduced in order to share common terms along the rest of the document. Finally, a brief outline of the approach we propose is given.

2.1 Introduction

Programming the software that controls robotic systems is not an easy task due to the diversity of hardware and software typically involved, and the complexity of problems that it is necessary to solve. This is not a new problem, and consequently, it has already received attention by other research groups. However, it has been very rare to see research groups "importing" architectures or systems that has been developed by others. In fact, reuse and recycling of code across laboratories is difficult and nowadays not very common. In general, the effort which is necessary to invest to learn and get enough experience in a solution provided by other laboratory is frequently considered greater or equal than the effort necessary to design and develop a new "home-grown" solution. In other cases, the adoption of approaches originated by other research groups means to abandon own ideas about how the problem should be solved. As a consequence, it is not strange to find as many approaches of solution as research groups. Evidently, this diversity of approaches is the result of a domain with is still too recent and young, and where it is also too costly and difficult to reproduce the results of other groups.

In this chapter, in section 2.3, we will summarize the approaches adopted by other research laboratories in order to deal with the complexity inherent in the programming of operative and successful robotic systems. But first, in section 2.2, we will introduce and define some terminology with the aim of establishing a common base of concepts for the rest of the document. In the last section of the chapter, section 2.4 we

will give a brief outline of the approach of solution we propose and an enumeration of the design principles that have driven the development of the work presented in this thesis.

2.2 Terminology

In the area of programming robotic systems there is a confusing terminology where many terms overlap. In this document, a programming tool is being presented, in particular, a software framework, but what is a software framework?. On the other side, in the domain, the most frequent solutions to reduce the complexity when programming robots are software architectures, but what is a software architecture?. Which differences, if any, are there between frameworks and architectures?, and how long do these concepts overlap?. The title of this thesis talks about a component-oriented programming framework, but what does the term "component-oriented" mean?, and what is a component?. And above all, what do all these concepts have to do with robotic systems?. In the rest of this section we will try to answer all these questions by defining each one of these terms in order to understand what each refers to, at least in the scope of the work presented in this document. The definitions given for each term have been mainly inspired by [Szyperski, 1999] and [Gamma et al., 1995].

2.2.1 Programming Languages

A programming language is a set of instructions and objects which allow writing algorithms in terms of these instructions and objects. In a specific programming language, only the algorithms that it is possible to express in terms of its instructions and objects are practicable. Thus, if the language is not generic enough, there will be problems that can not be expressed and solved using that language.

There are programming languages in the whole range of generality and completeness. There are languages quite specific for a domain with a small set of instructions and objects that allow to express all the problems we can find in this domain of knowledge. On the opposite, there are general languages in terms of which any function can be computed in the sense of Turing [Turing, 1937].

The utility of programming languages is that if they are generic enough they allow expressing programs in a higher level of abstraction, so the semantic gap between the language that machines understand and our language is reduced. But when programs grow in size and complexity we need constructs to give structure to our programs, such as functions, modules, classes, name-spaces, etc; what drives us to concepts like libraries, objects, frameworks, architectures and software components.

Evidently, to define a programming language in a specific domain it is necessary to acquire enough knowledge about the domain to be able to express any imaginable problem. If the knowledge is insufficient there will be problems that could not be solved without redefining the language. On the other side, if the language becomes too

generic, it might end up being useless, because other general-purpose languages could do the same. Some languages have been defined for programming robotic systems where, to avoid the problem of not making a too-narrow or too-generic language, they have been defined based on existing programming languages just by adding to them some new instructions and objects. We will comment some of them in section 2.3.

2.2.2 Libraries

A library is a set of useful and reusable software, usually expressed in a specific programming language, which has been designed to provide useful, general-purpose functionality. Thus, for instance in C++, a typical library is a set of predefined classes, functions and data structures defined normally inside a specific name space which can be incorporated in an application. The main characteristic of libraries is that they provide functionality to the programs that use them, but not structure. Figure 2.1 helps to illustrate this idea. When libraries are used the developer usually has to write the body, the core of the application that makes calls to the different libraries that the program may use.

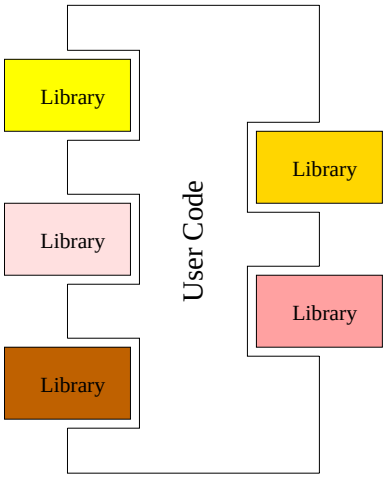

Figure 2.1: Using libraries.

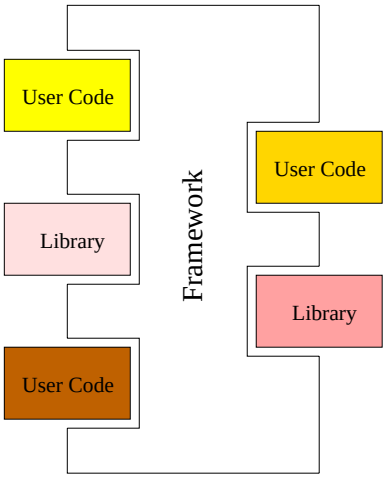

Figure 2.2: Using a framework.

2.2.3 Frameworks

A framework is a set of cooperating and reusable software that constitute a reusable design and structure for a particular domain of application. They have the same external aspect than libraries, since, for instance, a C++ framework may also take the form of a set of predefined classes, functions and data structures inside a specific name space. But the main difference between libraries and frameworks is that frameworks provide structure to the software that uses them. Figure 2.2 depicts graphically the idea.

In general, when a framework is used, it imposes the main body of the software we are developing. It establishes a skeleton for our software that then it is necessary to fulfill. A framework implies that the user has left some design decisions to framework implementers, and re-uses their design in the particular aspects the framework models. The main difference between a framework and a library is that using a framework the main body of the software under development is given by the framework (the skeleton), and the code we add to fulfill the skeleton and complete the application gets called by the framework at runtime. Just the opposite of using libraries where we establish our software main body and libraries get called from our code. It is frequent that frameworks define the control flow of the software where they are used, and usually the user code must follow particular conventions and naming schemes to integrate its code into the framework.

Frameworks allow to give some structure to programs without compromising generality too much. Making a framework constitutes an approach frequently taken when there is not enough knowledge to define a programming language in a specific domain. As it will be commented in the next section, section 2.3, some frameworks have been designed aimed specially to robotic systems, and, in fact, it is the approach we have taken in the work presented in this document.

2.2.4 Architectures

An architecture could be seen as a set of cooperating and reusable software which provides design and structure at different abstraction levels for a particular domain of software. Like frameworks, architectures also provides structure to the software where they are used. The difference between architectures and frameworks may be to a certain extent a bit blurred and fuzzy. In this document the concept of architecture presented in [Szyperski, 1999] has been adopted. Architectures provides software structural design at several layers of abstraction.

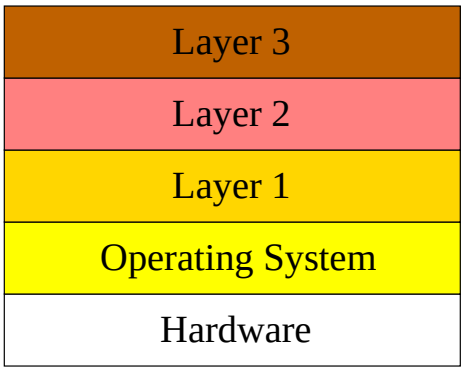

Figure 2.3: Strict-layered Architecture.

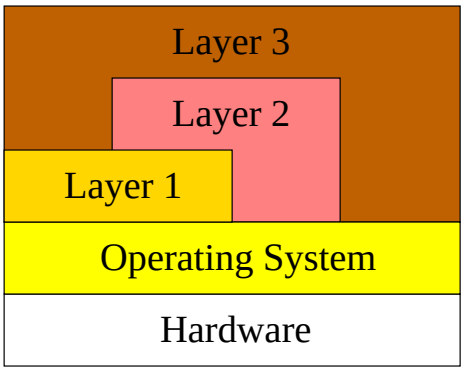

Figure 2.4: Non-strict-layered Architecture.

Architectures are usually classified in *strict-layered* architectures and *non-strict-layered* architectures. In strict-layered architectures a layer can only base its functionality on the operations and primitives offered by the layer immediately below. In turn,

it offers the operations and primitives with which the layer immediately above implements its own functionality. Figure 2.3 depicts a typical strict-layered architecture. In non-strict-layered architectures layers can also base its functionality on the operations and primitives in any of the layers below, not only strictly in the one immediately below, figure 2.4 shows an example.

Usually each layer in a software architecture constitutes the backbone of a software structure to which the user adds his/her code. The design of the software architecture guarantees the mechanisms of interaction between the objects or elements residing at different layers. Software architectures may be even more complex than that, some of them require the utilization of several programming languages, libraries and frameworks, or any other possible combination of them.

Frequently, a framework provides a specific infrastructure under which software architectures may be implemented. Thus, in general, software architectures are more stringent than frameworks, because they provide a more compromising and closer software design. Given an architecture designed and implemented for a specific domain, if there are systems in this domain that do not fit with the structural design established by the architecture, this one should be modified. Thus, like programming languages, architectures can be quite specific or generic, depending on the range of systems that match with the structural design they express.

There has been done a lot of work in software architectures aimed to robotic systems. A large research effort has been devoted to *hybrid architectures* [Arkin, 1998] [Kortenkamp et al., 1998] for autonomous mobile robots which are usually organized in three layers: the bottom or reactive layer, the intermediate or task control layer and the top or deliberative layer. The reactive layer is the closest to the hardware, so it deals directly with sensors and actuators, and tries to embody system behaviors. Typically, the behaviors correspond to software modules or a sort of combination of them. The second layer is a sequencer of behaviors in the lowest layer. The task execution layer is in charge of initiating, combining, and monitoring behaviors to achieve tasks defined in terms of reactive layer behaviors. The last layer, the deliberative one, is usually responsible for long-term deliberative planning, where plans are defined in terms of task carried out by the second layer. We will show several of them in section 2.3. As we will see, most of them constitute non-strict-layered architectures.

2.2.5 Software Components

The framework we present in this document is a "component-oriented" programming framework, but what is a component?. The concept of software components is not something sharp and clear, just the opposite, fuzzy and blurred. Out of the multiple definitions for software components we can find (chapter 11 in [Szyperski, 1999] is a good summary) we have chosen the following definition given in [Szyperski, 1999] (page 164):

> *"A software component is a unit of composition with contractually specified*

interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

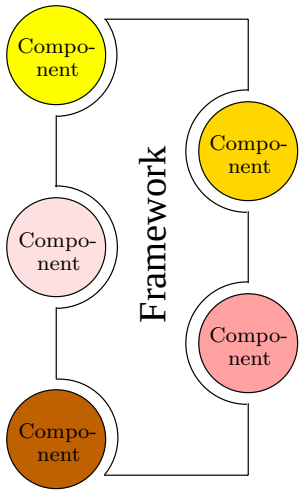

Figure 2.5: A component-oriented framework.

Thus, a software component is a piece of software that has been independently developed from where it is going to be used. It should offer a well-defined external interface that hides its internals, and it is independent of context until instantiation time. In addition, it can be deployed without modifications by third parties. Also, it needs to come with a clear specification of what it requires and provides in order to be able to be composed with other components. Deployment can be referred to as *binary deployment* and *source-code deployment*. An example of binary deployment is a binary program for a specific operating system. Another example is a java applet in byte codes. Both of them can be run directly without modifications. Binary deployment is specially used for proprietary software. For us, source-code deployment is referred when a piece of code can be added to other software at source code level without needing further modifications. Source-code deployment is more usual for open-source software.

All in all, a software component is characterized by the following features: uncoupling of external interface and internal implementation details, context-free design, ability to be subject to composition and integration with other components (integrability), and deployment (whether binary or source-code). Having these features, software components become deployable pieces of software that can be reused wherever needed.

The work presented in this document is considered a component-oriented framework because it allows designing systems in terms of composition and integration of software components. This framework provides means to design and built components and to compose and integrate them hierarchically and dynamically. Furthermore, it makes components share a minimal internal structure and a minimal external interface. The concept of a component-oriented framework is illustrated in figure 2.5.

2.3 Review of Related Research

Programming software for robotic systems is not a new problem. Multiple research groups have approached to the problem on their own way forging their solutions based on their own ideas and philosophy, know-how in the field, and experience and skillness with a determined set of programming tools. In this section we comment some of the approaches taken in other laboratories that we consider more significant or representative of the state-of-art in the field. We have classified them into three main groups: architectures, frameworks and programming languages, and others.

2.3.1 Architectures

The validation of different architectures originated by different groups is very difficult and costly. The same is true for transferring results between research groups. The main reason for this is that, in general, they have not been designed to be integrated together. In general, architectures for robotic systems are not easy-to-use software, their use and learning is not trivial, and getting to a level of expertise high enough to have productive results takes no little time. As a consequence, few studies comparing architectures quantitatively have been carried out with the significant exception of the work presented in [Orebäck and Christensen, 2003] which is an evaluation of three architectures for mobile robotics (Saphira [Konolige et al., 1997], TeamBots [Balch, 2000] and BERRA [Orebäck et al., 2000]) where a test problem was used for comparison.

A great research effort in the area of mobile robotics has established solid grounds for the combination of reactive and deliberative mechanisms (the hybrid deliberative/reactive paradigm [Arkin, 1998] [Murphy, 2000] [Kortenkamp et al., 1998]) to be widely considered as the best structural design to built control software for mobile robots. Although the ideas behind this paradigm are not only exclusive of mobile robotics, it is in this area where there has been a blossoming of multiple architectures that collectively are called hybrid architectures [Arkin, 1998] [Murphy, 2000]. The following paragraphs are devoted to the presentation of the architectures, most of them hybrid, that have been more influential or are more closely related to the work presented in this document.

2.3.1.1 Subsumption

This reactive architecture was proposed by Brooks [Brooks, 1986] and focuses on the reduction of system response times. The main idea is the building of robotic systems from the combination of basic perception/action behaviors. Some characteristic features include the followings:

- Use of behaviors as basic building blocks.
- Avoid as far as possible the utilization of world models that, in any case, should be restricted to a local scope.
- Take inspiration from biology.

The subsumption architecture makes use of supression/inhibition as coordination mechanisms for active behaviors. The user-defined hierarchy determines which competing behavior takes control on shared actuators. Pirjanian's work [Pirjanian, 1998] include an extensive analysis of alternative schemas for behavior fusion and action selection.

Purely reactive systems have proven their efficiency on multiple applications. However, they shown also some drawbacks [Tsotsos, 1995], being the most relevant:

- **Sensibility**: The objective of shortening response times can degrade stability if not managed properly. Simple noisy sensor readings could provoke undesirable effects on system performance.

- **Scalability**: The lack of modularity and mechanism to manage complexity make difficult to cope with problems of larger dimensions.

2.3.1.2 AuRA

The AuRA (Autonomous Robot Architecture) is a proposal by Ronald Arkin [Arkin and Balch, 1997]. It is a two-level hybrid architecture specially oriented to execute navigation tasks. AuRA implements ideas extracted from the study of biological systems and neurophysiology in robotic systems. It has been used in the construction of different robot applications in navigation, exploration and manipulation.

AuRA's structure comprises several components organized in two levels: a hierarchical deliberative component in the upper level and a reactive component in the lower level. The deliberative component, in turn, contains a mission planner, a spatial reasoner module and a plan sequencer. Linked to this level, the reactive component is designed as a motor schema controller.

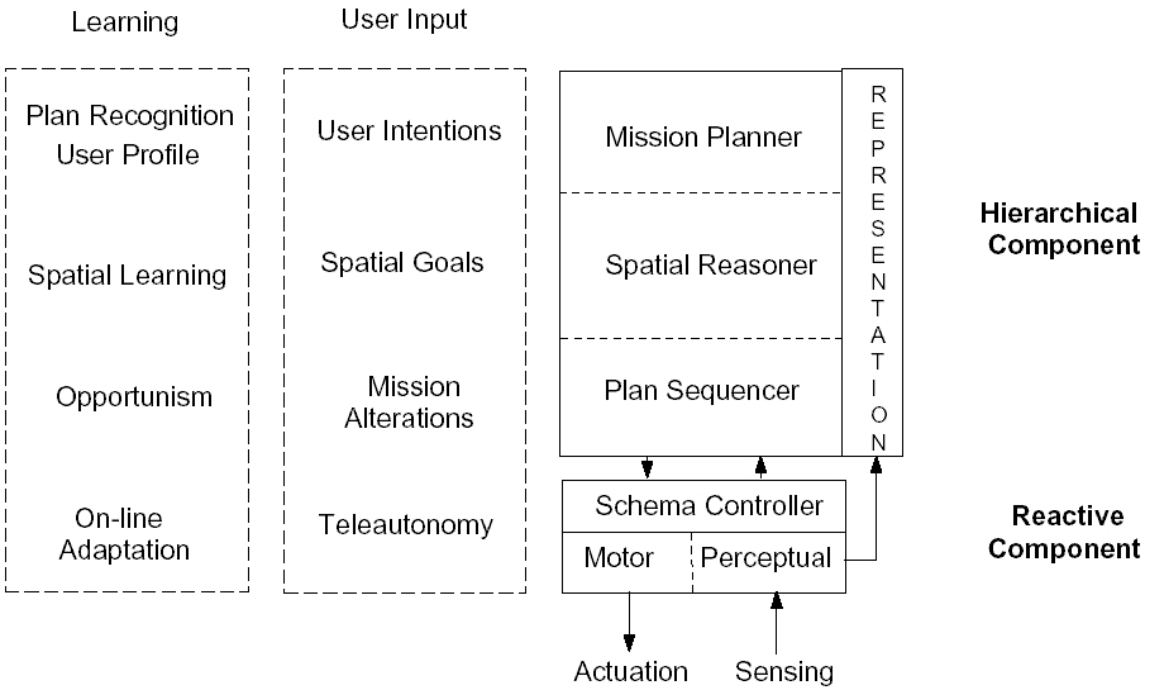

Figure 2.6: The AuRA architecture.

Inside the deliberative part, the mission planner generates high level goals and restrictions for system operation. The spatial reasoner module uses cartographic information to determine the appropriate sequence of trajectory segments the robot must

follow to complete it's mission. Finally, the sequencer translates each trajectory segment into a set of motor behaviors and demands their execution to the lower level.

At the reactive level, the schema controller monitors and controls at runtime the evolution of low-level behaviors. Each behavior or motor schema [Arkin, 1989] produces as output a response vector that is combined with the rest of motor schema's outputs to obtain the command for the physical robot. All behaviors operate asynchronously.

Once the reactive level's actions have been commanded, the deliberative level enters an idle state, waiting for either a successful task completion or an error signal. Errors are managed on an hierarchical sequence that begins in the sequencer and ends in the planner, if all lower levels fail to recover the system from the error situation.

The architecture is claimed to be highly modular. This aspect has been verified with the essay of different architecture configurations, changing the internal implementation of each level. Several adaptation and learning mechanisms have been incorporated to AuRA. Some examples are homeostatic control [Arkin, 1992], dynamic adaptation for behaviors using rule-based mechanisms [Clark et al., 1992], case-based reasoning for controlling behavior switching on environmental variations [Ram et al., 1992], or genetic algorithms for control loop tuning [Ram et al., 1994].

2.3.1.3 SFX

SFX (Sensor Fusion Effects) [Murphy, 2000] started out as an extension to AuRA, initially centered on sensor fusion and sensor error handling. This architecture comprises two levels, deliberative and reactive, inspired on biological cognitive models. A graphic scheme of SFX appears in figure 2.7. The sensor information is preprocessed locally before transmitted to both system layers for further analysis.

The deliberative component is divided into modules, each one implemented as a software agent interacting with the others. There is a supervisor agent, called the Mission Planner, that establishes the high level interface with the user and controls the evolution of the system. Below, three resource managers take care of sensor and effector allocation: the Task Manager, the Sensor Manager, and the Effector Manager. The Sensor Manager is of particular interest, as it contains specific strategies for evaluating sensor performance and solving problems. Other agents included in the deliberative level are a Cartographer and several performance monitors. The Cartographer is responsible for map making and path planning.

The reactive level subdivides in turn into two layers of strategic and tactical behaviors. The strategic behaviors represent the long term vision, trying to lead the system towards high level objectives. The tactical behaviors, on the contrary, face transitory situations, modifying the strategic outputs to overcome particular problems. In absence of contingencies, strategic indications prevail. An example of this organization is the navigation to goal (strategic) combined with obstacle avoidance (tactical).

SFX reactive control is similar to subsumption, but here the lower level tactical behaviors take control overriding commands from higher level behaviors. SFX has been

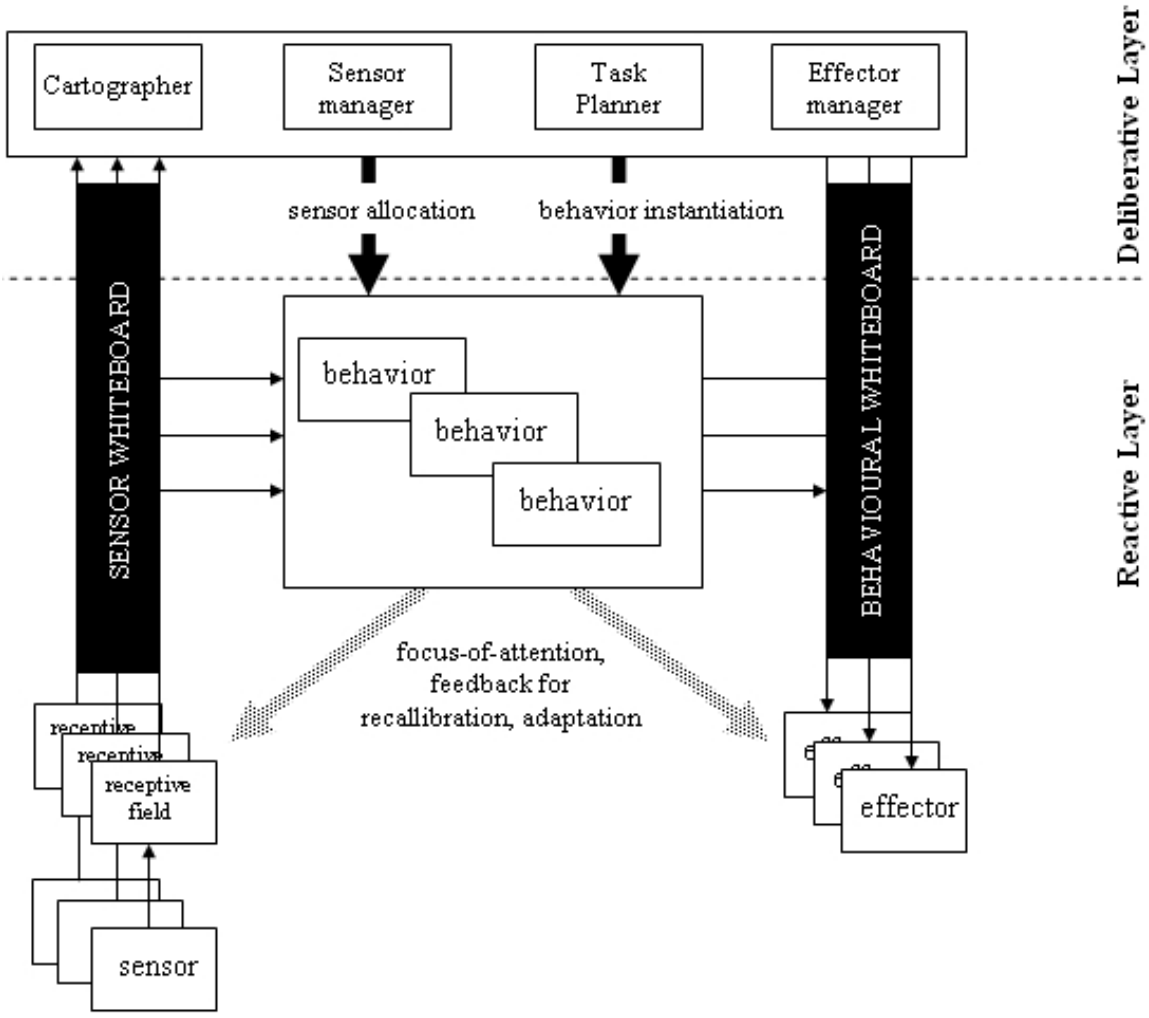

Figure 2.7: The SFX architecture.

used on diverse robotic applications, ranging from indoor office navigation to outdoor road following and rescue missions.

2.3.1.4 3T

The 3T architecture ("3 Tiers") results from the cooperation of several researchers like Peter Bonasso, James Firby, Erann Gat or David Kortenkamp. Closely related to this work are other architectures such as AAA [Firby et al., 1995] or ATLANTIS [Gat, 1992].

The main goal of 3T is reaching a robust behavior in task execution by means of combining reactivity and deliberation. The project also comprises the development of several software tools to support programmers in the design of robotic applications. 3T has been utilized in a great variety of environment and applications [Bonasso et al., 1997], including people location and recognition, trash collection, of-

fice context navigation and manipulator simulation.

The 3T architecture is organized in three levels or layers as depicted in figure 2.8: a reactive level of *skills or capacities* (the reactive layer), a *sequencer* (the task-sequencing layer), and a *planner* (the deliberative layer).

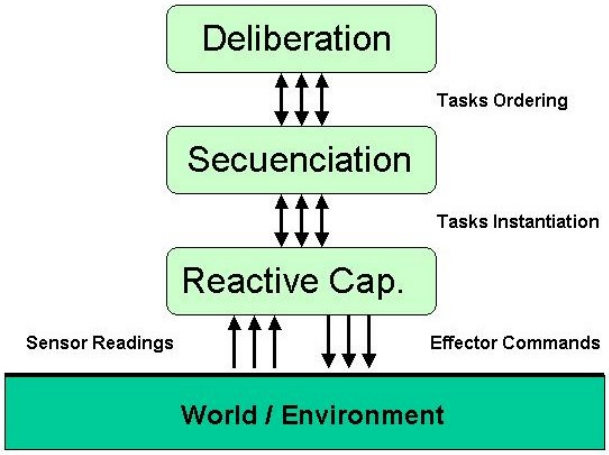

Figure 2.8: The 3T architecture.

The planner synthesizes all system goals in a list of tasks to perform. These, in turn, are decomposed in one or more sets of actions or RAPs (*Reactive Action Packages*) [Firby, 1989], that are scheduled for execution in the planner. The sequencer receives the active RAPs and controls their execution, by means of the selection of the required skills at the reactive level. Simultaneously, a set of event monitors get activated for notification to the sequencer when triggered. The sequencer modifies the task set composition in response to events, temporal violations or new high-level planning.

The capacities level integrates a set of environment dependant control actions (*situated skills*) that have been obtained systematically to avoid results conditioned by a particular robot or application context. This leads to a uniform representation for skills that eases their manipulation, including the following elements:

- Input and output specification.
- Initialization code and processing algorithm.
- Activation function and de-activation function.

Skills are controlled by means of a *skill manager* that establishes a uniform interface with the sequencer. Two types of signals are sent through this interface: commands directed to skills and events to be notified to the sequencer. The purpose is to hide low-level skill coordination details to the programmer, that can then concentrate development efforts on tasks.

The sequencing is carried out by the RAPs interpreter. This element uses a RAP library indexed by situation parameters that maps on different skills configurations.

The events constitute a special type of capacity, devoted to signalize the sequencer on the detection of circumstances relevant to the activity progress (task finalization, environment change, etc).

Above the sequencing/reaction pair, the planner adds global perspective to the system. It benefits, however, from the abstraction provided by the lower levels that contribute to reduce problem's dimensions. An important condition imposed on 3T to achieve the pursued objectives is to have the three defined levels operating concurrently in an asynchronous way.

A main issue in this architecture refers to the appropriate location of a given activity into the system. The analysis of four characteristics is proposed for this question:

- **Operation period**: Typical periods utilized in the different levels are milliseconds for the reactive level, tenth of second for the sequencer and ranging from seconds to tens of seconds for the planner.
- **Bandwidth**: Low-level skills can process high data volumes, while inter-level communications demand a very low bandwidth.
- **Functionality**: If an activity at a certain level implements mechanisms already included in the architecture it should be redesigned to allow default mechanisms to act (e. g. skills that perform action selection or RAPs that manage resources).
- **Flexibility**: Skills use to be already compiled and cannot be modified at runtime. Sequencing and planning, on the contrary, can be modified as they are based on interpreters.

The 3T architecture gives support for adaptive execution allowing several solving methods to be declared for each task. Each method has associated a set of applicability conditions that must be considered for its selection. In addition, each task includes a satisfaction test to determine when the task has completed successfully.

2.3.1.5 DAMN

DAMN (*Distributed Architecture for Mobile Navigation*) is a distributed architecture due to Julio Rosenblatt [Rosenblatt, 1995]. It is based on the execution of multiple behaviors that access to robot control through a voting mechanism. The objective is to integrate reactive mechanisms with deliberative components without having to impose a hierarchical structure.

This proposal has been utilized in different applications of operative robots that include path tracking, off-road navigation and remote-operation, combined with obstacles avoidance [Langer et al., 1994].

The DAMN architecture [Rosenblatt, 1995] comprises the following elements:

- The *DAMN Arbiter*, the control voting system.
- The *behaviors*.
- An *operation mode controller*.
- A *vehicle controller*.

Figure 2.9 shows the interconnections between the different components of the architecture.

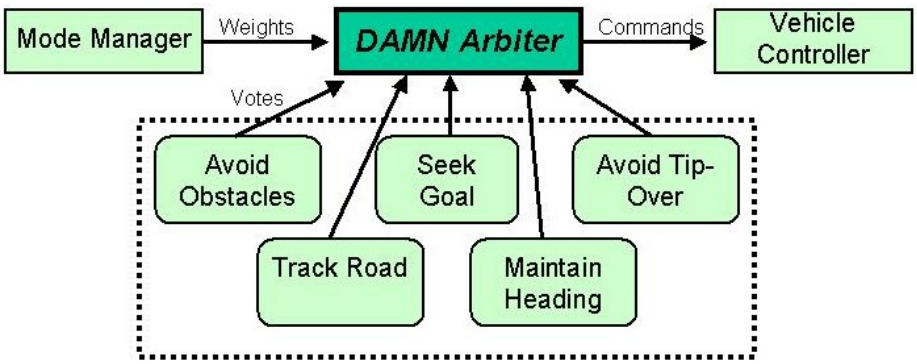

Figure 2.9: Components of the DAMN architecture.

The different behaviors, with independence of its level of competence, carry out negative and positive votes in the space of commands (turn, velocity, area of vision, etc.). The voting manager takes charge of fusing the results to elect the most voted option. This selection is the result of different processes that include the combination of the outcomes (weighted by the operation mode controller), filtering and interpolation. The operation is similar to the architecture of suppression, though here internal representations of the world can be employed.

In this architecture the behaviors are grouped in three levels of competence:

- Security.
  - Dynamics of the vehicle: Turn and velocity limits.
  - Obstacle Avoidance: Evaluation of collision in possible paths.
  - Auxiliary behaviors: Default movements and inertia.
- Action.
  - Monitoring of path.
  - Off-road.
  - Remote-operation.
- Objective.

- Sub-objectives.
- Gradient fields.
- Path planning.

The DAMN's design permits to combine behaviors using different operation frequencies, as in the case of reactive versus deliberative behaviors. The fusion of the commands generated determines the behavior of the system, so that the actions produced by modules with a greater weight are the ones that exercise more influence. Even so, higher level operation mechanisms can be employed to control the weights assignment, giving rise to a level of meta-control.

The adaptive execution in DAMN is based on the dynamic modification of the weights assigned to each behavior. It is also on this point where it is possible to introduce learning in the system, so that an adequate configuration of weights can be registered for its subsequent utilization in similar situations.

2.3.1.6 Saphira

In Saphira, three are the fundamental objectives considered for an autonomous mobile agent: robust task execution, people tracking and map construction [Konolige et al., 1997]. For accomplishing with these objectives, an architecture should incorporate three basic characteristics: coordination, coherence and communication. Saphira takes these premises to constitute an architecture utilized in the implementation of tracking applications, navigation and interaction with mobile robots [Guzzoni et al., 1997].

A layered architecture is proposed and built around an internal mechanism of representation, the local perceptual space or LPS. There is a perception part responsible for transferring sensor data to LPS and extracting information from them, and an action part on which different behaviors are executed. Several behavior types coexist: reactive at lower level, goal directed at intermediate, and task oriented at higher level. Figure 2.10 presents the different elements that integrate this proposal.

The architecture is conceived to establish a client/server relation with a robotic entity (*the robot server*). This vision improves the transportability, isolating the system from hardware particularities.

The control in Saphira is based on behaviors. The low-level reactive behaviors are defined and coordinated employing fuzzy logic [Saffiotti et al., 1997]. They are defined by means of a set of fuzzy rules and an updating function for a set of fuzzy variables.

The action is selected for each control channel (activity variable) averaging the different values of action obtained from the rules that conclude on that channel. The weighting function takes into account for each behavior a fixed priority value and a variable context of application. This context-dependant mechanism of coordination is employed both for reactive and goal oriented behaviors.

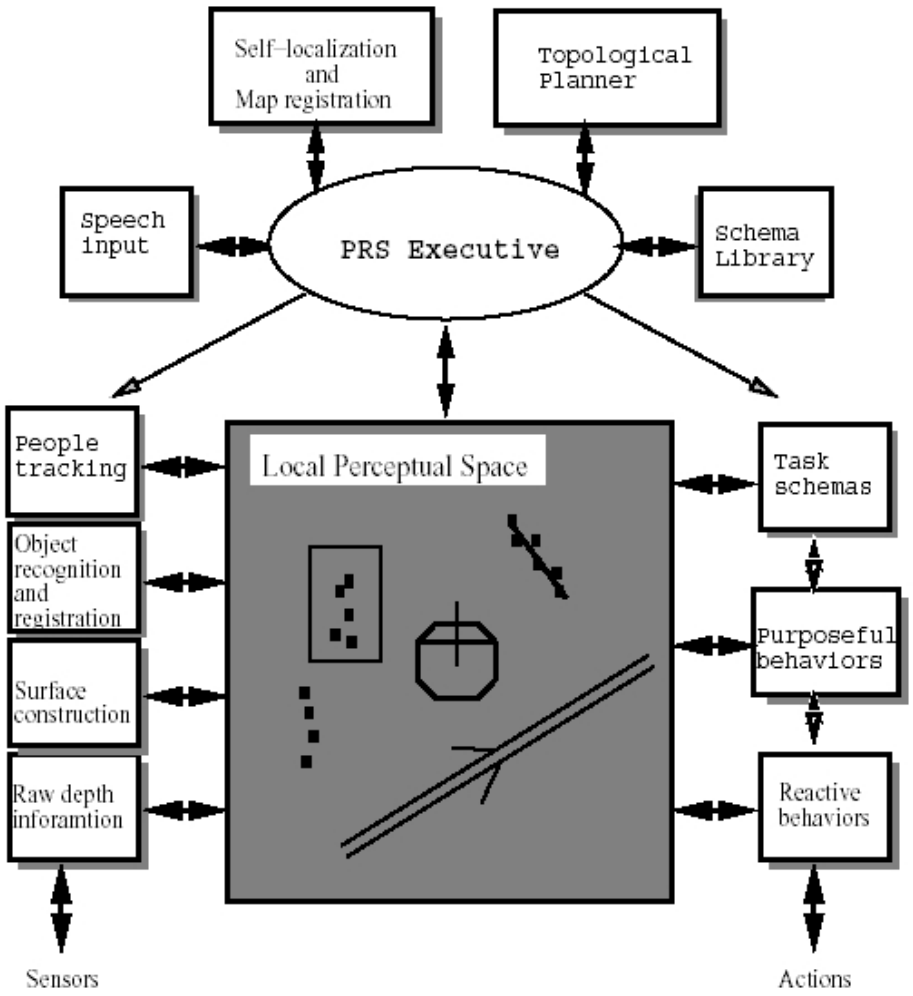

Figure 2.10: Elements of the Saphira architecture.

The coherence of the system relies on keeping updated some object descriptors of the environment, called *artifacts*, on the LPS. To accomplish this, characteristics and hypothesis of objects are generated, applying bottom up processing for symbolization of artifacts, and top down processing for verification of hypotheses. An example is the construction of a map of the environment extracting lineal characteristics from readings of ultrasonic sensors, which are combined with depth information to produce objects hypothesis like corridor, wall or door. The objects are compared with the list of artifacts for its updating, in case of coincidence, or extension, when a new object is detected. This process is called *anchoring* [Coradeschi and Saffiotti, 2003].

The selection and coordination of behaviors is performed by the PRS-Lite controller (*Procedural Reasoning System*). Some characteristics of this controller are the capacity of integrating goal and event directed tasks, reactivity, or hierarchical task decomposition. Additionally this module incorporates capacities such as the management of the continuous interaction processes, an extensive assembly of declarative control mechanisms, or the use of satisfaction levels as output after the execution of a task instead of success/failure notification.

PRS-Lite is based on the utilization of *activity schema*, that are defined as ordered sets of goals, integrated in turn by simple goals. A goal can belong to two basic categories: action or sequencing. The typical actions include test, assignment, execution, wait for condition or expansion/contraction. This last type permits the hierarchical expansion, giving rise to tree structures. The sequencing goals include jumps, branching and parallelization.

The operation of the system relies on the *intentions* objects, corresponding to the hierarchy of activity plans launched. Inside them, leaf nodes constitute the present assemblies of goals, and are selected for execution on each system cycle.

2.3.1.7 BERRA

BERRA (BEhavior based Robot Research Architecture) [Orebäck et al., 2000], is an architecture aimed at achieving scalability and flexibility as primary objectives. It is based on software components implemented as processes that can be transparently placed on a processing network. The implemented system makes use of the ACE (Adaptive Communication Environment) [Schmidt, 1994] library as supporting communication layer. ACE includes powerful patterns for client/server communication and service functions. OS dependent system calls are wrapped, allowing for portability across a wide range of operating systems.

The proposed architecture is structured in three layers: the deliberation layer, the task execution layer, and the reactive layer. The deliberate layer makes high level decision, derived from robot objectives or user orders, according to system state. The reactive layer integrates low level modules interconnected in a flexible network. The intermediate mission or task execution layer carries out the plans generated by the deliberative layer through the configuration of the reactive network to solve the task at hand.

The deliberative layer internal structure consist of a planner and a human robot interface (HRI). The HRI applies gesture and speech recognition to capture user commands that are then sent to the planner. The task execution layer contains two modules: a state manager and a localizer. The state manager is a finite state automaton in charge of module configuration at the reactive level, while the localizer retrieves and distributes position data. The reactive layer consists of a large set of modules that can be interconnected to configure a network of tight sensorimotor loops. The modules at this level are of three types: resources, behaviors and controllers. The resources capture data from sensors and serve them to behaviors (e.g. GoPoint, Avoid, Explore, MailDocking, DoorTraverse) for analysis. The behaviors's outputs are control propositions that controllers fuse to produce the final command to be executed by physical effectors. On figure 2.11, a schematic view of this architecture is depicted. The implemented system has been tested in a significant number of laboratory missions [Andersson et al., 1999].

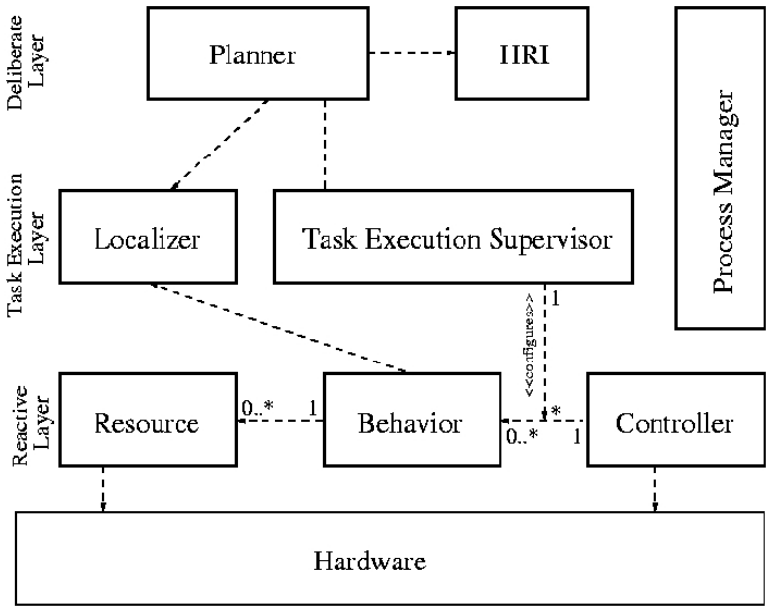

Figure 2.11: The BERRA architecture.

2.3.2 Frameworks and Programming Languages

In addition to architectures some software frameworks have been also created by different research groups following different motivations. In some cases, the frameworks were created to support a specific level of abstraction for implementing a more complex software architecture. This is the case of G^{en}oM developed at LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes), one of the frameworks we will comment next. G^{en}oM was developed to support the creation of the software architectures that control the set of robots used at that laboratory.

Other research groups have taken the approach of developing programming languages. In general, these languages are supersets of existent general-purpose programming languages to which operations and constructs to model concurrency, parallelism and task control have been added. Some of them also include abstractions to model inter process communications.

2.3.2.1 G^{en}oM

This proposal was originated at the CNRS-LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes) in Toulouse, carried out by Sara Fleury, Rachid Alami, Raja Chatila and Felix Ingrand, among others [Alami et al., 1998]. The basic objective is the integration of real-time modules in a robotic system, using a hybrid architecture. This imposes a series of requirements on robot functionality in order to attain a distributed real-time system that exhibits high predictability and scalability.

This framework, also known as G^{en}oM (Generation of Modules), has been essayed both at simulation level and on physical robots, mainly in coordination missions

[Alami et al., 1995]. The architecture defines three different levels: a *functional level*, an *execution level*, and a *decision level*. Figure 2.12 shows the structure of a low-level application example combining tracking with obstacle avoidance.

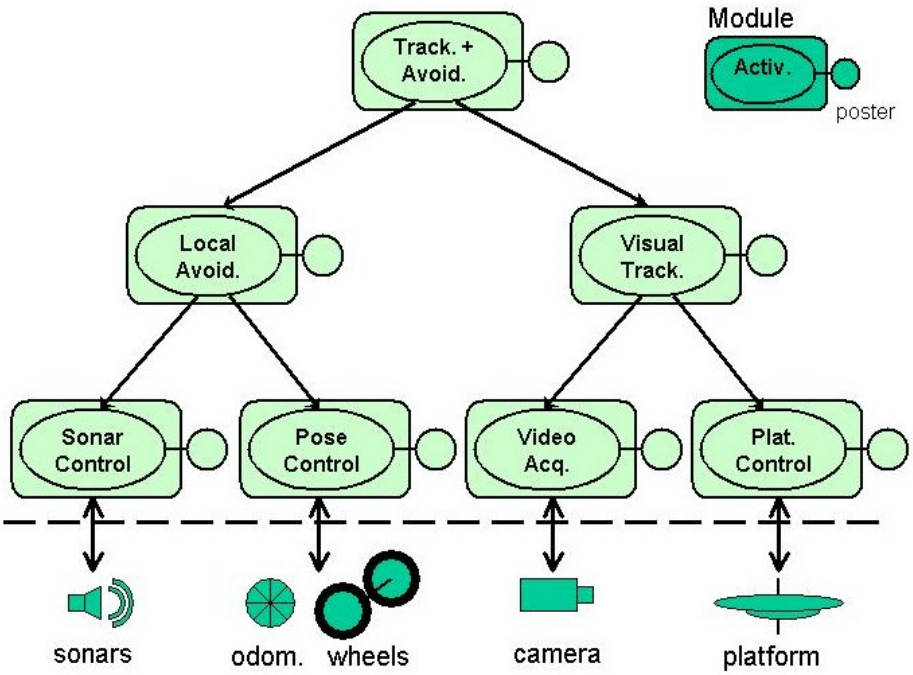

Figure 2.12: Application example based on LAAS architecture.

The functional level contains all the basic capacities for perception and action. These functions are encapsulated in inter-communicated controllable modules [Fleury et al., 1997] [Fleury and Herrb, 1998] that are linked to the physical level through an abstraction layer, the logical robot level. The purpose of this intermediate layer is to make the development independent from the physical robot available.

The executive level acts as interface between the functional and the decision level. It is a level with no planning capacity, that receives the action sequences from the decision level and translates them into dynamic control orders directed to the functional level. The executive level is structured internally in three main components:

- **Request monitor**: Receives requests from the decision level and uses the request database to map them into a set of modules to be activated.
- **Answer monitor**: Supervises the evolution of the execution in the functional level to notify detected events to the decision level.
- **Execution state database**: Contains rules to solve conflicts as a function of the information provided by the monitors.

The code used at the execution level is critical in the operation of the whole system and must be formally verified to guarantee its logical and temporal correctness.

Above the functional level, the system has a decision level in charge of interpreting the robot mission decomposing it into requests to the lower level. This level must be reactive and manages the sharing of resources among the applications. To ensure reactivity, the decision level is constituted by two entities: a planner and a supervisor. The planner generates the sequence of actions needed to fulfill a given objective, and it is used as a resource by the supervisor, who is responsible for the interaction with the lower level, the control of the plan execution and the reaction to events. Besides, the supervisor acts as user interface.

The supervisor makes use of two additional elements during plan execution: the execution modalities, to limit the searching space for planning, and a database of situation-driven procedures. The deliberation algorithms executed by the supervisor must have reduced execution times, in order to avoid an excessive control loop delay.

The decision level can be organized internally in multiple layers, depending on the application. All layers, however, adopt the same internal structure of planner/supervisor.

The typical operation cycles for the three levels are hundredths of a second for the functional level, tenths for the executive one and seconds for the decision level. Several tools are offered to verify the system performance, including an interactive test program, a computing time estimator and a chronogram generator.

LAAS architecture is conceived around the generic module concept depicted in figure 2.13. A module is defined as a software entity that offers services relative to a given resource. The services are attended on a client/server protocol basis, returning the result of the execution and a measure of its quality. The modules receive execution requests to begin a certain processing activity and control requests to modify its behavior. Relations between modules are established dynamically.

Data are interchanged among modules by means of posters, which are shared memory areas writable only from the owner and accessible from the rest of modules. Besides this functional posters, there are control posters that reflect the status of each module.

The execution state of the system is represented by means of an activity tree, that reflects a hierarchical decomposition formed by the launching of child activities from a parent activity. The activities to be integrated must be interruptible, and support controlled finalization and error detection. To facilitate this, the generic module structure contains the following elements:

- **Control level**: In charge of module management, this level receives the clients's requests, verifies them, solves potential conflicts, initialize and finalize activities, and returns answers back to the clients. It is also responsible for keeping the internal state representation of the module which is exported through the control posters.

- **Executive level**: Executes the activities demanded from the control level. This level contains one or more execution tasks, either periodic or not, that constitute

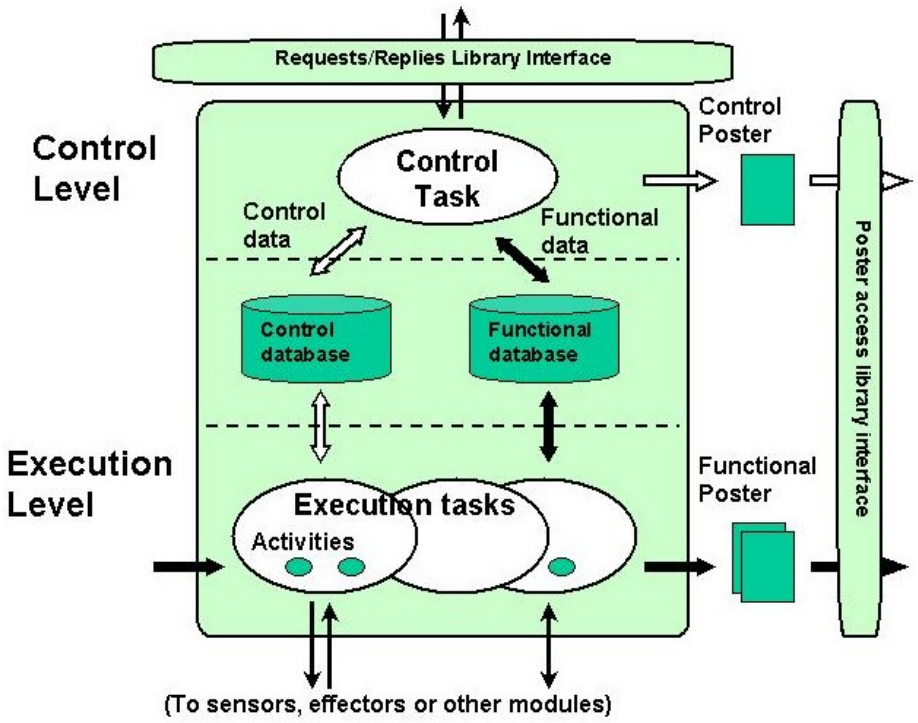

Figure 2.13: Internal structure of a generic module in LAAS.

the execution context for one or more activities.

- **Inter-level communication**: The exchange of information between levels takes place by means of two databases: a functional database and a control database.

An execution request maps to an activity that evolves following a state control graph, transiting on signals either from the control or the execution levels. Five are the possible states for an activity: "ETHER", "INIT", "EXEC", "ZOMBIE" y "INTER". The valid transitions for those states are "init", "exec", "failed", "abort" y "ended". The "EXEC" state decomposes into execution steps, named *codels*, that can be considered atomic from the module point of view. Usually, there are an initialization step ("start"), an iterative step for execution progress ("exec"), another for finalization ("end"), and finally a step for error conditions ("fail").

The modules are generated automatically from a formal description. This description includes items such as module identification, used data types, valid requests and associated elements (request type, input and output parameters, execution report code, detail of execution steps), the posters, the execution parameters (period, priority), etc.

The mechanisms of adaptation are based on the internal structure of the modules. These can incorporate error processing procedures in execution time that include strategies for recovery or notification of failures to clients.

2.3.2.2 ESL

ESL (*Execution Support Language*) is a language proposed by E. Gat [Gat, 1997] to codify execution-related knowledge in autonomous agents. It derives to a great extent from the RAP system [Firby, 1989], although with a more practical focus, oriented to flexibility and comfort of use. ESL is implemented as an extension of Common Lisp.

ESL incorporates constructions, among others, to implement the following characteristics:

- **Exception management**: Making use of the concept of conscious failure, that express the convenience that, since is not possible to build a system that do not fail under any circumstance, at least the failure situation should always be detectable. Some available constructions are FAIL, WITH-RECOVERY-PROCEDURES and WITH-CLEANUP-PROCEDURES.
- **Goal specification**: Permits to register methods to achieve different goals by means of the constructions ACHIEVE and TO ACHIEVE.
- **Task management**: ESL supports the concurrent execution of multiple tasks. It is possible, for example, to specify groups of tasks (TASK NET), synchronize the execution through events (WAIT-FOR-EVENTS, SIGNAL, CHECKPOINT-WAIT, CHECKPOINT), and indicate conditions of failure (WITH GUARDIAN).
- **Data base**: Instructions for management of a logical base, as are ASSERT, RETRACT or WITH-QUERY-BINDINGS.
- **Blocking mechanisms**: To avoid inconsistences in the management of shared variables.

2.3.2.3 TDL

TDL (*Task Description Language*) [Simmons and Apfelbaum, 1998] is a language designed to simplify the development of control programs for robots. It includes specific backup for the implementation of control at task level. TDL has been developed as an extension of C++, implemented on top of a communication library called TCM (*Task Control Management*).

This language incorporates the following characteristics: *tasks decomposition*, *monitoring of execution*, *synchronization.*, and *exception management.*

TDL derives from TCA (*Task Control Architecture*) [Simmons, 1992], a proposal that combines task level control with inter-process communication using message passing.

The basic representation utilized in TDL are task trees, where nodes are associated to specific actions whose execution result can either be success or failure. Task tree nodes may belong to the following types:

- **Command**: Constitute executable behaviors, and are located normally in the leaves of the task tree.
- **Goal**: These nodes represent high-level tasks, and their activation expands the tree in new child nodes that, in turn, can be goals or commands.
- **Monitor**: It is a node whose action can be invoked repeatedly on the detection of a specific event.
- **Exception**: Node associated to a specific type of failure, provoking either the activation of recovery procedures or a simple notification.

From the point of view of its manipulation, a node of the task tree can be in one of the following states: "disabled", "enabled", "active" or "completed". A node is disabled when the corresponding synchronization restrictions are not verified. When these conditions are met, the node is enabled and waiting for the capture of its execution resources. If this it is the case, the node becomes active until the execution finalizes, transiting to the "completed" state.

There are two types of restrictions: authorization and finalization. These restrictions indicate that a specific node, on the presence of a certain event, can initiate its execution (authorization) or finalize (finalization).

Here are included some examples of code:

1. Expansion and synchronization:

```
t1: spawn a(1); t2: spawn a(2); spawn b(3) with
    disable until t1 execution completed,
    disable expansion until t2 handling active;
```

2. Exceptions:

```
Goal navigateToLocn (double x, double y)
{
    with exception
        ("Overheating": handleOverheating(x, y),
         "no path": handlePlannerFailure()) do
         {
             ...
         }
}
```

3. Monitors:

```
Monitor monitorPickup ()
    max triggers = 1, max activations = 15,
    period = 0:0:1.5
{
    if (mailIsGone())
    {
        trigger;
        with (parallel) do
        {
            spawn speak("Thank you");
            spawn notifySender();
        }
    }
}
```

As part of this project, several programming support tools have been developed. Some examples include a compiler for the definitions of tasks and diverse visualization and debugging utilities.

2.3.2.4 SmartSoft

SmartSoft [Schlegel and Wörz, 1999b] is an object oriented framework devised to assist developers in the implementation of sensorimotor systems. It has been developed by Schlegel and Wörst at FAW (Research Institute for Applied Knowledge) at the University of Ulm in Germany. Like G^{en}oM, SmartSoft is a software framework developed to support a software architecture for a mobile robot [Schlegel and Wörz, 1999b].

Using Smarsoft it is possible to define a system as a set of *modules* which interact and inter-communicate between them. Modules have a uniform structure organized in a user area and a framework area. A module is a process, and it can be multithreaded. There is a thread provided and controlled by the framework which is transparent to module designers, that is responsible for all inter-communications between the inner threads of each module, and also for communications between modules. The user can organize the functionality of a module using multiple threads, if he/she considers that necessary.

