CoolBOT: A Component Model and Software Infrastructure for Robotics*

Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, José Isern-González, and Jorge Cabrera-Gámez

IUSIANI - Universidad de Las Palmas de Gran Canaria (ULPGC), Spain
{adominguez,dhernandez,jisern}@iusiani.ulpgc.es,jcabrera@dis.ulpgc.es

1 Introduction

In general, we face recurrently some common problems when programming robotic systems: multithreading and multiprocessing, distributed computing, hardware abstraction, hardware and software integration, multiple levels of abstraction and control, the development of a programming tool for a group of users which may become wide and diverse, etc. In this document we will introduce CoolBOT, a component oriented programming framework implementing primitives and mechanisms aimed to support the resolution of some of these common problems. This framework allows building systems by integrating "off-the-shelf" software components following a port automata model [SVK97] that fosters controllability and observability. Next section, Section 2, will introduce some of the recurrent problems we can find when developing the software infrastructure aimed to control a robotic system. In Section 3 we introduce a component oriented programming framework for programming robotic systems called CoolBOT which is the main subject of this chapter. In Section 4 we will outline the use of CoolBOT for building a real example. Finally in Section 5 we outline some of the conclusions we have drawn along the way of building and using CoolBOT.

2 Recurrent Problems

There are some questions and problems that appear repeatedly and recurrently in every robotic system which is not strange to solve again and again

* This work has been supported by research projects *PI2003/160* and *PI2003/165* funded by the Autonomous Government of Canary Islands (Gobierno de Canarias - Consejera de Educacin, Cultura y Deportes, Spain), by the MEC (Ministerio de Educacin y Ciencia, Spain) and FEDER research project *TIN2004-07087*, and by the ULPGC research projects *UNI2004/11*, *UNI2004/25* and *UNI2005/18*.

when we face the design of the system control software, and its implementation for each new system or project. Our own experiences and necessities created at our lab the real demand of investing resources and research effort in building a common software infrastructure implementing primitives and mechanisms to facilitate the resolution of some of these recurrent problems. In the next subsections we enumerate the problems we had in mind when developing the software infrastructure which is the subject of this contribution: CoolBOT, a C++ component-oriented programming framework for programming robotic systems.

2.1 Uniformity, Controllability, Observability and Robustness.

Certain level of uniformity in internal structure and external interface in software components is critical to allow a basic uniform treatment of components in spite of their individual functionality. This is a wide-spread and well-know principle in the operating systems community. In fact, one of the main challenges 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 systems impose on the programs they can execute a uniform internal structure and a uniform external interface in order to make them uniformly treatable, administrable and executable, independently of the functionality these programs have.

Moreover, 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. At the same time, component's internal uniformity sets a real basis for the development of debugging and profiling tools, and makes component design and implementation less error-prone.

Furthermore, robustness is a permanent design goal in every robotic project. Having systems integrated by components which can be observed and controlled externally constitutes a fundamental feature that facilitates the design of robust systems.

2.2 Concurrent/Parallel Programming

Concurrency and parallelism are serious issues to which is necessary to dedicate many efforts when programming robotic systems. Simultaneous and non-synchronized accesses 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 introduces even a higher level of difficulty. A developer should not be worried about simultaneous non-synchronized accesses to shared resources or critical sections, the programming infrastructure should care about all of it behind the scenes.

2.3 Real Time Requirements

In general, it is possible to establish a taxonomy dividing robotic systems in two big groups attending to their real time requirements. On one side there are systems with *hard real time* constraints. Their requirements correspond usually to the strict observation of deadlines in the achievement of different and/or multiple tasks in run-time, because otherwise, the safety of the system and its environment during operation can not be guaranteed. On the other side, we have systems with *soft real time* requirements. In these systems it is possible to miss spuriously deadlines for tasks without compromising system safe operation.

A software infrastructure aimed to programming robotic systems should support at least soft real time system implementations providing means to know when missed deadlines have happened in a system (*cognizant failures* [Gat92]). Supporting mechanisms and primitives for guaranteeing hard real time requirements would need a tighter coupling between the software infrastructure and the mechanisms and primitives it provides, and the underlying operating system where it runs.

2.4 Distributed Programming

Components should be integrable and reachable in a system with independence of the machines where they run. From a developer's point of view, whether components were in the same computer, or in a distinct one residing in the same computer network, should be transparent in terms of component integration and interconnection.

3 CoolBOT

CoolBOT [DB03] [DBHSIGCG04] is a C++ component-oriented framework for programming robotic systems that allows designing systems in terms of composition and integration of software components. Each software component [Szy99] is an independent execution unit 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.

CoolBOT has been mainly originated by very practical reasons. While developing several robotic systems 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) [DBHTCG00]. An active vision system termed *DESEO* (**DE**tección, **SE**guimiento y Reconocimiento de **O**bjetos)

[HTCGCS+99] aimed to detect, track and recognize faces, and an entertainment museum robot called *Eldi* [DBCGHS+01] 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 design and build CoolBOT, whose evolution can be tracked along different documents [DBAC00], [CGDBHS02], [DB03] and [DBHSIGCG04], being [HSDBGACG05] one of last ones. CoolBOT, the framework we will present in the rest of sections of this chapter, is the current result of this line of work, and its name does not stand for any acronym, it only reflects our desire of disposing of a robotic framework to make software easier and "cool" to integrate and reuse.

3.1 Port Automata Model for Components

In CoolBOT, components are modelled as *Port Automata* [SVK97]. This concept establishes a clear distinction between the internal functionality of an active entity, an automaton, and its external interface, sets of input and output ports. Fig. 1 displays the external view of a component where the component itself is represented by a circle, input ports (the i_k's) by the arrows oriented towards the circle, and output ports (the o_k's) by the arrows oriented outwards. Fig. 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 (the e_k's) caused either by incoming data through an input port, or by an internal condition, or by a combination of both. Double circles indicate automaton final states.

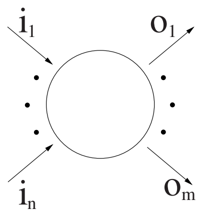


Fig. 1. Component external view.

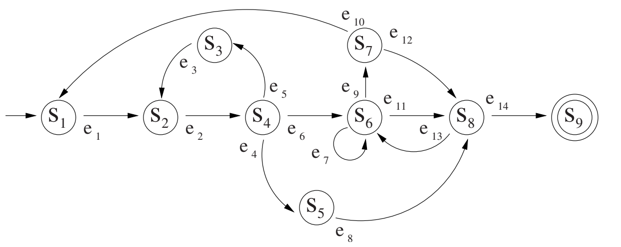


Fig. 2. Component internal view.

CoolBOT components interact and inter communicate each other by means of *port connections* established among their input and output ports. Data are transmitted through *port connections* in discrete units of information called *port packets*. *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.

3.2 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. 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.

Additionally, to guarantee external observation and control, CoolBOT components provide by default two important ports: the *control port* and the *monitoring port*, both depicted in Fig. 3.

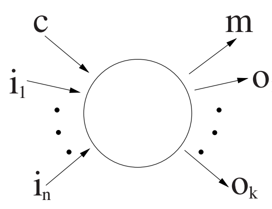


Fig. 3. The *control* port, *c*, and the *monitoring* port, *m*.