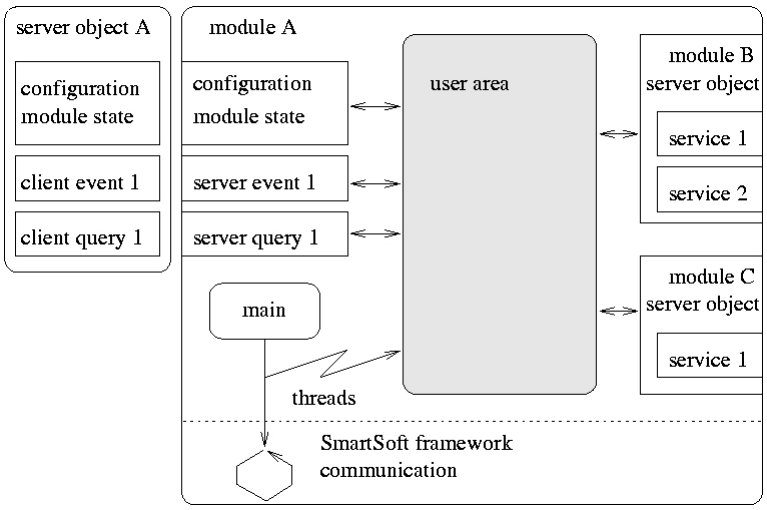

Figure 2.14: A SmartSoft module.

Each module provides different services, and those services are provided by means of proxy objects which the framework calls "server objects". A typical SmartSoft module appears in figure 2.14 with a server object. The model used for inter-communication between modules is a client-server paradigm. If a module wants to access to the services of another module, it needs to be provided with a server object from the second module. Equally, it acts in the same manner to offer its own services to other modules.

In SmartSoft, inter-communication between modules through server objects is

carried out using an established set of mechanisms or communication primitives that constitute several communications patterns, namely:

- **Command**: This is a primitive that implements an unidirectional communication pattern from the client module to the server module. This is a push communication primitive initiated by the client.
- **Command with Status**: This is a primitive which is similar to the previous one, the difference is that a status is returned by the server as a result of the command. This is a pull communication primitive initiated by the client.
- **Query**: This is a pull communication primitive initiated at the client side. The client makes a request and the server returns an answer. The client may get the answer for its request synchronously or asynchronously. It is possible to manage several requests before their answers being received.
- **Autoupdate Newest**: This is a push primitive initiated by the client. It implements a subscription mechanism of communication. While subscribed, a client receives data from a server. The client may either access to the most recent information received from the server, or block waiting for the next incoming information.
- **Autoupdate Timed**: This primitive is similar to the previous one, except that using this primitive a client can establish specific intervals of time at which it wants to receive data from the server.
- **Event**: This is a push primitive initiated by the client. With this primitive a client can subscribe itself in a server to receive signalizations of the occurrence of a specific event controlled by the server. At subscription time the client provides a condition under which the event should be signaled. The server is in charge of signaling clients when the event gets activated according to the conditions established by each client.
- **Configuration**: This is the most complex communication primitive. It has been devised to have a module (the master) controlling externally another module (the slave). Using this primitive a *master* module can interrupt another module in order to either reconfigure or abort it.

The main parts of the implementation of SmartSoft are based on the ACE software package [Schmidt, 1994] that ensures portability among different platforms. Low level communication routines are built on top of TCX [Fedor, 1993]. SmartSoft is an object-oriented framework for programming robotic systems that fosters asynchronous communications between modules and asynchronous execution inside them. The framework models inter-communications between modules using a client-server paradigm. Contrarily to G^{en}oM, the framework does not impose any structure on the user area inside each module, thus, it is up to the user the correct use of the

primitives provided by the framework. On the other side it provides a rich set of standard communication patterns which prevent the user to be worry about inter module communications. All in all, modules in SmartSoft constitute the main units for constructing robotic systems as they fit correctly in the definition we have adopted for software components in section 2.2.

2.3.3 Others

From a software engineering point of view the majority of the solutions devised to design and implement the software that controls robotic systems fall into the categories previously commented of architectures, frameworks and programming languages, but there are also other approaches. In this document we would like to make emphasis on one of them we consider very interesting, because it shares some of the ideas which are in the base of the work we present in this thesis. This approach is the Chimera operating system that is briefly outlined in the following paragraphs.

2.3.3.1 Chimera

Chimera [Stewart and Khosla, 1993] [Stewart, 1994] [Stewart and Khosla, 1996] [Stewart et al., 1997] is a real-time operating system developed at CMU (Carnegie Mellon University) aimed to control robotic systems. Chimera is a multiprocessor real-time operating system designed specifically to support the development of dynamically reconfigurable software for robotic systems. The typical hardware architecture where Chimera is applicable is an architecture containing one or more open-architecture buses housing multiple single board computers called *real-time processing units* (RTPUs).

The units of execution handled by the operating system are defined as software modules called port-based objects. A port-based object is an automaton which performs all its external communications by means of output and input ports. This conceptual object was inspired by the concept of port automata given by Steenstrup in [Steenstrup et al., 1983].

Port-based objects are also called *tasks*, because each one has its own flow of execution. At a given moment in a specific processor there may be several port-based objects running. Port-based objects communicate between them by means of its input and output ports. Every output and input port is mapped to what is called a *state variable*. Output ports are output state variables, and input ports are input state variables. Port-based objects communicate using this mechanism of output and input state variables.

The inter communication mechanism between port-based objects is based on the combined use of global shared memory and local memory for the interchange of data between software modules. Each port-based object in a given processor has its own local table of state variables (*local state variable table*). Moreover, there is a global state variable table shared by multiple processors that contains copies of all

local state variables corresponding to the port-based objects running in the system in a given moment. Tasks are executed cyclically iterating continuously through a task body where is coded the functionality of the port-based object. For each task, at the beginning of each iteration, the operating system transfers atomically the local output state variables to the shared global state variable table, and also transfers the input state variables from the global table to the local ones. This transferring is carried out automatically by the operating system implementing a two-level memory scheme of inter-communication. Conceptually, this mechanism of inter-communication between port-based objects is similar to the poster-based mechanism of communication utilized in G^{en}oM, but in this case, it is supported directly by the operating system.

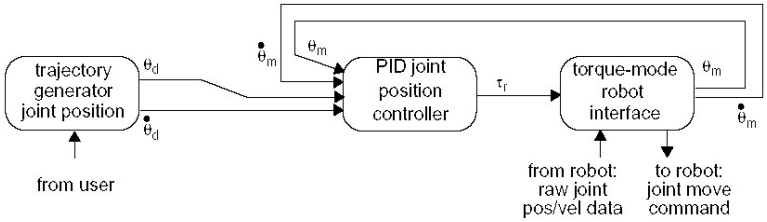

Figure 2.15: A typical control loop in Chimera.

Using interconnected port-based objects is possible to form multiple open and closed loops in order to define multiple control loops in a system. Figure 2.15 shows a typical control loop in Chimera. *Task sets* can be defined by forming different topologies of interconnected port-based objects. Task sets may be reconfigured statically and dynamically. A *job* is a high level description for a task set. Jobs can also be a collection of other jobs, so hierarchical composition is possible. A *control subsystem* is a collection of jobs running concurrently or in parallel. An *application* is one or more subsystems executing in parallel.

In Chimera, the port-objects are the basic units of composition to form topologies of components in order to build applications by integrating them. The operating system guarantees that port-based objects are deployable, because it imposes a clear structure for all of them: a clear interface of output and input ports and a cyclical internal structure. It is obvious that the concept of port-based object corresponds to what we have called a software component.

2.4 Proposed Approach: CoolBOT

The work presented in this thesis has been mainly originated by very practical reasons. While developing several robotic systems [Hernández-Tejera et al., 1999] [Cabrera et al., 2000] we got to a point where the necessity of some common software infrastructure was a clear demand. This infrastructure had to be generic enough to design any imaginable architecture for the projects we were involved at that moment. At the same time, it had to allow us to integrate software more easily. As one of the results of these projects we developed an agent-based software framework called **CAV** (**C**ontrol

Architecture for **A**ctive **V**ision Systems) [Domínguez-Brito et al., 2000b]. An active vision system termed **DESEO** (**DE**tención, **SE**guimiento y Reconocimiento de **O**bjetos) [Hernández-Tejera et al., 1999] aimed to detect, track and recognize faces, and an entertainment museum robot called **Eldi** [Cabrera et al., 2000] were developed using CAV as software infrastructure. CAV was a tool that allowed modelling software as networks of interconnected software agents. It provided mechanisms for intercommunication between agents, whether residing in a remote computer or in the local machine. It lacked many primitives and resources we consider were also necessary, like a more rich set of intercommunication mechanisms, and a support to make multithreading less error-prone and more systematic. Specially, it lacked mechanisms to facilitate software integration that still remained being an important problem for us. Further work and experiences using CAV has driven us to the present work passing by different phases which can be tracked in documents [Domínguez-Brito et al., 2000a], [Cabrera-Gámez et al., 2000] and [Domínguez-Brito et al., 2002] until the present work.

There are some questions and problems that appear repeatedly in every robotic system: Multithreading and multiprocessing, distributed computing, hardware abstraction, software integration of legacy code and third party code, different levels of abstraction defining different levels of achievement, the development of a programming tool for a group of users which may become wide and diverse, etc. For us the creation of a framework modelling and implementing mechanisms and techniques to support the resolution of these common problems was felt as a needed step forward.

2.4.1 Design Principles

In the next paragraphs we will introduce the design principles that have driven the design and development of a component-oriented programming framework for robotics called **CoolBOT** which is the subject of the present document.

2.4.1.1 Component-Oriented

The concept of software component introduced previously in section 2.2 defines software units which have a context-free design, a clear uncoupling between external interface and internal functionality, able to be subject to integration and composition with other components, and easily deployable. A software framework being able to program systems in terms of composing and integrating of components would be really an interesting programming tool where software integration would be fostered. This is an important feature, not only because it would avoid to build systems from scratch, which is a very common situation, but also to promote the integration of software from third parties.

As we have seen along the approaches commented in section 2.3 the use of "standard" building blocks for constructing systems is not new. Not only frameworks like G^{en}oM and SmartSoft define a building software block to build and integrate system software. Even architectures define blocks of integrable software fitting usually at each

level of abstraction. Think, for instance, of behaviors in the subsumption approach, or in the AuRA, SFX and DAMN architectures, and also of skills for the 3T architecture. The same idea is behind port-based objects in Chimera. Having architectures and frameworks, and evidently operating systems (think of normal processes and threads in any operating system) defining "standard" software building blocks in order to integrate bigger and more complex software allows to design and develop each block separately and independently.

2.4.1.2 Component Uniformity

Certain level of uniformity in structure in software components is critical to allow a basic uniform treatment of components in spite of their individual functionality. This is not a new idea, operating systems have applied it for years. Binary programs in any operating system have a given format imposed by the operating system. Only in this way it is possible that programs can run in any computer running under a given operating system. Based precisely on this idea, the Chimera operating system has been devised to build robotic systems from the integration of process-like entities called port-based objects as we have seen already in section 2.3.3.

Additionally, a uniform internal structure for components facilitates its observability and controllability, i.e. the possibility of monitoring and controlling the inner state of a component. We consider that these properties are key elements when defining robust systems, making its design and implementation less error-prone. At the same time, component's internal uniformity sets a real basis for the development of debugging and profiling tools.

In G^{en}oM we can find that this concept of uniform external interface and internal structure allows for the definition of different very useful tools for the developer like compilers and component testers. Obviously this is not a consequence that comes up spontaneously, but it is the result of a well-designed level of uniformity, shared by all the entities that conform a system, which is imposed by the framework to all modules.

2.4.1.3 Robustness

Robustness is a design principle in nearly every robotic system. We deem that a framework aimed to program them should provide mechanisms to promote robustness. In CoolBOT we have followed a principle of robustness based on the following motto (the *robustness motto*): *"A component-oriented robot system will not be robust and controllable if its components are not robust and controllable"*. Complementarily to the previous principle, we add the following considerations about robustness in individual components:

1. **Local Robustness**: A component must be able to monitor its own performance as a basic means of adapting itself to changing operating conditions. It must also implement its own adaptation and recovery mechanisms to deal with all

errors and abnormal situations which can be detected internally (*cognizant failure* [Noreils, 1990][Gat, 1992]).

2. **External Robustness**: Any error detected by a component that cannot be recovered locally by its own means, should be notified using standard means through its external interface, bringing the component to an idle state waiting for external intervention, that will order the component to continue, restart or abort operation.

Furthermore, we will consider a component controllable when it can be brought with external supervision - by means of a controller or a supervisor - through its external interface, along an established control path (*Principle of Controllability*). This is a feature we can also find in G^{en}oM where it is possible to drive modules along a small set of controlled states. Equally, in SmartSoft, there is the possibility of having a module (the master) controlling externally another module (the slave) using a specific communication pattern for configuring modules.

2.4.1.4 Modularity & Hierarchy

In a programming methodology where systems are constructed by integrating and composing components, it would be of great interest to dispone of constructs to organize components in a modular and hierarchical way. As systems grow in size and complexity it is necessary to provide constructs to give more structure and reduce complexity. Typical constructs are those that endow programs with modularity and hierarchy as we can found in most of the modern programming languages. It is not strange to find languages with constructs such as classes, name spaces, functions, modules in programming languages like C++, Java, Ada, etc.

In the scope of the framework we want to build, modularity and hierarchy would allow defining component composition as single components. Thus, in our component-oriented framework there will be *atomic* and *compound* components. Atomic components will be indivisible, i.e. they are not made up of other components. Compound components will be components which include in their definitions other components, whether atomic or not. Moreover, compound components must also integrate a supervisor for their monitoring and control. With this vision, a whole system is nothing else but a large compound component including several components, which in turn include another components, and so on, until this chain of decompositions finishes when atomic components are reached. Thence, a complete system might be envisioned as a hierarchy of components from a coordination and control point of view.

From the previous paragraph, it is clear that this concept of *compound* components would be what will endow systems with modularity and hierarchy. Similarly to the hierarchical constructs called **jobs** we can find in Chimera where it is possible to gather together port-based objects to form task sets. Additionally, a hierarchical structure promotes also a hierarchical treatment of errors which is also a common goal of many architectures like Aura.

2.4.1.5 Integrability & Incremental Design

We deem the framework should promote software integrability and incremental design. Evidently, the principle of building systems by integrating components promotes integrability. As collateral result, incremental design is not difficult to foresee. In this way, systems could be built bit a bit, integrating component by component, adding more functionality and capabilities as the development progresses.

We consider integrability and incremental design as key features for promoting integration of code with a variability of origins, and that would allow reuse of software originated from across different research groups. As an example, imagine a data base of software components implementing ready-to-use state-of-art algorithms in multiple fields of robotics. Systems could be built just integrating components, and components in this data base would be there just for everybody to test and use.

2.4.1.6 Distributed

Components should be integrable and reachable in a system with independence of in which machine they are running. That components were in same computer, or in a different one residing in the same computer network, should be indifferent in terms of component integration and interconnection. Therefore, integrating two components residing in different machines should be as easy as if they were in the same one, in the same manner that it is possible in SmartSoft to inter-communicate modules thanks to the ACE software layer. In this way, SmartSoft provides a rich set of standard communication patterns which prevent the user to be worried about inter module communications even if the different modules reside in different computers.

2.4.1.7 Reuse

Components are units that keep their internals hidden behind a uniform interface. Once they have been defined, implemented and tested they might be used as components inside any other larger component or system. Modern robot systems are becoming really complex systems and very few research groups have the human resources needed to build systems from scratch. Component-oriented designs represent a suitable way to alleviate this situation. We believe that research in robotics might enormously benefit from the possibility of exchanging components between labs as a mean for cross-validation of research results.

Code reuse can be promoted in multiple ways and at different levels. Examples are behavior reuse in subsumption architectures, and also in Aura and SFX; skill reuse in 3T; and module reuse in G^{en}oM and SmartSoft. In the case of the tool we want to construct we consider that the minimal units that conform a system should be deployable and integrable in order to make this minimal units able to be reused wherever needed. Finally, we have considered the concept of software component introduced in section 2.2.5 as the most convenient concept to embody these minimal units of integration.

2.4.1.8 Completeness & Expressiveness

The computing model underlying the framework should prove valid enough to build very different architectures for robotic systems and expressive enough to deal with concurrence, parallelism, distributed and shared resources, real time responsiveness, multiple simultaneous control loops and multiple goals in a principled and stable manner. Evidently, the more complete and general it is a framework, the less constrained will be the systems we can develop using it.

For us, it looks like making a framework providing enough functionality and means to solve some common problems inherent to robotic systems, which is neutral in terms of architectural design and structure, could be a good starting point to find out which features "a complete framework" should provide.

2.4.1.9 Operating System Support

Finally, another very practical principle of design we considered for the framework was to support CoolBOT in the two operating systems which are mainstream at the present moment: the Windows family of operating systems (Windows NT, 98, 2000 and XP), and GNU/Linux. Although these operating systems are not real-time, they can provide support and keep contraints for soft real-time systems [Ramamritham and Shen, 1998] [Gopalan, 2001], such as multimedia and active vision systems. We do not discard in the future the necessity of supporting the framework in some real-time operating systems such as RTLinux [RTLinux, 2003] or RTAI [RTAI, 2003].

Chapter 3

CoolBOT Fundamentals

In this chapter we will explain which ideas and concepts there are behind CoolBOT, and how these ideas and concepts have been put into practice to implement a software framework to program robotic systems.

3.1 Introduction

Ideally, software components should be something like electronic components or chips in electronic industry. It is many years that off-the-shelf chips can be bought and deployed anywhere. Each component has a clear functionality and also, a well established external interface. Furthermore, numerous standard tools exist to design electronic devices based on the composition, assembly and combination of these electronic components. A similar panorama would be desirable in terms of software in robotics.

The last paragraph expresses the key idea that has driven the design and development of CoolBOT from the beginning. Latest trends in Software Engineering are exploiting the idea of Components as the basic units of development and deployment when building complex software systems, specially if software reuse, modular composition and third-party software integration are important issues. CoolBOT should be understood as a component-oriented framework, in the sense defined in section 2.2.5.

The concept of software component is fundamental in CoolBOT. The main objective is to build robotic systems from the integration of software components. Which properties, features and/or attributes should software components have, to claim that systems can be build out of them by integration?. What is exactly a software component in CoolBOT?. How are components integrated?. And above all, how has all of this been implemented?. This chapter has been devised to answer such questions.

Next section, section 3.2, explains what a component is in CoolBOT. Section 3.3 describes the default features and traits that the framework imposes on all components. In section 3.4 the model of execution that CoolBOT components use is detailed. Section 3.5 is addressed to explain how components inter communicate. Section 3.6 introduces the concepts of *atomic* and *compound* components, and what they are for. Section 3.7

explains the network level in CoolBOT, how components can be accessed and used via a computer network, making them distributed. Finally, the last section, section 3.8, enumerates which objects and methods the framework makes available to developers and users in order to design and develop CoolBOT components, and built systems using CoolBOT abstractions and philosophy.

3.2 CoolBOT Components

As it has been mentioned previously in chapter 1, software typically involved in the control of robotic systems may be very heterogeneous, involving numerous hardware devices and software. Such a heterogeneity could be abstracted through a model of interaction among the different elements composing a system. From this point of view, a robotic system might be considered as a network of weakly coupled parallel and/or concurrent active entities interacting asynchronously in some way. It is the interaction among the entities involved and their local behavior what defines the task the system carries out. This concept of active entity is what in CoolBOT has been identified as a software component according to the definition of software component we enunciated in 2.2.5. Therefore, they are characterized by a set of important features, namely: uncoupling of external interface and internal implementation details, context-free design, and ability to be subject to composition and integration with other components. All them together make software components become deployable pieces of software that can be reused wherever needed.

3.2.1 Port Automata

In CoolBOT, components are modelled as **Port Automata** [Steenstrup et al., 1983][Stewart et al., 1997][Domínguez-Brito et al., 2000a]. This concept establishes a clear distinction between the internal functionality of an active entity, the automaton, and its external interface, its sets of input and output ports. Components define active entities which carry out specific tasks, and perform all external communication by means of their input and output ports. Components act on their own initiative, running in parallel or concurrently, and are normally weakly coupled, i.e. no acknowledgements are necessary when they communicate through their ports.

Components can be *atomic*, i.e. indivisible, or *compound* when they are made up of a composition or assemblage of other atomic and/or compound components. With independence of their type, atomic and compound components are externally equivalent, offering the same uniform external interface and internal control structure. These properties are extremely important in order to attain standard mechanisms that guarantee that any component can be externally monitored and controlled. Once a component, atomic or not, has been designed, implemented and tested, it can be used wherever it might be necessary. Therefore components constitute in CoolBOT the functional building blocks to program robotic systems.

Formally, CoolBOT defines software components as Port Automata. From [Steenstrup et al., 1983] and [Košecká et al., 1997], a port automaton P is formally defined as a generator $G = (L, Q, \tau, \delta, \beta, X, Y, F)$, where:

- L is the set of ports.
- Q is the set of states.
- $\tau \subseteq Q$ is the set of initial states.
- $X = \{X_i : i \in L\}$, where X_i is the input set for port i.
- $Y = \{Y_i : i \in L\}$, where Y_i is the output set for port i.
- $\delta : Q \times \sqcup_{i \in L} X_i \rightarrow Q$ is the transition map, where $\sqcup_{i \in L} X_i = \{(x, i) : x \in X_i\}$ is the disjoint union of the X_i's.
- $\beta = \{\beta_i : i \in L\}$, where $\beta_i : Q \rightarrow Y_i$ is the output map for port i.
- $F \subseteq Q$ is the set of final states.

All subject to the axiom that for each $q \in Q : \{x \in X_i : \delta(q, (x, i)) \neq \varnothing\} = \varnothing$ or X_i assuring that, in any state $q \in Q$, for any port i, either all elements of the input set X_i will be capable of being accepted or none of them will.

The concept of port automaton establishes a clear distinction between the internal functionality of an active entity, the automaton, and its external interface, the input and output ports. Figure 3.1 displays the external view of a component where the component itself is represented by a circle, input ports, i_i, by the arrows oriented towards the circle, and output ports, o_i, by arrows oriented outwards. As shown by the figure, the external interface keeps the component's internals hidden. Figure 3.2 depicts an example of the internal view of a component, concretely the automaton that models it, where circles are states of the automaton, and arrows, transitions between states. Transitions are triggered by events, e_i, caused either by incoming data through a port, or by an internal condition, or by a combination of port incoming data and internal conditions. Double circles indicate automaton final states.

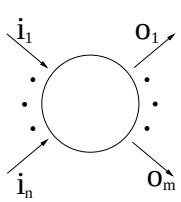

Figure 3.1: Component external view.

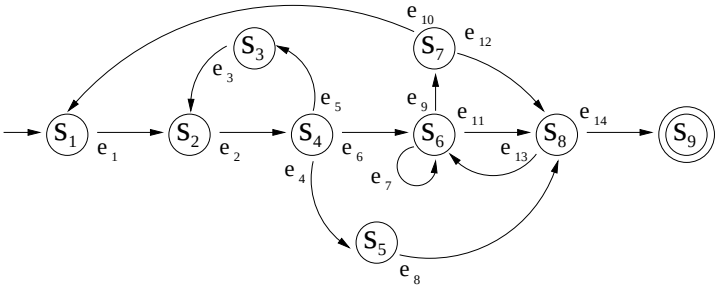

Figure 3.2: Component internal view.

3.2.1.1 Input Ports, Output Ports, Port Packets and Port Connections

In CoolBOT the port automaton's set L is constituted by two disjoint sets: a set of *input ports*, L_i, and a set of *output ports*, L_o; such that they verify that $L = L_i \cup L_o$ and $L_i \cap L_o = \varnothing$.

An output port and an input port may form a *port connection*. Data are transmitted through port connections in discrete units called *port packets*. *Port packets* are defined as discrete units of information which can be received through input ports, and/or issued through output ports. Therefore, *port packets* constitute the elements of the input and output sets X and Y. *Port packets* are also classified by their type, and usually each input and output port can only accept a specific set of port packet types.

To establish a *port connection* both input and output ports must be compatible, i.e. both ports must match exactly the type of port packets they can accept. CoolBOT components interact and inter communicate each other by means of port connections established among their input and output ports.

Additionally input and output ports can be *public* or *private*:

- *Public*: Input and output ports are visible and accessible from outside the component. Thence, other components may freely use these ports to establish new port connections.
- *Private*: Input and output ports are not externally visible. These ports are hidden inside the component interface, and have three main uses:
  - To support internal *timers* and *watchdogs* (sections 3.2.3 and 3.2.2.3).
  - To control, monitor and inter communicate internal components inside *compound components* (section 3.6.2).
  - To control, monitor and inter communicate *port threads* inside components (section 3.4.1).

3.2.1.2 Automaton States

The internal functionality of a component is defined by means of an automaton. The transitions between states of the automaton are determined either by port packets received through its input ports, or by any internal condition. Thus, in CoolBOT, in order to define the internal automaton of a component, each state is defined by means of several code sections:

- **Entry Section**: It is a portion of code which is executed each time the component enters into a state when it comes from a different one.
- **Exit Section**: It is a portion of code that gets executed when the automaton is about to leave the current state to transit to a different one.

- **State Transitions**: Each transition in any state is associated to a portion of code which is executed when the transition is triggered. In this portion of code, it is also indicated towards which state the transition drives the component.

3.2.2 Robustness

Following the *principle of robustness* or *robustness motto* mentioned in section 2.4.1 in chapter 2 (*"A component-oriented robot system will not be robust and controllable if its components are not robust and controllable"*) some important questions come up: when is a component robust?, how is it possible to guarantee that a component is robust?. Obviously, the robustness of a component can only be guaranteed by design and continuous testing, but we consider the framework should provide means and facilities to help component developers to build *robust components* where observability and controllability play fundamental roles. CoolBOT provides several framework constructs in order to support a robust design of components, namely: *observable and controllable variables*, *component exceptions* and *watchdogs*.

3.2.2.1 Observability and Controllability

Components should be observable enough to know whether they are working correctly or not, and in that case, they should be controllable enough to make some adjustment in their internal behavior to regulate and adjust their operation. Component functionality must be kept hidden behind its external port interface, however means to monitor and control such a functionality should be provided. CoolBOT introduces two kinds of variables as facilities in order to support monitoring and control of components.

- **Observable variables**: Represent features of components that should be of interest from outside, they are externally observable and permit publishing aspects of components which are meaningful in terms of control, or just for observability and monitoring purposes.
- **Controllable variables**: Represent aspects of components which can be externally controlled, i.e., modified or updated. Thence, through them the internal behavior of a component can be controlled.

Thus, components provide resources to guarantee their observability and controllability by means of these two sets of variables, evidently it is up to component designers to use observable and controllable variables conveniently to fulfill such a purpose.

3.2.2.2 Component Exceptions

Exceptions constitute a useful concept present in numerous programming languages (C++ [Stroustrup, 2000], Java [Arnold et al., 2000], etc.). Usually, exceptions are programming languages constructs that have been devised to separate error handling from

the normal flow of instructions in a program. Thence, exceptions are normally used to signal erroneous or anomalous situations at runtime, and its occurrence provokes the execution of a specific code to handle such situations through *exception handlers*.

Analogously, CoolBOT components may use exceptions to signal and handle erroneous, exceptional or abnormal situations during its execution. Concretely, when a CoolBOT component is defined during its design, it includes a list of *component exceptions*, declaring each exception using the following pattern:

```
On Exception: <exceptionId>
    <description>
    [<handler> [<retries> <period>]]
    [<onSuccessHandler>]
    [<onFailureHandler>]
```

where:

- `<exceptionId>` is an exception number to identify the exception.
- `<description>` is a description of the exception.
- `<handler>` is an optional handler to try an exception recovery procedure, optionally `<retries>` indicates the number of recovery attempts, and `<period>` specifies the period in milliseconds between attempts.
- `<onSuccessHandler>` is a handler to be executed in case of a successful recovery. It is optional.
- `<onFailureHandler>` is also an optional handler which is executed when all recovery exception tries have failed.

3.2.2.3 Port Watchdogs

Normally real time systems utilize *watchdogs* to signal a concrete type of abnormal situation that happens when a temporal deadline has not been reached. Normally, they are used to signalize that a task has not been completed within a specific amount of time.

CoolBOT provides a construct to associate a *port watchdog* with an input port, so that the watchdog is triggered when no port packets has been received through this port for a period of time. When this situation occurs a component exception associated with the watchdog is thrown.

3.2.3 Timers

CoolBOT also provides a *timer* construct, so that components may have timers to signal normal and abnormal situations (periodic task execution, task watchdogs, etc). As component exceptions and port watchdogs, timers fit in the port automata model

as internal conditions in a component that signal events triggering specific transitions in the component automaton.

3.2.4 Component Priorities

At execution time, CoolBOT components may have associated a specific priority, in terms of competing for CPU time. Specifically, the priority of a component is determined by a *priority policy* and a *priority* level. CoolBOT has two priority spaces, called *priority policies*, each with 16 different priority levels or *priorities*. The priority policies can be *normal* or *realtime*; inside each policy there are 16 priority levels where 0 is the lowest priority, and 15, the highest one. The priority spaces corresponding to both policies are disjoint spaces, where realtime policy priorities are higher than normal policy priorities. In this way, components with realtime policy are always prioritized, whatever priority they have, respect to any component with normal policy. Figure 3.3 depicts the range of component priorities.

The underlying operating system on top of which CoolBOT runs, assigns priorities to its units of execution – *processes* and *threads* – when they compete for CPU time. Depending on what operating system is running CoolBOT component priorities will be mapped differently.

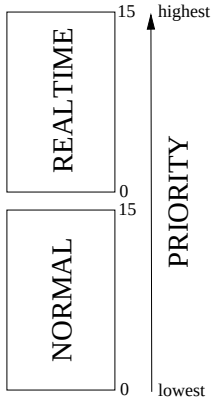

Figure 3.3: Component priorities.

As mentioned in section 2.4.1 of chapter 2, a principle of design was to implement CoolBOT into the two operating systems which are mainstream at the present moment: the Windows family of operating systems (Windows NT, 98, 2000 and XP), and GNU/Linux.

There are some differences between the thread models used by these operating systems, therefore the CoolBOT priority model maps distinctly in both. Concretely, in GNU/Linux, CoolBOT uses the *pthread* library that implements the *POSIX Threads* specification [IEEE, 1996], and maps component priorities into the POSIX priority model of scheduling policies (`SCHED_RR` and `SCHED_OTHER`) and priorities [Nichols et al., 1996] [Bover and Cesati, 2001]. The mapping of priorities in GNU/Linux is shown in figure 3.4. As to Windows, CoolBOT uses the Win32 standard API using native threads, and maps component priorities into the Win32 priority model of priority classes (`REALTIME_PRIORITY_CLASS` and `NORMAL_PRIORITY_CLASS`) and priority levels [Richter, 1997] [Solomon and Russinovich, 2000] [MSDN, 2002]. Figure 3.5 illustrates the priority mapping for Windows. In both operating systems, it is necessary to have respectively, `administrator` and `root` rights to use the *realtime* component priority policy, otherwise, the *normal* policy is the only policy available.

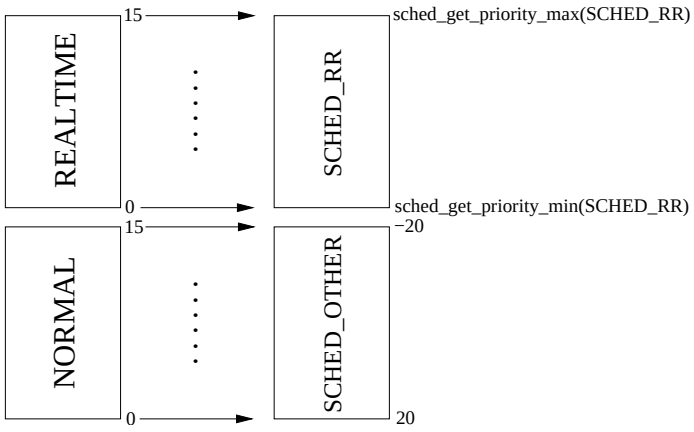

Figure 3.4: GNU/Linux component priority mapping.

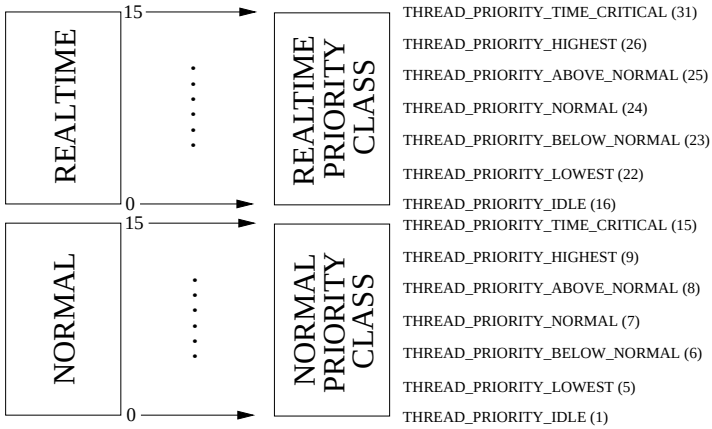

Figure 3.5: Windows component priority mapping.

3.3 Component Defaults

A component-oriented approach clearly demands certain level of uniformity among components (*component uniformity* design principle, chapter 2, section 2.4.1) in order to have components which are observable, controllable and integrable. Within CoolBOT this uniformity manifests itself in two important aspects:

- a *minimal uniform external interface* based on the concept of port automata which allows component external observation and control in terms of execution, and
- a *minimal uniform internal structure* which permits component execution to be externally observed and controlled.

Classic operating system theory [Silberschatz et al., 2001] [Stallings, 2000] considers as fundamental the principles of uniformity of interface and internal structure for its units of execution. Such a uniformity permits operating systems to deal with these

units of execution – *processes* and *threads* – atomically, and independently of the functionality they have. Furthermore, it allows defining useful tools as design frameworks, profilers and debuggers for their design and development.

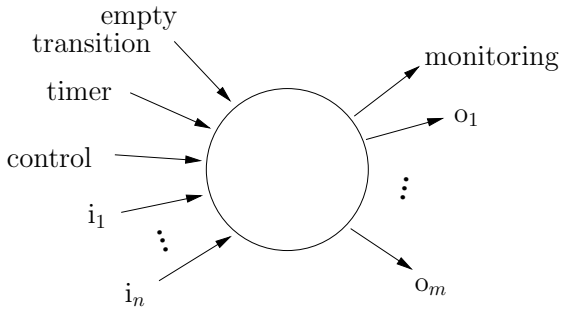

Figure 3.6: Default ports.

Similarly, component uniformity in terms of interface and internal structure is fundamental in CoolBOT. They constitute key elements to facilitate the design and definition of robust systems, making their development and implementation less error-prone, and establishing a real basis for designing and developing debugging and profiling tools for components.

CoolBOT imposes component uniformity by means of what it is referred to as *component defaults*. Component defaults are constituted by a set of features and characteristics that all CoolBOT components include by definition. They enumerate as follows:

- *control* and *monitoring* ports;
- default *observable* and *controllable* variables;
- the *default automaton*;
- default exceptions;
- a *default timer* and a *default timer* port;
- a default port for automaton empty transitions;
- and the *main* thread.

These default elements will be introduced and explained along the next subsections.

3.3.1 Control and Monitoring Ports

To guarantee external observation and control, CoolBOT components provide by default two important ports: the *control* port and the *monitoring* port, both depicted in figure 3.6.

- The **monitoring** port: This is a public output port by means of which component *observable variables* are published. Therefore, through this port, an external observer or supervisor can observe and monitor a component.

- The **control** port: This is a public input port through which component *controllable variables* are modified and updated, consequently an external controller or supervisor can control a component by means of this port.

So, component's *observable* and *controllable variables* are respectively read or published, and written through these two special ports. Note that this implies that for each component both ports should be able to accept enough port packet types to support publication of all its *observable variables*, and also to support updating of all its *controllable variables*. *Control packets* are defined as port packets that modify component control variables, and *monitoring packets* are defined as port packets used to publish changes in observable variables.

3.3.2 Default Observable and Controllable Variables

CoolBOT provides components with several default observable and controllable variables as tables 3.1 and 3.2 show. The **default observable variables** are the following:

Default Observable Variables		
Name	Symbol	Brief Description
state	*s*	Automaton state where the component is situated.
priority	*p*	Current component execution priority.
config	*c*	Asks for a supervised change of configuration, or confirms configuration commands.
result	*r*	Result of execution.
error description	*ed*	Error description after the occurrence of a locally unrecoverable exception.

Table 3.1: Default observable variables.

- **state**: At any instant during component´s life cycle this variable publishes the automaton state where the component is situated.

- **priority**: This observable variable is used to publish the component priority (section 3.2.4) at which the component is running.

- **config**: This observable variable has been devised to be used as a way to publish internal configurations changes, or to confirm configuration commands commanded externally. Actually it is only used in *compound* components (see section 3.6.2.1.1).

Default Controllable Variables		
Name	Symbol	Brief Description
new state	*ns*	Desired automaton state where the component has been commanded to go.
new priority	*np*	Desired execution priority the component has been commanded to have.
new exception	*nex*	Externally induced exception.
new config	*nc*	Component's configuration can be modified and updated during execution through this controllable variable.

Table 3.2: Default controllable variables.

- **result**: If necessary, using this observable variable components may return data as the result of a task execution. Components' designers and developers decide the type of data which is returned. In general component generates as result of its task execution data issued through any of the output ports of its external interface. But also, in some occasions, it might be necessary to produce a result when task execution has finished, in such a way that, an external supervisor or component controller could make control decisions based on this information. This variable has been mainly devised with this purpose in mind.

- **error description**: As commented in subsection 3.2.2, components use exceptions to signalize abnormal and exceptional situations during execution. Usually when an exception happens, the component tries to solve the erroneous condition applying the error handlers associated with the exception, if none of them are successful, the component enters into an erroneous situation. In this case, using this observable variable the component publishes a description of the unrecovered exception.

And, these are the **default controllable variables**:

- **new state**: Through this controllable variable a component can be commanded externally to go to a specific state of its automaton.

- **new priority**: This controllable variable allows to command components to change the priority at which they are running, so through this variable it is possible to command the component to acquire a new specific priority.

- **new exception**: During component execution the occurrence of an exception can be externally injected into the component by means of this controllable variable, whose main purpose is to serve as a means to test the behavior of a component against exceptions that might appear during execution.

- **new config**: Changes on the internal configuration of a component can be forced using this port, such changes are confirmed by means of the observable variable **config**. Actually it is only used in *compound* components (see section 3.6.2.1.1).

Bear in mind that the observable variable **config** and the controllable variable **new config** has been devised mainly for future uses. Actually these two variables are only used to control and supervise internal topology changes in *compound* components (see section 3.6.2.1.1 for a detailed explanation). A future use that is actually foreseen, but has not been implemented yet, is its use to control and observe components in order to make them adaptable to system resource availability [Hernández-Sosa, 2003].

As mentioned previously (subsection 3.3.1) component´s observable variables are issued and published through the *monitoring* port, and controllable variables are updated through the *control* port. For each one of the default variables shown in tables 3.1 and 3.2, CoolBOT provides a port packet type which permits its transmission and reception through these two ports. Hence, there are several default control packets and several default monitoring packets to support these default variables. Component developers can add new observable and controllable variables, but notice that control and monitoring packets corresponding to non default observable and controllable variables should be defined by the component designer.

3.3.2.1 Component Execution Control Loop

In general, for a component, a typical *execution control loop* uses the *control* port to exert control actions by means of control packets, and uses the *monitoring* port as the feedback to close the loop, figure 3.7 helps to illustrate this idea. That implies that control actions should be verified through the *monitoring* port, observing if any of the component observable variables has been affected by the exerted action. In this way, for instance, if a control action has been commanded to change the automaton state of a component, the verification of that action should be to check that the observable variable *state* has changed adequately in order to assure that the commanded state has been reached.

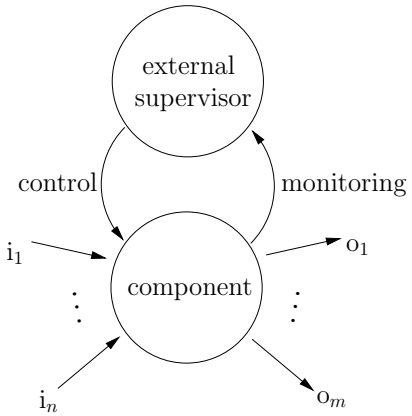

Figure 3.7: *Control* and *monitoring* ports: a typical component control loop.

3.3.3 The Default Automaton

Internally all components are modelled using the same default state automaton, the *default automaton*, shown in figure 3.8, that contains all possible control paths that a component may follow. In the figure, the transitions that rule the automaton are labelled to indicate the event that triggers each one.

Some of the labels corresponds to internal events: *ok*, *exception*, *attempt*, *last attempt* and *finish*. The remaining labels indicate events provoked by default controllable variable changes: ns_r, ns_{re}, ns_s, ns_d, *np*, and *nex* (see tables 3.1 and 3.2). Subscripts in ns_i indicate which state has been commanded, where subscript i can be any of the followings: r (**running** state), re (**ready** state), s (**suspended** state), and d (**dead** state).

The *default automaton* is said to be "controllable" because it can be brought externally in finite time by means of the *control* port to any of the controllable states of the automaton, which are: **ready**, **running**, **suspended** and **dead**. The rest of states are reachable only internally, and from them, a transition to one of the controllable states can be forced externally.

The **running** state, the dashed state in figure 3.8, constitutes or represents the part of the automaton that implements the specific functionality of the component, and it is called the *user automaton*. The *user automaton* varies among components depending on their functionality, and it is defined during component design and development. The initial state of a *user automaton* constitutes its *entry state*.

Having a look to figure 3.8 it is possible to observe how CoolBOT components evolve along their execution time, since they are *launched*, i.e. put into execution in the underlying operating system, until they finish their execution.

As soon as a component is launched, it is brought to **starting** state, where the component should capture resources needed for its operation and carry out its internal initialization. If any error comes up while requesting resources or during initialization, a component exception may be thrown. In this case, the automaton transits to **starting error recovery** state. Alternatively, if resource allocation and initialization is accomplished successfully, the component is brought into **ready** state.

A component gets to **starting error recovery** when a exception has been thrown in **starting** state. In this state the component executes periodically the *handler* of the thrown exception a maximum number of times (see *Component Exceptions* in section 3.2.2), this is shown in figure 3.8 with a transition labelled *attempt*, which is triggered by a timer. If the handler recovers successfully from the exception in any of the attempts, its *on-success handler* is executed, and then, the component transits again to **starting** state to continue on with resource allocation and initialization. Otherwise, the component executes the exception *on-failure handler*, then, is driven to **starting error** state, where the component publishes an error description through the controllable variable *error description*, and after that, it waits for external intervention through the *control* port, in order to jump to **dead** state.

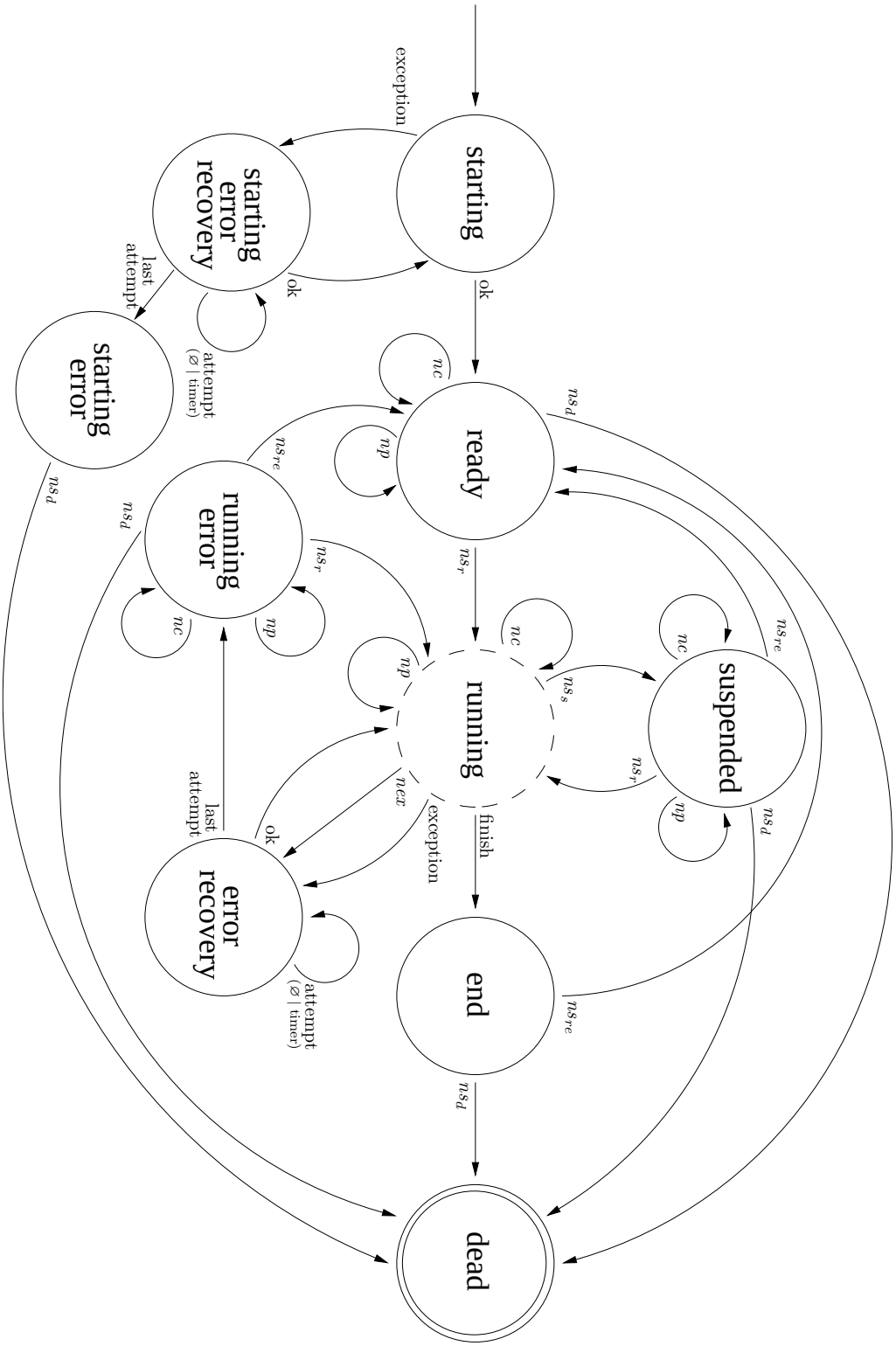

Figure 3.8: The Default Automaton.

At **ready** state the component waits idle either for initiating a task execution, getting into the *user automaton* – **running** state – by means of its *entry state*, or for its destruction, if it is driven to **dead** state. In **ready** state, the component can also be commanded to change its priority, or to accept a configuration command.

The **running** state is the part of the automaton – the *user automaton* – that endows the component with its particular functionality, therefore, it is in this state, where components accomplish their specific tasks. The *user automaton* has its own states and transitions, but the default automaton imposes the following requirements on it:

1. From any state of the *user automaton* it must be possible to drive the component to **suspended** state (transition labelled as ns_r in figure 3.8). This implies that the component should be externally interruptible at any user state, with a latency that will depend on its internal design. The component should save its internal status in case of continuing task execution. From **suspended** state the component can also be restarted, i.e. driven to **ready** state again, or brought to **dead** state. In **suspended** state, the component can also be commanded to change its priority, or to accept a configuration command.

2. It must be possible to change component priority at any user automaton state (transition labelled as *np* in figure 3.8), or to accept a configuration command (transition labelled as *nc*).

3. Some other default states must be accessible from any of the user automaton states, because they indicate component generic situations:

   - **Error recovery** state : It is reached when an exception is detected during task execution. As with **starting error recovery** state, in this state, an exception handler recovery procedure can be tried several times, until the maximum number of attempts have been exhausted unsuccessfully. In that case, the automaton goes to **running error** state. If the exception handler is successful in any of the recovery attempts, the automaton continues where normal task execution was interrupted previously. That implies that the component internal status must be preserved during exception recovery as well.
   - **Running error** state: When a component has not been able to recover itself from a component exception in the **error recovery** state, it transits into this state. Only external intervention can drive the component again to **ready** state to start a new task execution, or to **dead** state to finish component execution. Additionally in this state, component priority can be modified, and configuration commands can be accepted.
   - **End** state: If a component has finished its task, then it goes directly to **end** state, from which it can be brought to either **dead** state, or **ready** state. In this state the *result* observable variable is published, in order to return a value as a result of a task execution.

At **dead** state a component is supposed to finish its execution, so there, it should execute all its finalization routines, and release all resources that have been allocated.

3.3.4 Default Exceptions

CoolBOT defines several default exceptions that components can make use of. These exceptions are:

- *No Memory*: Indicates an out-of-memory situation during dynamic memory allocation.
- *Inconsistency*: Signals an inconsistency in the component automaton. For instance, if a component gets to any of the error recovery states of the default automaton without previously having thrown a component exception.
- *File Not Found*: Stands for a file-not-found situation when a file is not found where it should be.

Notice that, these default exceptions are defined by CoolBOT, but initially they have no handlers associated. Component designers and developers may assign their own handlers for them, and the number and period of recovery attempts.

3.3.5 Default Timer

CoolBOT components may have several timers, but initially, each one is provided with a *default timer*, which is usually used to implement the mechanism of exception recovery in the error recovery states (**starting error recovery** and **error recovery**) of the default automaton (see figure 3.8).

Usually components use private input ports to realize when their internal timers get triggered. Due to the fact that all components are provided with a *default timer* by CoolBOT, they are also equipped with a default private input port called *timer* to be used by the internal *default timer*. This is one of the input ports that appears in figure 3.6.

3.3.6 Other Defaults

A component, due to its internal design and functionality, may need to force a transition internally. This can be carried out using what is called an *empty transition*. To do so and to signal itself these kinds of transitions, all component is provided also with an additional private input port called *empty transition*. It appears also in figure 3.6.

There is another component default that will be explained in the next section, but that must be mentioned here since it is also a component default: the *main* thread. Its detailed explanation is deferred to the next section.

3.4 Component Nuts and Bolts

According to the definition for CoolBOT components given in section 3.2, components are *weakly coupled entities* that execute concurrently or in parallel, on their own initiative, in order to achieve their own independent objectives. Thence, components are not only data structures, but execution units as well. In fact, CoolBOT components are mapped as *threads* when they are in execution; Win32 threads [Solomon and Russinovich, 2000] [Richter, 1997] [MSDN, 2002] in Windows, and POSIX threads [Nichols et al., 1996] [IEEE, 1996] in GNU/Linux.

```
1 void Component::kernel()
2 {
3   PortPacket packet;
4
5   initialization();
6
7   while(true)
8   {
9     packet=waitForSomething(inputPorts);
10
11    if(!processPortPacket(packet)) break;
12  } // end of while
13
14  finalization();
15 }
```

Figure 3.9: Simplified C++ kernel code.

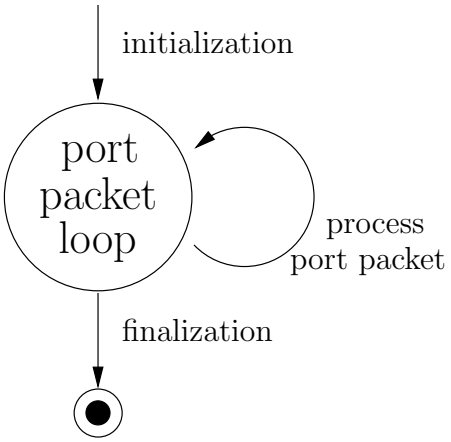

Figure 3.10: Simplified kernel.

At runtime a CoolBOT component executes a continuous loop processing port packets. Figure 3.9 lists a simplified C++ version of the code corresponding to this loop, and figure 3.10 depicts a graphical representation of it. This processing loop is referred to as the *component kernel*, and in it, the component carries out different actions depending on which input port has received each port packet, and in which state of its automaton the component is. This is why, components are said to be *input-port-driven*.

3.4.1 Port Threads

Frequently, it might be convenient that a CoolBOT component uses multiple threads to execute itself, in this case there would be some flows of execution running concurrently and/or in parallel, each one running a loop processing port packets. Figure 3.11 helps to illustrate the idea.

In CoolBOT, threads used inside a component to execute itself are called *port threads*. Although *port threads* are not components, they are also input-port-driven, in the sense that their execution is driven by the port packets they receive through a set of input ports. Thence, *port threads* are also processing port packet loops at runtime.

In a component a *port thread* is responsible for a subset of the whole set of input ports of the component. Thus, it is in charge of executing the transitions corresponding

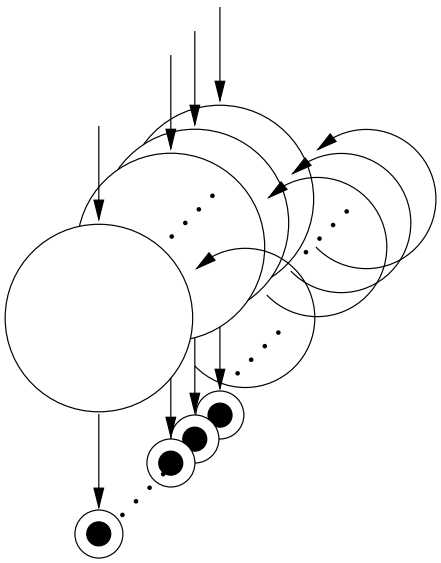

Figure 3.11: Multiple threads.

to this subset of input ports. Note that subsets assigned to different *port threads* should be disjoint. The same is applicable to output ports, thence, each *port thread* is also responsible for a subset of the component´s output ports. The kernel of a multithreaded component will include several *port threads* executing their own port packet loops having the appearance shown in figure 3.11.

Like components, *port threads* have also two ports for its external observability and control: an input port for control and an output port for monitoring. At runtime they follow an automaton as well, in particular the automaton displayed in figure 3.12 which consists of three states:

- **suspended**: Once a *port thread* has been launched, it gets to **suspended** state, where it remains idle until it is ordered, externally and through its control input port, to go to **running** or **dead** states.

- **running**: In this state the *port thread* executes continuously running component´s automaton transitions according to the port packets it receives through the input ports it is in charge of. From this state it can be driven to **suspended** or **dead** states by means of the control input port.

- **dead**: The *port thread* gets to this state to finish its execution.

For further clarification in figure 3.13 it is shown the C++ pseudo code corresponding to the automaton of figure 3.12. This is the *port thread kernel*, and constitutes the *port threads*' port packet loop. Observe that each time the port thread change their state, it publishes its current state (through its monitoring output port) – `publish(currentState)`.

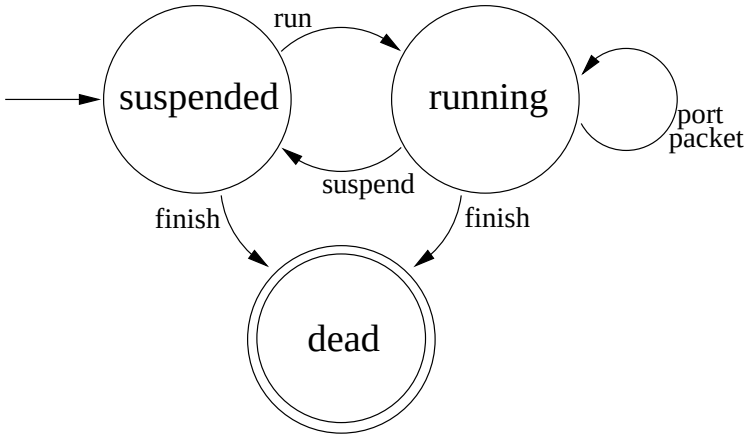

Figure 3.12: Port thread automaton.

3.4.2 The Main Thread

At runtime CoolBOT components, whether multithreaded or not, executes its automaton, composed by the *default automaton* and the *user automaton* explained in section

```
1 void PortThread::kernel(Component& component)
2 {
3   PortPacket packet;
4   InputPort port;
5
6   currentState=nextState=SUSPENDED;
7
8   while(currentState!=DEAD)
9   {
10    publish(currentState);
11
12    while(currentState==nextState)
13    {
14      switch(currentState)
15      {
16        case SUSPENDED:
17          packet=waitForSomething(controlPort);
18          nextState=packet.getState();
19          break;
20
21        case RUNNING:
22          port=waitForSomething(inputPorts);
23
24          if(port==controlPort)
25          {
26            packet=getPacket(controlPort);
27            nextState=packet.getState();
28          }
29          else runTransition(component,port,packet);
30
31          break;
32      }
33    } // end of while
34
35    currentState=nextState;
36  } // end of while
37
38  publish(currentState);
39 }
```

Figure 3.13: Pseudo C++ port thread kernel code.

3.3.3. A CoolBOT component can execute by means of multiple threads, but at least, it needs one thread to run. This is the thread that executes the automaton of the component, that controls and observes the component´s port threads, if any, and that keeps its internal state consistent, in the sense that, it is responsible for maintaining the consistency of the internal data structures that conform the internal state of the whole component. This thread is called the *main* thread. The *main* thread is also a component default provided by CoolBOT as commented in section 3.3.6.

The *main* thread controls the execution of the component, executing port transitions, transiting from state to state, and launching, suspending, running and killing or destroying – un-launching – port threads, if any. A detailed explanation of this aspect of the *main* thread is deferred to the next section.

During design and implementation of a multithreaded component it is necessary to indicate for each state which port thread should be running on each automaton state. At runtime the *main* thread will run and/or suspend each port thread depending on in which state of the automaton the component is along its execution. Remember that *port* threads have their own control and monitoring ports. Therefore, for each internal *port thread* the component should dispose of two internal private ports, an output one and an input one, to control and monitor its execution.

3.4.2.1 The Component Kernel

Figure 3.14 shows the pseudo C++ code that constitutes the core of all CoolBOT components at runtime. This is the port packet loop that keeps the execution of a component consistent. This is the code that is executed by the *main* thread of any CoolBOT component, and constitutes a more detailed version or the *component kernel* shown in figures 3.9 and 3.10. In a multithreaded component it will correspond to one of the threads appearing in figure 3.11.

Mainly, the *component kernel* consists of two nested while loops, where the outer one cycles indefinitely until the component finishes its execution, what happens when it arrives to **dead** state (see the *default automaton* on figure 3.8). This situation is codified inside the *component kernel* with a special value, `NOT_RUNNING`. The *component kernel´s* variable `currentState` takes that value when the automaton is about to finish. The inner loop executes the automaton transitions corresponding to each state of the component automaton, and it ends when a transition drives the component to a different state.

Observe that once the component has been launched, and before getting into the first while loop of figure 3.14, the component kernel initializes some local variables, `currentState`, `lastState`, `nextState`, etc., and launches its internal port threads, `launchPortThreads()`. Note from figure 3.12 that once *port* threads have been launched, they get to **suspended** state where they keep doing nothing, until they are ordered to go to other state.

At the beginning of each outer loop iteration of the kernel the observable variable *state*, table 3.1, is published by means of the component *monitoring port*, `publish-`

```
1 void Component::kernel()
2 {
3   TransitionFunction transition;
4   ControlTransitionFunction controlTransition;
5   InputPort signaledPort;
6   ControllableVariable controlVariable;
7
8   exceptionState=NO_STATE;
9   exception=NO_EXCEPTION;
10
11  lastState=NOT_RUNNING;
12  currentState=STARTING;
13
14  launchPortThreads();
15
16  while(currentState!=NOT_RUNNING)
17  {
18    publishState(); // Publish current state
19
20    setControlMask(currentState);
21    setMask(currentState_);
22
23    launchStopWatchDogs(currentState);
24
25    nextState=runEntrySection(currentState);
26
27    runPortThreads();
28
29    while(nextState==currentState)
30    {
31      flushObservableVariables();
32
33      // Get something from any port
34      signaledPort=waitForSomething(inputPorts);
35
36      if(signaledPort==controlPort)
37      {
38        controlVariable=
39            getIndex(signaledPort);
40
41        controlTransition=
42          getControlTransition(currentState,
43                               controlVariable);
44
45        // Execute control transition
46        if(controlTransition)
47          nextState=
48            controlTransition(signaledPort,
49                              controlVariable);
50      }
51      else // !=controlPort
52      {
53        transition=getPortTransition(currentState,
54                                     signaledPort);
55        // Execute transition
56        if(transition)
57          nextState=transition(signaledPort);
58      }
59    } // end of while
60
61    suspendPortThreads();
62
63    runExitSection(currentState);
64
65    flushObservableVariables();
66
67    lastState=currentState;
68    currentState=nextState;
69  } // end of while
70
71  unlaunchPortThreads();
72 }
```

Figure 3.14: Component kernel.

`State()`. Then, two masks are established, `setControlMask()` and `setMask()`, for the current automaton state. The first one, the *control mask*, masks the controllable variables that do not provoke any automaton transition in the current state. The second one, the *port mask*, masks the input ports that do not trigger any transition either. After that, if there should be any active watchdogs in the current state, they get started; and at the same time, any active watchdog that should be inactive, is stopped, `lauchStopWatchDogs()`. Next, the *entry* section of the current state, if any, is executed, `runEntrySection()`. Note that a change of state can be forced in this point, since `runEntrySection()` returns the next state to go. This has been devised for states like **starting** and **dead** in the *default automaton* (figure 3.8) that do not have any transition triggered by input port packets.

If the component is multithreaded, all internal port threads previously launched, that should be running in the current state, are ordered to change their internal state to **running** (see figure 3.12), `runPortThreads()`. In this way, they will be able to manage the transitions corresponding to the input ports they are responsible for.

Next, the kernel starts the inner loop that is in charge of executing the transitions corresponding to each state. Concretely, all observable variables that have changed are published, `flushObservableVariables()`, by means of the *monitoring* port. Mind here that the *main* thread executes also port transitions, so in a multithreaded component, transitions are executed by the *main* thread, concurrently (and/or in parallel) with the internal *port* threads that are running at the current automaton state. This is the reason of having the port threads, and the *main* thread as well, being in charge of the transitions associated to different disjoint subsets of the component´s input and output ports. It is mandatory that the *main* thread at least should be responsible for the component´s default ports, the *control* port and *monitoring* port of figure 3.6.

Following the nomenclature of Win32 API [MSDN, 2002] [Richter, 1997] [Solomon and Russinovich, 2000], CoolBOT input ports are *synchronization objects*, that is, they can have two states, *signaled* or *non signaled*. A component can block its execution, waiting until a *non signaled* input port is set to the *signaled* state. Usually, input ports keep *non signaled* while they have not received new port packets since the last time the input port´s owner, the component, accessed the port to get an incoming packet. They get *signaled* when a new port packet has just been received. Once they change their state to *signaled* they hold this state until one of the following operations is carried out:

- *Testing*: This is an operation to test whether an input port is *signaled* or not. In case of being *signaled*, the testing operation atomically makes the input port *non signaled*, and returns a value to indicate that the port was *signaled*. Otherwise, the operation returns simply a value specifying that the port was *non signaled*.

- *Waiting*: This operation is also used to test if an input port is *signaled* or not, but it operates differently. In case of not being *signaled* the operation blocks the caller. It will continue on waiting, blocked until the port gets *signaled*. Just at

the moment of being *signaled* again, the waiting operation atomically unblocks the caller, makes the port *non signaled*, and returns a value indicating that it was *signaled*. In case of being signaled, it operates in the same way that a testing operation. Finally, mind that a waiting operation can be *"timed"* in the sense that it is possible to specify how long the caller will be blocked waiting for a port signalization. If the port does not get *signaled* along the time specified, the waiting operation returns a value indicating that the port was not *signaled*. Notice that a timed waiting with time of zero is equivalent to a testing operation.

Therefore, and going on with the description of the kernel inner loop, once the observable variables are published, the component waits for (observe that is not a timed wait) that any not-masked input port gets *signaled*, `waitForSomething()`. So that, the component keeps blocked there until, at least, one of its input ports changes to *signaled*, due to the reception of a new incoming port packet. Notice in figure 3.13 that *port* threads use the same type of operation on input ports to wait for incoming port packets, `waitForSomething()`.

Observe that according to that, depending on at which rate or frequency a component is receiving port packets, the component (the threads that execute it) would be blocked for most of its execution time. Thus, components could be described as *data-flow-driven machines*, processing when they dispose of data in their input ports, and otherwise, keeping idle, waiting for processing new input port packets.

Once an input port has been signaled in the kernel inner loop, it is determined whether there is a transition associated with the *signaled* port. Notice that, if this port is the *control* port, it is necessary to find out which controllable variable has been updated, `controlVariable`, in order to select its associated transition. Finally the transition, whether a control transition or not, is executed. Note that the return value of running transitions is the state where the component should go.

The inner loop finishes when any of the triggered transitions returns a state which is different from the current one. What the outer loop does next is to suspend the port threads that are running at that moment, `suspendPortThreads()`. Bear in mind that the *main* thread, and the component´s internal *port* threads share the same address space. Having all port threads suspended outside the inner loop, guarantees that none of them accesses and/or modifies component´s internal data structures, that could be a danger for the consistence of the its execution.

After that, the *component kernel* executes the state *exit* section, `runExitSection()`, then, it publishes any observable variable that could have changed, and transits to the next state. In this way the kernel keeps iterating indefinitely until the next state to transit is `NOT_RUNNING`, that, as commented in a previous paragraph, occurs when the component gets into **dead** state and runs its *entry* and *exit* sections. Once the *component kernel* quits its outer loop, it destroys – un-launches – all its internal *port* threads, if any, and then finishes itself.

Finally, it is significant to pay attention to the fact that, the *component kernel*, and the kernels of the *port threads* it might have, are processing loops of input port

packets, where always that an input port gets *signaled*, its associated portion of code, an automaton transition, is executed. This transition is executed synchronously, in the sense that its corresponding flow of execution (run by either the *component kernel*, or one of the *port thread* kernels), runs completely the transition, and then, returns to continue on. Actions exerted on the component due to transitions originated by port packets received through the *control* port are called *control actions*, and the transitions that carry out then are called *control transitions*. Like all automaton transitions, *control transitions* are also executed synchronously. Actions derived from the execution of transitions originated by port packets received by the rest of the component´s ports are called *functional actions*, and their associated transitions are referred to as *functional transitions*. Their execution is synchronous as well. All in all, the execution of *control actions* is interleaved with the execution of *functional actions* corresponding to non control transitions. They do not interfere each other and, as a consequence, *functional actions* are *atomic* respect to *control actions*, and vice versa, in the same way that the portions of code called *codels* in G^{en}oM [Fleury et al., 1997] are also considered atomic. From figure 3.14 it is evident that automaton state *entry* and *exit* sections are also atomic in the same sense.

3.4.3 Input Port Priorities

At runtime, CoolBOT components keep an internal queue or fifo of signaling input ports. Input ports get inserted in it when they become *signaled*, and get extracted from it when they become *non signaled*. During execution the *component kernel* and the possible *port* threads keep reading and emptying this queue as they process port packets. Note that `waitForSomething()` function invocations in figures 3.9, 3.13 and 3.14 access the queue to know which input port gets *signaled* with new incoming port packets. On the other side, each time an external component sends a port packet through any of the component´s input ports, they get *signaled*, and therefore, queued into the fifo. It is important to notice that component´s input ports get inserted in the internal queue as they get *signaled*, and they are extracted from it in the same order they were inserted.

In general, using a queue of signaled input ports for all component´s input ports is enough, but it was considered interesting to have the possibility of establishing priorities of input ports. In this way, the *component kernel* and the *port* threads, if any, could process first port packets of input ports with higher priorities. As a consequence of this idea, CoolBOT provides the possibility of establishing different level of priorities for attending the input ports of a component.

3.5 Inter Component Communications

Analogously to modern operating systems that provide IPC (**I**nter **P**rocess **C**ommunications) mechanisms to inter communicate processes [Silberschatz et al., 2001], CoolBOT provides ***I**nter **C**omponent **C**ommunications*

or *ICC* mechanisms to allow components to interact and communicate among them. CoolBOT *ICC* mechanisms are carried out by means of input ports, output ports, and ports connections (section 3.2.1). More precisely, any ICC mechanism may be only put into practice through a previously established port connection.

Two main principles have driven the design and development of these ICC mechanisms:

- *Asynchronous Communications*: According to the definition of CoolBOT components given in section 3.2, they are *weakly coupled entities* that execute concurrently or in parallel, on their own initiative, in order to achieve their own independent objectives. To allow for these component's properties of *weakly coupling* and *independency*, all CoolBOT ICC mechanisms are *asynchronous*. That is, the operations of data sending and reception are completely uncoupled, and are carried out in different instants of time by different components.

- *Transparency and Minimization of Inter Component Synchronization*: The problem of *inter component synchronization* comes up when more than one component try to access the same data at the same time. In the case of ICC mechanisms, this problem of synchronization appears with the data structures and code that implements CoolBOT input and output ports, and port connections. ICC mechanisms has been implemented in a way that all synchronization details are kept hidden to make them transparent to their users, the components themselves. Additionally, a main motto have been adopted to minimize synchronization: *"Wherever it is possible, and for each component, try to keep local copies of any data needed to work independently, in this way, synchronization is only necessary to update such copies."* This motto will be referred as the *cache* motto.

Communications are one of the most fragile aspects of distributed systems. In CoolBOT, the rationale for defining standard methods for data communications between components is to ease inter operation among components that have been developed independently, offering optimized and reliable communication abstractions. The rest of this section will explain the basic ICC mechanisms that CoolBOT supports, and how these mechanisms are used on each typology of port connections provided by CoolBOT.

3.5.1 Basic ICC Mechanisms

As already commented in section 3.2.1 the inter communication of components consists of the sending and reception of port packets through port connections previously established. A port connection involves two components, the *sender*, which is the owner of the output port involved in the connection, and the *receiver* in the other end, the owner of the input port. To form a valid port connection between them it is mandatory that both ports admit the same types of port packets. Thus, once port connections have been established and formed between two or more components, ICC mechanisms are responsible for sending and receiving port packets through them.

It is important to highlight here that these basic ICC mechanisms are local, in the sense that they can only be used to inter communicate components running in the same machine. The problem of inter communicating distributed components will be addressed later in section 3.7.

ICC mechanisms are classified depending on which side involved in a port connection executes each mechanism, either the sender or the receiver, corresponding respectively to the output port and the input port taking part in the connection. Besides, they are also classified depending on which of them takes the initiative in the communication. Hence, *sender side* mechanisms are defined as those mechanisms that the sender executes on its output ports (usually to send port packets), and *receiver side* mechanisms are those mechanisms carried out by the receiver on its input ports (normally to access incoming port packets).

There are six sender side mechanisms: *active sending* (AS), *active sending with copy* (ASC), *passive sending* (PS), *signal sending* (SS), *sender shared writing* (SSW) and *sender shared reading* (SSR); and five receiver side mechanisms: *passive reception* (PR), *active reception* (AR), *signal reception* (SR), *receiver shared reading* (RSR), and *receiver shared writing* (RSW).

Figure 3.15 shows which pairs of sender side and receiver side ICC mechanisms are valid, note that whatever other combination is not possible. Output ports on the sender side and input ports on the receiver side use these mechanisms to inter communicate component through connections. Next, in the following subsections all these mechanisms will be described in detail.

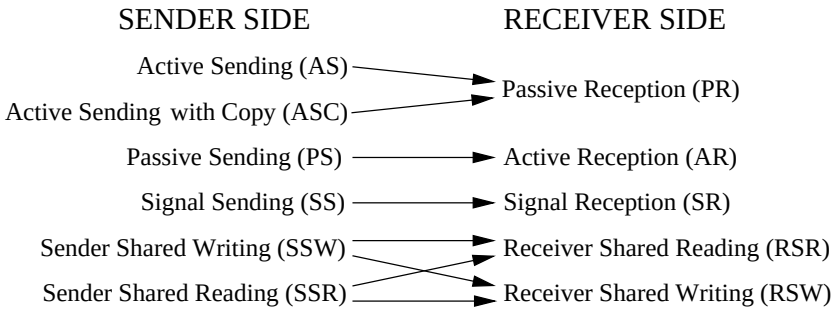

Figure 3.15: ICC mechanism pairs.

3.5.1.1 Active Sending (AS), Active Sending with Copy (ASC) and Passive Reception (PR)

Figure 3.16 illustrates the *active sending* (AS) mechanism by means of which, a component sends a packet through one of its output ports. To do so the output port must take part into a port connection, this is indicated on the figure with the arrow labelled *"port connection"*. The whole mechanism consists of three sequential steps:

1. *Acquisition*: To send data to the other end of a port connection it is necessary

to ask the output port for a port packet, labelled *"output packet"* in the figure, the sender gets a reference to it.

2. *Data Writing*: Next, the sender writes data in the output packet by means of the acquired reference.

3. *Sending*: The sender swaps the output packet with an internal packet in the input port at the other end of the connection, and change the state of the input port to *signaled*.

Figure 3.16 illustrates the active sending mechanism when the output port forms only one connection with an input port, but it is very common that an output port takes part into several port connections. In this case, the mechanism adds a new step, shown in figure 3.17, between the second and the third steps. Let us suppose that the port is involved in n port connections, then, once the second step of writing data has been finished, it is carried out a third sending step for each one of the connections. Specifically for each one of the first $n-1$ connections, the sending with copy step of figure 3.17 is added. Finally, for the last connection, the third step of sending with swapping of figure 3.16 is carried out. For this case, the ICC mechanism is called *active sending with copy* (*ASC*).

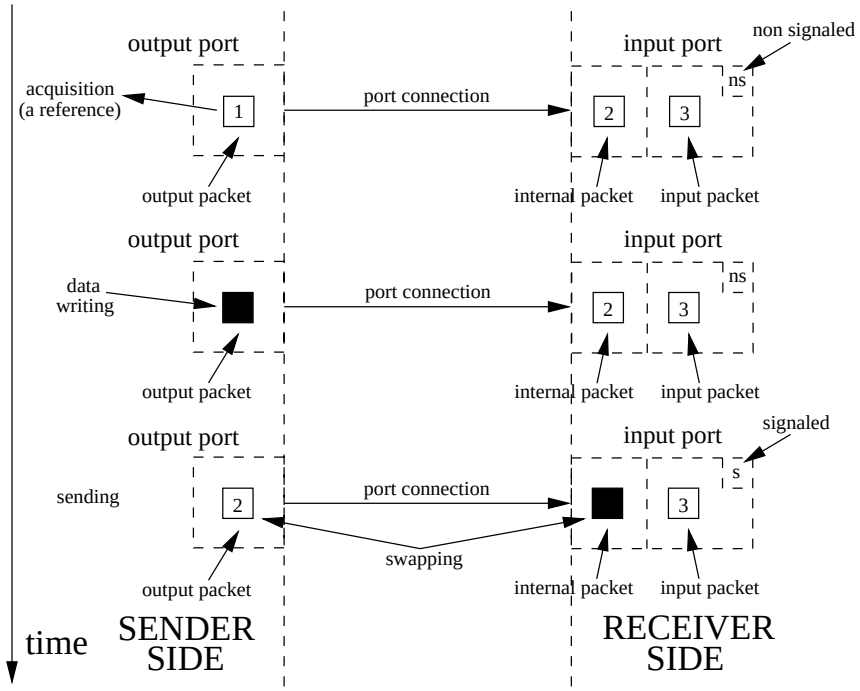

Figure 3.16: Active sending (AS).

All sending steps of figures 3.16 and 3.17 are atomic, in the sense that the sender is accessing information which is shared with the receiver. Keep in mind that the sender is responsible for doing the packet swapping or copying, and the input port signaling. These operations are protected internally by synchronization objects,

critical sections and *events* in Win32 [MSDN, 2002] [Richter, 1997], and *mutexes* and *conditions variables* in POSIX [Nichols et al., 1996], due to the fact that the input port may be involved in several port connections, and therefore, there may be several senders doing active sendings on it. Additionally, the receiver may be trying to access the port to read an incoming packet. Concerning synchronization costs, packet copy in figure 3.17 is the most costly one. Note that packet swapping in figure 3.16 has a very low cost in computational terms, it is just a swap of pointers or references.

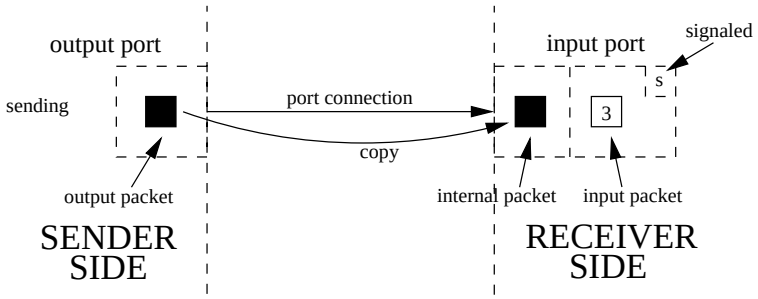

Figure 3.17: Active sending with copy (ASC).

It is said that these two ICC mechanisms (*AS* and *ASC*) are *"active"*, because in both cases the sender is responsible for transferring the port packet from its scope to the scope of the receiver. That implies that the sender usually makes the most costly operation, which happens when it has to make a copy of the output packet in the internal packet of the input port at the other end of the connection, as shown in figure 3.17. Notice that this situation occurs whenever the output port is involved in more than one port connection, what requires making a copy for each connection except for the last one. All in all, *active sending* and *active sending with copy* are sender side mechanisms that allow a sender, by means of a port connection, to situate a packet in the space of the receiver. Thus, it is said that the sender is the *"active"* entity.

Figure 3.15 indicates that the receiver side ICC mechanism corresponding to sender side mechanisms *active sending* and *active sending with copy* is *passive reception* (*PR*). In figure 3.18 appears the sequence of steps that conforms a *passive reception*. It comprises three steps:

1. *Waiting or Testing*: In this step the component asks for the state of the input port, whether *signaled or not*. To do so it may act in two ways, either waiting or testing:

   - *Waiting*: Corresponds usually to the situation in the inner loop of the component kernel of figure 3.14, when a component, the receiver, blocks waiting for an incoming port packet through any of its input ports. Note that, in this way, the component can be blocked waiting in case of the port has not been signaled.
   - *Testing*: The component just tests if the input port is signaled. In case of not being signaled the component does not get blocked, it can continue execution, so it is a non-blocking operation.

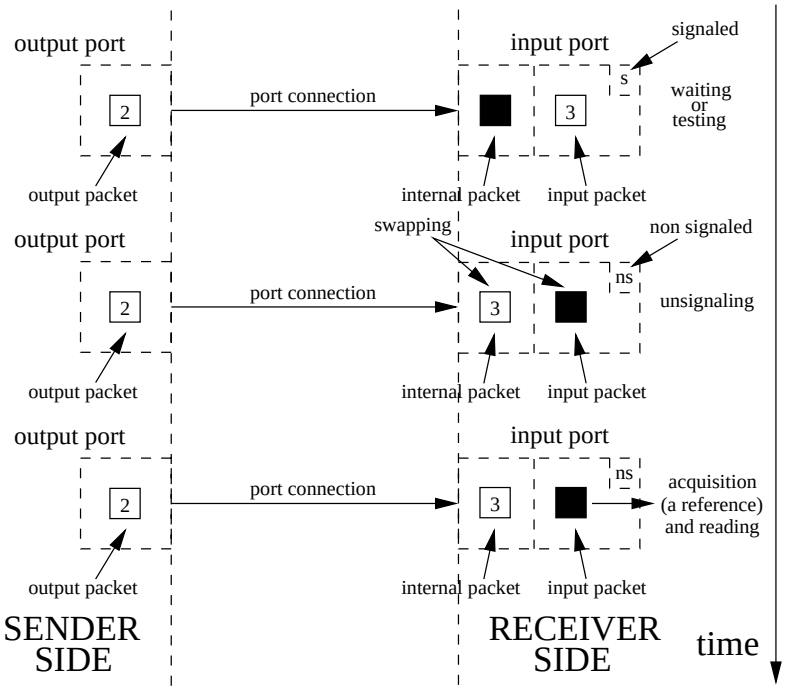

Figure 3.18: Passive reception (PR).

Signaling of the input port by means of any of the *active sending* mechanisms implies that the input port would end up as the first situation shown in figure 3.18, just after the sending steps of figures 3.16 and 3.17. A new packet has been swapped or copied in the internal packet, and the input port has been *signaled*.

2. *Unsignaling*: A waiting or testing operation on a *signaled* input port returns changing its state from *signaled* to *non signaled*. Atomically the *passive reception* mechanism swaps the internal packet and the input packet as well. Thus, in this way the data packet transferred by the sender will be now available for the receiver in the input packet.

3. *Acquisition and Reading*: Finally, to access the input packet, the receiver acquires a reference and reads it.

Note that the use of double buffering in the input port, using two packets, the internal packet and the input packet, minimizes synchronization when *active sending* mechanisms and *passive sending* are simultaneous, due to the fact that the critical section is only hold along the duration of a swapping of pointers or references. Once both packets have been swapped, the receiver can freely access to the information without worrying about synchronization issues.

The *passive reception* ICC mechanism is said to be *passive*, because once the receiver wakes up, it has the transferred information on its scope. In this way, just swapping packets, it can access the incoming packet. It can keep using that packet until it blocks to wait for another packet, and gets waken up again. Take into account

that keeping separated, by means of double buffering, the packet that is written (or swapped) by senders from the packet that is read by the receiver, uncouples senders from receivers making their internal working completely asynchronous.

Lastly, it is important to highlight that the combination of *active sending* mechanisms with *passive sending* allows for a type of interaction among components where senders, the *producers*, have the initiative in the communication, they are the *active* entities that complete the transferring of information into the scope of receivers, the *consumers*, which are the passive counterparts in this combination of ICC mechanisms.

3.5.1.2 Passive Sending (PS) and Active Reception (AR)

The combination of a *passive* sender with an *active* receiver to inter communicate two components is achieved with the combination of another two ICC mechanisms: *passive sending* and *active reception*, displayed in figures 3.19 and 3.20 respectively.

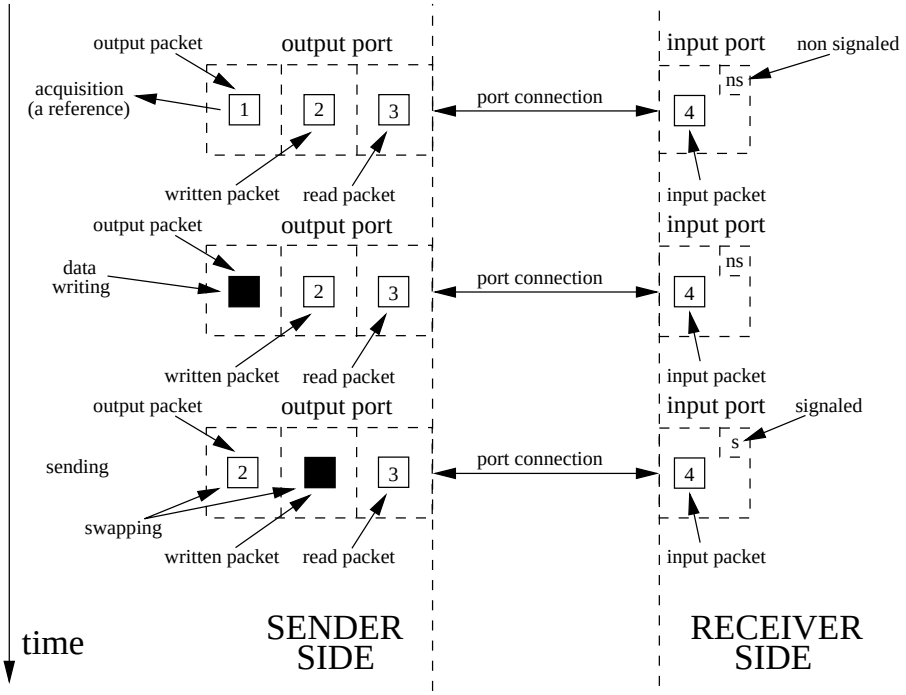

Figure 3.19: Passive sending (PS).

Similarly to *active sending* and *active sending with copy*, the ICC mechanism *passive sending* (*PS*) shown in figure 3.19 consists of three sequential steps:

- *Acquisition*: The sender asks the output port for an output packet to place the information that will be sent through the port connections where the output port takes part into. The sender gets a reference to the packet.
- *Data Writing*: Next, the sender writes the information that must be transmitted using the reference previously acquired.

- *Sending*: The sending of the packet consists of two actions:

  1. Firstly, the swapping of the output packet with an internal packet, labelled *"written packet"*, inside the output port where the information to be sent will be kept.
  2. Secondly, all input ports connected to the output port will be *signaled* to indicate that the sender has a new packet ready. Observe that the packet is still held in sender scope inside the output port.

Figure 3.20 displays the sequential steps that make up the *active reception* (*AR*) ICC mechanism. Analogously to the *passive reception* mechanism, it is formed by three steps:

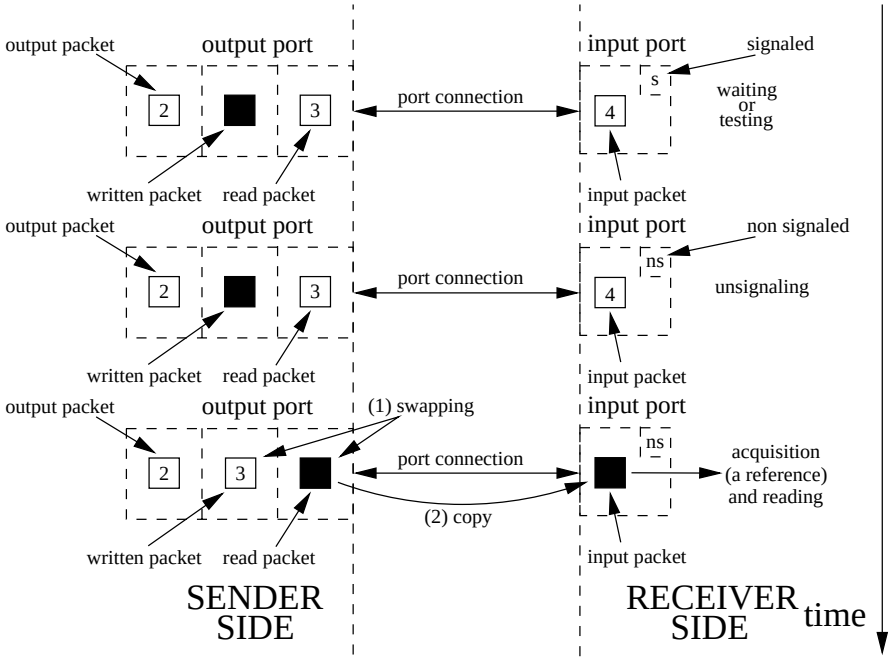

Figure 3.20: Active reception (AR).

- *Waiting or Testing*: Similarly to the waiting and testing steps of a *passive reception* in figure 3.18, an *active reception* can perform a waiting or a testing operation on the input port to find out whether it is *signaled* or not. Just at the moment of having the input port signaled by a *passive sending*, the last situation illustrated in figure 3.19, the input port changes and ends up as it is shown in figure 3.20.

- *Unsignaling*: The unsignaling just returning from a waiting or a testing operation on the input port, consists only in the change of the input port´s state from *signaled* to *non signaled*.

- *Acquisition and Reading*: Whenever a *passive sending* mechanisms is applied on the input port, the input port registers internally that its internal packet, labelled *"input packet"* in the figure is updated with respect to its counterpart at the other end of the connection, the packet labelled *"read packet"* in the output port. In case that the input packet has not yet been updated since the last port signalization, the receiver carries out the following actions in the output port:

  1. *Swapping*: Whether the *"read"* packet has not been updated with the last *"written"* packet, modified by the last *passive sending*, both packets get swapped by the receiver. Otherwise, if the *"read"* packet is up to date, nothing is done.
  2. *Copying*: The receiver makes a copy of the *"read"* packet into the input packet of the input port, at the other end of the connection, getting in this way the information emitted by the sender.

  Once the information is available as an input packet, the receiver acquires a reference to it, and reads the information.

  In case of having an up-to-date input packet in the input port, actions in the output port would not be necessary, in this situation the receiver gets just a reference to the input packet and reads it.

Notice that to carry out an *active reception* the receiver may need to access the output port at the other end of the connection, to access the packet transferred by the sender; this explains why in figures 3.19 and 3.20 the arrows labelled *"port connection"* are bidirectional. Note also that a receiver may make use of the *active reception* mechanism without utilizing the steps of waiting or testing and unsignaling previously mentioned. The *active reception* mechanism can be used only applying the last step of acquisition and reading, observe that in this case, the receiver acts differently depending on if the internal packet in the input port is up-to-date or not.

The combination of *passive sending* and *active reception* mechanisms has been devised to allow a sender to send information to multiple receivers with minimum synchronization costs. That is mainly the reason of using triple buffering in the sender side as it can be observed on figures 3.19 and 3.20. Using triple buffering the only critical section that the sender must go through is the swapping of the output and *"written"* packets in the output port during the sending step. It does not matter if simultaneously, there are readers accessing the port and copying the *"read"* packet. The sender does not need to wait blocked on a critical section, or on a mutex while there are receivers reading the output port, because the writer section, the *"written"* packet, is separated from the readers section, the *"read"* packet.

Thus, *passive sending* implies minimum synchronization costs for the sender, on the contrary, the receivers connected to the output port are responsible for transferring the information to their own scopes when they decide to do it, by means of *active receptions*. Not only they have to update their input packets when necessary, but they have also to update the *"read"* packet in the output port when it is needed as well.

For this reason, in this combination of ICC mechanisms it is said that the receiver is *active*. Receivers do the most costly work in terms of synchronization, swapping and copying of packets, if needed. Anyway, observe that using this combination of ICC mechanisms, copies are minimized and controlled transparently in a attempt to follow faithfully the *cache* motto.

3.5.1.3 Signal Sending (SS) and Signal Reception (SR)

Signal sending (*SS*) and *signal reception* (*SR*) constitute the combination of ICC mechanisms that carries out the simplest form of interaction between components, just the occurrence or signaling of an event.

The *signal sending* mechanism is depicted in figure 3.21, it consists of only one step:

- *Signaling*: Whenever the sender wants to communicate to other components the occurrence of an event, it can do so by means of signaling the output port. Signaling the output port means that all input ports sharing a connection with the signaled output port will change their state to *signaled* as well, as it is shown in figure 3.21

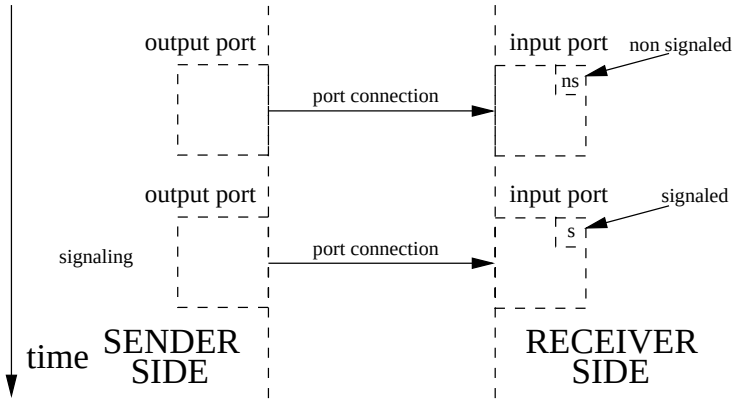

Figure 3.21: Signal Sending (SS).

The complementary ICC mechanism in the receiver side for a *signal sending* is a *signal reception*. Figure 3.22 displays how the *signal reception* ICC mechanism comprises two steps:

- *Waiting or Testing*: This step, similar to that of previous reception mechanisms of figures 3.18 and 3.20, occurs when the receiver is carrying out a waiting or a testing operation on the input port to know if it is *signaled* or not. As depicted in figure 3.22, after a *signal reception* mechanism the port ends up in a *signaled* state.

- *Unsignaling*: Once an input port is *signaled* by a *signal sending*, any waiting or testing operation on it will change its state from *signaled* to *non signaled* again.

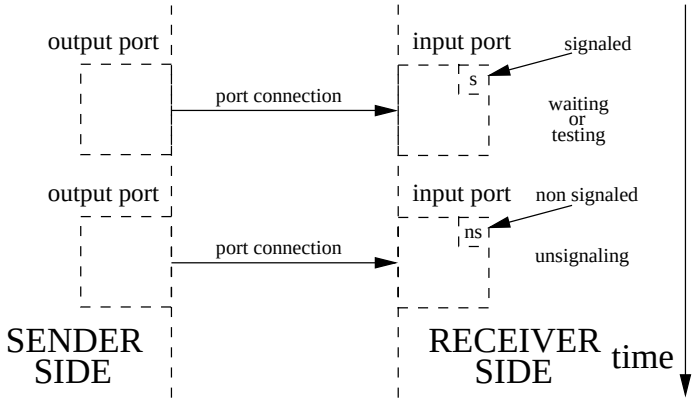

Figure 3.22: Signal Reception (SR).

3.5.1.4 Shared ICC Mechanisms

According to figure 3.15 there are four *"shared"* ICC mechanisms: *sender shared writing* (*SSW*), *receiver shared reading* (*RSR*), *sender shared reading* (*SSR*) and *receiver shared writing* (*RSW*). These mechanisms implement the model of shared memories for processes and threads utilized in multiple operating systems.

In CoolBOT an object which is shared by several components resides in an output port in the form of a port packet called *shared packet* as figures 3.23, 3.24, 3.25 and 3.26 illustrate. The *shared packet*, besides of being a port packet, supports a set of writing and reading operations which are user-defined. In general, objects shared between components using these ICC mechanisms are referred to as *shared packets*, and each type has its own writing and reading operations which are defined during its design by component developers/users.

The four *shared* ICC mechanisms CoolBOT provides guarantees that given a *shared packet*, multiple reading operations can be carried out on it simultaneously, whenever these readings operations belong to the set of reading operations the port packet accepts. On the contrary, any writing operation constitutes a critical section, so the mechanisms prevent the execution of any other operation, whether reading or writing, at the same time. All writing and reading operations must belong to the sets of writing and reading operations the *shared packet* has been designed to accept. In the following each one of these mechanisms is presented.

Figure 3.23 shows the sender side mechanism *sender shared writing* (*SSW*), it consists of two steps:

- *Acquisition*: As in other mechanisms already studied, in this step the sender just asks for a reference, in this case a reference to the shared packet stored internally in one of its output ports.
- *Writing*: This is the realization of a writing operation on the shared packet by means of the previously acquired reference. It is synchronized in such way that it prevents other readers or writers to do any operation on the packet. At the

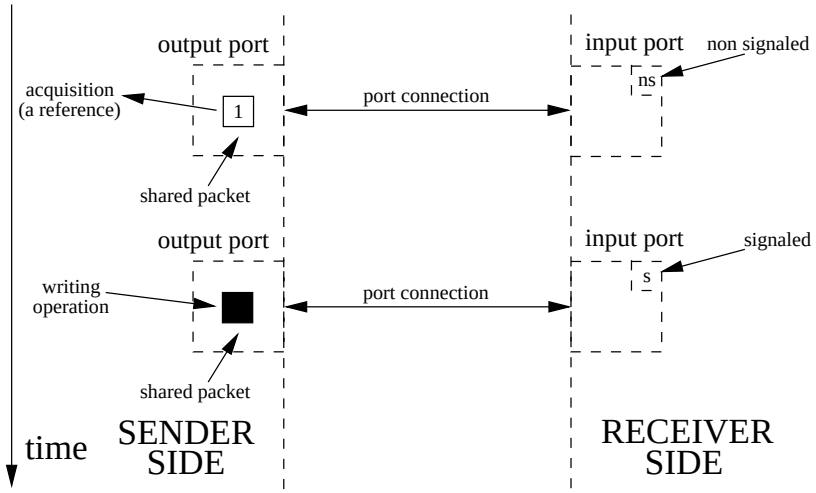

Figure 3.23: Sender shared writing (SSW).

same time all input ports connected to the output port will be signaled.

When an input port is signaled by way of a *sender shared writing*, besides of being signaled, it stores which writing operation was carried out on the shared packet at the other end of the connection. The aim of signaling input ports in figure 3.23 is to provide receivers with means to know when the shared packet is updated or written, and which written operation was done on them.

Receivers can read the shared packet by means of connected input ports using the *receiver shared reading* (*RSR*) ICC mechanism displayed in figure 3.24. The steps that usually conform this mechanism are explained as follows:

- *Waiting or Testing*: Similarly to the previous mechanisms *passive reception*, *active reception* and *signal reception*, a waiting or a testing operation permits to know whether an input port is *signaled* or not. The figure 3.24 illustrates the situation just after a writing operation has been carried out on the shared packet at the other end of the port connection, and the input port remains, thus, *signaled*.

- *Unsignaling*: Just returning from a waiting or testing operation in a *signaled* input port, the port is set again to *non signaled* state. Besides, the receiver can also retrieve information about which writing operation originated the signaling.

- *Reading*: The last step consists in doing a reading operation on the shared packet at the output port. It can be simultaneous with other readings realized by other components.

The first two steps can be avoided if the receiver is not interested in blocking and waiting until the shared packet gets written. The mechanism of figure 3.24 is the typical one when the receiver does not want to do polling and waste CPU time unnecessarily. The input port keeps internally (the port connection) a reference to the output port keeping the shared packet, so it can do a reading operation when needed.

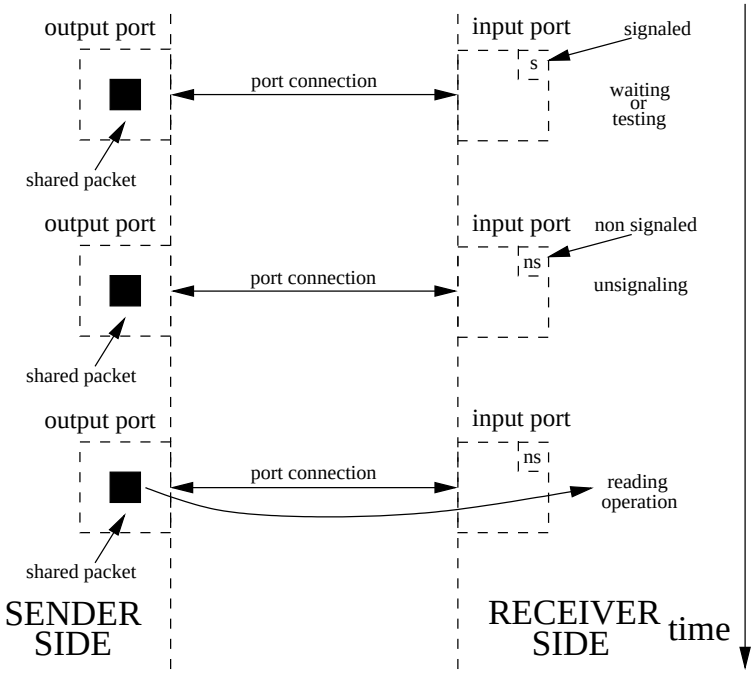

Figure 3.24: Receiver shared reading (RSR).

A sender can also do a reading operation on the shared packet in its own output port using the *sender shared reading* (*SSR*) ICC mechanism displayed in figure 3.25. Bear in mind that the shared packet is inside the output port, and even the sender can not access it freely. It must synchronize its access, since there may be other writers accessing it. The *sender shared reading* ICC mechanism allows for this situation. *Sender shared reading* is divided in two sequential steps:

- *Acquisition*: The component asks for a reference to the shared packet of the output port, and acquires it.
- *Reading*: In this state, the component carries out a reading operation using the reference.

Similarly to senders, receivers can perform writing operations on the shared packet of an output port to which they can be connected, they can do that using the *receiver shared writing* (*RSW*) ICC mechanism of figure 3.26. It consists of two steps:

- *Acquisition*: The component asks for a reference to the shared packet of the output port through the port connection.
- *Writing*: In this state, the component carries out the writing operation using the reference. As with *sender shared writing* input ports connected to the output port with the shared packet are signaled storing which writing operation was exerted on it as well.

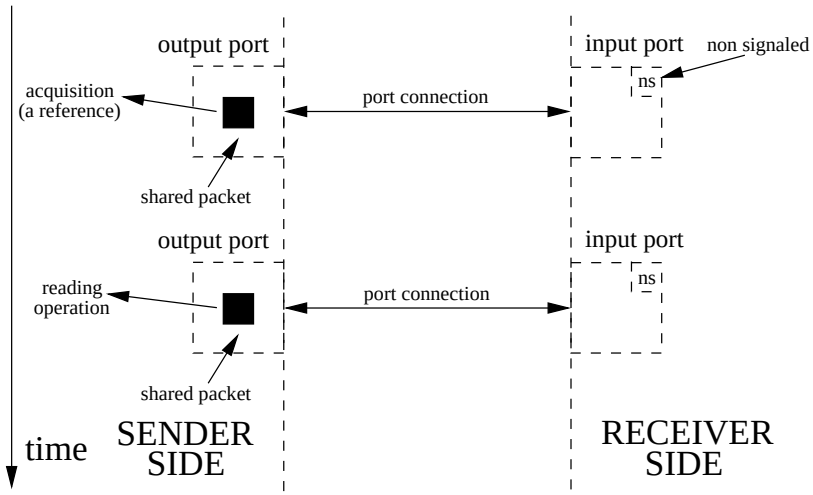

Figure 3.25: Sender shared reading (SSR).

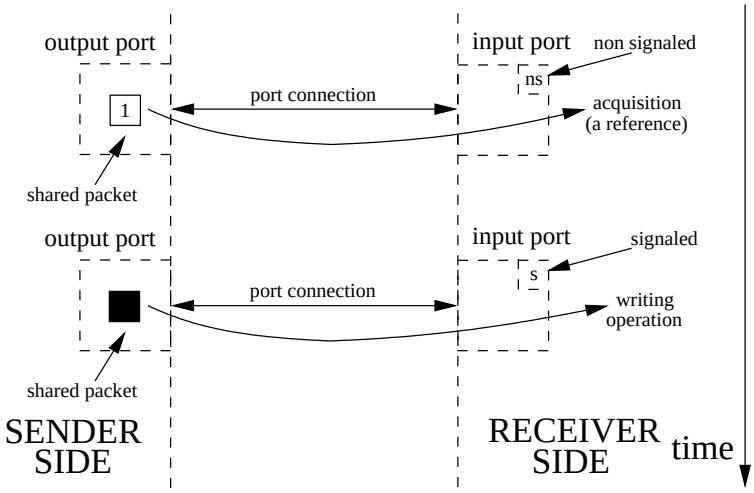

Figure 3.26: Receiver shared writing (RSW).

In general, a sender should be a writer and receivers only readers. The *sender shared writing* and *receiver shared reading* ICC mechanisms should be used in this case. But, in some situations when components share complex and large data structures which may be accessed indistinctly for writing and reading, it might be convenient to share memory in a way where the difference between writers and readers it is not so clear. It is for these situations that, these last two mechanisms, *sender shared reading* and *receiver shared writing*, have been added. All in all, these four shared ICC mechanisms (SSW, RSR, SSR and RSW) permit components to share memory by means of port connections, where any of them can be a writer or a reader, and above all, without the necessity of worrying about synchronization issues to do so.

3.5.2 Port Connections

According to what has been presented previously, CoolBOT components inter communicate by means of *port connections* formed by *output ports* and *input ports*. Remember that a *port connection* between an output port and an input port is only possible whether both ports match the type of port packets they accept. Besides, it is necessary that they also use internally valid combinations of sender side and receiver side ICC mechanisms (figure 3.15).

Output Ports		
Name	Symbol	Basic ICC Mechanisms
OTick	$\xrightarrow{t}$	SS
OGeneric	$\xrightarrow{g}$	AS, ASC
OPoster	$\xrightarrow{p}$	PS
OMultiPacket	$\xrightarrow{mp}$	AS, ASC
OLazyMultiPacket	$\xrightarrow{lmp}$	AS, ASC
OPriority	$\xrightarrow{pr}$	AS, ASC
OShared	$\xrightarrow{s}$	SSW, SSR
OPull	$\xrightarrow{pu}$	AS, ASC, PR

Table 3.3: Output port types.

More specifically, output ports and input ports will be compatible depending on what internal structure they have, and on which type of ICC mechanisms they use internally. Table 3.3 enumerates the eight types of output ports supported by CoolBOT, their symbology and the sender side ICC mechanisms each one uses internally (observe that the *OPull* output port uses also one receiver side ICC mechanism – PR). There are nine types of input ports as well. Table 3.4 resumes them, their symbology and the receiver side ICC mechanisms they utilize (notice that the *IPull* input port uses also sender side ICC mechanisms – AS and ASC).

Figures 3.27 and 3.28 show the different types of port connections supported by CoolBOT according to their type, given that the ports involved match the port packets they accept. In this figure it is also indicated below the arrows, the cardinality of each type of port connection. Next paragraphs will explain each one in more detail.

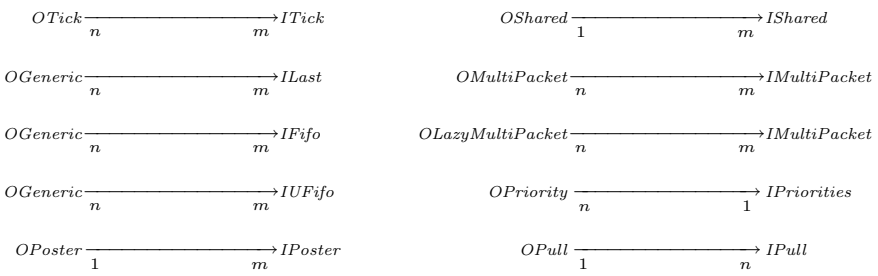

Figure 3.27: Port connections (n,$m \in \mathbb{N}; n,m \geq 1$).

Input Ports		
Name	Symbol	Basic ICC Mechanisms
ITick	$\xrightarrow{t}$	SR
ILast	$\xrightarrow{l}$	PR
IFifo	$\xrightarrow{f}$	PR
IUFifo	$\xrightarrow{uf}$	PR
IPoster	$\xrightarrow{p}$	AR
IMultiPacket	$\xrightarrow{mp}$	PR
IPriorities	$\xrightarrow{pr}$	PS
IShared	$\xrightarrow{s}$	RSW, RSR
IPull	$\xrightarrow{pu}$	PR, AS, ASC

Table 3.4: Input port types.

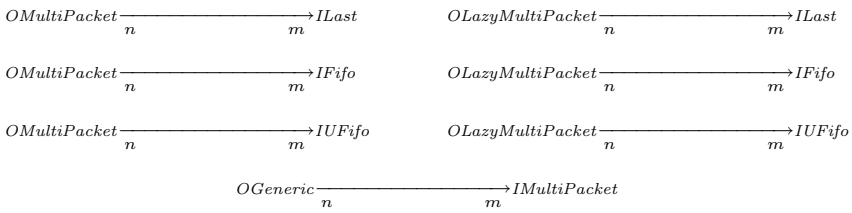

Figure 3.28: Simple multi packet connections (n,$m \in \mathbb{N}; n$,$m \geq 1$).

3.5.2.1 Tick Connections

Definition. A *tick* connection is an output port/input port pair formed by a *tick* output port (*OTick*) and a *tick* input port (*ITick*). Its internal structure appears in figure 3.29.

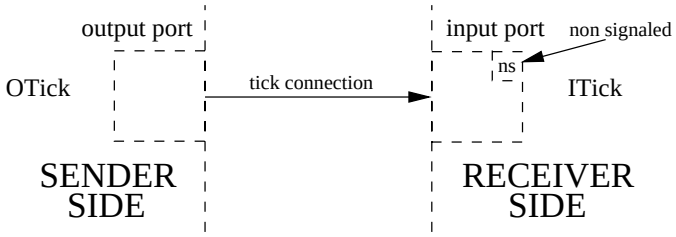

Figure 3.29: A *tick* connection.

Basic ICC Mechanisms. Observing tables 3.3 and 3.4 note that the types of ports involved in *tick* connections, *ITick* and *OTick*, use respectively *signal sending* (SS) and *signal reception* (SR).

Port Packet Types. *Tick* connections do not transport port packets.

Cardinality. Notice the cardinality of *tick* connections in figure 3.27, *tick* output ports and *tick* input ports can be involved in multiple tick connections.

Functionality. These are the simplest port connections provided by CoolBOT. *Tick* connections are mainly utilized to communicate the occurrence of events between components.

Signaling/Unsignaling. As displayed in figure 3.21, a *tick* input port, *ITick* gets signaled when a SS is applied on it. Once signaled, the input port keeps this state until a testing or waiting operation is exerted on it, as in a SR in figure 3.22.

3.5.2.2 Last Connections

Definition. A *last* connection is an output port/input port pair formed by a *generic* output port (*OGeneric*) and a *last* input port (*ILast*). Figure 3.30 depicts the internal structure of the port types that make up *last* connections.

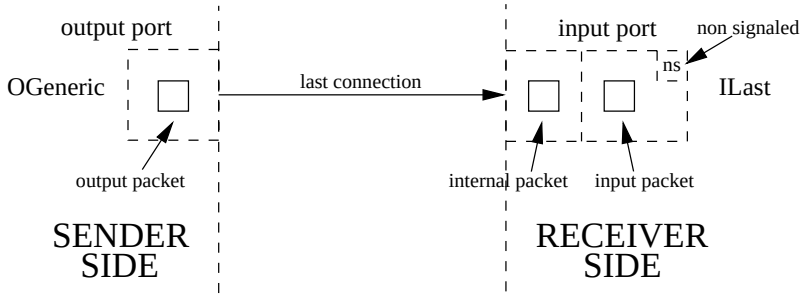

Figure 3.30: A *last* connection.

Basic ICC Mechanisms. The types involved in this type of port connections, *OGeneric* and *ILast*, use the ICC mechanisms of *active sending* (AS) or *active sending with copy* (ASC), and *passive reception* (PR) respectively, as shown in tables 3.3 and 3.4.

Port Packet Types. Ports involved in *last* connections, *OGeneric* and *ILast*, should match the type of port packet they transmit, each one may only be associated with one type of port packet.

Cardinality. Figure 3.27 illustrates clearly that the cardinality of this kind of connections is $n \rightarrow m$, so *OGeneric* and *Ilast* ports can be involved simultaneously in several *last* connections.

Functionality. *Last* connections constitute the simplest type of port connections that utilize AS, ASC and PR, since that the input and output ports that conforms them use the minimal internal structure necessary to implement such ICC mechanisms, as figures 3.16, 3.17 and 3.18 depict.

Signaling/Unsignaling. An *Ilast* input port is signaled when a sender exerts on it an active sending mechanism, either an AS or an ASC. It will hold its signaled state until a PR is applied on it, what would imply a testing or waiting operation which will change its state to *non signaled*.

3.5.2.3 FIFO Connections

Definition. A *fifo* connection is an output port/input port pair formed by a *generic* output port (*OGeneric*) and a *fifo* input port (*IFifo*). Figure 3.31 displays the internal structure of the port types that conform *fifo* connections.

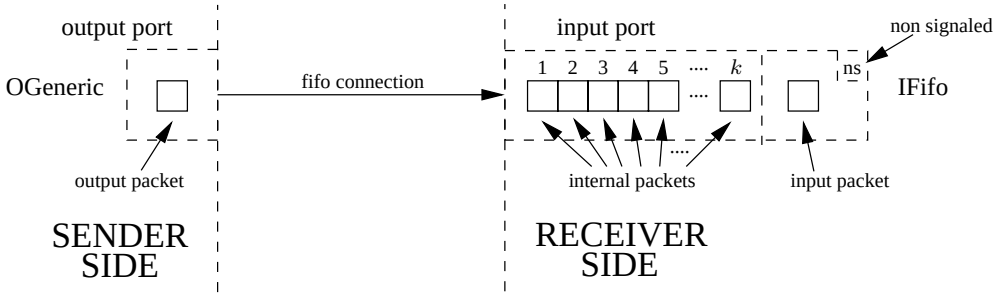

Figure 3.31: A *fifo* connection.

Basic ICC Mechanisms. According to tables 3.3 and 3.4 *generic* output ports and *fifo* input ports, *OGeneric* and *IFifo*, use *active sending* (AS), *active sending with copy* (ASC) and *passive reception* (PR) respectively to transfer port packets through the *fifo* connection. They use the same basic ICC mechanisms than *last* connections.

Port Packet Types. Likely *last* connections, ports taking part into *fifo* connections, *OGeneric* and *IFifo*, may be only associated with one type of port packet, and types admitted by both should match to form a connection.

Cardinality. The cardinality is $n \rightarrow m$ according to figure 3.27.

Functionality. *Fifo* input ports (*IFifo*) have several internal packets as appears in figure 3.31. In fact, it uses an array of internal packets that constitutes a **first-in-first-out** (**fifo**) structure or **queue** with a specific length (k in the figure). Concretely, an *active sending*, an AS or an ASC, means an insertion of a packet in the internal fifo. Similarly, a *passive reception*, an PR, signifies an extraction from the queue.

So, any *active sending* exerted on the *fifo* input port will be applied to the next available packet in the queue, if any. If the fifo is full, then the *active sending* will utilize the oldest packet, overwriting or swapping it with the new one. In the same way, any *passive reception* applied to it will extract the front packet, the oldest one, from the internal fifo. Namely, a *passive reception* will exchange the port´s input packet and the front packet (the oldest inserted one) of the internal fifo.

Signaling/Unsignaling. A *fifo* input port, IFifo, will be *non signaled* if its internal queue is empty, and will be *signaled* if this queue is not empty. In this way, a *non signaled fifo* input port will change its state to *signaled*, if any packet is inserted in it by means of the application of an AS or an ASC on it. In the same way, a *signaled fifo* input port will become *non signaled* when it gets empty by the application of some PRs on it.

3.5.2.4 Unbounded FIFO Connections

Definition. An *unbounded fifo* connection is an output port/input port pair formed by a *generic* output port (*OGeneric*) and an *unbounded fifo* input port (*IUFifo*). Internally, an *unbounded fifo* connection has the same internal structure that *fifo* connections, as it appears in figure 3.31

Basic ICC Mechanisms. Tables 3.3 and 3.4 indicate that ports involved in *unbounded fifo* connections use the same basic ICC mechanisms that *last* and *fifo* connections.

Port Packet Types. As with ports forming *last* and *fifo* connections, ports taking part into this type of connections, *OGeneric* and *IUFifo*, only admit one type of port packet.

Cardinality. They have the same cardinality that *last* and *fifo* connections.

Functionality. According to figure 3.31, *unbounded fifo* connections have the same structure that *fifo* connections. The only difference is that the length of the input port´s internal fifo grows when it is full, and any insertion, an AS or an ASC, is applied on it.

Signaling/Unsignaling. *IUFifo* input ports get *signaled* and *non signaled* in the same conditions that *IFifo* input ports.

Supplementary Comments. It is interesting to note that *last*, *fifo* and *unbounded fifo* connections are essentially equivalent in terms of functionality, because they differs only in the length of the internal fifo that is hold in the receiver side of the connection. Thus, *last* connections keep a fifo of packets of length 1, for *fifo* connections the fifo has a specific length k, and *unbounded fifo* connections allows a length which is variable. Therefore they all are *fifo* connections, the reason of having three specialized types is mainly due to performance reasons.

Finally, it is important to observe that the cardinality of these three types of connections is $n \rightarrow m$ (have a look at figure 3.27). Hence, *OGeneric*, *ILast*, *IFifo* and *IUFifo* ports may be involved in multiple connections simultaneously. It is also significant that *OGeneric* output ports can form simultaneously *last*, *fifo* and *unbounded fifo* connections.

3.5.2.5 Poster Connections

Definition. A *poster* connection is an output port/input port pair formed by a *poster* output port (*OPoster*) and an *poster* input port (*IPoster*). The internal structure of *poster* connections appears in figure 3.32.

Basic ICC Mechanisms. Observe in figure 3.32 that triple buffering is used in the sender side, the *poster* output port (OPoster); this is due to the types of ports involved in this kind of connections, *OPoster* and *IPoster*, that use respectively the *passive sending* (PS) and *active reception* (AR) ICC mechanisms illustrated in figures 3.19 and 3.20.

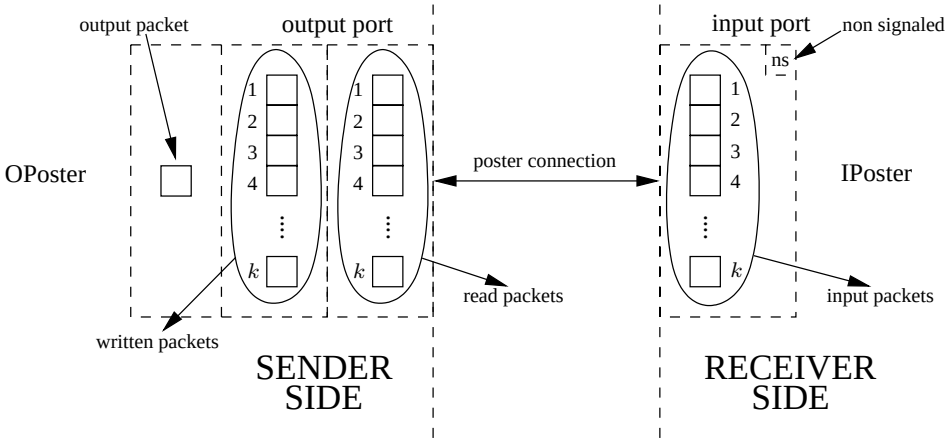

Figure 3.32: A *poster* connection.

Port Packet Types. *Poster* ports, *OPoster* and *IPoster*, may be only associated with one type of port packet. Like previous types of connections (except *tick* connections), they can form a *poster* connection if the port packet types they transmit match each other.

Cardinality. The cardinality is $1 \rightarrow m$, so *poster* input ports, *IPoster*, can only be involved in one *poster* connection. However, *OPoster* ports can form multiple connections simultaneously.

Functionality. As shown in figure 3.32, *poster* output and input ports are internally constituted by arrays of packets of a specific length (k in the figure). When PSs or ARs are applied on the connection, respectively in the sender or receiver sides, each ICC mechanism affects only one packet on each array, in other words, the mechanisms are applied in a per-index basis. Thus, a PS applied on the output packet i of the output port will affect packet i of the internal packets, and will signalize through the connection that the packet i has been updated. Similarly, an AR in the receiver side on the input packet i of the input port will affect only that packet, and its corresponding internal packet i at the other end of the connection.

Signaling/Unsignaling. A *poster* input port (IPoster) is *non signaled* whether all input packets in the array it keeps internally, are *non signaled*. The IPoster is *signaled* if any of these input packets is *signaled*. Remember that testing and waiting operations carried out along AR mechanisms, modify the state of the different packets from *signaled*, if any, to *non signaled*, and on the contrary, PS mechanisms make packets *signaled*.

Supplementary Comments. The rationale of *poster* connections is to allow for patterns of communication where there are one producer and one or more consumers (one sender and multiple receivers, cardinality $1 \rightarrow m$ in figure 3.27), and it is important to avoid high synchronization costs on the sender side. This type of port connection has been inspired by the *poster* construct that is used in G^{en}oM [Fleury et al., 1997] to inter communicate modules.

3.5.2.6 Shared Connections

Definition. A *shared* connection is an output port/input port pair formed by a *shared* output port (*OShared*) and a *shared* input port (*IShared*). Figure 3.33 displays the internal structure of this type of connections.

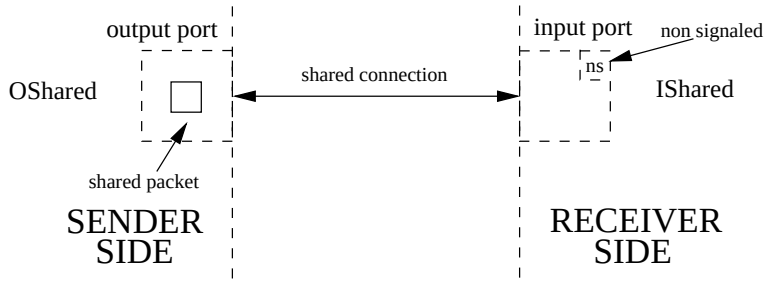

Figure 3.33: A *shared* connection.

Basic ICC Mechanisms. Tables 3.3 and 3.4 indicate that the basic ICC mechanisms used in *shared* connections are: *sender shared writing* (SSW), *receiver shared reading* (RSR), *sender shared reading* (SSR) and *receiver shared writing* (RSW). Remember from section 3.5.1.4 that the use of these ICC mechanisms implies that the *shared* output port taking part in the connection, contains the shared object. This object is a *shared packet*, a port packet that accepts by design a set of several writing and reading operations. Remember also that only these operations are guaranteed by the *shared* ICC mechanisms to be carried out without corruption of the shared object due to simultaneous accesses. Thus, using such mechanisms several writers and several readers can share the same information using *shared connections* involving the same output port.

Port Packet Types. *Shared* output and input ports, *OShared* and *IShared* admit each one only one type of port packet. Port packet types at both ends of a *shared* connection should match to form a valid connection.

Cardinality. *Shared* input ports, *IShared*, may take part only into one port connection. On the other side, *shared* output ports, *OShared*, may form multiple connections. This is the cardinality $1 \rightarrow m$ appearing in figure 3.27.

Functionality. The rationale of this type of connection and the basic ICC mechanisms that support it, is to provide a means to allow components to share memory, the *"shared packet"* of figures 3.23, 3.24, 3.25 and 3.26, under the abstraction of input and output ports.

Signaling/Unsignaling. A *shared* input port, *IShared*, gets *signaled* whenever a *shared writing* mechanism, SSW and RSW, is performed. The input port tracks internally the different writing operations realized on the output port to which it is connected, by means of an internal queue (not shown in figure 3.33). The input port remains *signaled* if this internal queue is not empty. The port becomes *non signaled* when it gets empty.

Each time a testing or waiting operation is carried out in the *IShared* port, an element is extracted from this signalization queue.

Supplementary Comments. Mind that *shared* connections have been devised to support the well known *"shared memory"* model of interaction between processes and threads, which is present in the IPC APIs of most modern operating systems. Thus, *shared* connections permit sharing memory (the *shared* packet of figures 3.23, 3.24, 3.25, 3.26, 3.33) between components in a transparent and operating-system-independent manner.

3.5.2.7 Multi Packet Connections

Definition. A *multi packet* connection is an output port/input port pair formed by a *multi packet* output port (*OMultiPacket*) and a *multi packet* input port (*IMultiPacket*). Its internal structure is shown in figure 3.34, observe that it has an array of output packets in the sender side at the output port, and two arrays of packets in the input port: an array of internal packets, and an array of input packets. All of them have the same length (k in the figure).

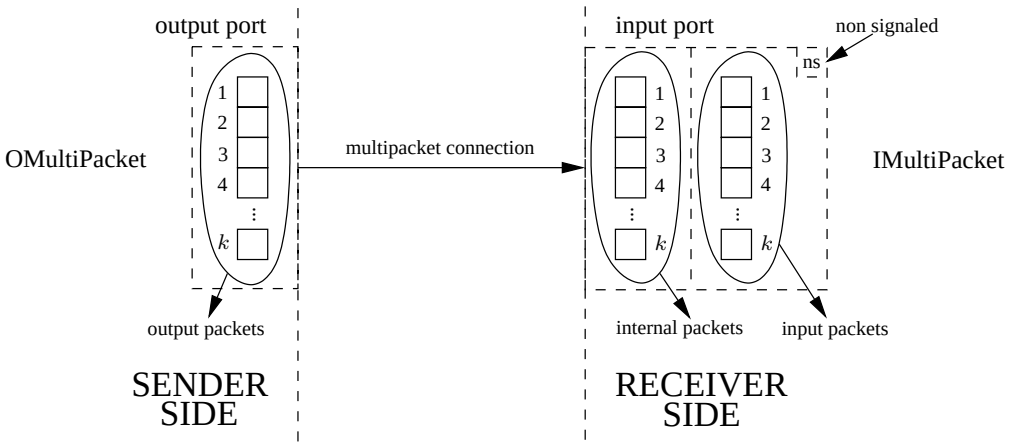

Figure 3.34: A *multi packet* connection.

Basic ICC Mechanisms. The basic ICC mechanism used by *multi packet* connections are *active sending* (AS), *active sending with copy* (ASC), and *passive reception* (PR), that is why double buffering is used in the receiver side, according to figure 3.34.

Port Packet Types. *Multi packet* connections may accept several types of port packets. Specifically, it can accept as many port packet types as elements each one of the arrays of packets involved in a connection has (k in figure 3.34). That is the main feature of this kind of port connection.

Cardinality. Cardinality, as in other previous connections, is $n \rightarrow m$. In this way, *multi packet* output and input ports may take part in multiple connections simultaneously.

Functionality. *Multi packet* can be seen as "big" *last* connections that pack together

a specific number of connections, k in figure 3.34. As with *poster* connections, the basic ICC mechanisms used in *multi packet* connections are applied internally in a per-index basis, affecting only the corresponding elements of a specific index in the arrays shown in figure 3.34 at both ends of the connection. The main utility of this type of connections is the possibility of sending different types of port packets through the same connection.

Signaling/Unsignaling. *Multi packet* input ports, *IMultiPacket* are *signaled* each time an AS or an ASC is applied on any of the packets of its internal array through any of the connections where the port is involved. It will remain *signaled* whenever an internal packet have been signaled, and no testing or waiting operation have been applied on it. Testing and waiting operations exerted on the input port, are carried out in a per-index basis. Returning from them will provoke the unsignaling of a specific internal packet. The port is *non signaled* if all its internal packets are *non signaled*.

Supplementary Comments. It is possible to connect individually specific elements of the output array of packets at the *OMultipacket* output port, to individual elements of the internal array of packets at the *IMultipacket* output port. The cardinality is also $n \rightarrow m$, as indicated in figure 3.27.

3.5.2.8 Lazy Multi Packet Connections

Definition. A *lazy multi packet* connection is an output port/input port pair formed by a *lazy multi packet* output port (*OLazyMultiPacket*) and a *multi packet* input port (*IMultiPacket*). It has the same internal structure that *multi packet* connections, as shown by figure 3.34. *Lazy multi packet* connections and *multi packet* connections differs exclusively in the type of output port that forms each one, *OLazyMultiPacket* and *OMultiPacket* respectively.

Basic ICC Mechanisms. *Lazy multi packet* connections use the same basic ICC mechanisms that *multi packet* connections: AS, ASC and PR (tables 3.3 and 3.4).

Port Packet Types. *Lazy multi packet* connections accept several types of port packets in the same terms that *multi packet connections*.

Cardinality. They also have the same cardinality that *multi packet* connections, $n \rightarrow m$ (figure 3.27).

Functionality. In terms of functionality, the only difference between *lazy multi packet* and *multi packet* connections is the behavior of the output port whose type is *OLazyMultiPacket*. An *OLazyMultiPacket* port does not carry out the sending of packets through the connection when ASs or ASCs are applied on it. It "accumulates" internally the different "sent" packets, and postpones their sending until a "flushing" operation is applied on the output port. This *flush* operation is specific to this type of output ports; thus, all sending steps of figures 3.16 and 3.17, corresponding to previous ASs or ASCs since the last *flush* operation, are delayed until a new *flush* operation is

exerted on the output port.

Signaling/Unsignaling. Obviously, due to the fact that the receiver side of this type of connections has the same type of input port that *multi packet* connections, *IMulti-Packet*, the situations to be *signaled* and *non signaled* are equivalent.

Supplementary Comments. As with *multi packet* connections, in *lazy multi packet* connections it is possible to connect individually any of the output packets at the *OLazy-Multipacket* output port, to any of the internal packets at the *IMultipacket* output port. In the same way, the cardinality is also $n \rightarrow m$, as appears in figure 3.27.

3.5.2.9 Priority Connections

Definition. A *priority* connection is an output port/input port pair formed by a *priority* output port (*OPriority*) and a *priority* input port (*IPriorities*). The internal structure of *priority* connections is depicted in figure 3.35.

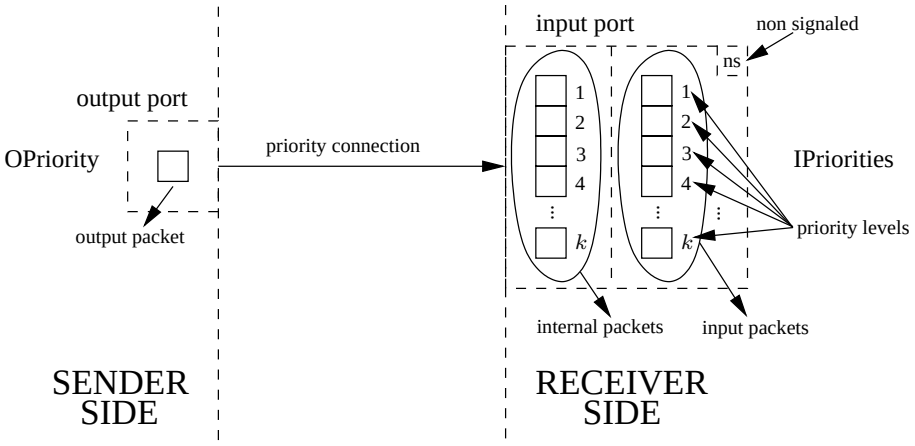

Figure 3.35: A *priority* connection.

Basic ICC Mechanisms. *Priority* connections make use of the basic ICC mechanisms: *active sending* (AS), *active sending with copy* (ASC), and *passive reception* (PR). Observe again the double buffering at the receiver side in figure 3.35.

Port Packet Types. Each part in a priority connection accepts only one type of port packet, and the types should match at both ends, the sender and the receiver sides, to form a valid *priority* connection.

Cardinality. *Priority* output ports, *OPriority*, can only take part into one connection, on the contrary, *priority* input ports, *IPriorities* may admit multiple connections simultaneously. This is the sense of cardinality $n \rightarrow 1$ in figure 3.27.

Functionality. *Priority* input ports, *IPriorities*, admit multiple connections at different levels of priority (k levels in figure 3.35). The number of priority levels is specific of each *IPriorities* port, and it is determined at instantiation time. *Priority* output ports,

OPriority, can connect to only one level of priority. Therefore, ASs and ASCs exerted on an *OPriority* will only affect the level of priority to which it is connected.

Signaling/Unsignaling. Any AS or ASC mechanism applied on an *OPriority* involved in a *priority* connection, will signalize the *IPriorities* port at the specific level of priority where the output port is connected. Each level of priority has associated a **time of persistence** that indicates how long a level of priority remains *signaled* at maximum if no new ASs or ASCs are applied on it. Furthermore, the level of priority can be masked and unmasked individually. Thus, the whole port will be *signaled* if any of its levels of priority which is not masked, is *signaled*. Similarly to other input ports, testing and waiting operations will turn *IPriorities* input ports *non signaled*. In fact, they remain *signaled* until all their levels of priority get *non signaled* by either the application of these operations, or by the expiration of the **persistence time** for all their levels of priority. Testing and waiting operations exerted on a IPriorities port return always indicating the highest priority which was *signaled*.

3.5.2.10 Pull Connections

In CoolBOT there are two basic communication models for port connections:

- *Push Model.* In a push connection the initiative for sending a port packet relies on the output port part; that is, the data producer (the sender) sends port packets on its own, completely uncoupled from its consumers (the receivers).
- *Pull Model.* A pull connection implies that packets are emitted when the input part of the communication, the consumer (the receiver), demands new data to process. In this model the consumer keeps the initiative, sending a request to the producer (the sender) whenever a new port packet is demanded.

All connections that have been introduced so far, observe the push communication model, thus, all of them constitute push connections, and they allow uncoupled interaction of components. However, in multiple cases, it is necessary a pull communication model. CoolBOT provides a pair of output/input ports, named *pull* connections, that allows for this kind of interaction.

Definition. A *pull* connection is an output port/input port pair formed by a *pull* output port (*OPull*) and a *pull* input port (*IPull*). The internal structure of *pull* connections is illustrated in figure 3.36.

Basic ICC Mechanisms. *Active sending* (AS), *active sending with copy* (ASC) and *passive reception* (PR) are the basic ICC mechanisms used by *pull* connections. These mechanisms are utilized to communicate the components in both directions, from the receiver to the sender for the requests, and from the sender to the receiver for the answers. That is the reason of having double buffering at both sides of *pull* connections, as it is shown in figure 3.36.

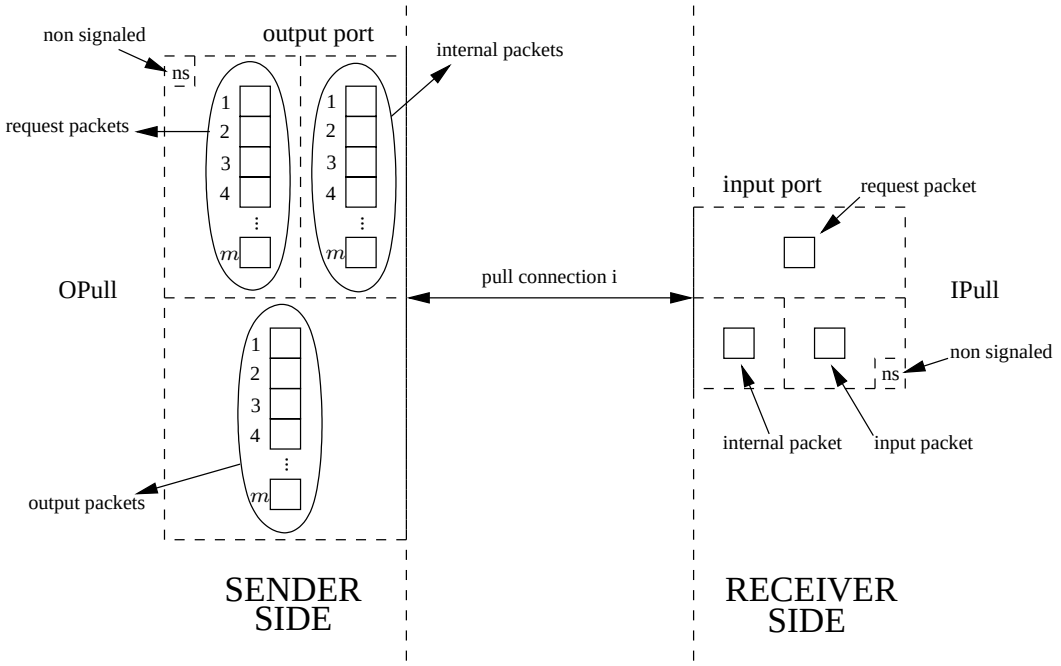

Figure 3.36: A *pull* connection.

Port Packet Types. *Pull* connections distinguish two types of port packets: one type for request packets, and another type for the answers, both types should match at both ends of a connection to establish a valid *pull* connection.

Cardinality. *Pull* output ports, *OPull*, can take part into multiple *pull* connections. In fact, corresponding items on each one of the arrays in the *OPull* output port in figure 3.36, are associated with one *pull* connection. Each *pull* input port, *IPull*, may only be connected to a one *pull* output port, *"pull connection i"* in the figure. This explains the cardinality $1 \rightarrow m$ which appears in figure 3.27.

Functionality. *OPull* output ports can manage multiple connections simultaneously, in this way, a sender can accept requests from multiple receivers. Therefore *pull* connections model a typical request/answer model of interaction between one producer, the sender, and multiple consumers, the receivers. Although in figure 3.36 the length of the internal arrays at sender side is m, *OPull* output ports change dynamically their length in order to accommodate all receivers to which they are connected.

Signaling/Unsignaling. *Pull* output ports, *OPull*, are *signaled* when they receive a request packet by means of an AS or and ASC through any of the connections into which it is taking part. Specifically, each connection gets individually *signaled* when ASs or ASCs are applied on it. The sender applies testing and waiting operations on the *OPull* to find out whether any requests have been received. Testing and waiting operations carried out in PRs in the sender side will make connections *non signaled*, and will allow knowing which requests have been received.

Pull input ports, *IPull* input ports will get *signaled* when answer packets are received by means of ASs or ASCs applied through the *pull* connection where it is involved. They remain *signaled* until a testing or waiting operation is applied on it by means of a PR in the receiver side.

3.5.2.11 Simple Multi Packet Connections

Figure 3.27 illustrates the main types of port connections provided by CoolBOT that have already been introduced in previous sections. Additionally, there is a set of port connections resulting from the combination of multi packet output and input ports (*OMultiPacket*, *OLazyMultiPacket* and *IMultiPacket*) with the output and input ports involved in *last*, *fifo* and *unbounded fifo* connections (*OGeneric*, *ILast*, *IFifo* and *IUFifo*). All these port connections are called collectively *simple multi packet* connections. Figure 3.28 shows all the possibilities.

Definition. *Simple multi packet* connections are defined as the pairs of output ports/input ports that appear in figure 3.28. Snapshots of their internal structure are shown in figures 3.38 and 3.37. The internals of *simple multi packet* connections resulting from the combinations of *OMultiPacket* and *OLazyMultiPacket* with *ILast*, *IFifo* and *IUFifo* are functionally equivalent to figure 3.37.

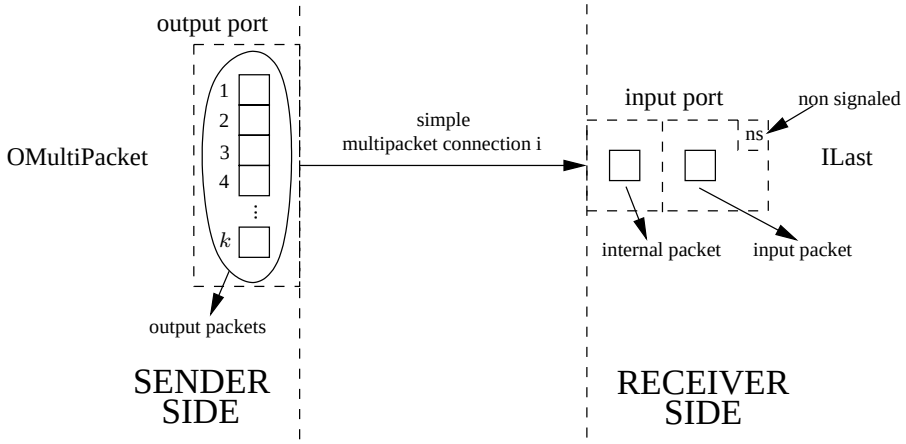

Figure 3.37: A *simple multi packet* connection combining an *OMultiPacket* and an *ILast*.

Basic ICC Mechanisms. *Simple multi packet* connections are possible due to the fact that the output and input ports involved in them, make use of the same compatible ICC mechanisms: AS and ASC in the output ports, and PR in the input ports (see tables 3.3 and 3.4). Besides, these ports have the same semantics of use. Notice that *priority* input and output ports (*OPriority* and *IPriorities*) use also the same type of ICC mechanisms, but due to the fact that their semantics of priorities are different, they can not be combined with other types of input and output ports. The same happens with *pull* output and input ports (*OPull* and *IPull*).

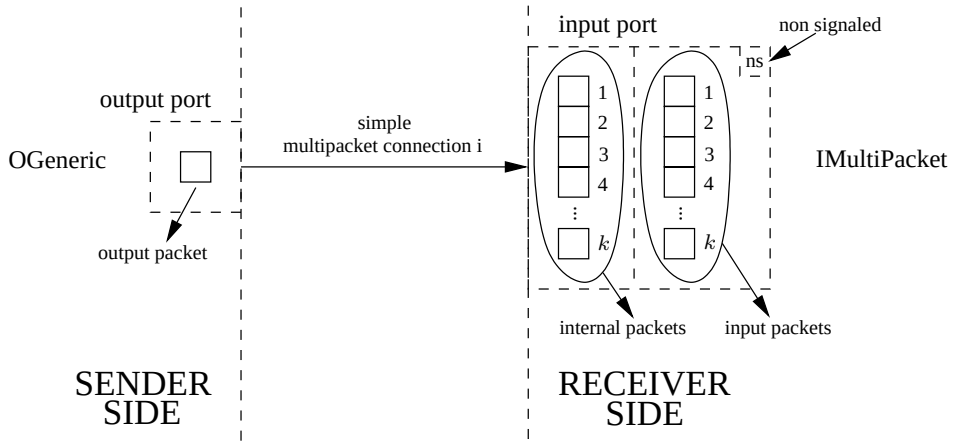

Figure 3.38: A *simple multi packet* connection combining an *OGeneric* and an *IMultiPacket*.

Port Packet Types. Observing figures 3.28, 3.37 and 3.38 it is important to note that *simple multi packet* connections implies that one of parts involved in the connection can only accept one type of port packet. This is because *OGeneric*, *ILast*, *IFifo* and *IUFifo* admit only one type of port packet. It is said that they are *simple packet* ports. On the opposite side, *OMultipacket*, *OLazyMultipacket* and *IMultipacket* ports accept multiple types of port packets and, accordingly, they are called *multi packet* ports. In particular, *multi packet* ports admit an array of types of port packets. All in all, in *simple multi packet* connections, the port connections are always carried out between a *simple packet* port and one of the elements of a *multi packet* port. It is mandatory that the port packet type accepted by the *simple packet* port matches the type accepted by the element of the *multi packet* port.

Cardinality. The cardinality of *simple multipacket* connections is $n \rightarrow m$ (figure 3.28). Having a look to figures 3.27 and 3.28, notice that, in this way *simple packet* and *multi packet* ports can take part in multiple types of port connections simultaneously.

Functionality. *Simple multi packet* connections offer the same functionality as *multi packet* and *lazy multi packet* connections, except that through them it is only possible to send a port packet due to the nature of *simple packet* ports.

Signaling/Unsignaling. *IMultipacket* input ports get *signaled* and *non signaled* in the same way that they do in *multi packet* and *lazy multi packet* connections. Similarly, the conditions for signaling and unsignaling of *ILast*, *IFifo* and *IUFifo* input ports are equivalent to the conditions they have in *last*, *fifo* and *unbounded fifo* connections.

3.6 Component Composition

In CoolBOT, components constitute the building blocks by means of which systems are constructed by integration and composition. In fact, it is possible to integrate different

components to form compositions or aggregates of components that, once integrated, can also be considered, in terms of composition, as components that, in turn, can be used to take part in new bigger compositions.

CoolBOT components are classified into two kinds: *atomic* and *compound* components. With independence of its type, components, whether *atomic* or *compound*, are externally equivalent in terms of composition. Next sections will introduce each of them in more detail.

3.6.1 Atomic Components

An *atomic* component is an indivisible component, i.e., it is not possible to split it into other components. Therefore, an *atomic* component is not a composition or aggregation of components. *Atomic* components have been devised to abstract hardware like sensors and effectors, encapsulate software libraries like third party software, and to implement specific algorithms designed to be reusable.

```
...

#include "coolbot.h"
using namespace CoolBOT;

...

namespace PioneerSpace
{
    ...

    class Pioneer: public Component
    {
        public:

            ...

        private:

            ...

    };

    ...
}
```

Figure 3.39: Component *Pioneer*: an atomic component.

CoolBOT is a C++ framework, and every CoolBOT component, *atomic* or not, is a C++ class. *Atomic components* participate of all concepts and abstractions introduced so far in this chapter. Along this section, the process of building an *atomic component* will be explained with the help of an example. At the same time, a more detailed vision of how components map into C++ code will be given.

Figure 3.39 shows the first level of organization of an *atomic* component called *Pioneer*. Observe how the component, the class *Pioneer*, inherits from a class called *Component*. All components inherits from this class. That is the mechanism the

framework uses to endow components with a default behavior, what has been referred to as *component defaults* in section 3.3. Following subsections will outline the process of building atomic components in CoolBOT, and how concepts and abstraction introduced so far in this chapter map into C++ code. All this will be done using the component of figure 3.39. From now on some C++ code will be shown in order to illustrate some framework aspects. Appendix A contains the CoolBOT programming style rules, that could give some more insights about the code, if considered necessary.

3.6.1.1 External Interface: Input and Output Ports

The external interface of public output and input ports of the *Pioneer* component appears in figure 3.40. In the figure, output and input port types are indicated by their symbols in parentheses. Consult tables 3.3 and table 3.4 for symbols associated respectively to output and input ports.

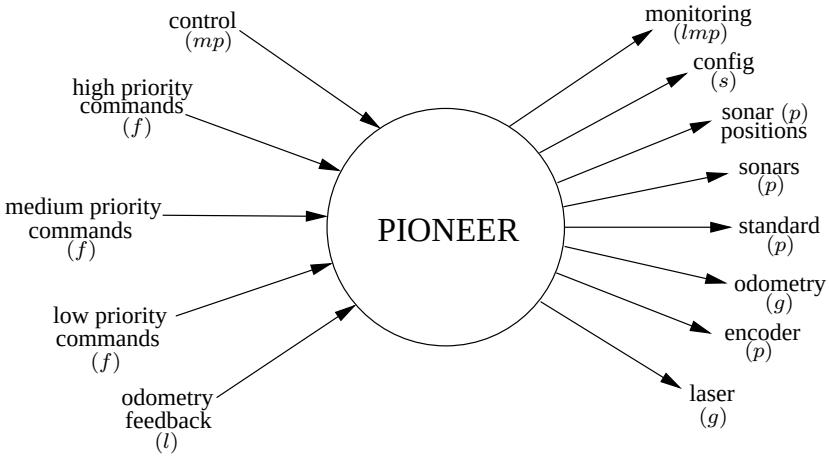

Figure 3.40: Component *Pioneer*: external interface.

The *Pioneer* component of figure 3.40 adapts the **ARIA** library (**A**ctiv Media **R**obots **I**nterface for **A**pplications) for the **Pioneer** family of robots of Activ Media Robotics. The component wraps the lowest level of functionality of **ARIA** [Activ Media Robotics, 2003] as a CoolBOT component. Figure 3.41 shows one of our Pioneer robots, in this case, it has a web camera mounted on a Directed Perception pan-tilt placed on the top of the robot. Tables 3.5 and 3.6 resume respectively the *Pioneer* component´s external interface of public output and input ports.

The code in figure 3.42 illustrates the definition of non default output and input ports for the *Pioneer* component. Figure 3.43 exemplifies the instantiation of two ports for the *Pioneer* component: the *high priority commands* output port and the *sonars* input port.

As to input ports, the *Pioneer* component uses internally three levels of priority; figures 3.44 and 3.45 illustrate respectively the code corresponding to the definitions of these three priority levels, and the mapping of the component´s input ports into each priority. Note how the *high priority commands*, *medium priority commands*, and

Figure 3.41: One of our Pioneer robots.

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is an *OLazyMultiPacket*.
config	This is an *OShared* output port. It publishes configuration data of the robot to which the component is attached.
sonar positions	Once connected to a physical robot this *OPoster* output port publishes the coordinates of the different sonars the robot provides.
sonars	By means of this *OPoster* output port the component provides robot´s sonar readings.
standard	Some additional information related to the robot is published in this *OPoster* output port: bumper status, motor stall, ...
odometry	Through this *OGeneric* output port the component publishes periodically the robot´s odometry.
encoder	This *OPoster* output port allows direct access to the motor encoders of the robot.
laser	This *OGeneric* output port resends information related to a **SICK** laser range scanner, if the robot is equipped with one.

Table 3.5: Component *Pioneer*: public output ports.

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which *Pioneer´*s controllable variables may be modified and updated. This component does not add any new controllable variable. It is a *IMultiPacket* input port.
high priority commands	*IFifo* input port that accepts commands to be sent to the robot the component is connected to. High priority input port.
medium priority commands	*IFifo* input port that accepts commands to be sent to the robot the component is connected to. Medium priority input port.
low priority commands	*IFifo* input port that accepts commands to be sent to the robot the component is connected to. Low priority input port.
odometry feedback	It is a *ILast* input port, by means of it the odometry data produced can be affected by a correction.

Table 3.6: Component *Pioneer*: public input ports.

low priority commands ports are functionally equivalent, they can send the same type of commands to the physical robot (consult [Activ Media Robotics, 2002] for more information about these commands). The only difference between them is that they belong to three different levels of input port priorities, the highest level corresponds to *high priority commands*, and the lowest to *low priority commands*. This feature is useful to prioritize some commands in specific moments of execution, for instance, an emergency stop, a fast velocity change to avoid and obstacle, etc. Obviously, these three ports should be used conveniently, as for instance, utilizing the highest command port, *high priority commands*, when the situation really requires to run a specific command immediately.

In the code outlined in figure 3.42, it is possible to observe that the *Pioneer* component also has private output and input ports (`_OutputPorts_` and `_InputPorts_` enumerations). Tables 3.7 and 3.8 resume briefly the use of this private output and input ports.

Private Output Ports	
Name	Brief Description
to reset odometry	*OGeneric* output port used to make the *listening* port thread to reset the odometry published by the component.
port thread control	*OMultiPacket* port to control the port threads, in this case, the *Pioneer* component has only one port thread: the *listening* port thread.

Table 3.7: Component *Pioneer*: private output ports.

The *Pioneer* component does not add any new controllable variable to the ones included by default by CoolBOT with component defaults (see section 3.3). The same happens with observable variables, as this component does not need further observable

variables apart from the ones provided by component defaults.

Finally, it is necessary to add that the *Pioneer* component does not use any *component watchdog* to supervise its input ports. Neither does it need any extra timer besides the *default timer* that CoolBOT provides for its internal functionality.

```
class Pioneer: public Component
{
    public:

        ...

        enum InputPorts
        {
            HIGH_PRIORITY_COMMANDS=DEFAULT_INPUT_PORTS,
            MEDIUM_PRIORITY_COMMANDS,
            LOW_PRIORITY_COMMANDS,
            ODOMETRY_FEEDBACK,
            INPUT_PORTS
        };

        enum OutputPorts
        {
            STANDARD=DEFAULT_OUTPUT_PORTS,
            ODOMETRY,
            SONAR_POSITIONS,
            SONARS,
            ENCODER,
            CONFIG,
            LASER,
            OUTPUT_PORTS
        };

        ...

    private:

        ...

        enum _InputPorts_
        {
            _EXCEPTION_IN_LISTENING_TASK_=_DEFAULT_INPUT_PORTS,
            _RESET_ODOMETRY_,
            _PORT_THREAD_MONITORING_,
            _LISTENING_CONTROL_,
            _INPUT_PORTS_
        };

        enum _OutputPorts_
        {
            _TO_RESET_ODOMETRY_=0,
            _PORT_THREAD_CONTROL_,
            _OUTPUT_PORTS_
        };

        ...
};
```

Figure 3.42: Component *Pioneer*: input and output ports.

```
Pioneer::Pioneer(const char* pRobotConnection)
{
    ...

    // Input port high priority commands: begin
    ppIPorts[HIGH_PRIORITY_COMMANDS]=
        new IFifo(CommandPacket::prototype(),
                  _COMMAND_FIFO_LENGTH_);
    // Input port high priority commands: end

    ...

    // Output port sonar: begin
    ppOPorts[SONARS]=
        new OPoster(SonarPacket::prototype(),1);
    // Output port sonar: end

    ...
}
```

Figure 3.43: Component *Pioneer*: ports instantiation.

```
class Pioneer: public Component
{
        ...

    private:

        ...

        enum _InputPortPriorities_
        {
            _LOW_=0,
            _MEDIUM_,
            _HIGH_,
            _INPUT_PORT_PRIORITIES_
        };

        ...
};
```

Figure 3.44: Component *Pioneer*: input port priorities.

```
int Pioneer::_pInputPortPriorities_[INPUT_PORTS+_INPUT_PORTS_]=
{
    _HIGH_, // CONTROL
    _HIGH_, // HIGH_PRIORITY_COMMANDS
    _MEDIUM_, // MEDIUM_PRIORITY_COMMANDS
    _LOW_, // LOW_PRIORITY_COMMANDS
    _LOW_, // ODOMETRY_FEEDBACK
    _LOW_, // _EMPTY_TRANSITION
    _LOW_, // _TIMER
    _HIGH_, // _EXCEPTION_IN_LISTENING_TASK_
    _LOW_, // _RESET_ODOMETRY_
    _HIGH_, // _PORT_THREAD_MONITORING_
    _LOW_, // _LISTENING_CONTROL_
};
```

Figure 3.45: Component *Pioneer*: input port priority mapping.

Private Input Ports	
Name	Brief Description
empty transition	This is the default *empty transition* port explained in section 3.3.6, it is an *ITick* input port.
timer	This is the default port *timer* (section 3.3.5) used by the *default timer*, it is also an *ITick* input port.
exception in listening task	Through this *ITick* input port the *listening* port thread communicates to the *main* one that an exception has been detected during execution.
reset odometry	*ILast* input port utilized by the *listening* port thread to reset the odometry which the component publishes through the public output port *odometry*.
port thread monitoring	*IMultiPacket* port for observing port threads. In this specific case, the *Pioneer* component has only a port thread: the *listening* port thread.
listening control	This is the control input port of the *listening* port thread. It is an *ILast* input port.

Table 3.8: Component *Pioneer*: private input ports.

3.6.1.2 Port Packets

As already introduced in section 3.2.1, *port packets* are the discrete information units that are sent and received by components through its output and input ports by means of port connections. Like components, *port packets* take the form of C++ classes in CoolBOT, and accordingly, each type of port packet is defined by a different C++ class.

The code in figure 3.46 resumes what a *port packet* is in CoolBOT. All *port packets* must inherit from the class *PortPacket* that appears in figure 3.46. *Port packets* are discrete data units that should be:

- *Able to be cloned*: it is possible to create or to instantiate a copy of any given port packet. This is imposed by CoolBOT using an abstract class: the class `CloningInterface` in figure 3.46.
- *Able to be copied*: it is possible to make what is called a "deep copy" [Stroustrup, 2000] of any given port packet. This is also an abstract class inherited by every port packet, the class `DeepCopyingInterface` of figure 3.46.
- *Able to be packed/unpacked*: it is possible to "pack" any port packet in a string of raw bytes, and also it is possible to "unpack" a port packet from a string of raw bytes, given that the type of the port packet is known (this is the well-known "serialization" interface that is found in multiple object-oriented programming languages [Arnold et al., 2000] and libraries [MSDN, 2002]). This is carried out using an abstract class as well, the class `PackingInterface` in figure 3.46.

  CoolBOT make uses of the **XDR** (e**X**change **D**ata **R**epresentation) library

[Bloomer, 1992] in order to pack and unpack port packets in strings of bytes in a machine-independent manner.

As an example, figure 3.47 shows a portion of code of a type of port packet used by the *Pioneer* component. As previously commented, a port packet type is defined as a C++ class. In the figure, it is a class called *OdometryPacket* which is the type of port packet used to publish the robot odometry through the component´s *odometry* output port depicted in figure 3.40.

3.6.1.3 Component Automaton

The *default automaton* of figure 3.8 is part of component defaults, so components inherit it from the *Component* class of figure 3.39. To complete the functionality of a component it is necessary to define what was called in section 3.3.3 the *user automaton*. Figure 3.48 shows the automaton of the *Pioneer* component already completed. As it can be observed the user automaton is only formed by one state called **main**.

The *Pioneer* component is an atomic CoolBOT component that uses internally the library **ARIA** to control a Pioneer robot. It uses the lowest level of functionality existing in **ARIA**, which consists mainly in accessing directly the functionality provided by the **Pioneer 2 Operating System** (**P2OS**) [Activ Media Robotics, 2002]. A Pioneer robot is a physical robot or a robot simulator (**SRIsim** provided also with the **ARIA** library) that executes the **P2OS**. A physical robot is commanded by means of a RS232 port, while the robot simulator uses a TCP/IP connection.

From now on, explanations will be given only referencing to a physical robot to which the *Pioneer* component is connected through a RS232 connection. All comments are applied identically to the robot simulator where the component just uses a TCP/IP

```
...

namespace CoolBOT
{
    ...

    class PortPacket: public CloningInterface,
                      public DeepCopyingInterface,
                      public PackingInterface
    {
        public:

            ...

        private:

            ...

    };

    ...
}
```

Figure 3.46: The *PortPacket* class.

connection instead of an RS232 connection.

Using the serial port the **P2OS** communicates periodically its internal state: bumpers, wheel velocities, wheel encoders, odometry, sonar readings, internal configuration, etc. The period for sending this information through the serial port is usually 100 milliseconds, although it can be configured to be 50 milliseconds as well. The **P2OS** uses the serial port also to receive any of the set of commands the robot can accept: move forward, translational velocity, rotational velocity, maximum velocity, reset odometry, etc. In this manner, an external computer using a *Pioneer* component may control a Pioneer robot, having it attached to one of its serial ports.

The behavior of the *Pioneer* component is not complex. The component starts execution trying to connect to the robot using one of the serial ports of the computer where it is running. This is carried out in the entry section of its **starting** state. Additionally, it asks for some memory in the same code section. If everything goes well, the component transits to **ready** state (see figure 3.48) where it waits idle until it is commanded to get into the user automaton. As it is shown in the figure, the *Pioneer* component´s user automaton has only a state named **main** state. In this state, the component is listening continuously the serial port to publish the information that is periodically sent through the port by the robot. The component only formats conveniently the information which it receives, and publishes it through its different output ports (figure 3.40). At the same time, in the **main** state, the component is attending all its input ports. Thus it sends to the robot any command that it receives through its command ports, and it corrects internally the odometry that is published when a correction is received through its *odometry feedback* input port. Once in **main**

```
...

#include "component/coolbot_portpacket.h"
using namespace CoolBOT;

...

namespace PioneerSpace
{
    ...

    class OdometryPacket: public PortPacket
    {
        public:

            ...

        private:

            ...

    };

    ...
}
```

Figure 3.47: An example of *port packet*: the *OdometryPacket* class.

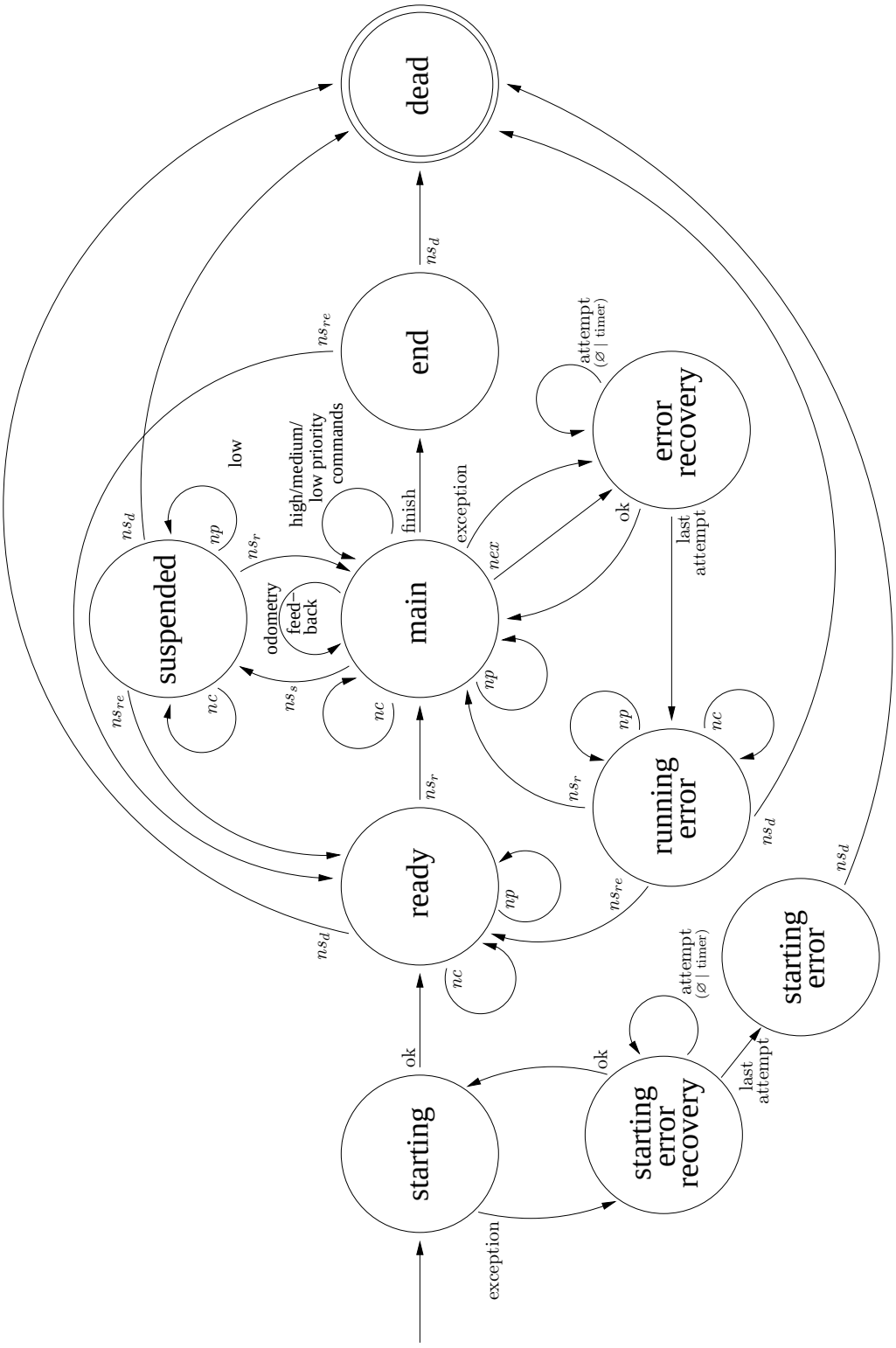

Figure 3.48: Component *Pioneer*: automaton.

state, the component is running there until, either it is commanded by means of the *control* port to go to one of the default automaton states, or, on the contrary, it gets into **error recovery** state because an exception has been raised.

Code of figure 3.49 illustrates how states, entry and exit sections and transitions are declared in the C++ code of a component. The piece of code of figure 3.50 exemplifies the typical definitions in CoolBOT for an entry section, an exit section and a transition for an automaton state, concretely, for the **main** state in the *Pioneer* component.

```
class Pioneer: public Component
{
    public:

        ...

        enum States
        {
            MAIN=DEFAULT_STATES,
            STATES,
            RUNNING_ENTRY_STATE=MAIN
        };

        ...

    private:

        ...

        // state starting: begin
        ...
        static int _startingEntry_(Component* pComp);
        static void _startingExit_(Component* pComp);
        // state starting: end

        ...

        // state main: begin
        ...
        static int _mainEntry_(Component* pComp);
        static void _mainExit_(Component* pComp);

        static int _mainTimer_(int port, Component* pComp);
        static int _mainExceptionInListeningTask_(int port, Component* pComp);
        static int _mainCommands_(int port, Component* pComp);
        static int _mainOdometryFeedback_(int port, Component* pComp);
        static int _mainResetOdometry_(int port, Component* pComp);
        // state main: end

        ...
};
```

Figure 3.49: Component *Pioneer*: automaton state declarations.

3.6.1.4 Port Threads

If it is considered necessary, components may be composed internally by several threads: the *main thread* that executes the *component kernel* and one or more threads called *port threads*, as was explained in section 3.4.1. In the particular case of the *Pioneer*

component, it has been estimated convenient to organize it internally into two threads: the *main* thread which executes the component kernel, and a *port thread*, the *listening* port thread. They have the following functionality:

- The *main* thread: This is the thread that executes the *component kernel*, and makes the component evolve along its automaton. Besides, when the component is in the **main** state, it is responsible for sending commands to the physical robot using the serial port.
- The *listening* port thread: This port thread is only active when the component is in the **main** state of the automaton, in the rest of states it is suspended (see figure 3.12). The principal functionality of this port thread is to listen to the connection with the robot to collect the information it sends, to format it conveniently, and then to publish it by means of the component´s output ports.

Each one of these two threads is responsible for a different subset of the output and input ports of the *Pioneer* component. Figure 3.51 shows the declaration of the

```
...

// state main: begin

int Pioneer::_mainEntry_(Component* pComp)
{
    Pioneer* pThis=static_cast<Pioneer*>(pComp);
    pThis->_debugEntrySection_();

    ...

    return pThis->_currentState_;
}

void Pioneer::_mainExit_(Component* pComp)
{
    Pioneer* pThis=static_cast<Pioneer*>(pComp);
    pThis->_debugEntrySection_();

    ...
}

...

int Pioneer::_mainCommands_(int port, Component* pComp)
{
    Pioneer* pThis=static_cast<Pioneer*>(pComp);
    pThis->_debugPortTransition_(port);

    ...

    return pThis->_currentState_;
}

...

// state main: end
...
```

Figure 3.50: Component *Pioneer*: an entry section, an exit section, and a transition.

identifiers for both threads, and figure 3.52 illustrates the C++ code that makes the mapping of which thread is responsible for each output and input port. For each state of a multithreaded component it is also necessary to indicate which thread is active in each automaton. The code of figure 3.53 illustrates a snippet of code showing how this is done in the *Pioneer* component. For each state there is a thread mask indicating which threads are active and running, and which ones are suspended.

```
class Pioneer: public Component
{
    public:

        ...

    private:

        ...

        enum _Threads_
        {
            _LISTENING_=_DEFAULT_THREADS,
            _THREADS_
        };

        ...
};
```

Figure 3.51: Component *Pioneer*: threads identifiers.

The rationale of using two threads inside the *Pioneer* component, the default *main* thread, and the *listening* thread, is based on the fact that their respective functionalities are disjoint and orthogonal. The *main* thread attends the component´s public output ports and send commands to the robot through the serial port. On the other side, the *listening* thread just listens to anything the robot sends through the serial port, and publishes this information by means of the component´s output ports. This can be verified having a look to the code of figure 3.52. Observe how the *main* thread is in charge of the majority of the public input ports of the component. Complementarily the *listening* port thread is responsible for the most of the output ports. There are also several private output and input ports through which both threads intercommunicate each other; their use and utility were already resumed in tables 3.7 and 3.6.

3.6.1.5 Exceptions

In addition to the default exceptions provided by the framework, the *Pioneer* component defines several additional exceptions. Table 3.9 resumes the different exceptions the *Pioneer* component uses.

Figure 3.54 shows the code corresponding to the declaration of the non default exceptions of the *Pioneer* component. There is also a typical prototype for an exception handler, in this case, an exception handler for the *connection error* exception. Furthermore, figure 3.55 illustrates a snippet of code where it is possible to see how and

exception is instantiated in a component, concretely the *connection error* exception. Below, it is outlined the body of an exception handler, the body corresponding to the prototype that appears in figure 3.54. Concretely, this exception handler gets called five times (see figure 3.55) before declaring the exception unrecoverable and driving the component to **running error** state. What is done in the handler is just to listen to the serial port for a specific time, if along all these attempts nothing is received then, the component considers the serial connection as definitively broken. In that case, the component would enter in **running error** state to wait for external intervention through the component´s *control* port. Otherwise, the component recovers itself from the exception, and returns to **main** state to continue normal execution.

Got to this point, a brief outline of the process of building an *atomic* CoolBOT component has been presented. Observe that *atomic* components participate without exception of all abstractions and concepts that have been presented so far, next section will show how *compound* components do participate of these abstractions as well.

```
...

int Pioneer::_pInputPortToThread_[INPUT_PORTS + _INPUT_PORTS_]=
{
    _MAIN, // CONTROL
    _MAIN, // HIGH_PRIORITY_COMMANDS
    _MAIN, // MEDIUM_PRIORITY_COMMANDS
    _MAIN, // LOW_PRIORITY_COMMANDS
    _LISTENING_, // ODOMETRY_FEEDBACK
    _MAIN, // _EMPTY_TRANSITION
    _MAIN, // _TIMER
    _MAIN, // _EXCEPTION_IN_LISTENING_TASK_
    _LISTENING_, // _RESET_ODOMETRY_
    _MAIN, // _PORT_THREAD_MONITORING_
    _LISTENING_ // _LISTENING_CONTROL_
};

...

int Pioneer::_pOutputPortToThread_[OUTPUT_PORTS + _OUTPUT_PORTS_]=
{
    _MAIN, // MONITORING
    _LISTENING_, // STANDARD
    _LISTENING_, // ODOMETRY
    _LISTENING_, // SONAR_POSITIONS
    _LISTENING_, // SONARS
    _LISTENING_, // ENCODER
    _LISTENING_, // CONFIG
    _LISTENING_, // LASER
    _MAIN, // _TO_RESET_ODOMETRY_
    _LISTENING_, // _PORT_THREAD_CONTROL_
};

...
```

Figure 3.52: Component *Pioneer*: mapping of output and input ports to port threads.

```
...

// state starting: begin

...

const bool Pioneer::_pStartingThreadMask_[_THREADS_]=
{
    true, // _MAIN
    false // _LISTENING_
};

// state starting: end

...

// state main: end

...

const bool Pioneer::_pMainThreadMask_[_THREADS_]=
{
    true, // _MAIN
    true // _LISTENING_
};

// state main: end

...
```

Figure 3.53: Component *Pioneer*: thread masks.

```
class Pioneer: public Component
{
    public:

        ...

        enum Exceptions
        {
            ROBOT_OPENING_FAILED=DEFAULT_EXCEPTIONS,
            P2OS_SYNCHRONIZATION_FAILED,
            CONNECTION_ERROR,
            ROBOT_PARAMS_LOAD_FAILED,
            EXCEPTIONS
        };

        ...

    private:

        ...

        // Exception CONNECTION_ERROR: begin

        static bool _connectionErrorHandler_(Component* pComp);

        // Exception CONNECTION_ERROR: end

        ...
};
```

Figure 3.54: Component *Pioneer*: exceptions declarations.

Exceptions	
Name	Brief Description
no memory	Default exception, see section 3.3.4
inconsistency	Default exception, see section 3.3.4
file not found	Default exception, see section 3.3.4
robot opening failed	This exception is thrown at the entry section of the **starting** state when it is not possible to open the serial port.
p2os synchronization failed	Once the component has opened the serial port correctly, it is necessary to follow an initial synchronization protocol to connect the **P2OS** running in the robot. This exception is launched if such a protocol happens to be erroneous. This synchronization is also performed in the entry section of the **starting** state, so only there, this exception might occur.
robot params load failed	This exception is raised when the component tries to load the robot parameter file and something goes wrong. Once the component has correctly completed the synchronization protocol with the **P2OS** running in the robot, the robot identifies itself. Based on this identification, the component loads a file with information about the robot: number of sonars, sonar positions, geometry, conversion factors, etc; this information is different depending on the robot model. This load happens also in the entry section of the **starting** state.
connection error	Once in execution, and having the component in the user automaton, i.e., in the **main** state, it might happen that the serial connection breaks down. Note that if, for instance, a wireless link is used with the robot, some data might be lost from time to time. Or perhaps, the serial wire connecting the robot has been disconnected, or cut, etc. This exception would be launched in such cases.

Table 3.9: Component *Pioneer*: exceptions.

3.6.2 Compound Components

The design principles of modularity, hierarchy and integrability, previously commented in chapter 2 (section 2.4.1), constitute the rationale behind *compound* components in CoolBOT.

Atomic components have been mainly devised to be used:

- to abstract low level hardware layers to control sensors and/or effectors,
- to interface and/or to wrap third party software and libraries,
- and to implement generic algorithms

in order to make them isolated pieces of deployable software in the form of CoolBOT components. Thanks to the uniformity of external interface and internal structure that CoolBOT imposes on components, CoolBOT components may be used as building blocks that hide their internals behind a public external interface. Components, once designed, developed, and tested enough, may be used as functional units wherever is necessary. Some questions come up naturally from these considerations. What about combinations of *atomic* components?. If these combinations were used as "single" components, would it be possible to use them in new combinations, as if they were *atomic* components as well?. Would it be interesting to consider combinations of components as "single" components that verify the uniformity of interface and structure that CoolBOT claims on components? Why?.

It is evident that *atomic* components are modular, in the sense, that they hide their internals, and offer only a public external interface. It would be a step forward, to have the possibility of integrating components in a way that, once integrated, they could be considered as "single" components, that hide also their internals, and offer an external interface of output and input ports, and observable and controllable variables, like any "single" component. In CoolBOT, this idea of a combination of components that, in turn, behaves like a "single" component is what have been called a *compound*

```
...

void Pioneer::_staticInitialization_()
{
    ...

    _ppExceptions_[CONNECTION_ERROR]=
        new Exception(CONNECTION_ERROR,
                      _ppExceptionsStrings_[CONNECTION_ERROR],
                      _ppExceptionsDescriptions_[CONNECTION_ERROR],
                      5, // attempts
                      1000, // milliseconds
                      _connectionErrorHandler_, // handler
                      NULL, // on-success handler
                      NULL); // on-failure handler

    ...
}

...

// Exception CONNECTION_ERROR: begin

bool Pioneer::_connectionErrorHandler_(Component* pComp)
{
    Pioneer* pThis=static_cast<Pioneer*>(pComp);

    ...
}

// Exception CONNECTION_ERROR: end

...
```

Figure 3.55: Component *Pioneer*: exception instantiation and an exception handler.

component.

Like *atomic* components *compound* components are modular, they may be integrated with other components, whether *atomic* or *compound*, to form other *compound* components. In this manner, hierarchies of components may come up from *compound* components. The hierarchy lowest levels appear when *atomic* components are reached.

A *compound* component, is a composition of instances of several components which can be either atomic or compound. Figure 3.57 illustrates this idea graphically, where the *compound* component **c** is a composition of two instances, one of an *atomic* component **a**, $\mathbf{a}_i$, and one of another *atomic* component **b**, $\mathbf{b}_i$, both shown in figure 3.56. Figure 3.58 depicts a *compound* component **d** made of an instance of the *compound* component **c**, $\mathbf{c}_i$, and an instance of the *atomic component* **b**, $\mathbf{b}_i$, evidencing that instances of *compound* components are functionally equivalents to *atomic* components in terms of composition and instantiation. Default ports (*empty transition*, *timer*, *control* and *monitoring* ports) are not shown.

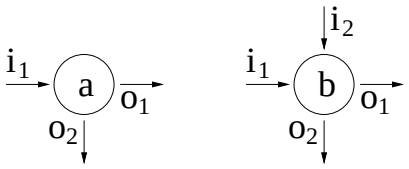

Figure 3.56: Two atomic components: **a** and **b**.

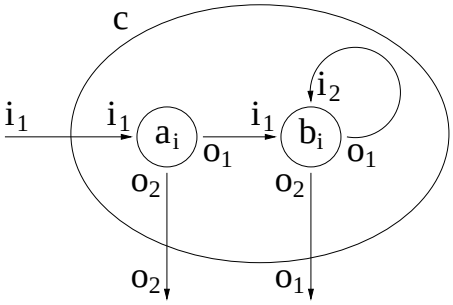

Figure 3.57: The compound component **c**, a composition of atomic components: **a** and **b**.

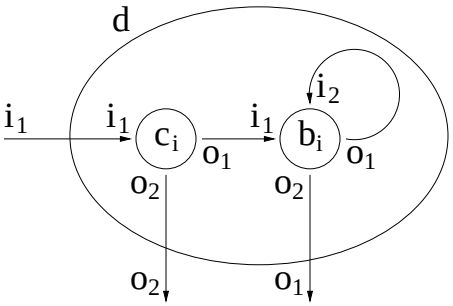

Figure 3.58: The compound component **d**: a composition of a compound component, **c**, and atomic component, **b**.

A *compound* component is a component that uses the functionality of instances of another *atomic* or *compound* components to implement its own functionality. Components whose instances are used inside a *compound* component are called *local* components, thus, **b** and **c** are *local* components of **d** in figure 3.58.

Compound components, like all CoolBOT components, are also port automata which offers an external interface of output and input ports, and that have their functionality defined internally by means of an automaton. Thence, they are provided by the framework with the same component defaults (see section 3.3) that *atomic* components.

3.6.2.1 The Supervisor

The automaton that coordinates and controls the functionality of a *compound* component is called its *supervisor*, and like *atomic* components it follows the control graph defined for the *default automaton* of figure 3.8. A *compound* component could be seen as an *atomic* component, that internally uses instances of other components to achieve its specific functionality. To emphasize this idea note that the main difference between an *atomic* and a *compound* component is that the *compound* component would have a line of code like this:

```
...
_ppComponents_[_PIONEER_]=new Pioneer("/dev/ttyS0");
...
```

that creates an instance of a component, a *Pioneer* component in this particular case, attached to a specific serial port (for example, "`/dev/ttyS0`" for GNU/Linux, or "`COM1`" for Windows). Code like this is typical in the constructors of *compound* components. But the operations that a *compound* component can carry out on its *locals* components are not only reduced to instantiation. Concretely, the *supervisor* of a *compound* component controls and observes the execution of its *local* components through their control and monitoring ports. Thus, it can have them running, suspended, waiting in *ready* state, etc. Furthermore, it can modify any of their controllable variables, default and non-default ones, and observe their behavior accessing all their controllable variables, whether default or not. In few words, in general, it has the possibility of monitoring and evaluating their level of progress and operation. On the other side, the external interfaces of output and input ports of its different *local* components allows connecting them forming diverse component topologies. Additionally, component instantiation is not only limited to the constructor of *compound* components, it can be performed at runtime wherever it is necessary.

Figure 3.59 clarifies the idea of the *supervisor* of a *compound* component; in the particular case of the figure, the *compound* component that appears there abstracts sensors and effectors in a typical mobile robot. In the figure the *supervisor* controls and observes the *local* components through their control and monitoring ports (labelled as **c** and **m**). The locals components have different functionalities. There is a **motion controller**, which is a component that have direct access to the robotic platform, and abstracts its low level vehicle motion interface. Another *local* component is the **ir/us/bumper server** that encapsulates the low level interface to the sensor suite typically found in any mobile robot (infrared sensors, sonars and bumpers), providing time-stamped sensory data scans at a specific frequency. It also detects emergency situations when the platform is too close to an obstacle. The **laser server** component

has a similar functionality that the **ir/us/bumper server** but applied, for instance, to a **SICK** laser range finder. The **battery monitor** component is just in charge of monitoring the robot battery levels. Finally, there is also a component called **state reflector** that fuses all robot sensor and effector data and publishes it in an integrated way. Observe how in the figure the supervisor is presented as a component inside the *compound* component, this is to highlight that the supervisor has its own independent flow of execution, the *component kernel* of the *compound* component, which constitute the flow of control that integrates the behavior of multiple components. For further clarification, the types of output and input ports are not shown in the figure.

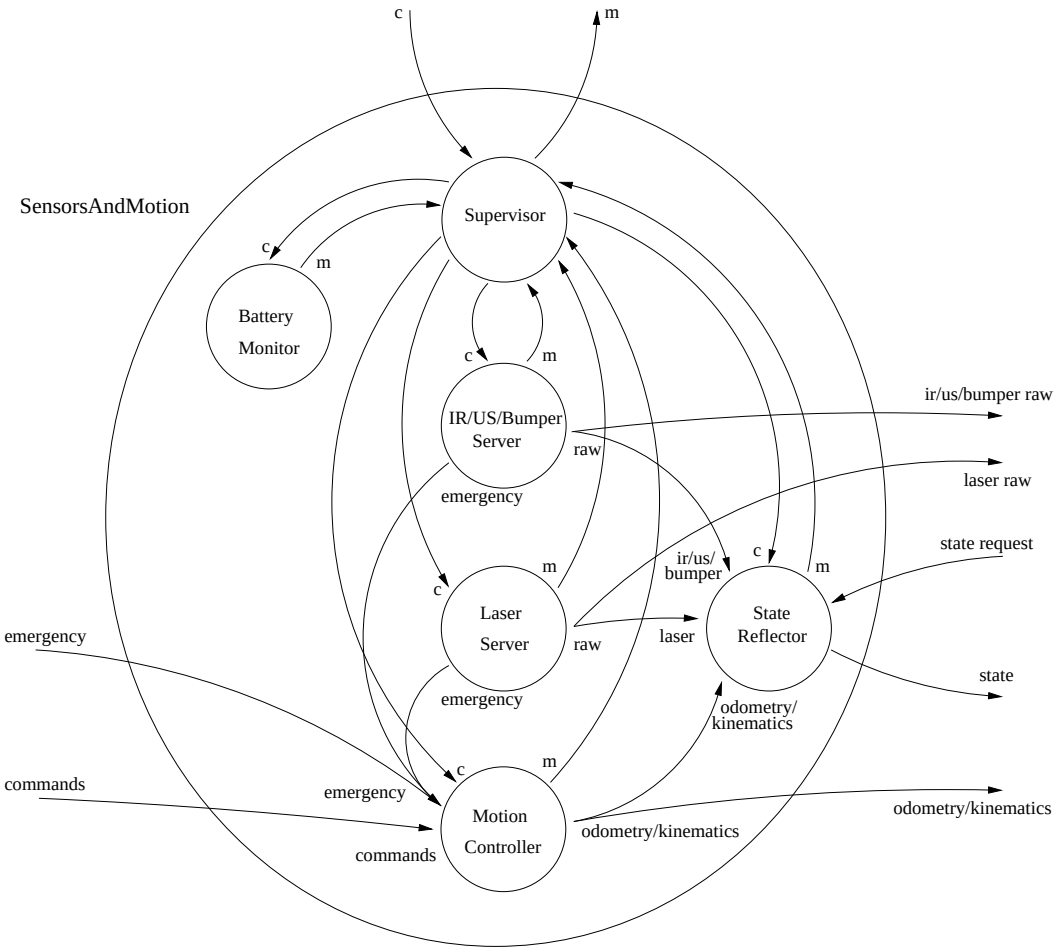

Figure 3.59: A *compound* component: the supervisor.

Note that, in the same way that the automaton of an *atomic* component concentrates the control flow of a component, the *supervisor* of a *compound* component concentrates the control flow of a composition of components. Note that the use of *compound* components at different levels of component composition, is equivalent to establish multiple levels of control loops by means of the integration of other components, conforming in such a way, not only a hierarchy of components, but also a *hierarchy of control* wherever they are used.

Figure 3.60 helps to illustrate graphically this idea, where there is a *compound*

component that internally makes use of instances of other components, one of which is also a *compound* component. This, in turn, has a *local* component that is *compound* as well. Only the default *control* and *monitoring* ports (**c** and **m**, respectively) are shown in the figure. Observe as the top level *compound* component offers an external interface that hides its internals, which allows it to be integrated in other components to conform bigger components, and higher level of control loops.

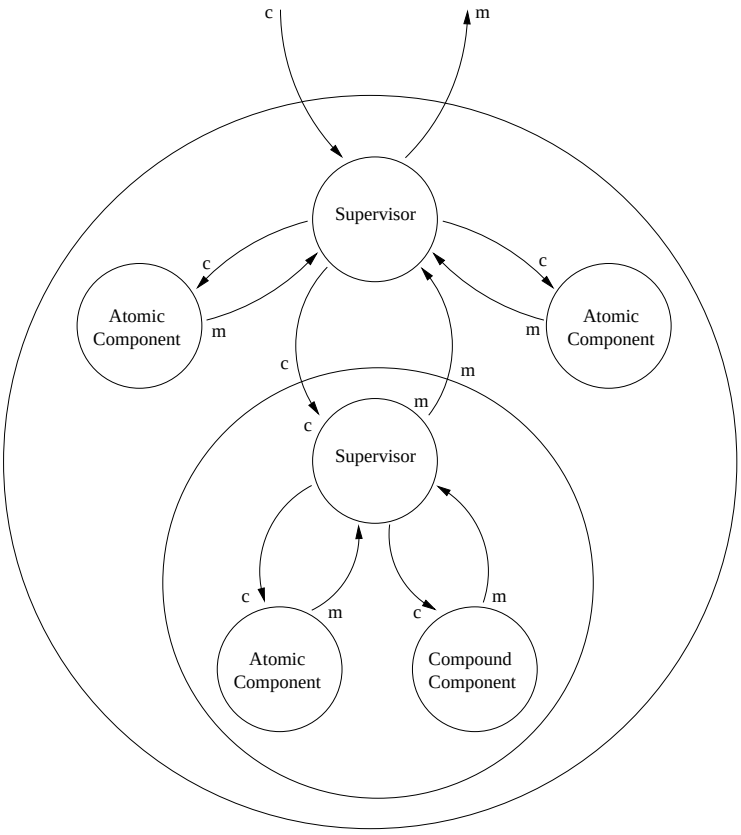

Figure 3.60: A *compound* components: a hierarchy of control.

3.6.2.1.1 Component Topologies: Internal and External Mapping A *compound* component establishes a topology among its *local* components by means of two kinds of port mappings:

- *Internal Mapping*: It is the set of all the port connections that internally configures a *compound* component. It is constituted by all the connection between *local* components´ ports. There are some examples of *internal mapping* in figures 3.57, 3.58 and 3.59. For example, in figure 3.59 the port connection between the output port **odometry/kinematics** of the component **motion controller**, and the input port of the same name in the *local* component **state reflector**.
- *External Mapping*: A *compound* component offers externally an interface of output and input ports. Not all ports of its *local* components may be accessible from

outside. When one output port or one input port offered by the external interface of a *compound* component corresponds to one port of one of its *local* components, this port has been externally mapped, and it is said to be part of the *compound* component´s *external mapping*. There are some examples of *external mapping* in figures 3.57, 3.58 and 3.59. For instance, in figure 3.59, the correspondence between the external input port **commands** of the *compound* component, and the input port of the *local* component **motion controller**.

The functionality of a *compound* component depends not only on the individual functionality of its *local* components, and the control flow that imposes the *supervisor*, but also on the topology of port connections the *local* components conform. This topology is established, as has been already commented, by means of *internal* and *external* mapping. There are no restrictions as to where and when a *supervisor* may establish both mappings, although it is obvious that a topology should be conveniently established after a *supervisor* commands all its *local* components to get into **running** state (figure 3.8).

Along the runtime life of a *compound* component the *internal* and *external* mapping that defines the topology that connects its *local* components might change. Bear in mind that to implement a specific functionality the *supervisor* of a given *compound* component might need the use of different component topologies in different states, possibly involving different component instances for each topology. Any change of topology is carried out by means of several possible actions: instantiation of new components, destruction of old ones, and configuration/deconfiguration of *internal* and *external* mapping. Out of all these operations only configuration/deconfiguration of *external* mapping must be externally driven. Take into account that the other operations can be completely hidden behind the *compound* component´s external interface.

If a *compound* component changes its internal topology, like, for example, substituting a component by another one. If such a change does not imply any modification in the *external* mapping, the component does not need to communicate such topology changes to any external component to which it may be connected, or to the *supervisor* of a *compound* component where it is included. But, if a change in the topology provokes a change of the *external* mapping of a specific input port, external components connected to this port should be disconnected, and connected once the *external* mapping in that port have been re-established. Changes of topology that involves changes in the *external* mapping of a *compound* component need external supervision, and this external supervision would be carried out by the *supervisor* of the *compound* component where the component is included. Obviously, if the *compound* component changing its *external* mapping were the top level component in a hierarchy of components, the external supervision would not be necessary.

The simplified C++ code of figure 3.61 resumes the algorithm that a *compound* component should follow when a change of topology must be carried out. Figure 3.62 contains a simplified C++ version of the algorithm the *supervisor* of a *compound* component should follow when one of its components asks for doing a supervised topology change. Observe like both algorithms, figures 3.61 and 3.62, use the default *new config*

observable variable (`newConfig`) to ask for supervision, and/or to confirm configuration changes to the *supervisor* of the component in the next upper level of the hierarchy of components. In the same way, the default *config* controllable variable (`config`) is used to receive answers and supervision from the upper level. Note that the algorithm of figure 3.62 is performed in transitions labelled as *nc* in the *default automaton*.

```
Component::mappingChange(Topology newMapping)
{
    suspendLocalComponents();
    disconnectLocalComponents(); // Internal mapping

    maps(internalMapping(newMapping));
    connectLocalComponents(newMapping); // Internal mapping

    if(isNeeded(externalMapping(newMapping)))
    {
        if(!isATopLevelComponent(this))
        {
            // Ask for supervision
            publish(config,externalMapping(newMapping));
            while{ waitForAnswer(controlPort(newConfig))!=doMapping };

            unmaps(externalMapping(newMapping)); // External mapping
            maps(externalMapping(newMapping)); // External mapping

            // Confirm mapping change completed
            publish(config,mappingDone);
        }
        else
        {
            unmaps(externalMapping(newMapping)); // External mapping
            maps(externalMapping(newMapping)); // External mapping
        }

    }
    runLocalComponents();
}
```

Figure 3.61: Changing mapping.

3.6.2.2 Exception Handling

Compound components can define its own exceptions too. The process of defining exceptions is exactly the same that the one used for *atomic* components. Observing the *default automaton* in figure 3.8 it is easy to realize that CoolBOT promotes the handling of exceptions at a local level first, and then if no solution is found, it demands supervision to the upper level. Thus, it is possible to distinguish two levels of exception handling at each level of control in a hierarchy of components:

- **Local Exception Handling**: A component, whether *atomic* or not, must deal with any exception first at a local level, running its associated handler in the way the *default automaton* establish (figure 3.8). If, as a result of the whole process of recovering from a exception, the exception persists because the component has not managed to cope with it, the component gets into **starting error** or **running error** states, and remains there waiting for external intervention from the *supervisor* in the upper level, if any, of the component hierarchy.

- **External Exception Handling**: Errors arriving to a supervisor from any of its *local* components must be managed first by this supervisor. They can be either ignored, propagated to higher levels in the hierarchy or handled as explained above.

When exceptions are handled within compound components several strategies can be envisaged, aside from the obvious re-instantiation of the faulty component. Let's suppose, for example, that we have several components that constitute equivalent alternatives for developing the same task, possibly using different resources, but offering the same external interface. Such components could be used alternatively to carry out a specific task and hence, a general strategy to cope with components in running error might be just *substitution* of one component with another one providing an equivalent interface and functionality. A complementary strategy may also be useful to avoid suspending a compound component whenever a member of the composition gets into running error. Equivalent components can be declared as *redundant* and executed concurrently or in parallel, so that if one of then fails, the others will keep the whole component running.

```
Component::supervisedMappingChange(Component component)
{
    suspendLocalComponentsConnectedTo(component);
    disconnectLocalComponentsConnectedTo(component); // Internal mapping

    if(has(externalMapping(component)))
    {
        if(!isATopLevelComponent(this))
        {
            // Ask for supervision
            publish(config,externalMapping(component));
            while{ waitForAnswer(controlPort(component,newConfig))!=doMapping };

            command(component,config,doMapping);
            while{ waitForAnswer(controlPort(newConfig))!=mappingDone };

            unmaps(externalMapping(component)); // External mapping
            maps(externalMapping(component)); // External mapping

            // Confirm mapping change completed
            publish(config,mappingDone);
        }
        else
        {
            command(component,config,doMapping);
            while{ waitForAnswer(controlPort(newConfig))!=mappingDone };

            unmaps(externalMapping(component)); // External mapping
            maps(externalMapping(component)); // External mapping
        }
    }

    connectLocalComponents(component);
    runLocalComponents();
}
```

Figure 3.62: Supervising a mapping change.

Then, for instance, when a *local* component instance gets into **running error** state, if a substitute exists, the *supervisor* will create an instance of it to carry out a substitution and keep the *compound* component working. The erroneous instance might be put into a queue of instances to be recovered. Instances in that recovery queue might be restarted periodically to check out if the running error persists. Imagine that there were a deadline for each instance in this recovery queue, so if the deadline expires the instance is deleted from the queue and destroyed. Otherwise, if any of them were recovered, the previous situation before its substitution could be restored. In that case, the restored component will return to occupy again its place in the system, and the substitute would be suspended and driven to **dead** state for its destruction.

In case of a *local* instance in a running error that could not be substituted, it might be added to the recovery queue previously mentioned. If its deadline in the queue were reached then the instance would be retired from the queue and destroyed. This might provoke the whole compound component to go to **running error**, or not, depending on its functionality.

An error that would need a special treatment is when a component hangs during execution. In such a situation, it could not attend its control and monitoring ports, turning it uncontrollable. When this exception were detected the component would be destroyed, this time using an operating system call like `kill()`, and obviously, it would not be added to the recovery queue.

3.7 Distributed Components

CoolBOT components reside in specific computers or machines when they are instantiated and executed. Frequently, the functionality of a component which runs in one machine may be needed in other computer elsewhere in a computer network. CoolBOT provides a mechanism based on "mediator" components [Gamma et al., 1995] called *proxy components*, and network processes called *CoolBOT servers*, to allow components to be accessible through a computer network.

3.7.1 Proxy Components

CoolBOT components inter communicate by means of their external interface of output and input ports when they are involved in port connections. At last term, output and input ports rely on the basic ICC mechanisms already introduced in section 3.5 to carry out the process of sending and receiving data between components in the form of port packets. As it was commented in the mentioned section, such ICC mechanisms just inter communicate components in the same machine, and more specifically, in the same process where the threads that constitute components run, and share resources.

One of the design principles considered during CoolBOT design and development was the transparency of communications among remote components (the "distributed" principle of section 2.4.1 in chapter 2). That principle claims that the inter

communication between components residing in different machines should be functionally equivalent to the inter communication between components running in the same machine in the same process. Consequently, components should use the same primitives to interact indistinctly with local and remote components. Such primitives are the ICC mechanisms used implicitly by the different typologies of output and input ports, that components use to send and receive port packets. Tables 3.3 and 3.4 enumerate respectively every output and input port typology, and which type of basic ICC mechanisms they use internally.

CoolBOT uses "mediators" [Gamma et al., 1995] components called *proxy components* to allow components to interact with remote components in a transparent way, through the same output and input port external interface they use to interact with local components. A *proxy component* is the representative of another component in a computer network. They are also CoolBOT components and participate of the same defaults and features that any other CoolBOT component.

3.7.1.1 Component Attachment

Figure 3.63 helps to illustrate graphically the rationale which is behind *proxy components*. Any component in a machine that should be accessible via a computer network must be attached to a *proxy component*. This process of attaching implies several actions.

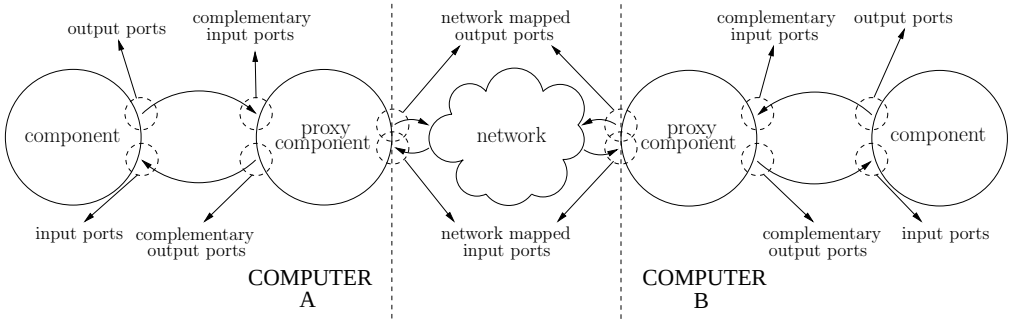

Figure 3.63: *Proxy components*.

- **Interface Adaptation**: *Proxy components* adapt their external interface of output and input ports to accommodate the external interface of the components to which they are attached. It is said that the *complementary input port* of an output port is an input port that accepts the same type of port packets that the output port, and whose typology is compatible enough to form a port connection with it. On the same terms, it is said that the *complementary output port* of an input port is an output port that accepts the same type of port packets that the input port, and whose typology is compatible enough to form a port connection with it. The complementary output and input ports corresponding to each one of the different types of output and input ports provided by the framework are indicated in figures 3.27 and 3.28.

Proxy components adapt their external public interface of output and input ports in such a way that they have, respectively, a complementary input and output port corresponding to each public output and input port of the component to which they are attached. It is said that during component attachment *proxy components* acquire the *complementary interface* of the component to which they are attached, figure 3.63 clarifies this concept. Having a complementary interface allows *proxy components* to establish port connections with any of the output and input ports of the component they act for the sake of.

- **Network Mapping**: *Proxy components* maps each one of the output and input ports of its complementary external interface into what is called, respectively a *network mapped input port* or a *network mapped output port*. To support network mapped ports, *proxy components* make use of the widely known **TCP/IP socket API** [Stevens, 1998]. Thus, a *network mapped output port* is the correspondence of an input port with a TCP/IP socket, when this socket is connected to a remote TCP/IP port, port packets received via the input port are sent through the TCP/IP socket. Analogously, a *network mapped input port* is the correspondence of an output port with a local TCP/IP socket which accepts new TCP/IP connections, and receives incoming port packets via the network connections that have already been established. Any port packet received through the TCP/IP socket is sent out through the output port to which the socket is associated. *Network mapped output ports* correspond to TCP/IP "client" sockets [Stevens, 1998] for sending outgoing port packets. On the other side all the *network mapped input ports* of a specific *proxy component* correspond to a TCP/IP "server" socket [Stevens, 1998] associated to a given TCP/IP port listening for incoming port packets and/or for accepting the creation of new TCP/IP connections. Figure 3.63 helps to show graphically the concept of network mapped ports.

  Finally, a *network mapped port connection* is defined as a *network mapped output port/input port* pair formed by a *network mapped input port* and a *network mapped output port*, such that the complementary output and input ports they map, are compatible enough to form a port connection.

- **Network Registration**: *Proxy components* need a TCP/IP port where to map all the input ports of its complementary external interface of input ports. TCP/IP ports are assigned by a network process called *CoolBOT server*. *CoolBOT servers* are explained in more detail in section 3.7.2, two of them appears in figure 3.65. Basically, they assign TCP/IP ports for *proxy components* in a specific machine, and at the same time they keep a data base registering all the proxies running in that machine, and which TCP/IP port they have assigned. Thus, for *proxy components*, *network registration* is referred to as the action of asking the local *CoolBOT server* for getting assigned a TCP/IP port, and being registered in its local data base. In the process of network registration *proxy components* provide also a name that will be the identifier that remote components will use to localize it in the local machine, this name is called its *network name*.

3.7.1.2 Functionality

The functionality of *proxy components* is resumed in figure 3.64 that depicts the *user automaton* corresponding to every *proxy component*. Take into account that the figure only shows the *user automaton*. Consult figure 3.8 to know how it fits inside the *default automaton*. Neither does the figure show arrows corresponding to transitions from the **main** user state to default automaton states. As shown by this figure, the *user automaton* of *proxy components* is not very complex, it consists of just one state called **main**.

At instantiation time, *proxy components* adapt its external interface of output and input ports, in order to have the complementary external interface that fits exactly to the external interface of the components to which they are attached.

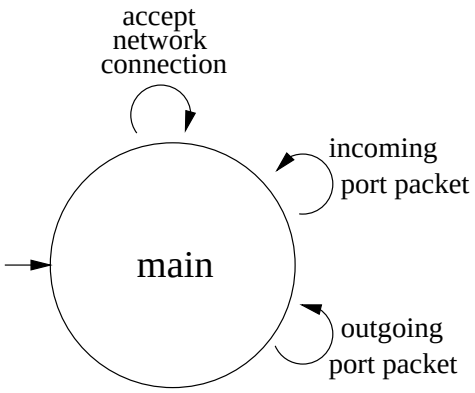

Figure 3.64: *Proxy components*: the *user automaton*.

Once *proxy components* are set in execution, they get into **starting** state where they ask their local *CoolBOT servers* for getting assigned a TCP/IP port. At the same time, they get registered in the internal data base of their corresponding local *CoolBOT server*. Then, at the same state, they map each one of their output and input ports into network mapped input and output ports. Finally, if everything goes well, they get to **ready** state, where they wait idle for being commanded to get into the *user automaton* for starting operation.

As soon as *proxy components* get into the *user automaton*, entering into the **main** state of figure 3.64, they start operation where they do mainly three tasks, that correspond exactly to the transitions that appear in figure 3.64:

1. **Accept network connections**: Remote *proxy components* may ask for establishing network connections. Network connections are demanded to map on them a port connection, between the components that the *proxy components* represent. If output and input ports at both ends in both components are compatible, a port connection is mapped. In this moment, the *proxy component* establishes a port connection with the component to which is attached using its complementary external interface. In this manner and from that moment on, incoming port packets via that mapped port connection can be adequately sent to the component the proxy is attached to.

2. **Receive incoming port packets**: Every port packet received via any of the network mapped input ports involved in a network mapped port connection, is sent directly through the output port it maps. In turn, this output port drives the port packet to the corresponding input port of the component to which the

proxy is attached. Port packets are received through the network packed in the form of strings of bytes. In section 3.6.1.2 it was commented that port packets are packed and unpacked in strings of bytes in a machine-independent manner using the **XDR** library [Bloomer, 1992]. *Proxy components* unpack incoming "packed" port packets that, then, are re-sent through the corresponding output ports towards the attached component.

3. **Send outgoing port packets**: Every port packet received through any of the input ports that conform the complementary external interface of *proxy components*, is sent out through its corresponding network mapping output port that should take part in a network mapped port connection with a remote *proxy component*. Port packets are packed in strings of bytes just before being sent through the network. As commented previously, they are packed in a machine-independent manner using the **XDR** library.

Proxy components also associate watchdogs with network connections that may provoke local exceptions. Timeouts of such watchdogs are specified when the *proxy components* are attached to their corresponding components. In the case of a watchdog violation, as any other CoolBOT component, *proxy components* try to recover locally from that violation, doing several attempts of a protocol of "sending something and get an answer", to confirm that the other part in the network connection is working correctly. Take into account that this may happen frequently if part of the network uses, for instance, a wireless infrastructure. Obviously, like any other component, if it is not possible to recover from a watchdog violation, the proxy will enter in **running error** state, and will wait for external intervention, usually from the *supervisor* of the *compound component* where it might have been included.

An important functionality of *proxy components* which is not shown in its *user automaton* in figure 3.64 is how to establish a port connection with a remote proxy. Bear in mind that in the figure it is only shown a transition to accept remote network connections where port connections will be mapped, but how does a *proxy component* ask for a connection?. This is a functionality of *proxy components* that does not appear in the *user automaton* in the figure, because the operation of doing a remote connection is a method of the component C++ class which is provided by CoolBOT, and may be called by the component itself or by its supervisor. This method is usually called through the component to which the *proxy component* is attached.

It is necessary to make here some comments about how *proxy components* deal with shared connections (section 3.5.2.6). Remember that a shared connection involves a shared output port (*OShared*) and a shared input port (*IShared*), and that the basic ICC mechanisms used are SSW, RSR, SSR and RSW as shown in tables 3.3 and 3.4. As explained in section 3.5.1, the main feature of these basic ICC mechanisms is that they share a port packet called "shared packet" which resides in the shared output port. As was commented, this packet is shared by all components involved in a shared connection with the output port. How does a *proxy component* publish through the network the port packet involved in a shared connection, if the component to which it is attached has a shared output port?. The approach used in CoolBOT is

to leave this decision to the developer of the shared packet. As it was also explained in sections 3.5.1 and 3.5.2.6, port packets involved in shared connections define a set of writing operations and a set of reading operations. The idea is that out of the writing operations some of them oblige the packet to be sent through the network, if there is a *proxy component* connected to the output port where the shared packet resides. So, *proxy components* will do a copy of the shared packet when any of these "network-forced-transfer" operations is signaled through a shared connection which has been mapped into the network. If any other writing operation is signaled the proxy does nothing.

In general, *proxy components* represent components to which they are attached in the network. Thence, once attached it is possible to establish remote port connections with remote *proxy components* residing in other computer. To do so, it is necessary to know the IP address or DNS name of the remote machine, and the *network name* of the remote *proxy component* as it was registered in its local *CoolBOT server*. With this information it is necessary to ask the remote *CoolBOT server* for the TCP/IP port assigned to the remote *proxy component*. Then, knowing the TCP/IP port it is possible to connect directly to the remote *proxy* and establish a network mapped port connection. On the other side, *proxy components* listen continuously to the TCP/IP port they were assigned, in order to accept connections with other remote *proxy components*. As to port packets, *proxy components* act for the sake of the components to which they are attached. In such a way that, all port packets coming via the network addressed to the component are routed to its *proxy component* that, re-sends them to the component using its complementary output ports. Thus, for a component there is no difference between receiving a port packet from a component residing in the same machine, or coming from a component running in a remote machine. In the same manner, any port packet addressed to a remote component and issued by the component, is received by the *proxy component*, that re-sends the port packet through the network in case that there were a port connection established with a remote machine.

3.7.2 CoolBOT Servers

A *CoolBOT server* is a process that runs in any machine or computer where there are CoolBOT components. Figure 3.65 depicts graphically the concept.

The main goal of *CoolBOT servers* is to provide the following services to *proxy components*:

- **TCP/IP Port Allocation/Deallocation**: In the process of component attachment, local *proxy components* register themselves with the *CoolBOT server*. Usually, the server selects randomly a TCP/IP port available in the underlying operating system, and assigns them to *proxy components*. Obviously, when a *proxy component* is detached from a specific component, the TCP/IP port is deallocated.

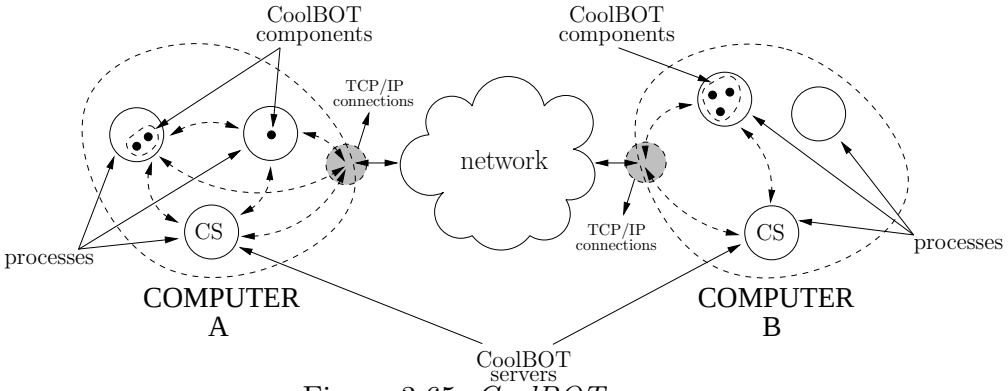

Figure 3.65: *CoolBOT servers.*

- **Registration/Deregistration of *Proxy Components***: Besides of assigning TCP/IP ports to *proxy components*, the *CoolBOT server* stores them in an internal data base. Concretely, it stores the correspondences between local *proxy components* and TCP/IP ports they have been assigned. On the other side, in the process of detachment of a *proxy component*, it gets de-registered from the *CoolBOT server*, i.e., it is deleted from the internal data base, and as commented in the previous paragraph, its TCP/IP port is released.

- **Look-Up Operations**: Remote *proxy components* need to know the TCP/IP ports at which the local *proxy components* are listening for network connections. *CoolBOT servers* accepts requests from remote *proxy components* to provide them with this information. To do so, it looks up the information in its internal data base. A remote *proxy component*, once found out the TCP/IP port corresponding to the local *proxy component* to which it wants to connect, establishes a network connection directly with it. From that very moment, all port packets interchanged between the components attached at both proxies are sent and received via the network transparently by means of them (see figure 3.63).

3.8 Scopes, Objects and Class Methods

CoolBOT supports all the concepts and abstractions presented along this chapter in the form of a hierarchy of C++ classes that conforms a namespace called `CoolBOT`. This hierarchy of classes provides a rich set of objects together with their methods to operate on components as operative units, and to program them internally.

Looking at programming languages the concept of scope is omnipresent. Having a look to C++, for example, we can find several scopes: global scope, the main function scope, global function scope, namespace scope, class scope, class member function scope, etc. Each scope allows the programmer the use of a specific subset of the constructs supported by the whole programming language. For instance, think about the scope of static class member functions in C++, where it is not possible to utilize the `this` pointer. Clearly, for each programming language several different scopes can be distinguished, and usually these different scopes establish a hierarchical principle

of organization. CoolBOT is a C++ framework where three particular scopes can be clearly distinguished:

- **Top Level Scope**: For any CoolBOT application this is the scope where the components situated at the top level of the hierarchy of components that constitute the application are instantiated and executed. From a user´s point of view it corresponds to the `main` function of the application.
- **Compound Component Scope**: This is the scope delimited by the declaration and definition of a *compound* component. From a user´s point of view it corresponds to a C++ class that implements a *compound* component.
- **Atomic Component Scope**: Analogously to the *compound component scope*, this is the scope delimited by the declaration and definition of an atomic component. From a user´s point of view it corresponds to a C++ class that implements an *atomic* component.

CoolBOT maps these three scopes in scopes supported by C++, evidently in any of this scopes any of the constructs and abstraction provided by C++ are also valid. Finally, in table 3.10 we resume the most important objects and associated methods provided by CoolBOT, and in which scopes they can be utilized.

Objects and Methods		
Object	Scopes	Operation
Automaton States	Atomic & Compound & Top Level	Run entry sections, run exit sections, run transitions, get the current state.
Output Ports	Atomic & Compound & Top Level	Instantiate output ports, establish and de-establish port connections, ICC sender side mechanisms, runtime verification of output port types, reconfigure output ports, and destroy output ports.
Input Ports	Atomic & Compound & Top Level	Instantiate input ports, ICC receiver side mechanisms, runtime verification of input port types, reconfigure input ports, mask/unmask input ports, mask/unmask controllable variables, and destroy input ports.
Timers	Atomic & Compound & Top Level	Instantiate timers, associate timers with *ITick* input ports, set timers, start timers, stop timers, and destroy timers.
Watchdogs	Atomic & Compound & Top Level	Instantiate watchdogs, configure watchdogs to supervise input ports (whatever type they have), associate callbacks with watchdog violations, start watchdogs, stop watchdogs, and destroy watchdogs.

continued on next page

continued from previous page

Objects and Methods		
Object	Scopes	Operation
Port Threads	Atomic & Compound & Top Level	Instantiate port threads, associate port threads with sets of input ports, run port threads, run port threads, suspend port threads, un-launch port threads, wait for port threads, and destroy port threads.
Exceptions	Atomic & Compound & Top Level	Instantiate exceptions, configure exceptions, throw exceptions, un-throw exceptions, operations to change the process of exception recovery in the default automaton, and destroy exceptions.
Components	Compound & Top Level	Instantiate components, configure components, control components (through their *control* ports), observe components (through their *monitoring* ports), connect *local* components (*internal mapping*), map external ports (*external mapping*), destroy components.
Proxy Components	Compound & Top Level	Instantiate *proxy components*, attach *proxy components*, detach *proxy components*, connect to remote *proxy components*, disconnect from remote *proxy components*, destroy *proxy components*.

Table 3.10: Scopes, objects and methods.

Chapter 4

Using CoolBOT

In this chapter some examples of simple robotic systems will be shown in order to illustrate the use of the framework for building robotic systems. It has not been devised to show in detail all the features and possibilities that CoolBOT provides, but to give a general vision about the methodology of constructing systems using CoolBOT.

4.1 Introduction

This chapter tries to answer several questions. Given a component that we must build endowing it with a specific functionality, which type of ports should it use?. There are several typologies of output and input ports with their own functionalities. Output and input ports take part into the external interface that components offer, so the type of output and input ports chosen at design-time has important effects in how components can be inter connected and used. Section 4.2 contains a discussion on which decisions might be made about which port types should be used in a specific design.

The next section, section 4.3, illustrates the use of CoolBOT in robotics, concretely implementing a reactive example in mobile robotics. The example is a typical one used in several well-known books [Arkin, 1998] [Murphy, 2000] to explain the reactive paradigm.

The following section, section 4.4, will continue describing how, from a reactive level and a set of developed and tested components implementing some behaviors, it is easy to integrate them using the constructs and abstractions that CoolBOT provides in order to make the robot perform a more complex task.

In section 4.5 it is proposed how a more formal description for tasks could be built using CoolBOT. This formal task approach uses some constructs of process algebra [Lyons and Arbib, 1989] [Lyons, 1990] [Košecká et al., 1997] to integrate components.

4.2 Which Port Type should be used?

In some circumstances, it is evident for a component which type of output or input port should be used for communication with other components. Some types of output and input ports have a clear functionality as, for instance, priority output and input ports, or tick output and input ports. But, in multiple occasions, it may not be clear which type of output or input port should be used. Obviously, in such cases, to make design decisions several questions should be taken into account. What does the component?. Does it operate driven by the port packets it receives through its input ports, or on the contrary, does it work asynchronously respect to its input ports?. Is it periodic in its operation?. Must it keep a frequency of operation?, is that frequency high?. Is it a producer of data for multiple components?, or, on the opposite, is it mainly a consumer of data from other components?. All previous questions guide design decisions oriented to chose the type of output and input ports that a component will offer as its external interface. Obviously the basic ICC mechanisms each type of output and input port use internally to carry out its functionality have a lot to do with such decisions. This section is addressed to put some grounds to support design decisions about the type of output and input ports that components offer through their external interfaces.

There is a common situation pattern where the election of the type of output and input ports can introduce significant computational costs in component inter communication. This is the well known pattern of "one producer and multiple consumers" illustrated in figure 4.1. This configuration of components usually involves a component, the producer, that produces data and publishes them in the form of port packets through one of its output ports. Other components, the consumers, that rely on those data in order to operate, access them establishing port connections with the output port through which the producer publishes those data. Depending on the number of consumers, on the length in bytes of the port packets emitted by the producer, and on the type of port connections involved, the cost of communications can be significant enough to prefer some output and input port types instead of others.

For the configuration of components in figure 4.1 several experiments have been carried out in order to measure the communication costs that imply the use of different types of port connections that might be used to implement the one-producer-multiple-consumers topology displayed in the figure. Three types of port connections has been considered for benchmarking: fifo connections, poster connections, and shared connections. For each type of port connection, it has been measured the cost of each ICC mechanism at both sides of the connections, the sender side, and the receiver side. The measurements have been performed for configurations combining one producer with 1, 5, 10, and 50 consumers, and for port packet lengths of 24 bytes, 128 bytes, 1 kbyte, 50 kbytes and 100 kbytes. To carry out such measurements the topology of figure 4.1 has been reproduced, in an application running in an Intel Pentium IV at 1.4 Ghz with 128 Mbytes of RAM, under a GNU/Linux Mandrake 9.1 distribution, and Microsoft Windows XP professional. The producer in all experiments was endowed with a working period of 100 milliseconds. The experiments were also tested at other different periods, concretely at 0, 10, 50 and 200 milliseconds, there were no signifi-

cant differences for both operating systems for working periods above 50 milliseconds. Results were different for periods under 50 milliseconds. Take into account that the closer the working period of the producer to the quantum of time assigned by the operating system scheduler (10-15 milliseconds in Windows, and 20 milliseconds for GNU/LINUX), the higher the synchronization costs of the different ICC mechanisms due to the overhead of the operating system's thread dispatching mechanism. Moreover, if the working period gets smaller than this quantum of time, the synchronization costs become really dominant caused by this overhead introduced by the operating system. To minimize the influence of the differences between the models of thread scheduling present in both operating systems, the components in all experiments (the producer and the consumers) have been executed at the same policy priority with the same priority level, concretely at *normal* policy with priority 8 (see figure 3.3 in section 3.2.4 for more information about priorities).

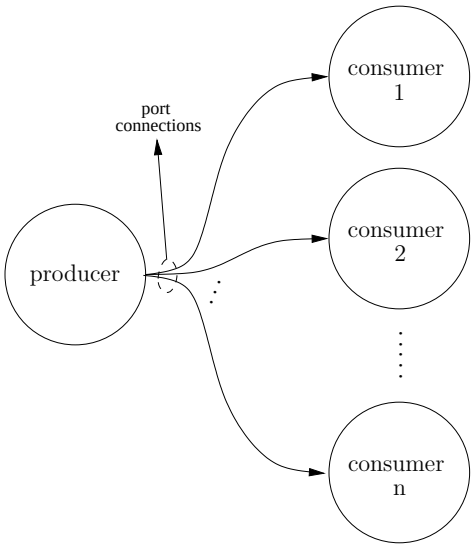

Figure 4.1: One producer of data and multiple consumers.

Tables in figure 4.2 show the results obtained for fifo connections in both operating systems. Tables in figure 4.3 contain the measurements corresponding to poster connections, and figure 4.4 shows the tables containing the results for shared connections. Time measurements were taken differently in both operating systems. In GNU/Linux was used the function `gettimeofday()` in `time.h` which has a resolution of microseconds. For Windows the pair of Win32 API functions `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` were utilized. Used together they allow measuring time with a resolution under microseconds. In particular, in the machine where the experiments were performed the resolution obtained using this pair of functions was of about 279 nanoseconds. We considered that at least a resolution of microsecond was enough to illustrate the differences between the three types of port connections under consideration.

The first interesting observation that comes up from the measurements of these three different types of port connections is that the measured costs for each ICC mechanism in each operating system are similar. This result is to a certain extend surprising due to the differences between the thread models used by each operating system, and also due to the fact that the libraries used by CoolBOT on each operating system in order to achieve multithreading are also distinct. A priori, we were expecting bigger discrepancies, in spite of having CoolBOT coded uniformly in both operating systems.

As a second meaningful observation, the measurements confirm the internal design given to each one of the types of connections. Thus for fifo connections, sender side

FIFO CONNECTIONS - GNU/LINUX - 100 MSECS.

CONSUMERS	ICC MECHANISMS		PORT PACKET LENGTHS				
			24 bytes	128 bytes	1 kbyte	50 kbytes	100 kbytes
1	AS	min	0.007	0.004	0.004	0.004	0.004
		max	0.055	0.119	0.095	0.078	0.037
		mean	0.01166	0.00745	0.00728	0.00733	0.00757
		σ	0.00952	0.00912	0.00903	0.00916	0.00904
		values	10000	10000	10000	10000	10000
	PR	min	0.001	0.001	0.001	0.001	0.001
		max	0.003	0.080	0.003	0.044	0.003
		mean	0.00198	0.00204	0.00189	0.00183	0.00189
		σ	0.00020	0.00082	0.00032	0.00070	0.00034
		values	10000	10000	10000	10000	10000
5	AS + ASCs	min	0.036	0.021	0.023	0.183	0.498
		max	0.209	0.681	0.182	0.397	0.796
		mean	0.04133	0.02492	0.02751	0.19870	0.52627
		σ	0.01454	0.01578	0.01441	0.01853	0.02710
		values	10000	10000	10000	10000	10000
	PR	min	0.001	0.001	0.001	0.001	0.001
		max	0.004	0.003	0.003	0.003	0.003
		mean	0.00256	0.00209	0.00196	0.00192	0.00185
		σ	0.00022	0.00008	0.00010	0.00021	0.00010
		values	50000	50000	50000	50000	50000
10	AS + ASCs	min	0.074	0.042	0.050	0.474	1.076
		max	0.963	0.364	0.371	0.891	1.780
		mean	0.07923	0.04669	0.05312	0.49233	1.14871
		σ	0.01930	0.01756	0.01756	0.02241	0.06998
		values	10000	10000	10000	10000	10000
	PR	min	0.001	0.001	0.001	0.001	0.001
		max	0.004	1.261	0.003	0.018	0.026
		mean	0.00260	0.00210	0.00194	0.00196	0.00201
		σ	0.00016	0.00009	0.00007	0.00011	0.00015
		values	100000	100000	100000	100000	100000
50	AS + ASCs	min	0.376	0.226	0.258	2.870	6.008
		max	26.740	53.854	27.502	58.623	105.013
		mean	3.24490	3.24054	3.39544	7.76583	13.24085
		σ	0.88558	1.34669	1.09989	2.56269	3.77843
		values	10000	10000	10000	10000	10000
	PR	min	0.001	0.001	0.001	0.001	0.001
		max	0.189	0.330	7.909	0.756	1.811
		mean	0.00328	0.00329	0.00338	0.00339	0.00391
		σ	0.00004	0.00005	0.00030	0.00015	0.00019
		values	500000	500000	500000	500000	500000

FIFO CONNECTIONS - WINDOWS - 100 MSECS.

CONSUMERS	ICC MECHANISMS		PORT PACKET LENGTHS				
			24 bytes	128 bytes	1 kbyte	50 kbytes	100 kbytes
1	AS	min	0.01760	0.00447	0.00419	0.00419	0.00447
		max	0.31903	0.04861	0.23467	0.06593	0.14834
		mean	0.02162	0.00580	0.00557	0.00538	0.00587
		σ	0.00657	0.00382	0.00452	0.00391	0.00413
		values	10000	10000	10000	10000	10000
	PR	min	0.00196	0.00196	0.00196	0.00168	0.00168
		max	0.00279	0.12264	0.05029	0.29305	0.12795
		mean	0.00218	0.00199	0.00196	0.00197	0.00197
		σ	0.00016	0.00139	0.00048	0.00296	0.00126
		values	10000	10000	10000	10000	10000
5	AS + ASCs	min	0.03101	0.02207	0.02458	0.18634	0.48610
		max	0.27964	0.08325	0.20477	0.78725	0.79060
		mean	0.04540	0.02469	0.02731	0.20151	0.52904
		σ	0.00788	0.00476	0.00499	0.01419	0.01908
		values	10000	10000	10000	10000	10000
	PR	min	0.00168	0.00168	0.00168	0.00168	0.00168
		max	0.05448	0.05364	0.17656	0.21204	0.05503
		mean	0.00219	0.00201	0.00195	0.00185	0.00179
		σ	0.00009	0.00005	0.00002	0.00009	0.00006
		values	50000	50000	50000	50000	50000
10	AS + ASCs	min	0.06984	0.04498	0.05084	0.47185	1.05209
		max	0.57437	9.71325	0.46486	29.95548	2.34695
		mean	0.11794	0.08281	0.08829	0.55941	1.21253
		σ	0.06222	0.11019	0.05412	0.32688	0.16775
		values	10000	10000	10000	10000	10000
	PR	min	0.00168	0.00168	0.00168	0.00168	0.00168
		max	4.49079	0.10392	0.09750	0.09331	0.18578
		mean	0.00234	0.00203	0.00203	0.00196	0.00199
		σ	0.00027	0.00002	0.00001	0.00014	0.00012
		values	100000	100000	100000	100000	100000
50	AS + ASCs	min	0.28495	0.23103	0.26484	2.83584	5.98652
		max	2.44919	38.27833	6.90311	158.04048	21.20074
		mean	0.32068	0.25748	0.29004	3.00924	6.97317
		σ	0.06754	0.38578	0.09406	2.34611	0.92282
		values	10000	10000	10000	10000	10000
	PR	min	0.00168	0.00168	0.00168	0.00168	0.00168
		max	21.07642	21.98576	21.61364	21.14542	21.22560
		mean	0.00259	0.00245	0.00250	0.00267	0.00249
		σ	0.00225	0.00196	0.00239	0.00365	0.00307
		values	500000	500000	500000	500000	500000

Figure 4.2: Fifo connections: measurements in GNU/Linux and Windows. Working period of 100 milliseconds. Measurements in milliseconds.

ICC mechanisms are more costly that receiver side ICC mechanisms. Having a look at any of the tables of figure 4.2 it is easy to observe that the PRs (*passive receptions*) have a small cost that remains constant independently of the number of consumers involved in the connection, and the size of the port packets taking part into the communications. On the contrary, sender side mechanisms, AS (*active sendings*) and ASCs (*active sendings with copy*) get more costly, mainly as the number of consumers grows, and also as the port packet size increments. This is due not only to the cost of copying in the ASCs, but also to inter component synchronization whose cost grows with the number of components involved in the same fifo connection. Bear in mind that the minimum values for each ICC mechanism depicted on the different tables are the closest ones to the real cost of each ICC mechanism with minimum synchronization. All in all, the results of the experiment for fifo connections confirms that in this type of connections the sender is the side that assumes the most part of the cost in the communication between components.

As to poster connections, at first sight we get to a similar conclusion. The measurements confirm the design given to poster connections, and observing the tables of figure 4.3 we can notice that in general, the consumers perform the most costly operations in poster connections. Remember that PSs (*passive sendings*) do not make any port packet copy, they consist in just a swapping of pointers or references, and the signaling of the poster input ports involved in the same connection. Results confirm that but indicate also that when the number of consumers grows, synchronization costs increase also, even to a level where the cost performed at the server side is similar to the work carried out at the receiver side. This is due to the fact that when the sender emits a PS it must wait for the releasing of a critical section where several consumers (readers) might be accessing the poster, even in the case that the sender has only to perform a swapping of pointers. As to the receiver side, ARs (*active receptions*) has a cost which is mainly proportional to the size of the port packets. As a summary of the results obtained for poster connections, the results confirm the design given to poster connections, where, receivers carry out the most costly operations due to port packet copies, but also they unveil the significance of synchronization costs in communications. Thence, the more components are taking part into a connection, the more dominant synchronization costs get during inter component communications. And in the case of poster connections, it is the sender side the part which assumes the majority of the synchronization costs.

With respect to shared connections the measures shown in the tables of figure 4.4 provide results which are similar to the ones obtained for poster connections except for the fact that the sender side ICC mechanism involved in this type of port connection, the SSWs (*sender shared writings*), adds an additional cost of port packet copying to the synchronization costs incurred by the PSs. This is not strange because the shared packet designed for this experiment only implements a writing operation and a reading operation, and both of them make a copy of the shared port packet. Evidently this is why, in general, in shared connections the sender and receiver sides have the same synchronization costs than its corresponding sides in poster connections. Then to this cost it is necessary to add the cost of the different writing and reading operations im-

plemented by the shared port packet. Note that it is not the same to make a raw copy of a 100-kbyte port packet than to make a query to a shared data-base, or to a complex data structure shared by multiple components, imagine for example a data structure like the LPS (*Local Perceptual Space*) of Saphira [Konolige et al., 1997]. Thence depending on the different writing and reading operations implemented by the shared packet they may become dominant in terms of timing costs in the communications performed through shared connections.

A third important comment that comes up from the results obtained in these benchmarking experiments is that inter component synchronization is not for free. This is very clear observing the results collected for poster and shared connections where the timing costs for the sender side ICC mechanisms grow proportionally to the number of components involved in the same connection.

In addition, it is important to highlight again that in the experiments the producer and the consumers were executed at the same priority in order to reduce the influence of priorities in the measurements. Take into account that, for instance, giving a higher priority to the producer would reduce the mean cost of the sender side ICC mechanisms for all types of connections, although in this case the underlying operating systems could have some more influence in the results as they use different approaches to solve priority inversion problems (see references [Nichols et al., 1996] and [Bover and Cesati, 2001] for GNU/Linux, and [Richter, 1997] and [Solomon and Russinovich, 2000] for Windows).

There is a question we have not answered yet. Which one of the types of port connections under consideration is more convenient in this topology of one producer multiple consumers?. Which one should we use?. From the measurements we have collected for each type of connection we know they behave according to its internal design, although inter component synchronization costs are dominant when the number of components involved in the same connection is high. Anyway we think the following rules or recipes of design can be useful to guide design decisions about the external interface of components in this case:

- **High Frequency Producers**: A producer working at a high frequency that must provide data to multiple consumers, should consume as less time as possible doing communications, so poster connections looks like the correct choice for data that should be updated at high frequencies, because the sender side mechanisms in this type of connections are less costly. We have to take into account that if the number of consumers grows then to avoid the increasing of synchronization times in the producer, it should be assigned a higher priority than the priority at which its consumers run. Another form of reducing synchronization cost would be having the consumers running at working frequencies lower than the operating frequency at which the producer executes.

  It is not difficult to find high frequency producers in a robotic system. They correspond usually to components that collect information from sensors and publish it in some "normalized" way in order to make sensory information available to the rest of components that conform a system.

POSTER CONNECTIONS - GNU/LINUX - 100 MSECS.							
CONSUMERS	*ICC MECHANISMS*		*PORT PACKET LENGTHS*				
			24 bytes	*128 bytes*	*1 kbyte*	*50 kbytes*	*100 kbytes*
1	*PS*	*min*	0.007	0.004	0.004	0.005	0.004
		max	0.846	0.570	0.571	0.201	0.288
		mean	0.01191	0.00832	0.00878	0.01424	0.02173
		σ	0.01416	0.01161	0.01809	0.02723	0.05125
		values	10000	10000	10000	10000	10000
	AR	*min*	0.004	0.003	0.004	0.053	0.130
		max	0.008	0.005	0.021	0.158	0.158
		mean	0.00470	0.00366	0.00506	0.06864	0.14714
		σ	0.00048	0.00048	0.00049	0.00451	0.00483
		values	10000	10000	10000	10000	10000
5	*PS*	*min*	0.022	0.015	0.016	0.016	0.015
		max	0.213	0.194	0.198	0.508	0.983
		mean	0.02626	0.01781	0.01886	0.02151	0.02549
		σ	0.01683	0.01558	0.01594	0.04397	0.08711
		values	10000	10000	10000	10000	10000
	AR	*min*	0.002	0.002	0.003	0.047	0.115
		max	0.006	0.005	0.007	0.087	0.180
		mean	0.00473	0.00339	0.00476	0.07219	0.14222
		σ	0.00026	0.00016	0.00008	0.00397	0.00572
		values	50000	50000	50000	50000	50000
10	*PS*	*min*	0.043	0.028	0.030	0.031	0.031
		max	0.425	0.394	0.404	1.035	1.947
		mean	0.04677	0.03162	0.03330	0.03601	0.03883
		σ	0.02018	0.01911	0.01947	0.05331	0.10480
		values	10000	10000	10000	10000	10000
	AR	*min*	0.003	0.002	0.003	0.044	0.117
		max	0.596	1.259	1.265	1.508	2.585
		mean	0.00458	0.00365	0.00478	0.06971	0.14300
		σ	0.00030	0.00011	0.00019	0.00440	0.00573
		values	100000	100000	100000	100000	100000
50	*PS*	*min*	0.217	0.146	0.150	0.156	0.153
		max	36.162	104.137	28.177	58.892	55.403
		mean	3.46154	3.41774	3.40677	7.99262	13.63811
		σ	1.16199	1.63294	0.98278	1.96347	2.79372
		values	10000	10000	10000	10000	10000
	AR	*min*	0.002	0.002	0.003	0.038	0.098
		max	0.755	2.134	0.396	22.746	41.568
		mean	0.00588	0.00529	0.00667	0.09132	0.19383
		σ	0.00037	0.00045	0.00051	0.00808	0.01587
		values	500000	500000	500000	500000	500000

POSTER CONNECTIONS - WINDOWS - 100 MSECS.							
CONSUMERS	*ICC MECHANISMS*		*PORT PACKET LENGTHS*				
			24 bytes	*128 bytes*	*1 kbyte*	*50 kbytes*	*100 kbytes*
1	*PS*	*min*	0.00559	0.00447	0.00419	0.00419	0.00447
		max	0.06817	0.04386	0.04498	0.14443	0.22573
		mean	0.00761	0.00601	0.00610	0.01015	0.01644
		σ	0.00560	0.00484	0.00520	0.02002	0.04059
		values	10000	10000	10000	10000	10000
	AR	*min*	0.00531	0.00419	0.00559	0.05420	0.11929
		max	0.05559	0.05336	0.05839	0.23550	0.31596
		mean	0.00654	0.00480	0.00619	0.06532	0.13837
		σ	0.00116	0.00085	0.00113	0.00516	0.00752
		values	10000	10000	10000	10000	10000
5	*PS*	*min*	0.01872	0.01592	0.01481	0.01481	0.01509
		max	0.10029	0.08968	0.11845	0.43581	0.25590
		mean	0.02259	0.01840	0.01763	0.02527	0.03323
		σ	0.00735	0.00618	0.00674	0.02977	0.05353
		values	10000	10000	10000	10000	10000
	AR	*min*	0.00503	0.00447	0.00531	0.05503	0.11761
		max	0.18103	0.20561	0.33384	0.27825	0.62745
		mean	0.00668	0.00504	0.00634	0.06832	0.14584
		σ	0.00030	0.00007	0.00012	0.00315	0.00209
		values	50000	50000	50000	50000	50000
10	*PS*	*min*	0.03688	0.02905	0.02905	0.02989	0.02961
		max	0.90291	0.51263	4.91571	2.00109	4.67406
		mean	0.09058	0.07162	0.07481	0.21666	0.39184
		σ	0.06860	0.06071	0.08029	0.25342	0.52602
		values	10000	10000	10000	10000	10000
	AR	*min*	0.00391	0.00391	0.00447	0.04079	0.10672
		max	4.06057	3.24958	0.20142	5.44399	5.56691
		mean	0.00653	0.00533	0.00628	0.07079	0.15026
		σ	0.00017	0.00016	0.00016	0.00417	0.00550
		values	100000	100000	100000	100000	100000
50	*PS*	*min*	0.20058	0.14639	0.14806	0.15086	0.15030
		max	39.97072	4.62405	10.91312	9.31124	13.66542
		mean	0.38953	0.32739	0.33826	0.85253	1.58058
		σ	0.51546	0.33439	0.36633	1.31015	2.63345
		values	10000	10000	10000	10000	10000
	AR	*min*	0.00251	0.00223	0.00307	0.03911	0.00251
		max	169.82773	21.27198	23.94690	99.45538	104.33784
		mean	0.03747	0.01585	0.01783	0.08511	0.18848
		σ	0.02035	0.01011	0.01047	0.02942	0.08670
		values	499936	499935	499935	499912	499887

Figure 4.3: Poster connections: measurements in GNU/Linux and Windows. Working period of 100 milliseconds. Measurements in milliseconds.

- **High Frequency Consumers**: If the consumers must operate at high frequencies, it seems interesting to keep their working pace to avoid them the most costly part in communications. In this case it is evident that fifo connections are the most convenient choice, because the communication cost at the receiver side in this type of port connections is small, and depends mainly on the length of the port packets.

  Typical high frequency consumers are, for instance, components that control actuators in robotic systems. Imagine, for example, a component controlling the motor joints of a pan-tilt robotic camera, or the motors guiding a mobile robot, or the motor joints of a robotic arm.

- **Sharing of Complex Data Structures**: If multiple components should share a complex data structure which is big enough to make its copying costly in terms of time, or costly in terms of memory if multiple copies of it were kept by several components, the most rational choice would be the use of shared connections. Take into account that the cost in this type of connections not only depends on synchronization issues, but also on the writing and reading operations implemented by the shared packet, which should be in this case the shared complex data structure. Thus, choosing conveniently these operations in order to make them not too costly can make shared connections the better option.

  Using complex data structures shared by multiple components is not a rare situation in robotic systems. As an example, think, for instance, about the LPS (*Local Perceptual Space*) in Saphira [Konolige et al., 1997], which is a complex data structure shared by multiple components.

As a final comment, it is necessary to make emphasis in the fact that the results of these experiments are applicable to the rest of typologies of port connections which have not been studied for benchmarking, because all of them use the same types of basic ICC mechanisms to inter communicate components. Tick connections are the only exception, because they utilize the SS (*signal sending*) and SR (*signal reception*) ICC mechanisms. Note that by design the cost of this mechanisms is purely due to inter component synchronization, so they should increase as the number of components involved in the connections grows.

4.3 A Reactive Example

CoolBOT has been conceived to promote integrability, incremental design and robustness of software developments in robotics. In this section, a first example will be shown to illustrate how such principles manifest in systems built using CoolBOT. As a first example, it will be illustrated a very basic reactive level in a typical mobile robot. The example has been chosen from [Murphy, 2000] (chapter 4, page 107) in order to use a well-know example as a reference.

SHARED CONNECTIONS - GNU/LINUX - 100 MSECS.							
CONSUMERS	*ICC MECHANISMS*		*PORT PACKET LENGTHS*				
			24 bytes	*128 bytes*	*1 kbyte*	*50 kbytes*	*100 kbytes*
1	*SSW*	*min*	0.006	0.004	0.005	0.043	0.068
		max	0.160	0.042	0.226	0.619	0.579
		mean	0.01050	0.00825	0.00973	0.05776	0.10921
		σ	0.01204	0.00975	0.01030	0.03315	0.05987
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.003	0.002	0.003	0.065	0.136
		max	0.012	0.004	0.006	0.183	0.427
		mean	0.00382	0.00320	0.00465	0.07902	0.14525
		σ	0.00044	0.00040	0.00059	0.00358	0.00434
		values	10000	10000	10000	10000	10000
5	*SSW*	*min*	0.017	0.016	0.016	0.066	0.116
		max	0.213	0.188	0.193	0.853	1.257
		mean	0.02006	0.01790	0.01832	0.07587	0.12971
		σ	0.01715	0.01502	0.01545	0.04559	0.07930
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.002	0.002	0.003	0.057	0.118
		max	0.005	0.004	0.006	0.137	0.195
		mean	0.00383	0.00299	0.00445	0.08134	0.15675
		σ	0.00012	0.00007	0.00016	0.00223	0.00511
		values	50000	50000	50000	50000	50000
10	*SSW*	*min*	0.030	0.028	0.030	0.077	0.132
		max	0.415	0.383	0.389	1.779	2.821
		mean	0.03242	0.03052	0.03235	0.08573	0.14192
		σ	0.02056	0.01836	0.01883	0.05927	0.09306
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.002	0.002	0.003	0.057	0.105
		max	2.661	0.004	0.029	1.950	2.221
		mean	0.00394	0.00311	0.00450	0.08134	0.15350
		σ	0.00013	0.00007	0.00017	0.00259	0.00639
		values	100000	100000	100000	100000	100000
50	*SSW*	*min*	0.146	0.141	0.141	0.196	0.245
		max	44.015	43.818	58.598	32.290	55.302
		mean	3.29366	3.24404	3.43093	8.20397	13.68922
		σ	1.04410	1.06730	1.35748	1.80957	2.88766
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.002	0.002	0.003	0.041	0.097
		max	0.227	0.441	2.190	32.371	39.379
		mean	0.00436	0.00451	0.00658	0.09808	0.19990
		σ	0.00018	0.00018	0.00032	0.00623	0.01290
		values	500000	500000	500000	500000	500000

SHARED CONNECTIONS - WINDOWS - 100 MSECS.							
CONSUMERS	*ICC MECHANISMS*		*PORT PACKET LENGTHS*				
			24 bytes	*128 bytes*	*1 kbyte*	*50 kbytes*	*100 kbytes*
1	*SSW*	*min*	0.00587	0.00475	0.00559	0.06174	0.11817
		max	0.08632	0.14276	0.07878	0.23942	0.37044
		mean	0.00855	0.00621	0.00755	0.07634	0.13899
		σ	0.00624	0.00493	0.00510	0.03133	0.05059
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.00475	0.00419	0.00475	0.07375	0.13186
		max	0.11510	0.06230	0.14750	0.23774	0.32406
		mean	0.00504	0.00448	0.00590	0.08137	0.14997
		σ	0.00159	0.00083	0.00185	0.00500	0.00649
		values	10000	10000	10000	10000	10000
5	*SSW*	*min*	0.01676	0.01481	0.01620	0.08381	0.14359
		max	0.19081	0.09051	0.09051	0.33803	1.05851
		mean	0.02013	0.01714	0.01893	0.10786	0.17587
		σ	0.00751	0.00593	0.00618	0.02717	0.04964
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.00475	0.00419	0.00475	0.06397	0.12208
		max	0.22349	0.18913	2.97300	0.48693	6.96066
		mean	0.00539	0.00467	0.00612	0.08260	0.15108
		σ	0.00008	0.00008	0.00011	0.00207	0.00880
		values	50000	50000	50000	50000	50000
10	*SSW*	*min*	0.03045	0.02794	0.02961	0.09470	0.15617
		max	0.90626	2.55172	0.71517	4.13153	3.19733
		mean	0.08469	0.07021	0.07309	0.29117	0.50220
		σ	0.07199	0.07575	0.06180	0.25459	0.45931
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.00391	0.00391	0.00447	0.03883	0.08381
		max	0.18941	1.22278	4.00191	10.62258	6.51535
		mean	0.00498	0.00485	0.00590	0.07939	0.14801
		σ	0.00012	0.00008	0.00027	0.00275	0.01349
		values	100000	100000	100000	100000	100000
50	*SSW*	*min*	0.14499	0.14136	0.14136	0.21818	0.28607
		max	52.96818	62.15958	13.20391	16.19228	35.42713
		mean	0.35528	0.34071	0.34105	0.91344	1.79486
		σ	0.66125	0.73824	0.41547	1.26972	2.79441
		values	10000	10000	10000	10000	10000
	RSR	*min*	0.00251	0.00223	0.00335	0.03883	0.10169
		max	29.64846	22.51403	72.92770	57.78975	40.36044
		mean	0.02130	0.01104	0.01328	0.11406	0.22753
		σ	0.01065	0.00456	0.00529	0.05818	0.15509
		values	499884	499884	499885	499893	499892

Figure 4.4: Shared connections: measurements in GNU/Linux and Windows. Working period of 100 milliseconds. Measurements in milliseconds.

The objective of this section is to illustrate how a mobile robot can be endowed with different capabilities by means of integrating components in an incremental way using CoolBOT. Thus, initially the robot will only be able to avoid obstacles moving away from them, and finally it will end up with a "wanderer" conduct with obstacle avoidance.

4.3.1 An Avoiding Component

Along the examples we will be using the P2DX Pioneer robot of displayed in figure 3.41. To interface with it, the *Pioneer* component introduced in chapter 3 (section 3.6.1) will be utilized. Using that component, an atomic component, called *PF Avoiding* has been designed and built to endow this robot with the capability of avoiding obstacles. The external interface of public output and input ports of this component appears in figure 4.5. Output and input port types are indicated by their symbols in parentheses. Consult tables 3.3 and 3.4 for the symbols associated respectively to output and input ports. Tables 4.1 and 4.2 describe briefly each one of the public output and input ports the component offers. Non default observable variables (*period*, *distance* and *persistence*) and non default controllable variables (*new period*, *new distance* and *new persistence*) appear at the bottom of the figure, they are explained respectively in tables 4.3 and 4.4.

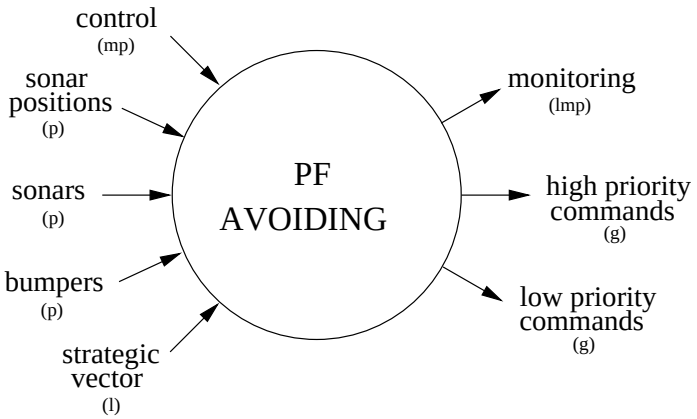

Figure 4.5: Component *PF Avoiding*: external interface.

The *PF Avoiding* component makes use of a **potential field** approach[Arkin, 1998] [Murphy, 2000] to perform obstacle avoidance using robot sonar readings as sensory information. In particular, for each sonar reading the component will associate a *repulsive potential field* that verifies the following equation:

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is always an *OLazyMultiPacket* for all components.
high priority commands	This is an *OGeneric* output port. Through this port the component sends commands to the mobile robot. It is connected to the homonym public input port of the *Pioneer* component. It transports high priority robot commands.
low priority commands	This is an *OGeneric* output port. Through this port the component sends commands to the mobile robot. It is connected to the homonym public input port of the *Pioneer* component. It transports low priority robot commands.

Table 4.1: Component *PF Avoiding*: public output ports.

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which controllable variables may be modified and updated. It is always an *IMultiPacket* input port for all components.
sonar positions	The *Pioneer* component publishes the positions in robot coordinates of each one of its sonar sensors. The *PF Avoiding* component uses this *IPoster* input port to know how many sonar sensors the robot has and their locations on the robot. It should be connected to the *Pioneer* component´s homonym public output port.
sonars	The *PF Avoiding* component disposes of this *IPoster* input port, in order to receive the sonar readings the *Pioneer* component publishes periodically through its *OPoster* public output port called *sonars*.
bumpers	This *IPoster* input port is used to receive periodically information about the status of the bumpers available in the mobile robot. Information about bumpers is published through the *Pioneer* component's public *standard* output port, so it should be connected to that output port.
strategic vector	This is an *ILast* input port through which an strategic vector can be provided to the robot.

Table 4.2: Component *PF Avoiding*: public input ports.

$$\overrightarrow{r_i}(\overrightarrow{p},\overrightarrow{p_i}) = \begin{cases} \overrightarrow{u}, & \text{if } d \leq d_{min} \\ \left(\frac{d_{min}}{d}\right)^2 \overrightarrow{u}, & \text{if } d_{min} < d < d_{max} \\ \overrightarrow{0}, & \text{if } d > d_{max} \end{cases} \tag{4.1}$$

such that, $\overrightarrow{p}$ is any position in robot coordinates; $\overrightarrow{p_i}$ is the i^{th} robot sonar reading in

Non Default Observable Variables	
Name	Brief Description
period	This observable variable indicates at which period operates the component.
distance	This observable variable is d_{max} in equation 4.1. It indicates the minimum distance at which the robot should start moving away from any detected obstacle.
persistence	This observable variable contains the time of persistence during which an strategic vector received through the *strategic vector* input port is valid.

Table 4.3: Component *PF Avoiding*: non default observable variables.

Non Default Controllable Variables	
Name	Brief Description
new period	Through this controllable variable it is possible to change the working period of the component. It updates the *period* observable variable.
new distance	Through this controllable variable it is possible to change how close the robot may get to any obstacle (d_{max} in equation 4.1). It modifies the *distance* observable variable.
new persistence	By means of this controllable variable the persistence of the strategic vectors received by the component can be modified. It updates the observable variable *persistence*.

Table 4.4: Component *PF Avoiding*: non default controllable variables.

robot coordinates as well; $d = |\overrightarrow{p} - \overrightarrow{p_i}|$ is the euclidean distance between $\overrightarrow{p}$ and $\overrightarrow{p_i}$; $\overrightarrow{u} = \frac{\overrightarrow{p}-\overrightarrow{p_i}}{|\overrightarrow{p}-\overrightarrow{p_i}|}$ is an unitary vector between $\overrightarrow{p}$ and $\overrightarrow{p_i}$; d_{max} is a threshold distance from $\overrightarrow{p_i}$ specifying the minimum distance at which repulsion starts taking effect; and d_{min} is the minimum distance from $\overrightarrow{p_i}$ where $\overrightarrow{r}$ saturates to $\overrightarrow{u}$.

The equation 4.1 returns a vector, $\overrightarrow{r_i}$, whose angle is the orientation that the robot should take to run away from the obstacle, and whose modulus takes values between 0 and 1, expressing which fraction of a maximum velocity should be commanded to the robot to runaway from the obstacle. The closer there is an obstacle, the faster the robot runs away from it. Obviously when the obstacle is too close, the modulus saturates to 1, that happens when $|\overrightarrow{p} - \overrightarrow{p_i}| \leq d_{min}$. On the opposite, an obstacle has no influence when it is beyond the distance threshold d_{max}. Figure 4.6 shows graphically an example of the repulsive potential field introduced by each sonar reading corresponding to equation 4.1.

Usually, a specific robot has several sonar sensors, and each one of them induces

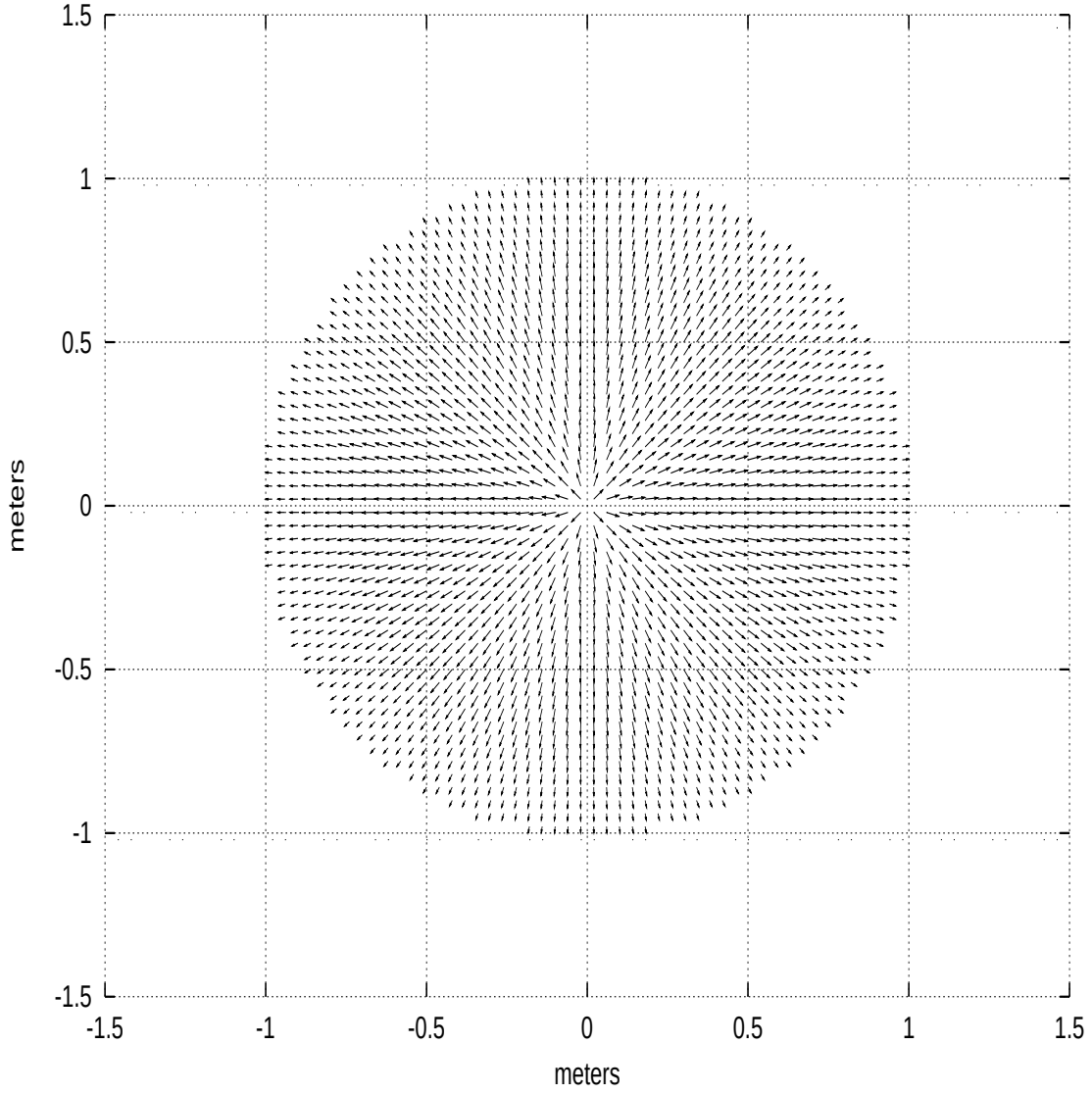

Figure 4.6: Component *PF Avoiding*: repulsive potential field.

a repulsive field. Thus, if there are n sonar sensors, there will be n $\overrightarrow{r_i}$ vectors exerted on the robot in the way shown in equation 4.1. Usually, they are combined following the next equation:

$$\overrightarrow{r}(\overrightarrow{p}) = \frac{1}{n}\sum_{i=1}^{n} \overrightarrow{r_i}(\overrightarrow{p}, \overrightarrow{p_i}) \tag{4.2}$$

which calculates the average of the repulsive vectors induced by all sensors.

The *user automaton* corresponding to the *PF Avoiding* component is exposed in figure 4.7. Transitions to *default automaton* states are not displayed. It consists of the following states:

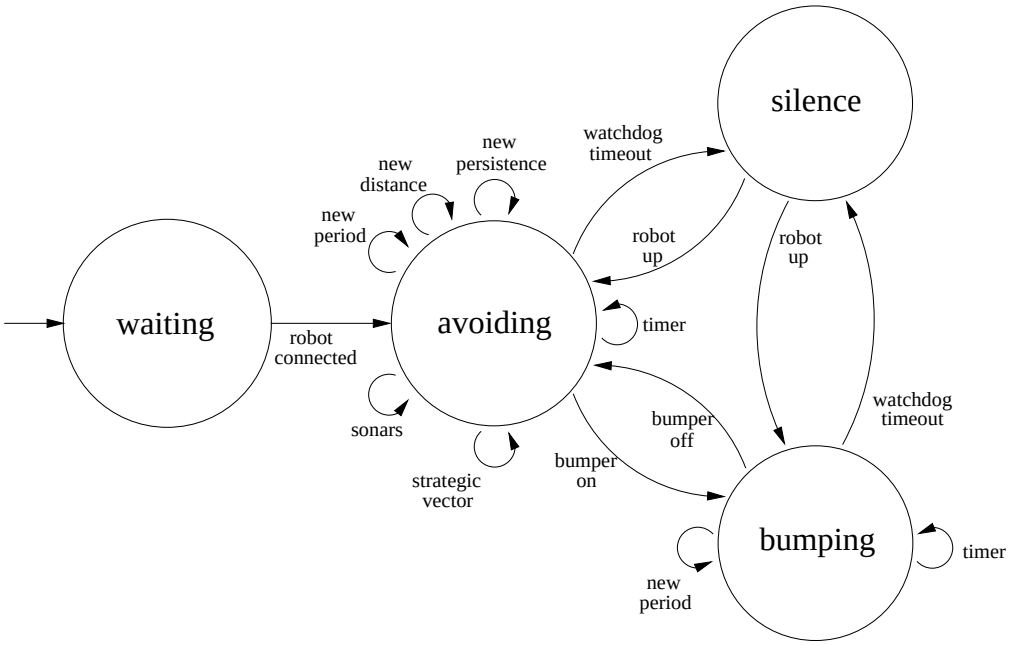

Figure 4.7: Component *PF Avoiding*: user automaton.

- **waiting**: This is the *user automaton* entry state of the *PF Avoiding* component. In it, the component remains idle, waiting until it gets connected to a *Pioneer* component instance, through its public input ports: *sonar positions*, *sonars* and *bumpers* (table 4.2 describes each one of them). Once connected the component transits to the **avoiding** state.

- **avoiding**: This is the main state of the *PF Avoiding* component. Several transitions are possible:

  - **new period**: The *PF Avoiding* component operates with a period determined by the *period* observable variable, which can be, in turn, updated by means of the *new period* controllable variable (transition **new period** in figure 4.7).
  - **sonars**: Through the *sonars* input port the *PF Avoiding* component receives individual robot sonar readings. The component accumulates the repulsive vector corresponding to each incoming sonar reading as they are received.
  - **strategic vector**: The *strategic vector* input port may be used by an external component to influence the behavior of the *PF Avoiding* component. If an strategic vector has been received through that input port, it stores the vector internally, and associates it with a time stamp. In the **timer** transition will be explained how this vector influences component´s behavior. Strategic vectors should verify that their moduli take only real values between 0 and 1.
  - **new persistence**: As commented above each time a strategic vector is

received, it is time-stamped. The component only memorizes internally the last one strategic vector received. This vector has associated a time of persistence, a time period during which it may influence the behavior of the component. Once this time expires, the strategic vector becomes obsolete, and it is ignored. This time of persistence is determined by the *persistence* observable variable, which may be modified and updated by means of the *new persistence* controllable variable.

- **new distance**: It is possible to change how close the robot may get to obstacles by means of modifying the *distance* observable variable. As indicated in table 4.3, this variable contains the value d_{max} of equation 4.1 which is used to calculate the repulsive vectors originated by each sonar reading. This observable variable is changed by means of updating the *new distance* controllable variable (see table 4.4). This component makes d_{min} in equation 4.1 equals to a half of the value of d_{max} ($d_{min} = \frac{d_{max}}{2}$).
- **timer**: With a periodicity determined by the *period* observable variable, this transition is triggered once per period. In general, each time this transition occurs the component will have accumulated several repulsive vectors corresponding to various sonar readings. Take into account that they are accumulated individually as they are received through the *sonars* input port. In general, sonar readings received in each period do not correspond to all the sonar sensors available in the robot. Thence, only a few sonar readings are available in each period. The *PF Avoiding* component only takes into account the sonar readings corresponding to the last period, so, it accumulates only the repulsive vectors corresponding to them. When the **timer** transition gets called the component averages these last-period repulsive vectors. Additionally, if there is an active strategic vector whose persistence time has not expired yet, then this vector is also averaged with the last-period repulsive vectors. Otherwise the strategic vector is ignored. The result of this average of vectors will be used to command the robot through the *low priority commands* output port: the modulus of the vector will be used to command the robot velocity, and its orientation to command the robot orientation.
- **bumper on**: Besides of receiving sonar sensor information in order to carry out obstacle avoidance, the *PF Avoiding* component monitors the status of the bumpers in the robot. If any of the bumpers gets activated the robot has crashed into an obstacle, in this case this transition drives the component to **bumping** state.
- **watchdog timeout**: The *PF Avoiding* component has a watchdog associated to the *bumpers* input port. Note from table 4.2 that this input port should be connected to the *standard* output port of the *Pioneer* component. Bear in mind that if the robot is operating correctly, the *Pioneer* component should sent periodically something through this output port with a period of 50 or 100 milliseconds, depending on how the robot has been configured. Thus, by monitoring the *bumpers* input port with a watchdog long enough,

it is possible to detect when the robot is "out of line", i.e., when the connection to the robot has fallen down. In this case the component transits to **silence** state.

- **bumping**: The *PF Avoiding* component gets into this state when the robot has collided with an obstacle. Just entering this state, the robot is stopped by sending a stop command through the *high priority commands* output port (table 4.1), that should be connected to the homonym input port of the *Pioneer* component. Take into account that a robot hitting an obstacle is an emergency situation, and the robot should be stopped immediately in order to avoid any harm to it. That is the reason of using an output port connected to the *Pioneer*'s *high priority commands* input port, commands sent through it have higher priorities, and therefore, will be commanded to the robot first. As soon as the component gets into **bumping** state and the robot is stopped, the *PF Avoiding* tries to separate the robot from the obstacle. To do so, velocity and orientation commands are sent periodically through the *low priority commands* output port in order to run away from the obstacle (**timer** transition). Once it is detected that the robot is not touching anything (**bumping off** transition) the component returns to **avoiding** state. Finally, if it happens to be a watchdog timeout in the *bumpers* input port, in the same manner that in the **avoiding** state, the component transits to **silence** state (**watchdog timeout** transition).

- **silence**: This state is reached if the communication with the robot is lost for a period of time long enough (the watchdog associated with the *bumpers* input port). After that, the component sits idle waiting until the robot "returns to live", i.e., the component starts receiving something from it again, in that case, the component transits to the state where it was previously, **avoiding** or **bumping** (**robot up** transitions).

4.3.2 The Avoiding Level

The combination formed by the *Pioneer* and the *PF Avoiding* components constitutes a minimum level of obstacle avoidance for a Pioneer robot. The interconnections between them are displayed in figure 4.8. When this component configuration is executed in our robot the observable behavior is that the robot remains still if there are no obstacles close enough. In the event that an obstacle gets closer, the robot will move away from it, trying to keep always a minimum distance between it and any obstacle (d_{max} in equation 4.1). Consequently, this behavior could allow a person to herd the robot, as it behaves in order to keep this minimum distance from obstacles.

Finally, observe that in this configuration of figure 4.8, the *strategic vector* is not being used, the behavior of the robot is completely determined by the *PF Avoiding* component. Next section will introduce a new component called *Strategic PF* that will make the robot behave in a more elaborated way using the *PF Avoiding* component's *strategic vector* input port.

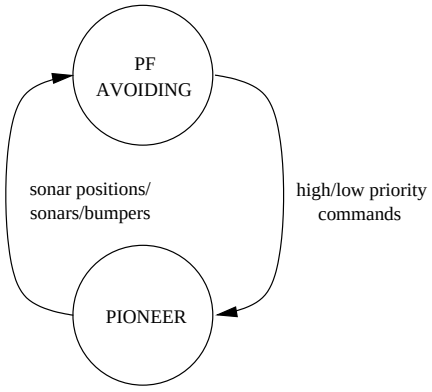

Figure 4.8: The avoiding level.

4.3.3 A Strategic Component

The *PF Avoiding* component presented in the previous section 4.3.2 endows a mobile robot with a repulsive behavior against obstacles, even in the case that the robot hits one of them. But by design, the component allows to be influenced in its behavior by means of a public input port called *strategic vector*. Remember that, as it was commented, at each working period, a vector called *strategic vector* may be added and averaged to the repulsive vectors generated by the different obstacles that the robot´s sonar sensors have detected. Additionally, the *PF Avoiding* associates a persistence time to the last strategic vector that has been received, if that time expires without receiving a new one, the strategic vector is ignored. Thus, once it has expired, only repulsive vectors are taken into account to command the robot. It is important to highlight that the modulus of a strategic vector represents the fraction of the maximum robot translational velocity and it must be in the range of values $[0, 1] \in \mathbb{R}$. Bear in mind that this is also a property that the repulsive vectors induced by obstacles have too (as equation 4.1 confirms).

The use of strategic vectors in the *PF Avoiding* makes possible to command the robot externally to move it in a specific direction. Taking advantage of this feature of the *PF Avoiding* component, a new atomic component called *Strategic PF* component was designed and built. Its external interface of public output and input ports appears in figure 4.9 with its non default observable and controllable variables included at the bottom. In parentheses, the types of input and output ports are indicated by its respective symbols (consult tables 3.3 and 3.4). Moreover, tables 4.5, 4.6, 4.7, and 4.8 describe respectively its public output and input ports, and its non default observable and controllable variables.

Likewise the *PF Avoiding* component, the potential field approach is behind the functionality of the *Strategic PF* component. This component operates also periodically, generating at each period an strategic vector that is sent through its *strategic vector* public output port. The component works in four modes that corresponds directly with the commands that the component may receive through its *commands* public input port: *inactive*, *move*, *goto* and *docking*. Observing the figure 4.10 it is

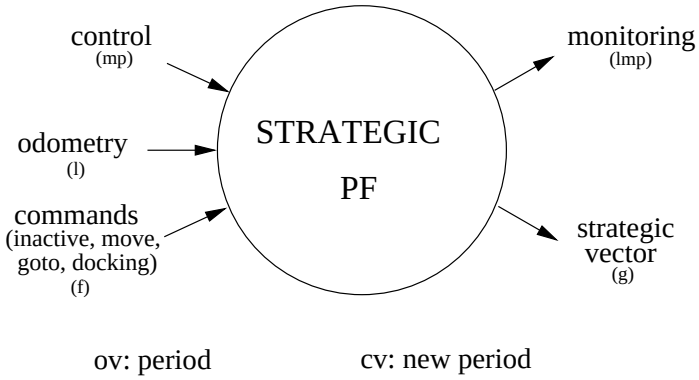

Figure 4.9: Component *Strategic PF*: external interface.

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is an *OLazyMultiPacket*.
strategic vector	This is an *OGeneric* output port. It has been devised to feed the *PF Avoiding* with strategic vectors.

Table 4.5: Component *Strategic PF*: public output ports.

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which controllable variables may be modified and updated. It is an *IMultiPacket* input port.
odometry	This is an *IFifo* input port through which the component receives odometry information. This odometry information is usually available by connecting this input port to the *Pioneer* component´s homonym output port.
commands	This is an *IFifo* input port used by this component to receive commands that may change its internal activity. The possible commands are: *inactive*, *move*, *goto* and *docking*.

Table 4.6: Component *Strategic PF*: public input ports.

Non Default Observable Variables	
Name	Brief Description
period	This observable variable indicates at which period operates the component.

Table 4.7: Component *Strategic PF*: non default observable variables.

evident that they correspond also to states of its *user automaton*. In each mode the component generates a strategic vector using a different potential field, except in the

Non Default Controllable Variables	
Name	Brief Description
new period	Through this controllable variable is possible to change the working period of the component. It updates the *period* observable variable.

Table 4.8: Component *Strategic PF*: non default controllable variables.

inactive mode where it does nothing.

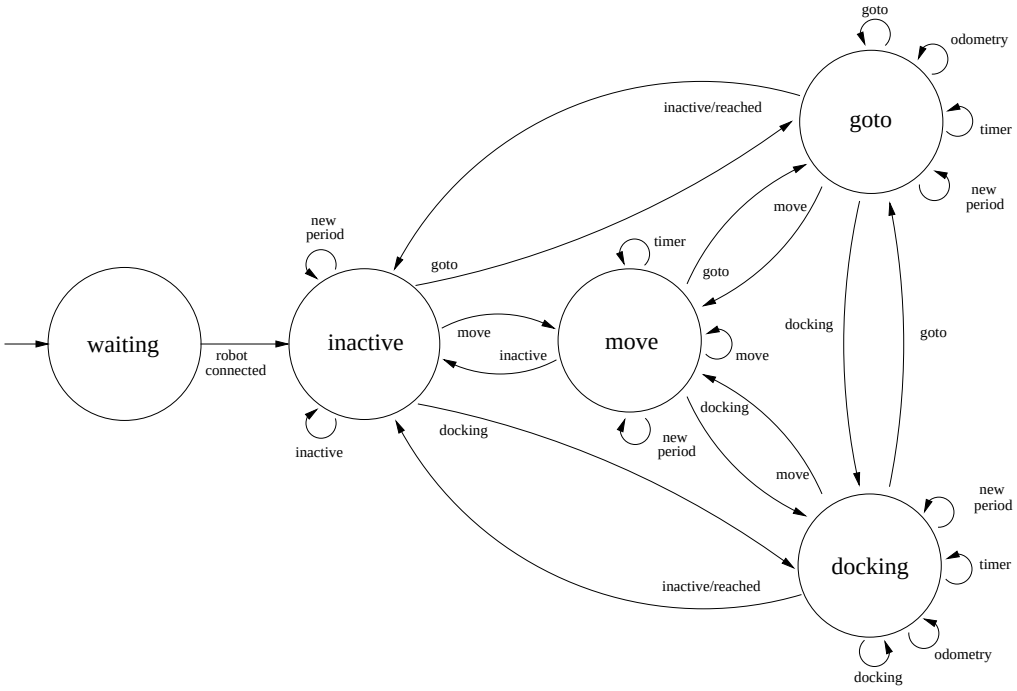

Figure 4.10: Component *Strategic PF*: user automaton.

Thus, when the component is in *move* mode it generates an strategic vector by means of a *uniform* potential field which verifies the following equation:

$$\overrightarrow{un}(\overrightarrow{p}) = \overrightarrow{v} \tag{4.3}$$

such that $\overrightarrow{v}$ is a vector whose modulus verifies that $0 \leq |\overrightarrow{v}| \leq 1$. This is a constant potential field that generates always the same strategic vector wherever the robot is situated. Figure 4.11 illustrates an example of an uniform potential field according to equation 4.3.

In *goto* mode the component acts differently. In this case the *Strategic PF* generates strategic vectors based on an *attractive* potential field. This is a potential field centered in a point of the space that follows the following equation:

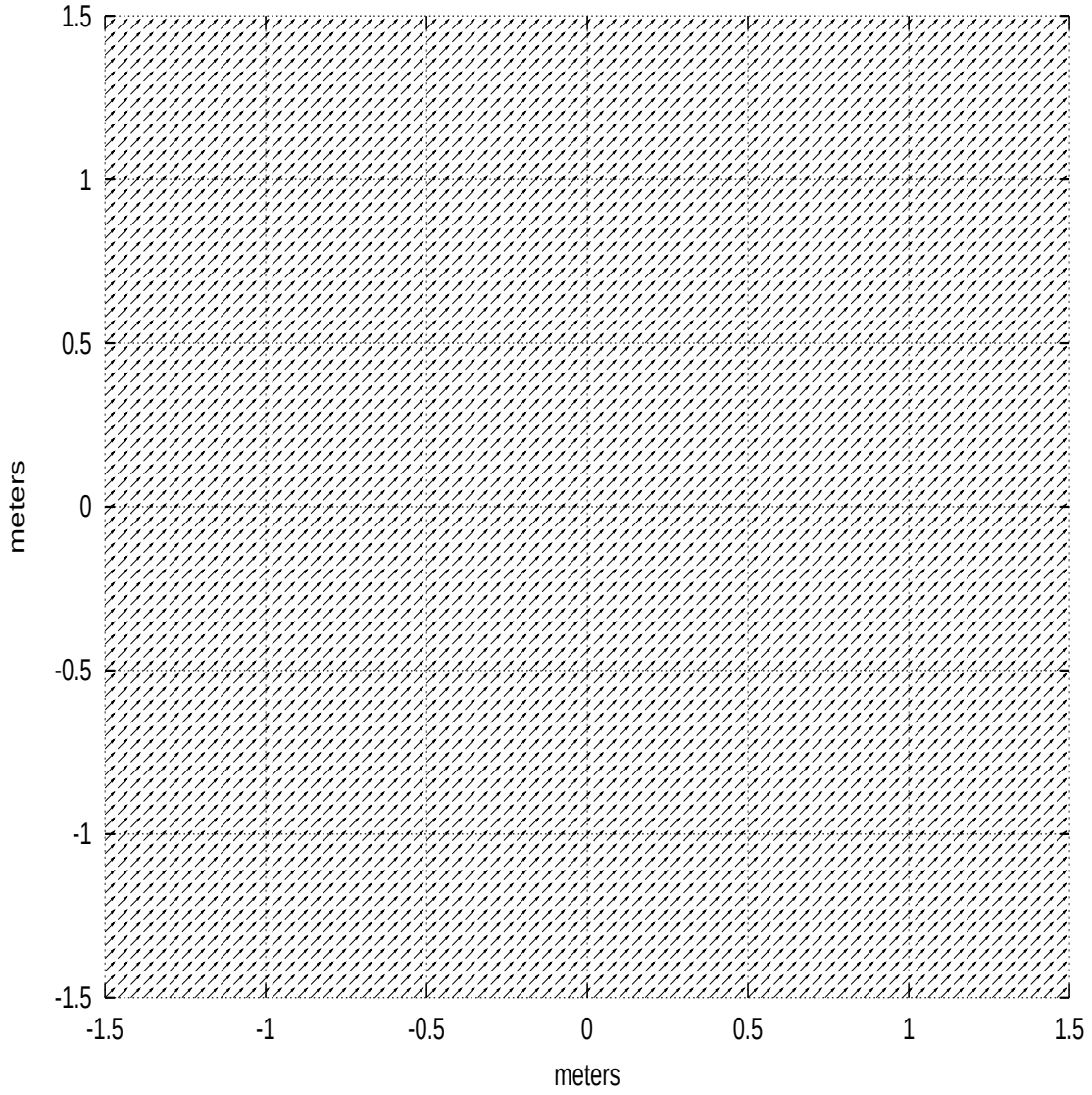

Figure 4.11: Component *Strategic PF*: uniform potential field.

$$\overrightarrow{a}(\overrightarrow{p},\overrightarrow{g}) = \begin{cases} \overrightarrow{u}, & \text{if } d \geq d_{max} \\ \frac{d-d_{min}}{d_{max}-d_{min}}\overrightarrow{u}, & \text{if } d_{min} < d < d_{max} \\ \overrightarrow{0}, & \text{if } d < d_{min} \end{cases} \tag{4.4}$$

such that, $\overrightarrow{p}$ is any position in robot coordinates; $\overrightarrow{g}$ is the position of the goal also in robot coordinates; $d = |\overrightarrow{g} - \overrightarrow{p}|$ is the euclidean distance between $\overrightarrow{g}$ and $\overrightarrow{p}$; $\overrightarrow{u} = \frac{\overrightarrow{g}-\overrightarrow{p}}{|\overrightarrow{g}-\overrightarrow{p}|}$ is an unitary vector between $\overrightarrow{g}$ and $\overrightarrow{p}$; d_{max} is a threshold distance where the potential field starts to fall in modulus because $\overrightarrow{p}$ is too close to the goal point $\overrightarrow{g}$; finally, d_{min} is the minimum distance to the goal where it is possible to be. Figure 4.12

displays an example of an attractive potential field described using equation 4.4.

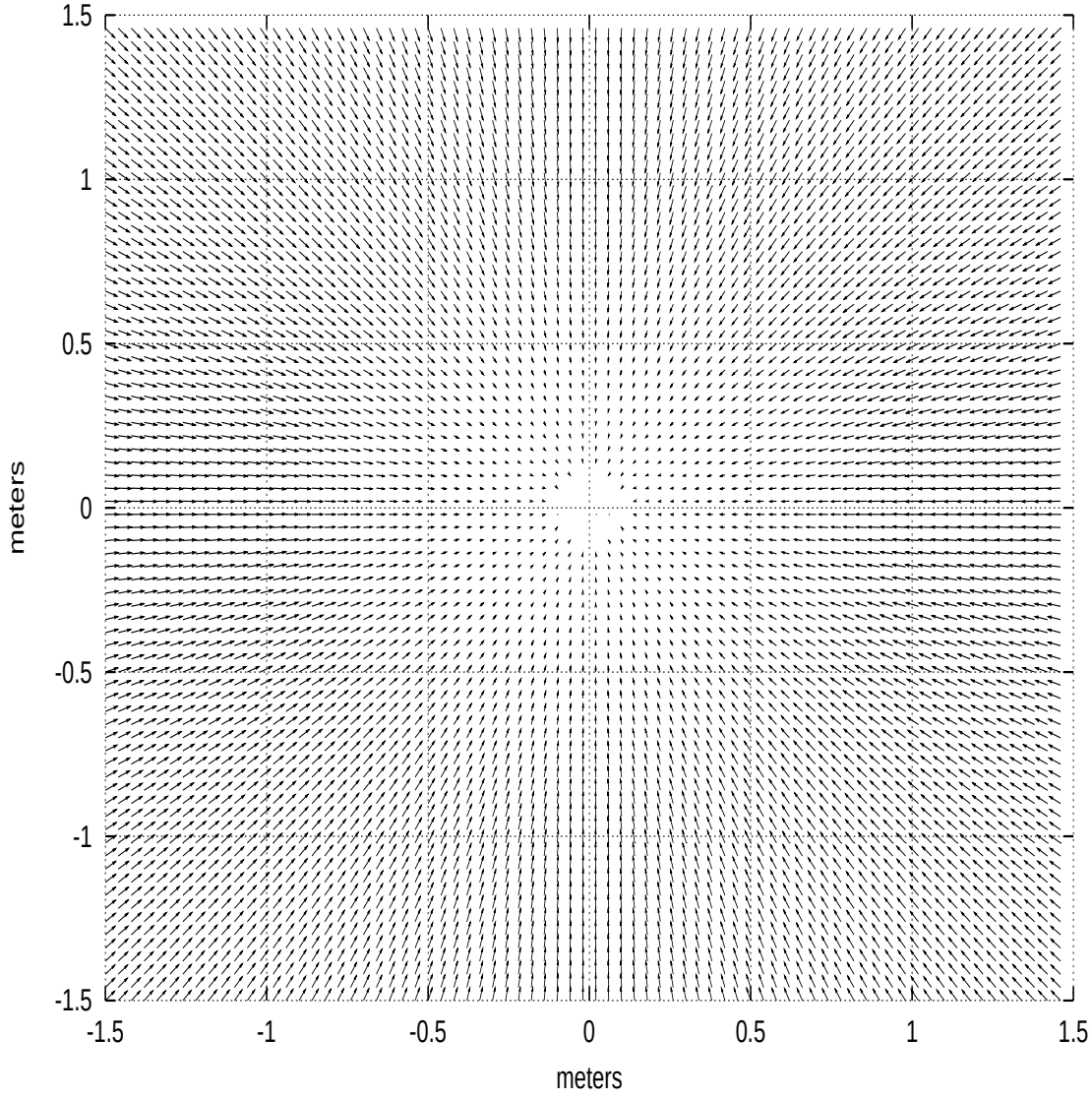

Figure 4.12: Component *Strategic PF*: attractive potential field.

In *docking* mode the *Strategic PF* component generates strategic vectors using a *docking potential field* centered in a goal point. The docking field is useful to lead a robot to a specific goal position, but doing the approaching to the goal point keeping the robot in a range of orientation angles. Mathematically, the docking potential field is described by the following equation:

$$\overrightarrow{d}\,(\overrightarrow{p},\overrightarrow{g}) = \begin{cases} \overrightarrow{u}, & \text{if } d \geq d_{max} \\ \frac{d-d_{min}}{d_{max}-d_{min}}\,\overrightarrow{u}, & \text{if } d_{min}<d<d_{max} \text{ and } \theta_{\overrightarrow{u}} \in [\theta_a,\theta_b] \\ \frac{d-d_{min}}{d_{max}-d_{min}}\,\overrightarrow{u}_{\frac{\pi}{2}}, & \text{if } d_{min}<d<d_{max} \text{ and } \theta_{\overrightarrow{u}} \in (\theta_b,\theta_c] \\ \frac{d-d_{min}}{d_{max}-d_{min}}\,\overrightarrow{u}_{-\frac{\pi}{2}}, & \text{if } d_{min}<d<d_{max} \text{ and } \theta_{\overrightarrow{u}} \in [\theta_c,\theta_a) \\ \overrightarrow{0}, & \text{if } d<d_{min} \end{cases} \quad (4.5)$$

such that, $\overrightarrow{p}$ is any position in robot coordinates; $\overrightarrow{g}$ is the goal point also in robot coordinates; $d = |\overrightarrow{g} - \overrightarrow{p}|$ is the euclidean distance between $\overrightarrow{g}$ and $\overrightarrow{p}$; $\overrightarrow{u} = \frac{\overrightarrow{g}-\overrightarrow{p}}{|\overrightarrow{g}-\overrightarrow{p}|}$ is an unitary vector between $\overrightarrow{g}$ and $\overrightarrow{p}$; $\theta_{\overrightarrow{u}}$ is the angle that $\overrightarrow{u}$ forms with the positive x axis; θ_a and θ_b are two angles that determine an angular sector surrounding the docking goal point where the docking is possible; $\theta_c = \frac{\theta_a+\theta_b}{2} - \pi$ is the opposite angle to $\frac{\theta_a+\theta_b}{2}$; $\overrightarrow{u}_{\frac{\pi}{2}}$ is $\overrightarrow{u}$ rotated $\frac{\pi}{2}$ (anticlockwise); $\overrightarrow{u}_{-\frac{\pi}{2}}$ is $\overrightarrow{u}$ rotated $-\frac{\pi}{2}$ (clockwise); d_{max} is a threshold distance where the potential field starts to fall in modulus because $\overrightarrow{p}$ is too close to the goal point $\overrightarrow{g}$; d_{min} is the minimum distance to the goal where it is possible to be. Figure 4.12 visualizes the equation with an example. Docking fields are common to drive a robot towards a docking station, usually in industrial scenarios, where the robot should get "parked" in the docking station, and where normally the robot should approach it from only specific directions [Murphy, 2000].

The *Strategic PF* component presents the *user automaton* displayed in figure 4.10 (*default automaton* transitions and states are not shown). Note that the different component´s modes of operation correspond to different states in the automaton, they can be described as follows:

- **waiting**: This is the *user automaton* entry state. In it, the component remains idle, waiting until it gets connected to a *Pioneer* component instance, through its *odometry* public input port (see table 4.6). Once connected the component transits to the **inactive** state.

- **inactive**: The *Strategic PF* component gets to this state when it is in *inactive* mode. Getting to this state the component stops the robot by sending a $\overrightarrow{0}$ vector through its *strategic vector* (note that, even in this case, the *PF Avoiding* component will keep the robot avoiding obstacles). In this mode the component accepts modification of its *new period* controllable variable (table 4.8) in order to modify its *period* observable variable (table 4.7), and consequently its period of operation (**new period** transition). Besides of changing its working period, in this state the component can receive commands through its *commands* input port (table 4.6). Depending on which command has been received, it remains in **inactive** state, or transits to **move**, **goto** or **docking** states (respectively **move**, **goto** and **docking** transitions).

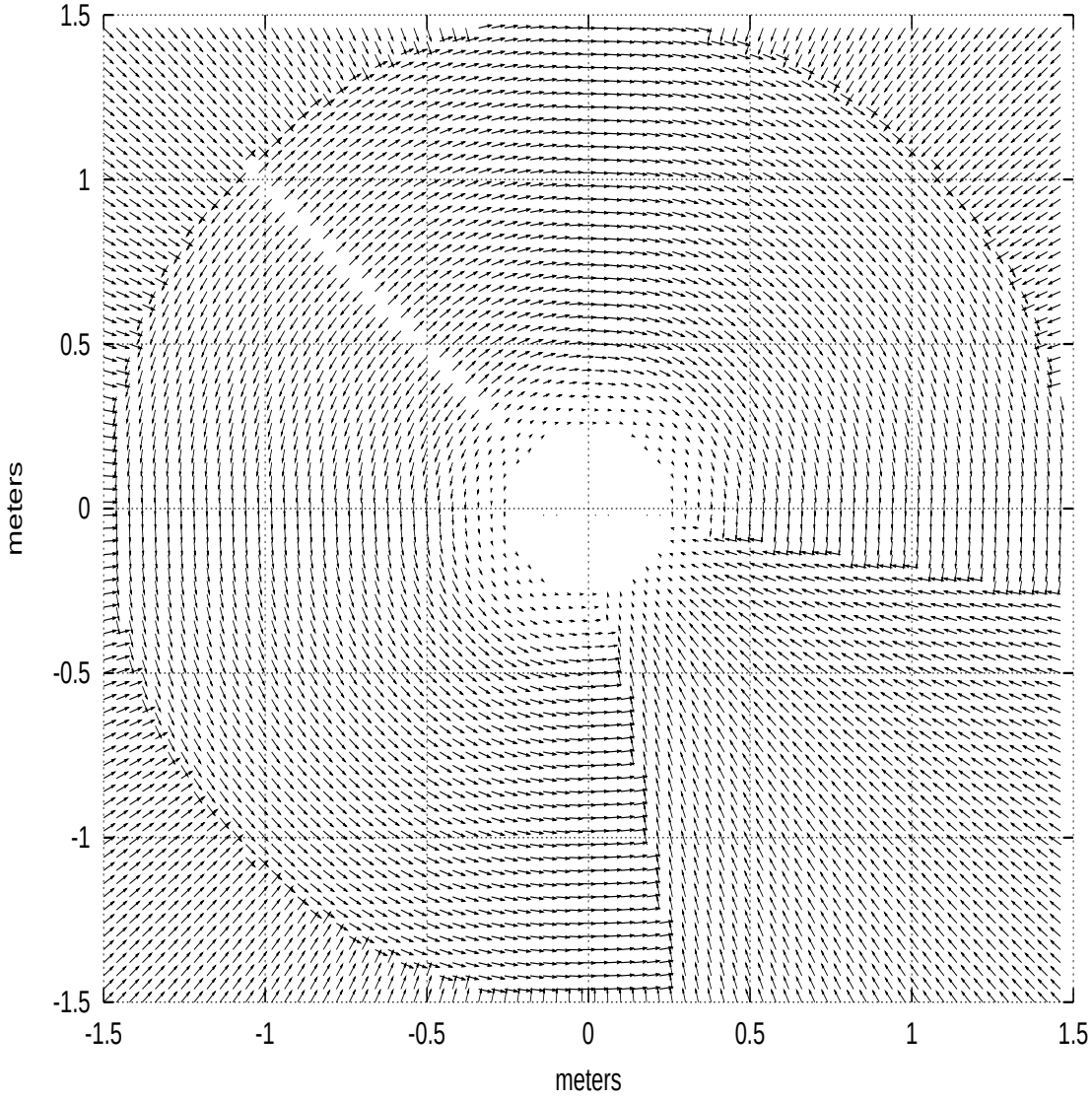

Figure 4.13: Component *Strategic PF*: docking potential field.

- **move**: The *Strategic PF* component stays in this state when it is in *go* mode. In this state the component accepts changes of its period of operation (**new period** transition), and operates periodically generating strategic vectors based on a uniform potential field (**timer** transition). The vector $\overrightarrow{v}$ in equation 4.3 is contained in the command that provoked the component to enter in **move** state. While the component remains in *move* mode the vector $\overrightarrow{v}$ establishing the direction and modulus of the uniform potential field can be changed (**move** transition) with new *move* commands. In addition, the component can receive other commands to change its mode of operation, and consequently to change to any of the other states (**inactive**, **goto** and **docking** transitions).

- **goto**: The *Strategic PF* component remains in this state when it is in *goto* mode. Staying in this state the component accepts changes of its working period

(**new period** transition), and operates periodically generating strategic vectors based on an attractive potential field (**timer** transition). The parameters that govern this attractive field are provided by the *goto* command that has driven the component to that state. These parameters are: d_{min}, d_{max} and the goal point $\overrightarrow{g}$ (equation 4.4). Receiving new *goto* commands through the *commands* input port, any or all the parameters that define the attractive potential field can be modified (**goto** transition). Moreover, the component can receive other commands to change its mode of operation in order to transit to **inactive**, **move** and **docking** states (respectively **inactive**, **move** and **docking** transitions).

- **docking**: The *Strategic PF* component stays in this state when it is in *docking* mode. In this state the component accepts changes of it period of operation (**new period** transition), and generates periodically strategic vectors based on a docking potential field (**timer** transition) following equation 4.5, where the different values that defines the potential field are provided by a *docking* command received through the component´s *commands* input port. Such parameters are: the goal point $\overrightarrow{g}$, the threshold distances d_{min} and d_{max}, the angles θ_a, θ_b, and the minimum distance from obstacles (d_{max} in equation 4.1), because during docking it may be necessary to get closer to obstacles. New *docking* commands received through the *commands* input port, may modify any or all the parameters that define the docking potential field by means of which the component generates strategic vectors for docking (**docking** transition). Besides, the component may receive other commands to change its mode of operation in order to transit to **inactive**, **move** and **goto** states (respectively **inactive**, **move** and **goto** transitions).

4.3.4 A Wander Component

In this point, a new atomic component called *Wander* will be defined to endow a robot with a wandering behavior, taking advantage of the components that have been designed and built so far: the *Pioneer*, *PF Avoiding* and *Strategic PF* components. The *Wander* component offers the external interface of public output and input ports illustrated in figure 4.14. Non default observable and controllable variables appear at the bottom. In parentheses, the types of input and output port are indicated by its respective symbols (consult tables 3.3 and 3.4). Tables 4.9, 4.10, 4.11, and 4.12 describe respectively its public output and input ports, and its non default observable and controllable variables.

Figure 4.15 displays the *Wander* component´s *user automaton*. It consists of one state called **wandering**. In this state the component operates periodically sending *goto* commands (**timer** transition) through its *command* public output port (table 4.9), in order to change the operation mode of the *Strategic PF* component. In each period it is sent a *goto* command with a different goal point $\overrightarrow{g}$ which is chosen in an aleatory way. Evidently the working period should be chosen long enough to allow the robot to approach a goal point for a while before driving it to another different point in the

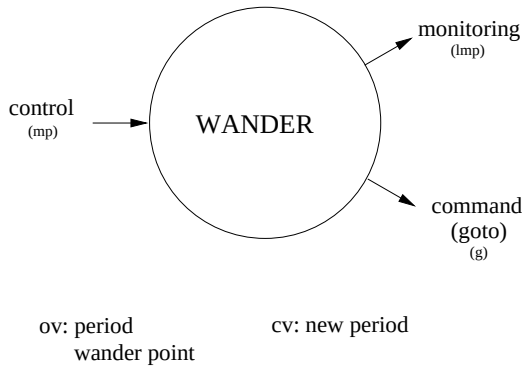

Figure 4.14: Component *Wander*: external interface.

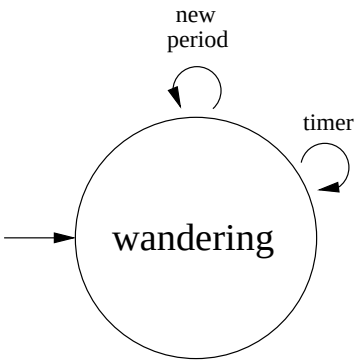

Figure 4.15: Component *Wander*: user automaton.

next period.

Staying in this state it is also possible to change the period of operation by means of the *new period* controllable variable that, in turn, will modify the *period* observable variable that governs the component´s working period (**new period** transition). Tables 4.11 and 4.12 contains a brief description of the non default observable and controllable variables that the component offers.

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is an *OLazyMultiPacket*.
command	This is an *OGeneric* output port. It has been devised to send *goto* commands to the *Strategic PF* component.

Table 4.9: Component *Wander*: public output ports.

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which controllable variables may be modified and updated. It is an *IMultiPacket* input port.

Table 4.10: Component *Wander*: public input ports.

4.3.5 The Wandering Level

Figure 4.16 shows the combination of the components that would make a Pioneer robot wander around while avoiding obstacles. This would be a wandering level built on top of the avoiding level presented in section 4.3.2 and displayed in figure 4.8. Obviously some operational decisions that should be necessary to made would be the

Non Default Observable Variables	
Name	Brief Description
period	This observable variable indicates at which period operates the component.
wander point	This observable variable indicates at which aleatory point the component has decided to lead the robot.

Table 4.11: Component *Wander*: non default observable variables.

Non Default Controllable Variables	
Name	Brief Description
new period	Through this controllable variable it is possible to change the working period of the component. It updates the *period* observable variable.

Table 4.12: Component *Wander*: non default controllable variables.

different operation frequencies of the different components. It is obvious that the *Wander* component should work at a low frequency with a period of operation of maybe minutes. On the other side, the operation period at which the *Strategic PF* generates strategic vectors should be short enough to refresh the persistence time of such vectors in the *PF Avoiding* component. Clearly, the *new persistence* controllable variable of the *PF Avoiding* component should be chosen accordingly. Bear in mind also that the faster the *PF Avoiding* component generates robot commands, the faster the robot response will be, so the choice of its period of operation should be short enough to have a response fast enough. Remember that there is a low limit for that period which is the period at which robot sensory information is received, either 50 or 100 milliseconds, furthermore,several cycles of 50 or 100 milliseconds are necessary to receive readings from all the sonar sensors available in the robot.

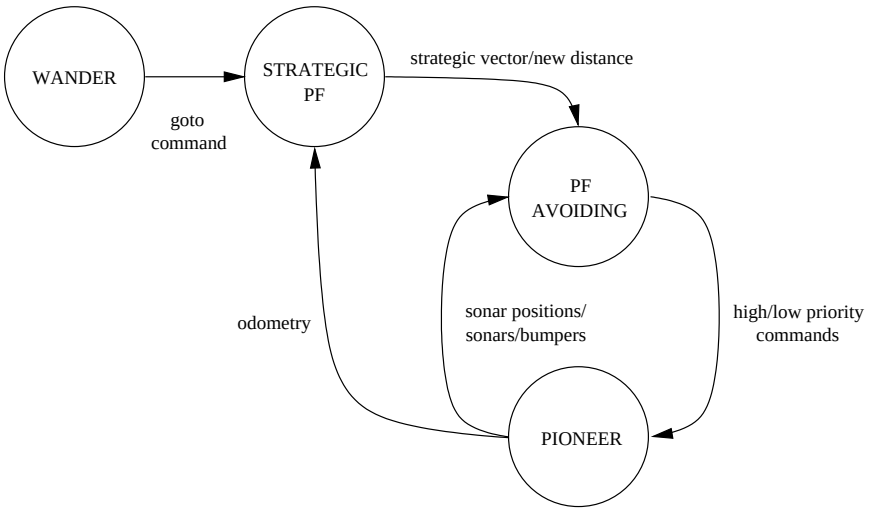

Figure 4.16: The wandering level.

4.4 What about a Task?

Quoted from Arkin ([Arkin, 1998], page 267):

> *"Action-oriented perception requires that perception be conducted in a top-down manner on an as-needed basis, with perceptual control and resources determined by behavioral needs."*

Basically, it establishes that the perceptual information that a system should use and take into account, is driven and determined by the tasks the system must carry out. Thence, the perceptual information that is necessary in order to accomplish a task determines the computational resources and the sensory information needed in a system.

As commented by Arkin in [Arkin, 1998] perceptual information in behavior-based systems organize in three general ways:

- **Sensor Fission**: This is referred to as when a behavior access directly sensory information from which it extracts the perceptual information it needs. Finally, based on this information it exerts on system effectors the actions it considers necessary. Figure 4.17 helps to understand the concept. An example of sensor fission is the control loop of figure 4.8, where the *PF Avoiding* component (an avoiding behavior) extracts the necessary perceptual information it requires, and generates directly the effector actions the system needs in order to avoid obstacles.

- **Sensor Fusion**: It is frequent that different sensory information coming from different sensors provide the same type of perceptual information. Consider, for instance, a robot that detects the obstacles surrounding it by using three different sensors: a ring of sonars, a laser range finder, and a camera in a pan-tilt platform. Consider a behavior that uses that perceptual information in order to emit a system response by means of effector actions. As the information that comes from these three different sensory sources may be redundant, complementary or contradictory, some filtering could be necessary in order to feed the behavior with perceptual information that fuses and integrates the data coming from these different sensors. This filtering to fusion perceptual information is what is called *sensor fusion*. Figure 4.18 illustrates the idea as it might be put into practise in terms of CoolBOT, observe how a component fuses the perceptual information in order to feed another component.

- **Sensor Fashion**: Frequently a behavior along its runtime life-cycle makes use of different perceptual information, because it is a requirement imposed by the particular functionality which is active at that moment, this is what is called *sensor fashion*. In figure 4.19 a typical configuration for sensor fashion is displayed. In the figure, the component (the behavior) uses different perceptual information depending on its internal state and its internal context of execution. Imagine for example a behavior that drives a mobile robot from one point to another using an a-priori map, while avoiding obstacles. Such a behavior is split up in different

phases (for example: door traversing, freeway navigation, corridor navigation and docking) where the component makes use of different perceptual information in order to complete each phase successfully.

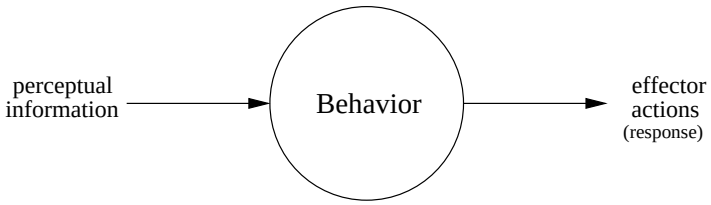

Figure 4.17: Sensor fission.

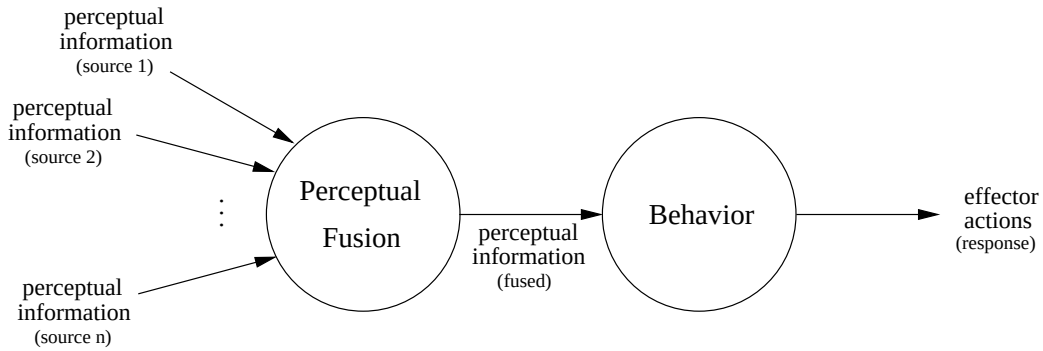

Figure 4.18: Sensor fusion.

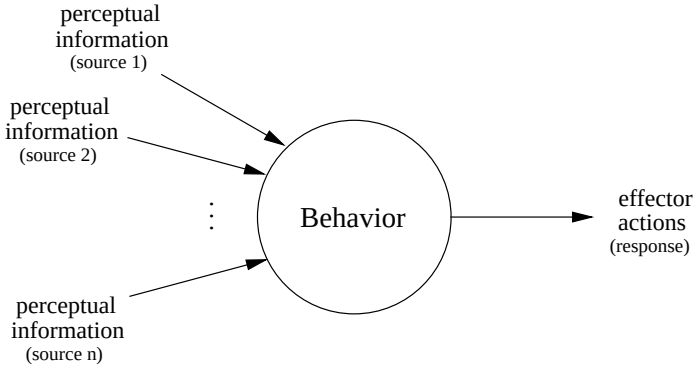

Figure 4.19: Sensor fashion.

From figure 4.17 and 4.18 it is not difficult to see that component configurations typical of sensor fission and sensor fusion have a straight implementation using the abstractions and means that CoolBOT provides. As commented, the control loop of figure 4.8 is typically a sensor fission configuration of components. It is not difficult to modify the control loop of that figure in order to integrate more sensory information doing sensor fusion. For example, the information obtained from a laser range finder could be utilized if a new component that fuses laser and sonar information were introduced to feed the *PF Avoiding* component.

The objective of this section is to illustrate by means of an example how to organize components in order to carry out a task split up in different phases that use different perceptual information (*sensor fashion*). The concrete example that will be shown here has been inspired by the example of perceptual sequencing commented in Arkin [Arkin, 1998] (pages 279-283). Our example consists in making the robot used in the previous section (section 4.3) perform the task of going home, i.e., going to a homing area where it should dock. The task requires also that the robot avoids obstacles.

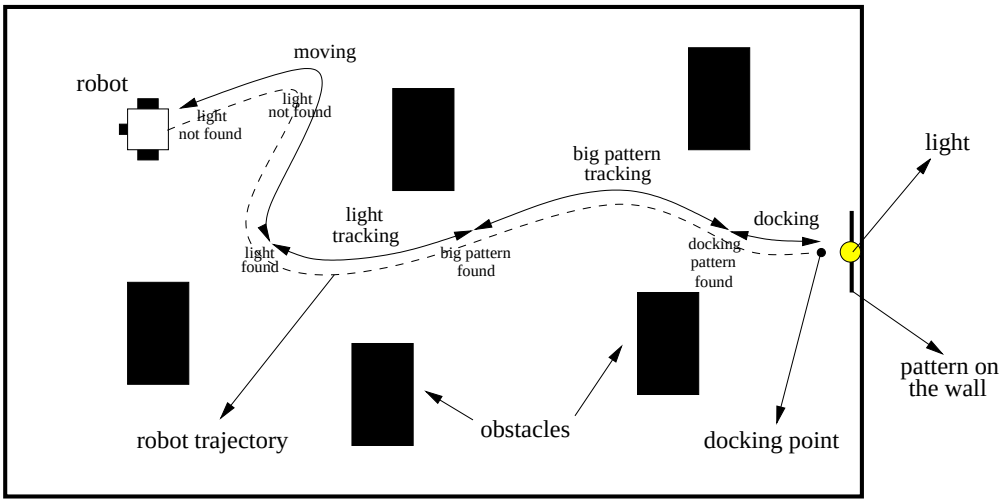

Figure 4.20: The Go Home task: the scenario.

Figure 4.20 displays the scenario considered for this task, a room with a docking area in one side of the room. This docking area is indicated by a panel situated on one of the room walls, and a bright light above it. On the panel there are two interleaved patterns: a big pattern which is a big "H" letter, and a small pattern composed by two circles called the *docking pattern*. They appear in figure 4.21, they are referred together as the *homing pattern*. Thus, there are three main perceptual landmarks in the scenario that will be used by the robot to perform the task of going home. The first landmark is the light located just above the panel where the homing pattern is stuck. It can be detected from nearly any point in the room using a camera mounted in a pan-tilt platform placed on the robot (see figure 3.41). The second landmark is the big "H" letter, the big pattern, which is easily detected from nearly any position in the room when the robot looks frontally at it. The third one, is the docking pattern of circles which should be detected when the robot is close to the homing position. This is a point situated at a specific distance (for instance, half a meter) from the panel, just in front of the middle point between the docking pattern's circles. The circles that form the docking pattern verify that at the minimum distance to the wall both are inside the robot camera´s field of vision.

As it is shown in figure 4.20 the task is divided into four main phases: **moving**, where the robot just moves for a while to find a position where it is able to detect the light above the homing pattern; **light tracking**, a phase where once the light is

detected the robot servo navigates by tracking it to get closer in order to detect the big "H" letter; **big pattern tracking**, as soon as the robot localizes visually the big pattern, it servo navigates towards it to reach a position where it is able to recognize visually the docking pattern; **docking**, finally, when the robot is so close to the docking area that it recognizes the docking pattern, the robot navigates more precisely (slower) to get docked conveniently at a specific distance from the wall where is situated the docking pattern; in this last phase the robot utilizes the sonar rings available on the robot to control approximately the distance to the wall.

We will make use of most of the components presented so far: the *Pioneer* component, the *PF Avoiding* component, and the *Strategic PF* component. Besides, two new components will be designed and built: the *Vision Server* component, and the *Go Home* component, that once integrated together will perform the required task. These two new components will be introduced in the next sections.

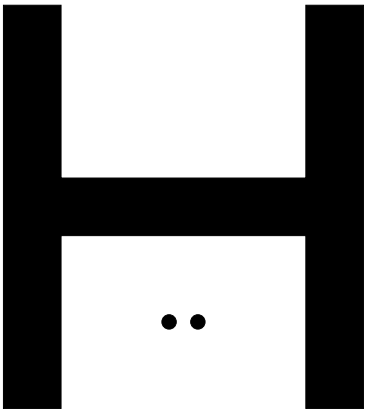

Figure 4.21: The Go Home task: the homing pattern.

4.4.1 A Vision Component

In order to obtain visual perceptual information from the camera situated in the pan-tilt support placed on top of the robot, an *atomic* component called *Vision Server* has been designed. Its external interface appears in figure 4.22, and as in previous figures, the types of output and input port are indicated between parentheses (consult tables 3.3 and 3.4). The output and input ports that conform their external interface are briefly explained in tables 4.13 and 4.14. This component does not need any new additional observable or controllable variables.

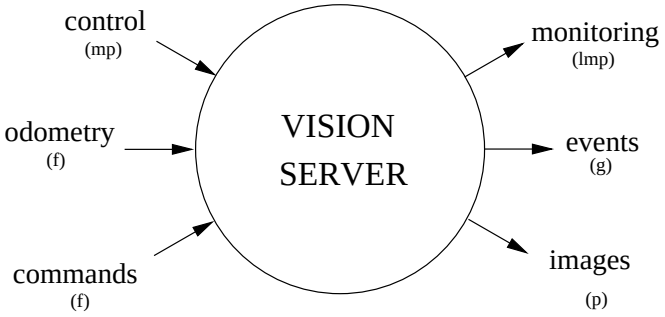

Figure 4.22: Component *Vision Server*: external interface.

The *Vision Server* component´s *user automaton* is shown in figure 4.23. Transition to *default automaton states* are not displayed. The *Vision Server* component has three modes of operation that correspond to the states appearing in the figure:

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is always an *OLazyMultiPacket*.
events	This is an *OGeneric* output port through which the component signalizes some events, such as the lost of a visual target it was tracking, or when it has finished a process of visual scanning.
images	This is an *OPoster* output port. This component serves time-stamped images, together with the pose of the camera in robot coordinates at which the image was sampled, and, if necessary, the image coordinates of the pattern that has been found in the image.

Table 4.13: Component *Vision Server*: public output ports.

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which controllable variables may be modified and updated. It is always an *IMultiPacket* input port.
odometry	This is an *IFifo* input port through which the component receives the odometry of the robot, in order to calculate the pose associated to the images that it publishes through its *images* output port.
commands	This is an *IFifo* input port. The component receives through it commands that affects the component's modes of operation. There are three different commands: *inactive*, *scan*, *next* and *track*.

Table 4.14: Component *Vision Server*: public input ports.

- **inactive**: This is the entry state of the *user automaton*. In this state, the component remains idle waiting for a command through its *commands* input port that would drive it to the other modes of operation, corresponding to **scanning** and **tracking** states (**scan** and **track** transitions respectively).

- **scanning**: When the component gets into this state it is because it has received a *scan* command through its *commands* input port. In this state the component will command the pan-tilt camera to make a scan delimited by two pan angles (tilt is not used). During the scanning, the component will take image samples at specific angles positions, will associate them with a time stamp and the pose of the camera in robot coordinates, and will publish them using the component´s *images* output port. The *scan* command that drives the component to this state must provide the pan angles that will limit the scan, and the angle increment that will be used for sampling images along the scanning process. Additionally, it must receive a *next* command through its *commands* input port to sample each image along the whole process (**next** transition). Once completed the scanning, the component emits a *scan-finished* event through its *events* output port and returns to **inactive**

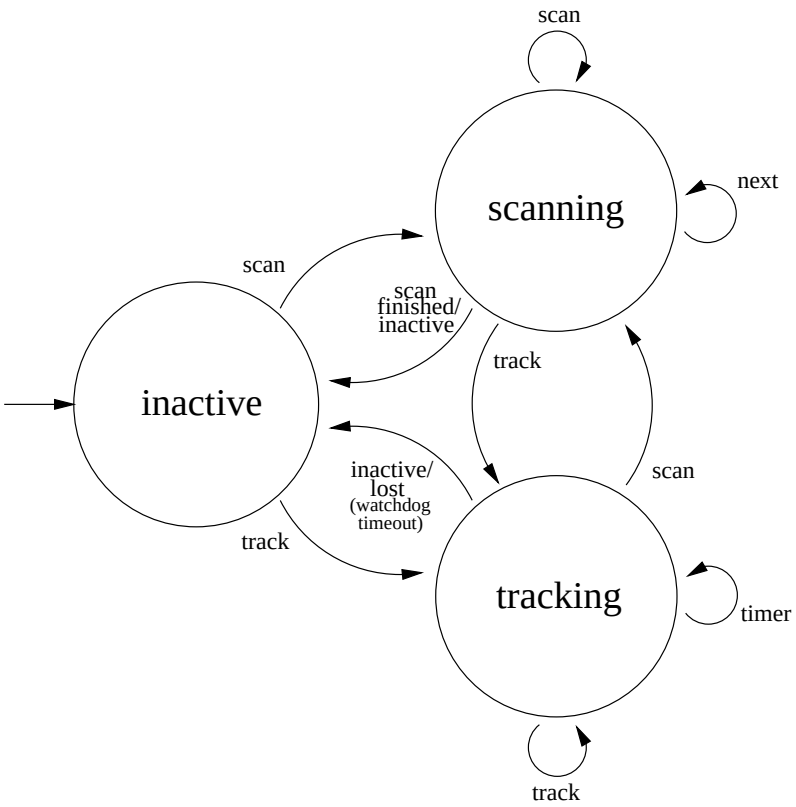

Figure 4.23: Component *Vision Server*: user automaton.

state (**scan finished** transition). Moreover, the component can be explicitly commanded to other *user automaton* states through the *commands* input port (**inactive** and **track** transitions). If a new *scan* command is received, a new scanning is started (**scan** transition). For all these cases, when the component is commanded to other state or when a new scanning is started the component sends also a *scan-finished* event through the *events* output port.

- **tracking**: The component enters this state when it receives a *track* command. There are two types of *track* commands: *track-pattern*, that makes the component find an image pattern in sampled images; and *track-light*, that makes the component find the most brighter area in images. *Track* commands indicates also a working period for the component when it is in this mode of operation. Initially, the target, either the pattern or the light, should be in the camera´s field of view, and once detected the camera tracks it using the pan-tilt module where the camera is mounted. From that moment on, the component periodically tries to find it on the field of view and acts conveniently in order to track the target (**timer** transition). Each time the target is detected the sampled image together with a time stamp, the pose of the camera, and the target's location in image coordinates is published through the component´s *images* output port. Traking is supported by a Kalman-filtering approach to keep the pattern inside the camera´s field of view [Hernández-Tejera et al., 1999]. Additionally, in case of tracking as target, an image pattern, the component implements and adaptive

correlation technique [Guerra-Artal, 2002] for tracking image patterns. By the reception of a new *track* command the target and/or the frequency of operation may be changed (**track** transition). Additionally, staying in this state the component may be ordered to go to **inactive** and **scanning** receiving the corresponding commands through the component´s *commands* input port (**inactive** and **scan** transitions respectively). If, during tracking, the component is not able to find the target in sampled images, the component emits a *lost* event thorugh the *events* input port and transits to **inactive** state (**lost** transition).

4.4.2 A Go Home Component

In figure 4.24, it is displayed the external public interface of the *atomic* component called *Go Home* which is the component that controls and sequences the different phases of operation of the homing task (figure 4.20). Like the *Vision component*, neither does this component need any non default observable and controllable variables. In tables 4.15 and 4.16 brief descriptions of its public output and input ports are given.

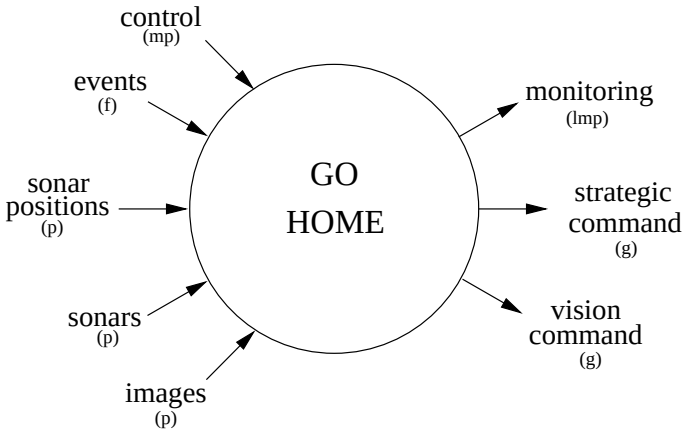

Figure 4.24: Component *Go Home*: external interface.

Public Output Ports	
Name	Brief Description
monitoring	This is the default *monitoring* output port by means of which the component publishes all its observable variables. It is always an *OLazyMultiPacket*.
strategic command	This is an *OGeneric* output port. It has been devised to command the *Strategic PF* component.
vision command	This is also an *OGeneric* output port through which the component may send commands to the *Vision Server* component.

Table 4.15: Component *Go Home*: public output ports.

The *Go Home* component has the *user automaton* that appears in figure 4.25 (*default automaton* transitions are not shown, except for the **end** state). As we can see

Public Input Ports	
Name	Brief Description
control	This is the component´s default *control* port through which controllable variables may be modified and updated. It is always an *IMultiPacket* input port.
events	This is an *IFifo* input port through which the component can be signaled of the occurrence of events by the *Vision Server* component, it should be connected to the homonyn output port of this component.
sonar positions	This is an *IPoster* input port. Through this port the component connects directly to the *Pioneer* component´s *sonar positions* output port that publishes the position of the robot sonar sensors in robot coordinates.
sonars	This is an *IPoster* input port. It is used to receive sonar readings from the robot by means of connecting to the *Pioneer* component´s *sonars* output port.
images	Through this port the component receives time stamped images with their associated pose (and a target location – whether the brighter area in the image, or a specific image pattern – in image coordinates) from the *Vision Server* component. It is an *IPoster* input port.

Table 4.16: Component *Go Home*: public input ports.

the different phases that have been considered for the completion of the going-home task in figure 4.20 correspond clearly to different states in the automaton. Now, in the following paragraphs the states that conform the *user automaton* are described:

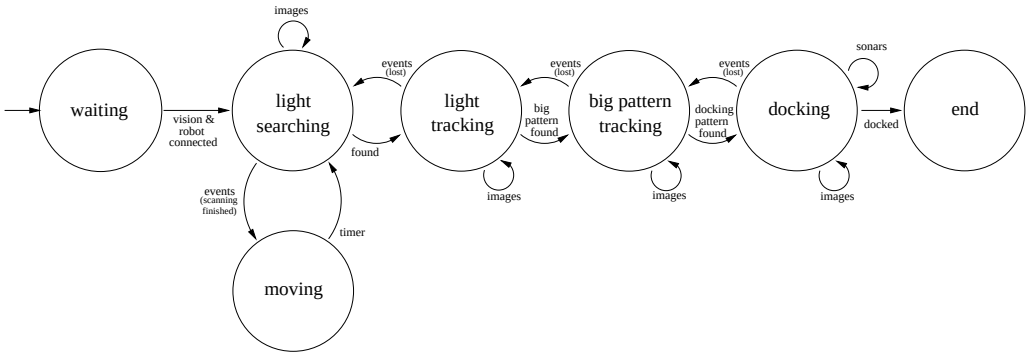

Figure 4.25: Component *Go Home*: user automaton.

- **waiting**: This is the *user automaton* entry state where the component waits for being connected to the *Vision Server* and *Pioneer* components. As soon as it is connected the component transits to the **light searching** state.
- **light searching**: In this state the robot remains stationary, and the component sends a *scan* command to the *Vision Server* component which is ordered to do a complete visual scanning, sampling images along the whole range of angles

which is possible to reach moving the pan joint of the pant-tilt unit to which the camera is attached. During scanning, images are sampled and received by the component with its corresponding pose in robot coordinates (**images** transition). For each image a brighter area detection algorithm is applied to detect the light situated above the homing area (figure 4.20), if the light is not found a *next* command is sent to the *Vision Server* in order to make it sample a new image. On the opposite, if the light is found in one of the images, the robot gets oriented to that direction, and the component transits to **light tracking** state (**found** transition). If the whole scanning range is completed without finding any light in sampled images (because maybe it is occluded by an obstacle), the robot transits to **moving** state (**events** transition).

- **moving**: In this state the component chooses an aleatory direction and makes the robot move in this direction for a while. How long the robot is moving is established at instantiation time, and obviously, it depends on the velocity we want the robot to move. As soon as this moving time has expired the component returns to **light searching** state to continue looking for the light again (**timer** transition). To move the robot in an the aleatory direction the component sends a *move* command to the *Strategic PF* component. Notice that *move* command implies the application of an uniform potential field (*Strategic PF*, section 4.3.3), so the robot will start moving just towards an aleatory direction at a specific velocity for a specific amount of time.

- **light tracking**: In this state the component makes the robot to servo-navigate towards the homing area using visual tracking by means of the *Vision Server* component. At the very moment of entering this state, the component sends a *track-light* command to the *Vision Server* component, in order to keep the light in the camera´s field of view using the pan-tilt unit. Remember that the *track-light* command that initiates visual tracking indicates also at which period of operation the visual component must track the light. During light tracking, through the *images* input port the component accesses the different images where the light is found. Each image is accompanied by a time stamp, its corresponding camera pose in robot coordinates, and the location of the brighter area in images coordinates. (**images** transition). Using this information the robot is led to go towards the homing area utilizing *move* commands sent through the *strategic command* output port (sent to the *Strategic PF* component). Additionally, the component searches in each image it receives the pattern with the big "H" letter of figure 4.21. The component has a pool of patterns corresponding to the big "H" letter seen from different distances, the component tries to find any of them. In such a case, the component is driven to **big pattern tracking** state (**big pattern found** transition). If the *Vision Server* component it is not able to track the light and loses it, the component stops the robot, and transits to **light searching** state (**events** transition).

- **big pattern tracking**: This state is analogous to **light tracking** state, because the component also carries out visual tracking, but instead of tracking the light

above the homing pattern, it tracks one of the patterns of the pool of patterns with the big "H" letter (figure 4.21) it found in the previous **light tracking** state. In order to do so, a *track-pattern* command is sent to the *Vision Server* component indicating also at which period visual tracking should be carried out. From that moment on, the component will access periodically new sampled images containing the tracked pattern (**images** transition). Like in the previous state, each image has its own time stamp, a corresponding camera pose in robot coordinates, and also the location of the pattern in image coordinates. Using this information the component drives the robot towards the homing position. Along the tracking process, on each image the component tries to find the docking pattern (the two circles in figure 4.21). The component has also a pool of patterns containing samples of this pattern seen from different distances. If it succeeds finding one of them, the component transits to **docking** state (**docking pattern found** transition) that constitutes the last phase of the going-home task. If, by any circumstance, the *Vision Server* component fails to track the pattern and loses it, the component stops the robot and, returns to **light tracking** state (**events** transition).

- **docking**: In this state the component drives the robot tracking visually the docking pattern found in the previous state in order to get situated at the docking point of figure 4.20. Evidently, the component gets to this state when the robot is close enough to find visually any of the patterns of the pool of docking patterns. The component makes use again of a *track-pattern* command sent to the *Vision Server* component, in order to receive periodically new sampled images containing the docking pattern, each one having a time stamp, an associated camera pose in robot coordinates, and the location of the pattern on the image (**images** transition). Equally, like in **light tracking** and **big pattern tracking** it uses this information to drive the robot towards the docking position. The process of docking is finished when the docking pattern occupies a specific big area on the images where the pattern has been found. In order to confirm that, the robot also monitors the distance between the robot and the room wall, where the panel containing the docking pattern, is situated. For that it makes use of the information coming through its *sonars* input port corresponding to the sonar readings published by the *Pioneer* component. As soon as it is close enough to the wall, and the pattern is big enough on the images, the component considers it has docked correctly, and the task finishes successfully transiting to **end** state. Evidently the robot is stopped at this point, and the *Vision Server* and *Strategic PF* components are commanded to get inactive. If the *Vision Server* component fails tracking the docking pattern, the component stops the robot and returns to **big pattern tracking** state (**events** transition).

- **end**: This is the **end** state of the *default automaton* of figure 3.8. In this state the component has finished successfully its operation.

Note that the *Go Home* component codifies in its *user automaton* the sequencing of phases in which the *Go Home* task has been split up, and how in each phase

different perceptual information, and different algorithms are used on that perceptual information. Therefore, it is evident that in CoolBOT sensor fashion and the sequencing of different phases that use different perceptual information at different instances of time during the performing of a task with different execution contexts are easily mapped in the automaton of a component.

4.4.3 The Go Home Task

Figure 4.26 shows a feasible configuration of components that would carry out the task of making our robot go to a homing area and get docked. Contrarily to the previous reactive examples shown in figures 4.8 and 4.16 where all the components were run at the same priority level (priority 8 at *normal* policy), and due to the complexity of the task at hand we have established three priority levels between the components of figure 4.26. At the highest one (priority 15 at *normal* policy) we have situated the *Pioneer*, the *PF Avoiding* and the *Vision Server* components, at the medium level (priority 11 at *normal* policy) we set the *Strategic PF* component, and at the lowest priority level (priority 8 at *normal* policy) there was only one component running, the *Go Home* component.

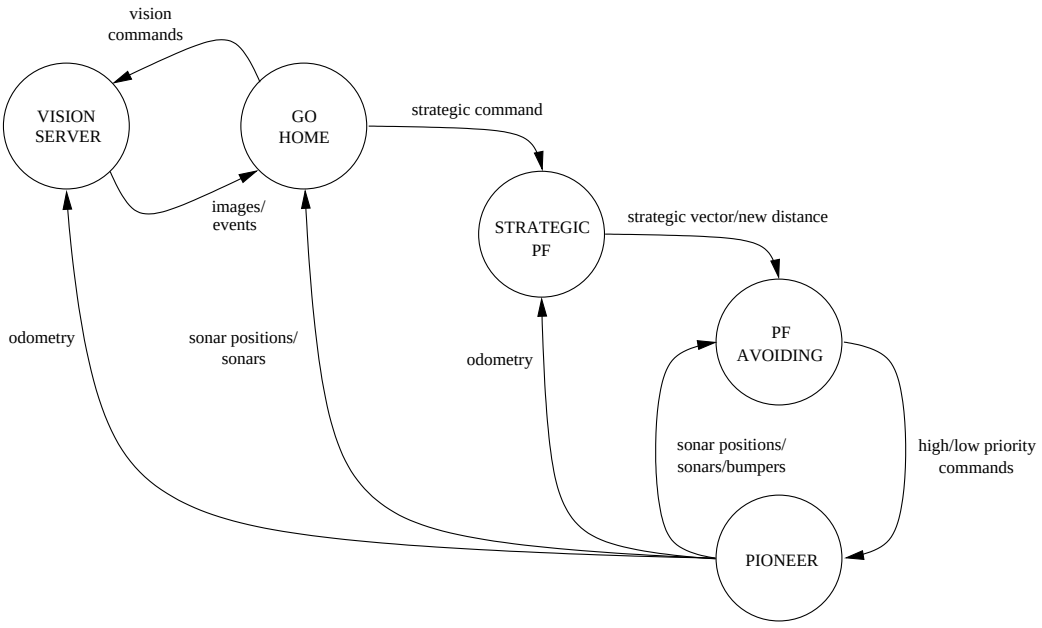

Figure 4.26: The Go Home task.

Observing figure 4.26, it is easy to see that the logic and control of the going-home task resides completely in the *Go Home* component. Using the same configuration of components but interleaving another component instead of it, would make the robot do another task, like for instance, look for an specific object. Obviously adding new functionalities to the *Visual Server* component will endow the system with the possibility of having more perceptual information what will make the robot behave in a more elaborated way, and perform more complex tasks.

4.5 A More Formal Approach for Tasks

In this section, it is presented how it might be possible, using CoolBOT, to make a description of tasks in a more formal way. The formal approach that has been selected has been inspired by Košecká in [Košecká et al., 1997] and [Košecká, 1996], who exploits the idea for representing robotic tasks as network of processes, as initially proposed by Lyons in its $\mathcal{RS}$ (Robot Schemas) model [Lyons and Arbib, 1989] [Lyons, 1990]. In Košecká´s work, tasks are considered as networks of processes modelled as automata conforming a finite state machine model. Additionally, the description and specification of tasks is done by means of a small set of operators of process algebra which allows defining tasks as networks of processes in a formal way. This set of process algebra operators defines six different operators to specify task compositions: **sequential composition**, **parallel composition**, **conditional composition**, **disabling composition**, **synchronous recurrent composition** and **asynchronous recurrent composition**.

Evidently, due to the fact that CoolBOT components are also automata, the approach proposed by Košecká´s work should be easily applied to CoolBOT components. In order to define tasks in the same terms and with the same constructs proposed by Košecká´s work, six different *compound* components implementing each of the Košecká´s operators will be presented along the rest of this section. After that, we will illustrate how to describe the examples presented in the previous sections 4.3 and 4.4 in terms of these *compound* components.

Some terminology must be introduced first in order to understand the different process algebra operators as they will be formally defined. Thus, it is said that a component has performed a *successful execution*, if it has carried out a complete task execution getting correctly to the **end** state of its automaton (figure 3.8). Alternatively, it is said that it has terminated a *failed execution*, if it has finished task execution getting to **running error** state corresponding to an exception which has become locally unrecoverable. In addition, it is said that it has been *aborted*, if it has been driven to **suspended** state in order to be re-started or killed.

4.5.1 Sequential Composition

The **sequential composition** operator is represented by the symbol **;**, this is the simplest operator and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a; b$ is such that an instance of c, c_i, behaves like an instance of a, a_i, until this one finishes, and then it behaves like an instance of b, b_i. When b_i finishes, c_i finishes with the same state as b_i.

The *compound* component that implements the sequential composition operator has a *supervisor* with the *user automaton* shown in figure 4.27 (*default automaton* transition and states are not shown, except for **end** and **running error** states). At instantiation time the component must be provided with the components over which the operator will be applied. Once it has been instantiated and ordered to run, it, in

turn, commands the first component to run. Then it waits for the finalization of its execution, whether successfully or not. After that, it orders the second component to run and waits for its conclusion, and depending on if it has been successful or not, the whole *compound* component finishes accordingly.

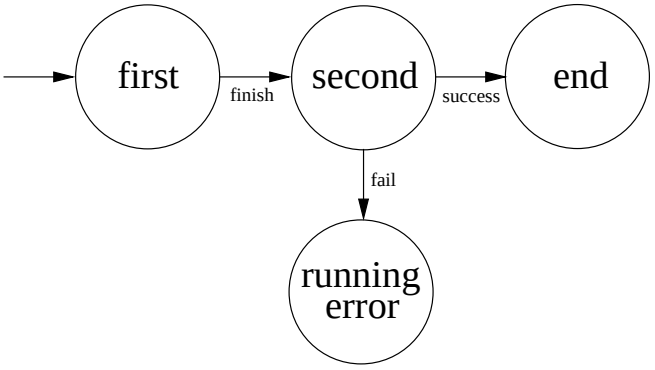

Figure 4.27: Sequential composition: user automaton.

This operator is clearly aimed to situations where we want two components to be executed sequentially in an unconditional way, no matter what are their final results. An example would be to order a robot to go sequentially to two points. Imagine that we have a component called `Goto` which performs the task of driving our robot to go to a specific point in robot coordinates. So we could do something like:

```
...
Component* gotoAB=
    new SequentialComposition(new Goto(pointA),
                              new Goto(pointB));
...
```

making the robot going first to `pointA`, and then to `pointB`, but even in the case, it does not reach the first point, we want it to have a try for the second one. Obviously, the `SequentialComposition` class is the *compound* component implementing the sequential composition operator, note that this component maps all output and input ports of its operands in its external interface, so it adapts its external interface in order to accommodate the operands it receives at instantiation time. The other *compound* components implementing the rest of process algebra operators adapt their external interface in the same way.

4.5.2 Conditional Composition

The **conditional composition** operator is represented by the symbol **:**, and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a < v >: b(v)$ is such that an instance of c, c_i, behaves like an instance of a, a_i, until this one finishes successfully computing v as a result of its processing, then it behaves like an instance of b, b_i, which uses v as input data.

When b_i finishes, c_i finishes with the same state as b_i. If a_i finishes unsuccessfully, c_i finishes with the same state as a_i.

The *supervisor* of the *compound* component that implements the conditional composition operator has the *user automaton* displayed in figure 4.28 where *default automaton* transitions and states are not shown (except for **end** and **running error** states). Observe the differences between the automaton in the figure, and the automaton corresponding to the sequential operator in figure 4.28, note that the main difference is that the second component is run if the first one succeeds in its execution. There is another difference which is not evident from the figure, but it is clear in the formal definition of both operators: in a conditional composition if the first component finishes successfully it returns a result (v in the previous paragraph) which is in turn fed to the second component when it is ordered to run. Evidently, to return a result of a successful execution it is possible to use the default *result* observable variable (table 3.1 in section 3.3.2 in chapter 3), but there is not any default controllable variable to feed a component with data when it starts running. To do so, components involved in conditional compositions that must receive such information should define a controllable variable aimed to that and called *starting data*.

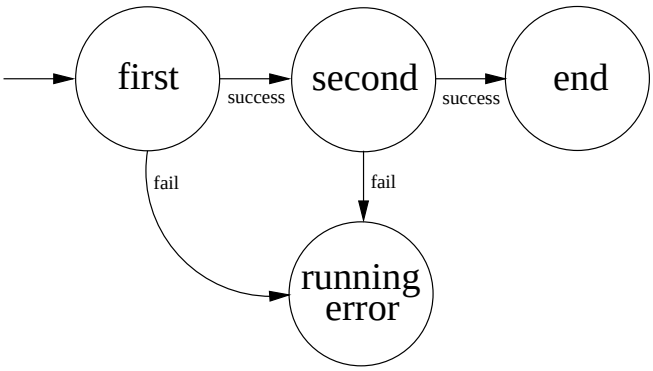

Figure 4.28: Conditional composition: user automaton.

A clear application of a conditional composition is when we want a component to be run, if a previous one has been executed with successful results. An example might be something like:

```
...
Component* gotoPattern=
    new ConditionalComposition(new Localize(pattern),
                               new NavigateTo(pattern));
...
```

such that, for example, `Localize` is a component that localizes a pattern using the pant-tilt camera mounted on the robot, and `NavigateTo` is a component that servo-drives the robot navigating towards the pattern doing visual tracking with the pan-tilt camera. Obviously if the pattern is not found the operator fails because it was not able to find the pattern, and, thus, it is unable of accomplishing the second part of the composition. The `ConditionalComposition` class is the compound component that implements the conditional operator, and like the `SequentialComposition` class at

instantiation time it also adapts its external interface to the output and input ports of its operands.

4.5.3 Parallel Composition

The **parallel composition** operator is represented by the symbol $\|$, and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a\|b$ is such that an instance of c, c_i, behaves like an instance of a, a_i, and an instance of b, b_i, running in parallel (or concurrently), and the state of the composition is a state pair which combines the states of both instances (see [Košecká, 1996] for details); c_i finishes with the same state as the last finished instance, either a_i or b_i. If c_i is suspended both instances are suspended as well.

The *compound* component that implements the parallel composition operator presents a *supervisor* with the *user automaton* illustrated in figure 4.29, *default automaton* transitions and states are not displayed (except for **end** and **running error** states). This component just runs in parallel (or concurrently) the two components that constitute its operands, and then wait for them to finish their execution. Depending on the result of the execution of the component that has finished last, the composition will terminate either successfully, or failing, transiting respectively to **end** or **running error** states.

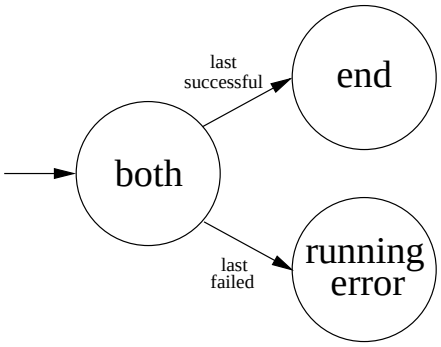

Figure 4.29: Parallel composition:
user automaton.

An example of the application of the parallel composition operator is when we want two components to be executed in parallel or concurrently in order to perform a specific activity in a coordinated way, as for example:

```
...
Component* doMapping=
    new ParallelComposition(new Explore(),
                            new ConstructMap());
...
```

where `Explore` is a component that makes a robot wander around and `ConstructMap` is another component that builds a map of the environment. The `ParallelComposition` class is the *compound* component which implements the parallel composition operator,

and like the other process algebra operators, at instantiation time it adapts its external interface to the output and input ports of its operands.

4.5.4 Disabling Composition

The **disabling composition** operator is represented by the symbol ♯, and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a\sharp b$ is such that an instance of c, c_i, behaves like an instance of a, a_i, and an instance of b, b_i running in parallel (or concurrently), and its state is the state pair conformed by the states of both instances. The compound instance c_i finishes when one its local components finishes, and it takes the same state as this first finished instance, either a_i or b_i, the other is aborted.

The *supervisor* of the *compound* component implementing to the disabling composition operator has the *user automaton* displayed in figure 4.30, *default automaton* transitions and states are not displayed (except for **end** and **running error** states). Like in the parallel composition, in this *compound* component the components that constitute the operands of the composition are ordered to run in parallel (or concurrently), but in this case, the *supervisor* waits only for the termination of one of them. Once one of them has finished, the *supervisor* aborts the other one, and skips to **end** or **running error** states depending on how the component that finished first has terminated, either successfully or failing.

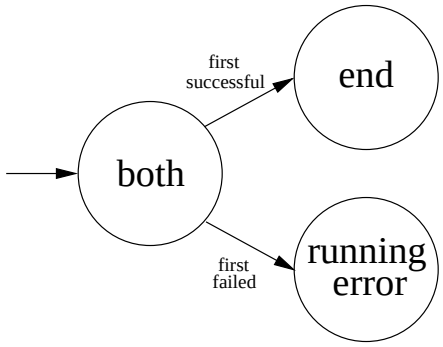

Figure 4.30: Disabling composition: user automaton.

An example for this type of composition is, for instance, the component called `Explore` previously commented, and a component called `Localize` previously commented as well. Thus we could do something like:

```
...
Component* localizePattern=
    new DisablingComposition(new Explore(),
                             new Localize(pattern));
...
```

where we would have the robot wandering around and looking for a pattern. If the pattern was found, the whole composition would finish. Obviously the `DisablingComposition` class is the *compound* component corresponding to the disabling composition

operator, and like the other components implementing process algebra operators presented so far, it adapts its external interface to the output and input ports of the operands it receives at instantiation time.

4.5.5 Synchronous Recurrent Composition

The **synchronous recurrent composition** operator is represented by the symbol **:;**, and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a < v >:; b(v)$ is is recursively defined as $a < v >:; b(v) = a < v >: (b(v); (a < v >:; b(v)))$.

In figure 4.31, we can see the *user automaton* corresponding to the *supervisor* corresponding to the *compound* component that implements the asynchronous recurrent composition operator. *Default automaton* transition and states are not displayed in the figure (except for **end** and **running error** states). Observing the figure, initially the component that constitutes the first operand in the composition is run, if its execution is successful, then the second component (the second operand) is commanded to run. Once that second one has finished, whatever has been its result, the first component is run again, and the whole loop repeats indefinitely until the first component fails. In such a case, the whole composition finishes transiting to **running error** state. Note that like in the conditional composition operator, each time the first component finishes successfully, it may generate data as a result of its execution that, then, is fed it to the second component when it starts running.

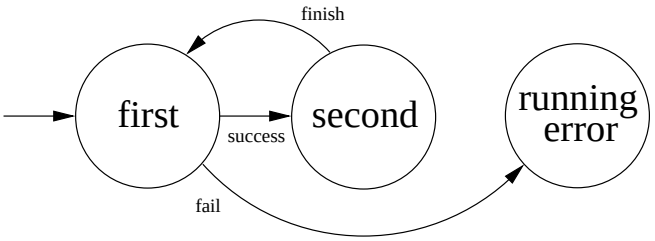

Figure 4.31: Synchronous recurrent composition: user automaton.

An example of this composition might be the following:

```
...
Component* gotoPattern=
    new SynchronousComposition(new Localize(pattern),
                               new NavigateTo(pattern));
...
```

where the same components used in the example given for the conditional composition operator are used, the difference is that in this case the robot keeps trying to navigate towards the pattern while it can localize the pattern. It will finish its navigation only when it fails localizing the pattern. The `SynchronousComposition` class is the *compound* component that implements the synchronous recurrent composition operator, and like the rest of operators, it adapts its external interface to the output and input ports of the operands with which it has been instantiated.

4.5.6 Asynchronous Recurrent Composition

The **asynchronous recurrent composition** operator is represented by the symbol **::**, and is defined in the following terms. Let a and b be two components, whether atomic or not, then the *compound* component $c = a < v >:: b(v)$ is such that is recursively defined as $a < v >:: b(v) = a < v >: (b(v) \| (a < v >:: b(v)))$.

The *compound* component that implements the asynchronous recurrent composition operator presents a *supervisor* having the *user automaton* appearing in figure 4.32 where *default automaton* transition and states are not displayed in the figure (except for **end** and **running error** states). In this composition, the *supervisor* initially runs the component that constitute the first of its operands. Once it has finished, if it succeeded, then the *supervisor* runs in parallel (or concurrently) the second component (the second operand), and again, the first component. From that moment on, each time the component corresponding to the first operand finishes successfully, a new instance of the second operand, and the first operand again are run in parallel (or concurrently). The *supervisor* keeps doing that indefinitely until the first operand fails. In that case the *compound* component, transits to the state called **waiting** in the figure. At that point the component just wait for the termination of all the instances of the second operand that has been ordered to run in the previous state called **main**. Once all of them has finished the whole composition will transits to **end** or **running error** states accordingly to how the last one has terminated. Like in the conditional operator, when each execution of the first operand finishes successfully it may generate data as a result of its task execution. In that case, those data are fed into each instance of the second operand when they get started.

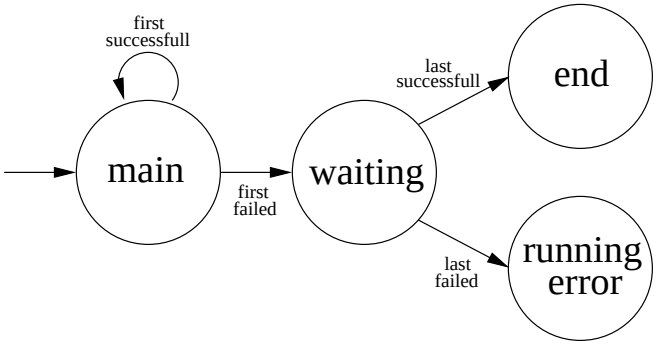

Figure 4.32: Asynchronous recurrent composition: user automaton.

An example of this composition might be the following:

```
...
Component* gotoPattern=
    new AsynchronousComposition(new LocalizeNewPattern(patterns,
                                                        foundPatterns),
                                new Track());
...
```

where `LocalizeNewPattern` is a component that visually localizes a pattern included in the set `patterns` but not included in `foundPatterns`. Each time this component

finishes successfully finding a pattern, it subtracts the pattern from `patterns` and adds it to `foundPatterns`, and returns the pattern. After that the asynchronous composition operator instantiates a new `Track` component that makes the robot track visually the found pattern. Then it repeats the operation until `LocalizeNewPattern` finds all the patterns in `patterns`, or until it is not able to find any one else. Execution of the whole composition finishes when the last instantiated `Track` component loses its target. Using this example the robot is able to visually follow multiple patterns, and keep tracking until it loses all of them. Evidently, the `aynchronousComposition` class is the *compound* component that implements the asynchronous recurrent composition operator, and like the rest of operators, it adapts its external interface to the output and input ports of the operands with which it has been instantiated. Observe that, contrarily to other operands, and due to the fact that this operand creates new instances of its operands, usually it is necessary a kind of shared memory between them to keep the consistence of the execution of the different instances. In the example this shared memory was the two sets of patterns: `patterns` and `foundPatterns`.

Finally it is important to comment here that for all *compound* components presented so far that implement this small set of process algebra operators, if the operand that finally determines the last state of the composition, generates data as a result by means of its default *result* observable variable, the composition will return such data as result of the execution of the whole composition.

4.5.7 Using the Operators

Now we will show how the different configurations of components of figures 4.8, 4.16 and 4.26 can be formulated in terms of the process algebra operators presented along this section. Thus, for the configuration of figure 4.8, a code like this will be possible:

```
...
Component* reactiveAvoiding=
    new ParallelComposition(new Pioneer("/dev/ttyS0"),
                            new PFAvoiding( // working period
                                            100, // milliseconds
                                            // minimum distance to obstacles
                                            300, // millimeters
                                            // persistence time
                                            1000 // milliseconds
                                           ));
...
```

for having the robot wandering around a bit as figure 4.16 shows, we would have:

```
...
Component* reactiveAvoiding=
    new ParallelComposition(new Pioneer("/dev/ttyS0"),
                            new PFAvoiding( // working period
                                            100, // milliseconds
                                            // minimum distance to obstacles
                                            300, // millimeters
                                            // persistence time
                                            1000 // milliseconds
                                           ));
Component* strategicWithAvoiding=
    new ParallelComposition(reactiveAvoiding,
                            new StrategicPF( // working period
```

```
                                          500 // milliseconds
                                          ));
Component* wanderAround=
    new ParallelComposition(strategicWithAvoiding,
                            new Wander( // working period
                                        300000 // milliseconds
                                      ));
...
```

and for the *Go Home* task configuration shown in figure 4.26:

```
...
Component* reactiveAvoiding=
    new ParallelComposition(new Pioneer("/dev/ttyS0"),
                            new PFAvoiding( // working period
                                            100, // milliseconds
                                            // minimum distance to obstacles
                                            300, // millimeters
                                            // persistence time
                                            1000 // milliseconds
                                          ));
Component* strategicWithAvoiding=
    new ParallelComposition(reactiveAvoiding,
                            new StrategicPF( // working period
                                            500 // milliseconds
                                            ));
Component* addingTheVisionServer=
    new ParallelComposition(strategicWithAvoiding,
                            new VisionServer());
Component* goHome=
    new DisablingComposition(addingTheVisionServer,
                             new GoHome());
...
```

which finishes when the `GoHome` instance finishes. It is important to notice that the *compound* components corresponding to the different process algebra composition operators do not make any mapping between their operands, except for adapting their external interfaces to accommodate the output and input ports of them. Port connections among operands need be done by the user/developer in order to run such configurations conveniently.

Chapter 5

Conclusions and Future Work

Along this document a new framework to facilitate the programming of robotic systems has been presented. In this final chapter we will draw some conclusions about where this work has brought us, and we will outline future lines of development.

5.1 Introduction

Traditionally software integration efforts when programming robotic systems has been undervalued. However the heterogeneity of the problems that is necessary to face provokes that an important effort during the development of this kind of systems must be devoted to system integration, mainly to system´s software integration. Software integration for robotic systems has been one of the main problems to which the work presented in this document has been addressed. Nonetheless, some other problems common in the field of programming robotic systems have been taken into account as well. In the rest of this chapter we will comment with more detail where we think we have got to. Thus, in the next section (section 5.2) some final comments will be given about other approaches to the same problem that have been carried out, or are under development in other laboratories, and that we consider closer to the approach proposed in this document. The following section (section 5.3) is a summary of the conclusions we have gathered from the work presented here. Finally, in section 5.4, we will comment briefly which future trends the work presented in this document might follow.

5.2 Final Comments

5.2.1 SmartSoft

SmartSoft [Schlegel and Wörz, 1999a] [Schlegel and Wörz, 1999b] is a framework aimed at providing components for building robotic systems that we have already com-

mented in chapter 2. We consider this framework quite close to the work we present in this document, so we would like to make some further comments taking into account what was said in chapter 2, and also its most recent changes [Schlegel, 2003].

By design, SmartSoft does not impose any restrictions on the internal architecture of the modules it uses to model systems and, in this sense, they are modelled as opaque units. From now on, we will refer indistinctly to SmartSoft modules as components.

The only organizational principles provided by the framework occur at the communication level. In SmartSoft components interact using a small set of basic communication patterns that implement several client/server, publisher/subscriber and master/slave protocols. In its most recent version SmartSoft implements the following primitives for communication patterns: *send* (the former command primitive), *query*, *push-newest* (the former *autoupdate newest* primitive), *push-timed* (the former *autoupdate timed*), *event*, *wiring* (this is a new primitive) and *state* (the former *configuration* primitive). As was commented, the send and query patterns implement, respectively, a one-way and request/response interaction. Push-newest and push-timed are used, respectively, to distribute data from one producer to n consumers on an irregular or regular basis. The event pattern is used for asynchronous notification of events. The wiring and state patterns are devoted to control and coordination issues. The wiring pattern is a new pattern that provides support for connecting components dynamically, i.e. at runtime. The state pattern is used for setting up a basic activity coordination mechanism cancelling blocking calls or forcing state changes in modules.

SmartSoft has some strong points of coincidence with CoolBOT. Both frameworks conceived a perception-action system as a network of components whose functionality and capabilities arise from the coordinated interaction of individual components. And for that purpose both frameworks provide a reduced set of communication patterns, or ICC mechanisms as termed in CoolBOT, that essentially implement similar basic set of communication protocols with an emphasis on asynchronicity. Both frameworks also hide the difficult issues of concurrency and synchronization to the programmer. There are, however, some differences. The most noticeable is that CoolBOT provides the concept of port type as a second layer over the basic low-level communication patterns. Fifos, posters, etc. are readily available on CoolBOT while they must be implemented by the programmer in SmartSoft. Certainly, CoolBOT could have left the task of implementing these ports to programmers, but that would have been a very frequent source of programming errors and - more importantly - there would not be warranties about the interoperability of two components communicating through ports of the same type, as far as they had been implemented by different programmers.

Another important difference between SmartSoft and CoolBOT relies on the assumptions that each framework poses on the internal architecture of components. In CoolBOT we have considered that composing a complex system using a modular set of components requires not only communication, but also monitoring and control facilities. While in principle SmartSoft considers components as opaque modules or units of interaction, CoolBOT imposes a common internal architecture on all components,

mainly reflected on the use of the same state automaton and a minimal common interface for monitoring and control, i.e. the control and monitoring ports. Nevertheless, SmartSoft incorporates the state pattern that seems to be not a mere communication pattern, but a pattern concerned with control and coordination issues. It is rather obscure if using this pattern components can still be considered as opaque entities, or there must exist a minimal internal control structure that must be shared among components.

Finally, the wiring pattern of SmartSoft stands for what in CoolBOT is the set of operations for establishing and de-establishing port connections at runtime. Evidently this is a key feature when we deal with robotic systems. Changing the conditions and contexts in which a system is evolving should be reflected in changes of the topology of components which conforms and integrate the software that controls that system.

5.2.2 G^{en}oM

G^{en}oM [Fleury et al., 1997] [Fleury and Herrb, 1998] is part of a multilevel robotic architecture developed at LAAS that has been applied in a large number of projects and used over many different robots. Like SmartSoft, G^{en}oM was also commented in chapter 2. G^{en}oM (Generator of Modules) is the layer of the LAAS architecture responsible of defining the components (formerly termed modules in LAAS jargon) of a robotic system and is mainly concerned with the definition of the internal architecture of components. It considers that a component is made up of three entities: a set of algorithms, an execution engine and a communication library. Algorithms are formulated as elementary pieces of code or codels. These codels constitute the atomic instructions that the execution engine sequences, and are always run atomically, i.e. they can't be interrupted. The execution engine is generic and is in charge of executing the codels, managing the internal state of the component and handling data and control flow. A communication library is used to offer communication services to components.

In G^{en}oM's most recent version, a strong emphasis has been placed on making codels as independent as possible of the rest of the framework. The idea is to clearly separate code that is part of the framework (execution engine and communication library) from the code that really constitute the component (*codels*). Codels are grouped into a codel library and is this library the only part of the system that need to be shared between researchers to exchange components. Both the execution engine and the communication library are part of the default framework and independent of codels. The execution engine sequences codels that implement component's services. This sequencing is modelled as a state automaton with codels associated to states and transitions between states, very much like it is done in CoolBOT. Interestingly, the execution engine is designed to offer bounded execution time warranties to components and, although it is presently not available, execution engines specially suited to hard-real time or particular operating systems could be designed. The communication library seems to offer only a basic poster pattern and it looks to be much less developed that the counterparts of SmartSoft or CoolBOT.

G^{en}oM has been a strong source of inspiration for modelling CoolBOT's components and, not surprisingly, they share many features. The main - as opposed to SmartSoft - is that components in both frameworks have a well defined internal architecture organized around a state automaton. This characteristic, that does not compromise the functionality of the component, seems to be central in order to achieve properties like reactivity, uniform error recovery mechanisms and generic control and monitoring procedures for both approaches. As a framework, CoolBOT makes available a richer set of communication facilities between local or distributed components, in the form of port types and ICC mechanisms, than G^{en}oM. The concept of codels, although much more delineated in G^{en}oM, is also shared between both frameworks. G^{en}oM associates codels to services, states and state transitions. In CoolBOT, states and state transitions have associated handlers that play the same role. Nevertheless, a major difference between both approaches exists in the definition of codels and handlers. Currently, CoolBOT components are in the last term C++ program files, while G^{en}oM codels are declared using a devoted language making explicit many important parameters and options. This effectively allows for decoupling generic definition of components (i.e. codels) from specific instantiations of the same components. Eventually, this information could be useful for a low level planner or scheduler.

5.2.3 Orocos

The EU-funded OROCOS project [Orocos, 2003] has been launched with the aim of "defining the robot software of the future" and shares many goals with SmartSoft, G^{en}oM and CoolBOT. The LAAS lab is a partner in the OROCOS project, together with K.U. Leuven and CAS/KTH (Stockholm), and Christian Schlegel, designer of SmartSoft, is also associated to this project.

Orocos is still an on-going project that has three open fronts. K.U. Leuven, the project leader, has been working on providing a realtime control framework using realtime versions of the GNU/Linux OS. This work has produced two main contributions. One is a layer for abstracting device drivers, called the Framework Device Interface (FDI). The other is a portable abstraction of operating system's details regarding system calls, threads interface, IPC mechanisms, etc. This layer is termed as the Framework Operating System Interface (FOSI). Another of the partners, CAS/KTH in cooperation with Christian Schlegel is working on achieving a CORBA-based communication framework that should be considered as a new version of SmartSoft. In this front there seems to be interest on addressing real-time communication issues using real-time implementations of CORBA. Finally, the LAAS group has been working on a second version of G^{en}oM that should provide OROCOS framework with an event based, realtime and programmable execution engine for handling component's control flow. As already pointed out, OROCOS is still an open project whose achievements need to be evaluated.

5.3 Conclusions

Some conclusions have been drawn from the work that has been presented in this thesis. They have been organized into two groups: general conclusions, and conclusions related to the experiments.

5.3.1 General Conclusions

5.3.1.1 Uniformity

Having independent and asynchronous units of execution that share the same basic internal structure and the same basic external interface for monitoring and control, allows a basic treatment of such entities in spite of the functionality they have individually. This is a wide-spread and well-know principle in the operating system community. In fact, one of the main questions an operating system developer must face is how to model processes and threads for a specific operating system, and which services and primitives it provides for such a model. All operating system imposes on the programs they can execute, an uniform internal structure and an uniform external interface in order to make them uniformly treatable, administrable and executable by the operating system independently of the functionality these programs have. As a consequence, programs for a given operating system are deployable units of software that does not need any further modification to be executed.

Chimera is a clear example of this idea applied in the robotic field as we commented in chapter 2. Chimera is a real time operating system, aimed to robotics and developed at CMU [Stewart and Khosla, 1993] [Stewart and Khosla, 1996] [Stewart et al., 1997] [Stewart, 1994] where the minimal units of execution that the operating system handles, are port-based objects with a uniform structure and interface. These features guarantee reusability, deployment and recycling of these port-based objects between machines running under Chimera.

Following the same idea, CoolBOT imposes some uniformity on the units of execution it defines, the CoolBOT components. This uniformity makes components externally observable and controllable, and treatable by the framework in an uniform and consistent way. Furthermore, the uniformity the framework imposes makes them also integrable with other components to built more complex systems, in a way that converts them in elements of deployable software.

5.3.1.2 Deployment, Reuse and Recycling of Components

CoolBOT components, once they have been designed, built and tested, constitute software units providing independent functionalities that may be used where needed. CoolBOT components are black boxes that hide their functionality behind an interface of output and input ports together with a set of observable and controllable variables. In a given system it is easy to interchange components having the same external interface

and similar functionality.

5.3.1.3 Visibility

Access to component internal state and structure by means of the default *monitoring* and *control* ports, makes components observable and controllable. The internal operation of any component can be observed by means of its *monitoring* port. On the other side, components can be controlled by means of their *control* port in order to drive them to a specific controlled state.

5.3.1.4 Inter Component Communications

Having multiple flows of execution sharing the same computer resources and intercommunicating between them to achieve a specific functionality is a recurrent problem when programming robotic systems. And, although it is a frequent problem, and due to the complexity of these kind of systems, it is not a trivial question to which it is necessary to pay much attention during design and implementation. In fact, oftely, it is also the source of critical and hard-to-identify bugs.

CoolBOT fosters asynchronous execution and inter-communication, what facilitates runtime uncoupling and promotes execution driven by data, i.e., components consume CPU resources when they have something to process at their input ports. Inter component communication (ICC) is carried out by means of port connections formed by pairs of output and input ports. CoolBOT provides a rich typology of output and input ports that allow for a wide set of port connections, modelling multiple patterns of interaction between components. All these types of port connections utilize internally the same set of basic inter component communication mechanisms that minimize inter component synchronization (the cache motto), and uncouple the processing of components involved in any connection (asynchronous communications). In addition, remote components can be reachable locally by means of an infrastructure of *proxy components* and *CoolBOT servers*.

A component developer does not have to worry about synchronization issues due to simultaneous accesses between components, neither does it have to worry about if the component he/she is implementing will interact only with local or remote components, at design time this is irrelevant. The same objects and methods are used to communicate with local and remote components. Thus, in terms of inter component communications, he/she only has to worry about the output and input ports his/her component will offer through its external interface, and the port packets that these output and input ports will accept.

5.3.1.5 Multithreading

Concurrency and parallelism are serious issues to which is necessary to dedicate many efforts when programming robotic systems. Simultaneous and not-synchronized ac-

cesses to the same resources by multiple units of execution are in the origin of a wide set of problems. The use of multiple operating systems with different thread models introduce even a higher level of difficulty.

In CoolBOT, multithreading is modelled at three levels: the level of *compound* components where a *supervisor* (a thread) runs the component's automaton, and monitors and controls a set of component instances (and even *port threads*, if necessary); the level of *atomic* components where the *main* thread executes the automaton and controls and monitors a set of *port threads*; and finally, the level of *port threads* where threads are modelled as input-port-driven threads. In a given system the threads running at a specific instant of time synchronize each other, irrespective of their thread level, in the same way, using port connections formed by output and input ports, that in the case of *port threads* will be private output and input ports. That supposes an important saving of effort at the time of programming robotic systems. In CoolBOT, the different units of execution (*compound* and *atomic* components, and *port threads*) are modelled as active entities, driven by the data they receive through their input ports, and that produce their results using their output ports. Thus, a developer, once he/she models a component in these terms, should not be worried about simultaneous non-synchronized accesses to shared resources or critical sections, the framework cares about all of it behind the scenes.

5.3.1.6 A Model for Exception Handling

Robustness is a permanent design goal in every robotic project, however it is also difficult to achieve due to the complexity involved: sensors and effectors have their own set of specific operating errors; the same is applicable to different operating systems; equally each data link protocol and hardware generates its own specific mal-function situations; on the other side, some necessary legacy, third-party or reused code may utilize its particular scheme for error handling (for instance, C++ exception versus C return values); in addition, different programmers and developers involved in the same project can pose different strategies for error recovery; and finally, it is necessary to add that the problem the system has to solve has its own particular set of errors. All this makes difficult to design from scratch a "normalized" model for dealing with errors, exceptions, and faulty situations in every system. We think this problem would be simplified if the framework we use to program systems provided us with a coherent model for dealing with errors.

CoolBOT promotes an uniform approach for handling faulty situations, establishing a local and an external level of exception handling. Exceptions are first handled locally in the components where the exceptions come out. If they can not be handled at this local level, they are deferred to the *compound* component where these components are included. This handling scales, in turn, going up in a hierarchy of components. In addition to that, it is possible to provoke externally the occurrence of an exception in a given component (by using the *new exception* controllable variable). This feature permits a better testing and debugging of components when they take part into larger systems, systems of which the component was ignorant when it was designed and built.

It should be pointed out that this feature can be exploited without having to modify a single line of code. This is an interesting feature in order to develop better and more robust components and systems, because it is not only possible to test a component individually, but also to test what happens when something goes wrong in a complex system conformed by multiple components.

5.3.1.7 Strong Design Requirements

Certaninly, CoolBOT imposes a number of strong requirements on components, namely:

- The uniformity of internal structure and external interface.
- The modelling of components as functional units of independent execution that performs all its external communication by means of its external interface of ports accepting a specific set of port packets
- The uniform approach for component exception handling that the framework imposes when building components.

These are strong design requirements that could be considered as too-tightening demands that may limit in excess programmer´s freedom. Our experience is just the opposite. These design requirements should be better considered as framework facilities that disciplines the programmer/developer and obliges him/her to pay much attention to component design and engineering. All that makes software less error-prone.

5.3.1.8 Generality

CoolBOT has been devised in order to facilitate the development of software aimed at controlling robotic systems, but evidently it might be also useful for building software in other computer science fields that share the same problems and issues, and where the abstractions, the programming philosophy and the infrastructure that CoolBOT offers are also useful. As an example, a field where we consider that CoolBOT could be applied with interesting expectatives is multi-agent systems.

5.3.1.9 Asynchronous Model of Execution

In general, CoolBOT favors a programming methodology that fosters concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing. Thus, in CoolBOT, the behavior of a whole system comes up from the integration of the components taking part into the system, and from the coordination of the interaction between the behaviors of all of them. Computationally, a working system, in terms of CoolBOT, is a network of components wired through port connections, whose execution may be distributed along the whole network, and is driven by the data the port connections transport.

5.3.1.10 Control vs. Functionality

CoolBOT provides means to separate control and functionality when programming systems. It is not difficult to separate control and functionality in components. In *compound* components, control may be concentrated exclusively in the *supervisor*; the functionality can be obtained by means of the integration and coordination of the different *local* component instances. In *atomic* components, the *main* thread may be used as the repository of the control code of a component, meanwhile the functionality the component provides may be implemented in one or more *port threads*. Notice that, in this way, it is possible to reduce the *main* thread to a minimum, which would just need to pay attention to the updates that the controllable variables of the component may receive. Having a minimal *main* thread makes a component completely responsive to control actions exerted on it, making possible to achieve faster control loops.

5.3.1.11 Operating System Support

Currently, CoolBOT is supported under the Windows family of operating systems (Windows NT, 98, 2000 and XP), and GNU/Linux. Given that those operating systems are not real-time, and that CoolBOT relies on the thread model of the underlying operating systems to map its support for multithreading, the framework can not guarantee any realtime requirement by itself. However, CoolBOT provides some mechanisms and resources which usually are characteristic of real time operating systems (timers, watchdogs, etc.), and it can keep soft real-time requirements, since the operating systems where it runs actually can keep such requirements [Ramamritham and Shen, 1998] [Gopalan, 2001].

5.3.2 Experimental Conclusions

5.3.2.1 Port Connections and ICC Mechanisms

Some experiments have been carried out to clarify which typology of port connections is more convenient for one-producer-multiple-consumers configurations. The results obtained from these experiments have confirmed the internal design given to each one of the types of port connections under study, although some interesting conclusions have been also drawn. One of them is that synchronization costs can be dominant in some situations, specially when the number of components involved in the same port connection grows. Other interesting conclusion is that results in the operating systems above which CoolBOT currently executes (the family of Windows operating systems – Windows NT, 98, 2000 and XP – and GNU/Linux) are very similar in spite of the thread models each one utilizes internally. All in all, results have allowed us to establish some rules or recipes of design regarding some common situations, such as high frequency producers, high frequency consumers and the sharing of complex data structures, that would be useful for guiding design decisions when building CoolBOT components.

5.3.2.2 Incremental Development

As proof-of-concept some basic examples has been built and tested using CoolBOT. From a mobile robot initially showing an avoiding obstacle behavior that allowed to herd it, the examples scaled up in small steps in order to add new capacities to the robot. Thus, we got to a robot with a wandering behavior that could avoid obstacles, and after that, we ended up with a more capable robot able to get to a docking station on its own. Along all these steps of achievement we have shown that component composition promotes and facilitates incremental development, and incremental debugging. We think following such an approach of development would facilitate the construction of more complex and capable systems out of a wide and diverse set of capable components.

5.3.2.3 Component Reuse

One of the main motivations that gave birth to the work presented in this document was the necessity of a methodology to avoid duplication, reconstruction from the scratch, and re-engineering of software already operative. The approach followed in CoolBOT has been based on defining components as units of deployment that may be easily integrated wherever needed. In the basic examples used as proof-of-concept of the framework the same components were used, in different systems without further modifications.

5.4 Future Work

In this document we have presented a tool for programming robotic systems in the form of a framework of C++ classes that favors a specific programming methodology based on component construction and integration. Also, a few examples illustrating some meaningful characteristics of the framework have been shown, but, nevertheless, there are many exciting lines of future developments within CoolBOT. The following paragraphs enumerate the next steps we think will be important sources of future work.

5.4.1 Development Tools

Currently, developing CoolBOT components once they have been designed, is a quite systematic and repetitive process, at least to get to a first working component skeleton. Usually the process consists in starting with the skeleton code of a component, and modifying it in order to obtain the functionality we want. There is already a small set of developed components, and component examples illustrating the most common patterns of use. Although systematic, this process is cumbersome and boring. A compiler that were able to generate component skeletons automatically from a description code, would be more than desirable. It will not only reduce the programming skills and knowledge about CoolBOT internals currently needed for programming components,

but it will also reduce drastically programming errors shortening to the same extent development time.

A tool that would be also clearly interesting is a component debugger where it could be possible to observe any component through its *monitoring* port, and modify its internal behavior using its *control* port using friendly graphical user interfaces.

Finally we consider also of great interest the availability of a component profiler, a tool that were able to show chronograms relating execution times of the different running components conforming a specific system.

5.4.2 Support for Real Time Operating Systems

Currently, CoolBOT operates under the Windows family of operating systems (Windows NT, 98, 2000 and XP), and GNU/Linux. Although CoolBOT provides some mechanisms and resources which usually are characteristic of real time operating systems (timers, watchdogs, etc.), CoolBOT is not a real-time framework. At last term, it relies in the thread model that the underlying operating system utilizes, therefore if the operating system is not real-time, then, consequently, the framework is not real-time. The experience with the operating systems where CoolBOT works actually indicates that the framework is useful in robotic systems with soft-real time requirements like, for instance, active vision systems and entertainment robotics. Evidently, this is possible because such operating systems really allow to keep soft-realtime requirements [Ramamritham and Shen, 1998] [Gopalan, 2001]. In order to give support to robotic applications with hard real-time constraints we are considering to port CoolBOT to one real-time operating system like RTLinux [RTLinux, 2003] or RTAI [RTAI, 2003] might be an interesting line of future work, providing that, on one side CoolBOT already implements an operating system abstraction layer which would facilitate its porting to other operating systems. On the other side, surely it would be very attractive to know how ICC mechanisms could take advantage of some typical features available in such operating systems, such as the use of real-time watchdogs, the possibility of influencing the operation of the real-time scheduler, etc.

5.4.3 Network Support

Currently, CoolBOT has a network support based on a connection-oriented model of communication between components residing in different computers in a network. Components can only reach other components in other machines if they are already in execution. We consider an interesting feature, a service to instantiate components remotely. Maybe to do so it would be interesting the porting of the current network support based on TCP/IP sockets to a CORBA [Henning and Vinoski, 1999] implementation where there are plenty of network services available.

5.4.4 Component Graphical Interfaces

Graphic interfaces are particularly a source of problems when code reuse and recycling is a requirement, specially if portability across different operating systems is an issue. Windowing APIs rely usually on specific operating systems, and the appearance they endow to applications may be very different. Sometimes the effort necessary to design and implement a consistent visual-look for the applications conforming a project is bigger than expected, due to the complexity of obtaining consistency in appearance, and keeping, at the same time, the code portable. We consider that un-coupling the graphical interface of components from the functionality they give, and abstracting the graphical API that is used at last term in each machine under which CoolBOT is executed, could reduce considerably the effort devoted to the design of graphical interfaces when constructing components and systems.

5.4.5 Development of Complex Demonstrators

In this document some basic small examples have been presented to illustrate the features and means that CoolBOT offers to robotic system developers. Evidently, the more components we have, the more complex the systems we could build would be. How long does CoolBOT scale?. Only the development of complex demonstrators out of a wide-enough set of robust ready-to-use components will bring an answer to this question.

5.4.6 Framework Promotion

A framework like CoolBOT without users has really no interest. We think CoolBOT is a valid and useful approach to program robotic systems, and promoting their use is obviously an objective that we have in mind. Evidently, the development of complex demonstrators is a mean of promotion by itself, but other means of diffusion are also interesting: the elaboration of how-to documents and tools such as compilers, debuggers and profilers, more examples showing component patterns of use, the creation of a repository of diverse and different ready-to-use components reflecting the implementation of robotic state-of-art algorithms, licensing the CoolBOT source code using the GPL software license [FSF, 2003], supporting vehicle controllers for the most popular mobile robots, supporting real time operating systems, etc.

5.4.7 CoolBOT as a Long-Term Experimental Tool

For us, CoolBOT is a long-term experimental tool of which this work is only a first design and development effort. The development of programming tools, the definition and adding of an uniform and uncoupled graphical interface model for components, the porting of CoolBOT to real-time operating systems, the built of complex demonstrators, and the promotion of the framework to increase the user base, all them constitute

future objectives we foresee that, finally, will help to validate this methodology on the long-term.

Appendix A

CoolBOT Programming Style

C++ code that implements CoolBOT has been written according to the syntactic style and coding rules that will be presented in this appendix.

A.1 Naming

As to naming there are four main rules of coding style, the first one of them affects identifiers of macros, enumeration constants and constants, the second one affects the rest of identifiers, the third one is related to type identifiers, and the last one can affect all kind of identifiers.

A.1.1 Macros, Enumeration Constants and Constants

Identifiers of macros and constants that are used as macros, and enumeration constants are spelled completely in uppercase letters. Besides, if they are formed by more than a word, these are separated by a "_" character, figure A.1 illustrates the rule.

```
...
#define THIS_IS_A_MACRO ...
...
const int THIS_IS_A_CONSTANT = ...;
...
enum AnEnumeration { FIRST_VALUE, ..., LAST_VALUE };
...
```

Figure A.1: Macros and constants codification.

A.1.2 Other identifiers

Identifiers of:

- class data members, whether static or not,
- class member functions, whether static or not,
- global variables, global objects and global functions,
- automatic variables and objects defined inside a block, even if they are affected by the `static` modifier,
- any function formal parameter,
- and variables, objects and functions defined in file scope, i.e., variables, objects and functions defined out of class or name space scope, whether affected or not by the `static` modifier.

should be spelled in lowercase letters. In case of being formed by more than one word, the second one and the rest of words should start with a capital letter and they are written without any other character between them, illustrated by figure A.2.

```
...
AType aGlobalOrFileScopeVariable;
extern OtherType aGlobalVariable;
static AnotherType aFileScopeVariale;

JustAType aGlobalOrFileScopeFunction(...) { ... }
extern ATypeAgain aGlobalFunction(...) { ... }
static OneMoreType aFileScopeFunction(...) { ... }
...
class AClass
{
    public:

        static SomeType aStaticDataMember;
        static AnyType aStaticMemberFucntion(...);

        SomeOtherType aDataMember;
        AnyOtherType aMemberFucntion(OneType aParameter,...)
        {
            TheLastType anAutomaticVariable;
            ...
        }
        ...
};
...
```

Figure A.2: Generic naming of identifiers.

A.1.3 Types

Type identifiers (classes, structs, typedefs, ...) should always be spelled in lowercase letters starting with an uppercase. If the identifier is compound by more than a word,

all of them are spelled in lowercase letters starting also with an uppercase and written without any other character or letter separating them. Figure A.3 illustrates this rule.

```
...
enum AnEnumeration { ... };
...
class AClass { ... };
...
typedef AType ATypeAlias;
...
struct AnotherClass { ... };
...
```

Figure A.3: Naming for type identifiers.

A.1.4 Prefixes and Suffixes

Identifiers can be affected by several prefixes and suffixes. These prefixes and suffixes are used to indicate the class access modifiers affecting an identifier and, to know whether the identifier is either a pointer or a reference.

- **Access Prefixes and Suffixes**

  In the scope of a class, any identifier, even for subtypes, should include a prefix or/and a suffix, depending on its access modifier, according to the next rules:

  - **Private access**: Any identifier with the modifier `private` will be prefixed by a "`_`" character and also suffixed by another "`_`" character.
  - **Protected access**: Any identifier with the modifier `protected` will be prefixed by a "`_`" character.
  - **Public access**: Public identifiers will not be affected by any access prefixes or suffices.

  Figure A.4 illustrates these rules.

- **Pointer and Reference Prefixes**

  Identifiers referred to pointers and references will be respectively prefixed according to:

  - **Pointers**: An identifier of a pointer will be prefixed by a "`p`" lowercase letter, if it is a double pointer, will be prefixed by two "`p`" lowercase letters. By three "`p`" ´s if it is a triple pointer, and so on.
  - **Pointers to Members**: An identifier of a pointer to a class data member will be prefixed by a "`pm`" prefix.
  - **References**: An identifier of a reference will be prefixed by a lowercase "`r`" letter.

```
...
class AClass
{
    public:
        ...
        class APublicSubType { ... };
        AType aPublicDataMember;
        OtherType aPublicMemberFunction(...);
        ...
    protected:
        ...
        class _AProtectedSubType { ... };
        AType _aProtectedDataMember;
        OtherType _aProtectedMemberFunction(...);
        ...
    private:
        ...
        class _APrivateSubType_ { ... };
        AType _aPrivateDataMember_;
        OtherType _aPrivateMemberFunction_(...);
        ...
};
...
```

Figure A.4: Access prefixes and suffixes.

- **References to Members**: An identifier of a reference to a class data member will be prefixed by a "`rm`" prefix.
- **Function Pointers**: An identifier of a function pointer will be prefixed by the "`pf`" prefix.
- **Member Function Pointers**: An identifier of a pointer of a class member function will be prefixed by the "`pmf`" prefix.

When an identifier is affected by any of these prefixes, the first word of the identifier should start with a capital letter, contrary to what was indicated in section A.1.2 (see figure A.5). Rules affecting pointers and references are not applied to function identifiers, whether member functions or not, and neither to macros, constants used as macros, and enumeration constants.

A.2 Brace Style

This section is dedicated to where and how to open and close braces to define code blocks. The generic rule used in CoolBOT coding has been to put an opening brace "`{`" starting a block in its own line and the same to finish the block using a closing brace "`}`", as shown in figure A.6.

When a block involves just a few short sentences, it is possible to put everything in the same line, braces and sentences, see figure A.7.

```
...
AType* pAGlobalOrFileScopePointer;
OtherType** ppAGlobalOrFileScopeDoublePointer;
AnotherType& rAGlobalOrFileScopeReference;
OneMoreType (*pfAGlobalFunctionPointer)(...);

AgainAType AClass::* pmADataMemberPointer;

AndAgainTheSameType AClass::& rmADataMemberReference;

ReturnType (AClass::* pmfAMemberFunctionPointer)(...);

 ...

class AClass
{
    public:
        ...
        MyType* pAPublicPointerDataMember;
        ...
    protected:
        ...
        YourType** _ppAProtectedDoublePointerDataMember;
        ...
    private:
        ...
        HisType& _rAPrivateReferenceDataMember_;
        HerType& _aPrivateMemberFunction_(...);
        TheirType (* _pfAPrivateFunctionPointer_)(...);
        ...
};
...
```

Figure A.5: Prefixes for pointers and references.

A.3 Indentation

As to indentation the rule in CoolBOT is to indent with a `tab` space everything that is included in a block, even blocks consisting just in only one sentence without braces. Note, however, that braces delimiting the block, if any, are not indented. Furthermore, code gets indented also with a `tab` space when it follows a `public`, `protected` or `private` access modifier in the definition of a class, as can be observed in figures A.6 and A.7.

A.4 Other Comments and Remarks

The different figures along this document try to illustrate all the coding style rules using in CoolBOT, but if there were any doubt respect to any aspect that has remained ambiguous, be free to have a look at the real code to observe them. All CoolBOT code

```
...
AType aFunction(...)
{
    ...
    for(....)
    {
        ...
    }
    ...
    if(...)
    {
        ...
    }
    else
    {
        ...
    }
    ...
}
...
class AClass
{
    ...
};
...
```

Figure A.6: Brace coding for blocks.

```
...
class AClass
{
        ...
    private:
        ...
        void _defaults_()
        { _aDataMember_=...; _anotherDataMember_=...; ...; }
        ...
        void _release_()
        {  _justOtherDataMember_=...; _oneMoreDataMember_=...; ...; }
        ...
};
...
```

Figure A.7: One line blocks.

is spread under a directory root that should have been called `coolbot`.

Bibliography

[Activ Media Robotics, 2002] Activ Media Robotics (2002). Pioneer 2/PeopleBot Operations Manual Version 11. http://robots.activmedia.com.

[Activ Media Robotics, 2003] Activ Media Robotics (2003). Aria Reference Manual Version 1.2.0. http://robots.activmedia.com.

[Alami et al., 1995] Alami, R., Aguilar, L., Bullata, H., Fleury, S., Herrb, M., Ingrand, F., Khatib, M., and Robert, F. (1995). A General Framework for Multi-robot Cooperation and its Implementation on a Set of Three Hilare Robots. In *4th International Symposium on Experimental Robotics (ISER'95)*, pages 26–39, Stanford, CA, USA.

[Alami et al., 1998] Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An Architecture for Autonomy. *International Journal of Robotics Research (Special Issue on Integrated Architectures for Robot Control and Programming)*, 17(4):315–337.

[Andersson et al., 1999] Andersson, M., Orebäck, A., Lindström, M., and Christensen, H. I. (1999). ISR: an Intelligent Service Robot. Intelligent Sensor Based Robotics. Lecture Notes in Artificial Intelligence, Springer Verlag, Heidelberg.

[Arkin, 1989] Arkin, R. C. (1989). Motor Schema-Based Mobile Robot Navigation. *International Journal of Robotics Research*, 8(4):92–112.

[Arkin, 1992] Arkin, R. C. (1992). Homeostatic Control for a Mobile Robot: Dynamic Replanning in Hazardous Environments. *The International Journal of Robotics Research*, 9(2):197–214.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior Based Robotics*. The MIT Press.

[Arkin and Balch, 1997] Arkin, R. C. and Balch, T. R. (1997). AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence (JETAI)*, 9(2-3):175–189.

[Arnold et al., 2000] Arnold, K., Gosling, J., and Holmes, D. (2000). *The Java Programming Language*. Addison-Wesley Java Series. Addison Wesley.

[Balch, 2000] Balch, T. (2000). Teambots. http://www.teambots.org.

[Bloomer, 1992] Bloomer, J. (1992). *Power Programming with RPC*. O'Reilly & Associates, Inc.

[Bonasso et al., 1997] Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D., and Slack, M. (1997). Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2):237–256.

[Bover and Cesati, 2001] Bover, D. P. and Cesati, M. (2001). *Understanding the LINUX Kernel*. O'Reilly, First edition.

[Brooks, 1986] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

[Cabrera et al., 2000] Cabrera, J., Hernández, D., Domínguez, A. C., Castrillón, M., Lorenzo, J., Isern, J., Guerra, C., Pérez, I., Falcón, A., Hernández, M., and Méndez, J. (2000). Experiences with a museum robot. Workshop on Edutainment Robots 2000, Institute for Autonomous Intelligent Systems, German National Research Center for Information Technology, Bonn, 27-28 September, Germany. Also available through URL ftp://mozart.dis.ulpgc.es/pub/Publications/eldi5p.ps.gz.

[Cabrera-Gámez et al., 2000] Cabrera-Gámez, J., Domínguez-Brito, A. C., and Hernández-Sosa, D. (2000). Coolbot: A component-oriented programming framework for robotics. Dagstuhl Seminar 00421, Modelling of Sensor-Based Intelligent Robot Systems, To appear in Springer Lecture Notes in Computer Science in summer 2001. Centro de Tecnología de los Sistemas y de la Inteligencia Artificial (CeTSIA), University of Las Palmas de Gran Canaria, Edificio de Informática y Matemáticas, Campus Universitario de Tafira, 35017 Las Palmas, SPAIN.

[Chappell, 1996] Chappell, D. (1996). *Understanding ActiveX and OLE - A Guide for Developers & Managers*. Microsoft Press.

[Clark et al., 1992] Clark, R. J., Arkin, R. C., and Ram, A. (1992). Learning Momentum: On-line Performance Enhancement for Reactive Systems. In *IEEE International Conference on Robotics and Automation*, pages 111–116, Nice, Francia.

[Coradeschi and Saffiotti, 2003] Coradeschi, S. and Saffiotti, A. (2003). An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96. Special issue on perceptual anchoring. Online at http://www.aass.oru.se/Agora/RAS02/.

[Coste-Maniere and Simmons, 2000] Coste-Maniere, E. and Simmons, R. (2000). Architecture, the Backbone of Robotic Systems. Proc. IEEE International Conference on Robotics and Automation (ICRA'00), San Francisco.

[Domínguez-Brito et al., 2000a] Domínguez-Brito, A. C., Andersson, M., and Christensen, H. I. (2000a). A Software Architecture for Programming Robotic Systems based on the Discrete Event System Paradigm. Technical Report CVAP 244, Centre for Autonomous Systems, KTH - Royal Institute of Technology), S-100 44 Stockholm, Sweden.

[Domínguez-Brito et al., 2002] Domínguez-Brito, A. C., Hernández-Sosa, D., and Cabrera-Gámez, J. (2002). Programming with Components in Robotics. Waf 2002 - III Workshop Hispano-Luso en Agentes Físicos, Murcia.

[Domínguez-Brito et al., 2000b] Domínguez-Brito, A. C., Hernández-Tejera, F. M., and Cabrera-Gímez, J. (2000b). A Control Architecture for Active Vision Systems. Frontiers in Artificial Intelligence and Applications: Pattern Recognition and Applications, M.I. Torres and A. Sanfeliu (eds.), pp. 144-153, IOS Press, Amsterdam.

[Fedor, 1993] Fedor, C. (1993). TCX - An Interprocess Communication System for Building Robot Architectures: Programmer´s Guide to version 10.xx. Carnegie Mellon University, Pittsburgh, Pennsylvania.

[Firby, 1989] Firby, R. J. (1989). *Adaptive Execution in Dynamic Domains*. PhD thesis, Departament of Computer Science, Yale University.

[Firby et al., 1995] Firby, R. J., Kahn, R. E., Prokopwicz, P., and Swain, M. J. (1995). An Architecture for Vision and Action. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 72–81, Montreal, Canada.

[Fleury and Herrb, 1998] Fleury, S. and Herrb, M. (1998). G^{en}oM: Manuel d´Utilisation. LAAS Report N°98xxx, http://www.laas.fr.

[Fleury et al., 1997] Fleury, S., Herrb, M., and Chatila, R. (1997). G^{en}oM: A Tool for the Specification and the Implementation of Operating Modules in a Distributed

Robot Architecture. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 842–848, Grenoble, Francia.

[FSF, 2003] FSF (2003). The GNU Project. http://www.gnu.org. The Free Software Foundation.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

[Gat, 1992] Gat, E. (1992). Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 809–815, San Jose, CA, USA.

[Gat, 1997] Gat, E. (1997). ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents. In *Proceedings of the IEEE Aerospace Conference*, pages 319–324, Aspen, CO, USA.

[Gopalan, 2001] Gopalan, K. (2001). Real-Time Support in General Purpose Operating Systems. ECSL Technical Report TR92, Experimental Computer Systems Labs, Computer Science Department. State University of New York at Stony Brook.

[Guerra-Artal, 2002] Guerra-Artal, C. (2002). *Contribuciones al Seguimiento Visual Precategórico*. PhD thesis, Departamento de Informática y Sistemas. Universidad de Las Palmas de Gran Canaria.

[Guzzoni et al., 1997] Guzzoni, D., Cheyer, A., Julia, L., and Konolige, K. (1997). Many Robots Make Short Work. *AI Magazine*, 18(1):55–64.

[Henning and Vinoski, 1999] Henning, M. and Vinoski, S. (1999). *Advanced CORBA Programming with C++*. Addison-Wesley Professional Computing Series. Addison-Wesley.

[Hernández-Sosa, 2003] Hernández-Sosa, J. D. (2003). *Adaptación Computacional en Sistemas Percepto-Efectores. Propuesta de Arquitectura y Políticas de Control*. PhD thesis, Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria.

[Hernández-Tejera et al., 1999] Hernández-Tejera, M., Cabrera-Gámez, J., Castrillón-Santana, M., Domínguez-Brito, A. C., Guerra-Artal, C., Hernández-Sosa, D., and Isern-González, J. (1999). *DESEO: an Active Vision System for Detection, Tracking*

and Recognition, volume 1542, pages 376–391. International Conference on Vision Systems, Las Palmas de Gran Canaria, Spain. Springer-Verlag Lecture Notes on Computer Science. ISBN 3-540-65459-3.

[IEEE, 1996] IEEE (1996). POSIX Standard Specification. IEEE Std 1003.1, 1996 Edition. http://standards.ieee.org. IEEE Standards Association. NoneInternational Standard ISO/IEC 9945-1: 1996 (E) IEEE Std 1003.1, 1996 Edition (Incorporating ANSI/IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995, and 1003.1i-1995).

[Konolige et al., 1997] Konolige, K., Myers, K., Saffiotti, A., and Ruspini, E. (1997). The Saphira Architecture: a Design for Autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(1):215–235.

[Kortenkamp et al., 1998] Kortenkamp, D., Bonasso, R. P., and Murphy, R. E. (1998). *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press.

[Kortenkamp and Schultz, 1999] Kortenkamp, D. and Schultz, A. C. (1999). Integrating robotics research. *Autonomous Robots*, 6:243–245.

[Košecká, 1996] Košecká, J. (1996). *Supervisory Control Theory of Autonomous Mobile Agents*. PhD thesis, University of Pennsylvania, GRASP Laboratory.

[Košecká et al., 1997] Košecká, J., Christensen, H. I., and Bajcsy, R. (1997). Experiments in Behavior Composition. *Robotics and Autonomous Systems*, 19:287–298.

[Langer et al., 1994] Langer, D., Rosenblatt, J. K., and Herbert, M. (1994). A Behavior-Based System for Off-Road Navigation. *IEEE Journal of Robotics and Automation*, 10(6):776–782.

[Lyons, 1990] Lyons, D. M. (1990). A process-based approach to task representation. *IEEE Proceedings Robotics and Automation*, pages 2142–2147.

[Lyons and Arbib, 1989] Lyons, D. M. and Arbib, M. A. (1989). A formal model of computation for sensory-based robotics. *IEEE Transactions on Robotics and Automation*, 5(3):280–293.

[Monson-Haefal, 2001] Monson-Haefal, R. (2001). *Enterprise JavaBeans*. O'Reilly.

[MSDN, 2002] MSDN (2002). Microsoft Visual Studio .NET Documentation. Microsoft Develover Network Web Site - msdn.microsoft.com, Visual Studio .NET.

[MSDN, 2002] MSDN (2002). Microsoft Visual Studio .NET Documentation. MSDN Library: Windows Development. Windows Base Services: DLLs, Processes and Threads. Microsoft Develover Network Web Site - msdn.microsoft.com, Visual Studio .NET.

[Murphy, 2000] Murphy, R. R. (2000). *Introduction to AI Robotics*. The MIT Press.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. P. (1996). *Pthreads Programming*. O'Reilly.

[Noreils, 1990] Noreils, F. R. (1990). Integrating Error Recovery in a Mobile Robot Control System. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 396–401. IEEE Computer Society Press.

[Orebäck and Christensen, 2003] Orebäck, A. and Christensen, H. I. (2003). Evaluation of Architectures for Mobile Robotics. *Autonomous Robots. Kluwer Academic Publishers*, 14:33–49.

[Orebäck et al., 2000] Orebäck, A., Lindström, M., and Christensen, H. (2000). BERRA - A Behaviour Based Robot Architecture. Proc. Int. Conf. on Robotics and Automation (ICRA), San Francisco.

[Orocos, 2003] Orocos (2003). The Orocos Project. http://www.orocos.org.

[Pirjanian, 1998] Pirjanian, P. (1998). *Multiple Objective Action Selection and Behavior Fusion Using Voting*. PhD thesis, Dpt. of Medical Informatics and Image Analysis, Aalborg University.

[Ram et al., 1994] Ram, A., Arkin, R. C., Boone, G., and Pearce, M. (1994). Using Genetic Algorithms to Learn Reactive Control Parameters for Autonomous Robotic Navigation. *Journal of Adaptive Behavior*, 2(3):277–305.

[Ram et al., 1992] Ram, A., Arkin, R. C., Clark, R. J., and Moorman, K. (1992). Case-Based Reactive Navigation: A Case-Based Method for On-line Selection and Adaptation of Reactive Control Parameters in Autonomous Robotic Systems. Technical report, Georgia Tech.

[Ramamritham and Shen, 1998] Ramamritham, K. and Shen, C. (1998). Using Windows NT for Real-Time Applications: Experimental Observations and Recommendations. IEEE Real-Time Technology and Applications Symposium (merl.com/people/shen/pubs/rtas98.pdf).

[Richter, 1997] Richter, J. (1997). *Advanced Windows*. Microsoft Press, Third edition.

[Rosenblatt, 1995] Rosenblatt, J. K. (1995). DAMN: A Distributed Architecture for Mobile Navigation. In *Proceedings of the AAAI Spring Symp. on Lessons Learned from Implememted Software Architectures for Physical Agents*, Stanford, CA, USA.

[RTAI, 2003] RTAI (2003). Realtime Linux Application Interface for Linux. http://www.rtai.org. Dipartimento di Ingegneria Aerospaziale, Politecnico di Milano.

[RTLinux, 2003] RTLinux (2003). Real Time Linux. http://www.rtlinux.org. FSM-Labs Inc.

[Saffiotti et al., 1997] Saffiotti, A., Ruspini, E. H., and Konolige, K. (1997). Using Fuzzy Logic for Mobile Robot Control. In Prade, H., Dubois, D., and Zimmermann, H. J., editors, *International Handbook of Fuzzy Sets and Possibility Theory*, volume 5. Kluwer Academic Publishers Group, Norwell, MA, USA, and Dordrecht, The Netherlands.

[Schlegel, 2003] Schlegel, C. (2003). Overview of the OROCOS::SmartSoft Approach. http://www1.faw.uni-ulm.de/orocos/.

[Schlegel and Wörz, 1999a] Schlegel, C. and Wörz, R. (1999a). Interfacing Different Layers of a Multilayer Architecture for Sensorimotor Systems using the Object Oriented Framework SmartSoft. Third European Workshop on Advanced Mobile Robots - Eurobot´99. Zürich, Switzerland.

[Schlegel and Wörz, 1999b] Schlegel, C. and Wörz, R. (1999b). The Software Framework SmartSoft for Implementing Sensorimotor Systems. IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS'99, Kyongju, Korea.

[Schmidt, 1994] Schmidt, D. C. (1994). The Adaptative Communication Environment. An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the 11 th and 12 th Sun User Group Conferences*, San Jose, USA.

[Silberschatz et al., 2001] Silberschatz, A., Galvin, B. P., and Gagne, G. (2001). *Operating System Concepts*. John Wiley & Sons Inc.

[Simmons, 1992] Simmons, R. (1992). Concurrent Planning and Execution for Autonomous Robots. *IEEE Control Systems*, 12(1):46–50.

[Simmons and Apfelbaum, 1998] Simmons, R. and Apfelbaum, D. (1998). A Task Description Language for Robot Control. Proc. International Conference on Intelligent Robotics and Systems, Vancouver, Canada.

[Solomon and Russinovich, 2000] Solomon, D. A. and Russinovich, M. E. (2000). *Inside Microsoft Windows 2000*. Microsoft Programming Series. Microsoft Press, Third edition.

[Stallings, 2000] Stallings, W. (2000). *Operating Systems: Internals and Design Principles*. Prentice Hall International Editions. Prentice Hall.

[Steenstrup et al., 1983] Steenstrup, M., Arbib, M. A., and Manes, E. G. (1983). Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences*, 27:29–50.

[Stevens, 1998] Stevens, W. R. (1998). *UNIX Network Programming. Networking APIs: Sockets and XTI*, volume 1. Prentice Hall, Second edition.

[Stewart, 1994] Stewart, D. B. (1994). *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*. PhD thesis, Carnegie Mellon University, Dept. Electrical and Computing Engineering, Pittsburgh.

[Stewart and Khosla, 1993] Stewart, D. B. and Khosla, P. (1993). Chimera 3.1: the Real-Time Operating System for Reconfigurable Sensor-Based Control Systems. Advanced Manipulators Laboratory, The Robotics Institute and Department of Electrical and Computer Engineering, Carnegie Mellon University.

[Stewart and Khosla, 1996] Stewart, D. B. and Khosla, P. (1996). The chimera methodology: Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering*, 6(2):249–277.

[Stewart et al., 1997] Stewart, D. B., Volpe, R. A., and Khosla, P. (1997). Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering*, 23(12):759–776.

[Stroustrup, 2000] Stroustrup, B. (2000). *The C++ Programming Language*. Addison Wesley, Special Edition edition.

[Szyperski, 1999] Szyperski, C. (1999). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.

[Tsotsos, 1995] Tsotsos, J. K. (1995). Behaviorist intelligence and the scaling problem. *Artificial Intelligence*, 75(4):135–160.

[Turing, 1937] Turing, A. M. (1936-1937). On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proc. London Maths. Soc., ser. 2*, volume 42, pages 230–265.

Index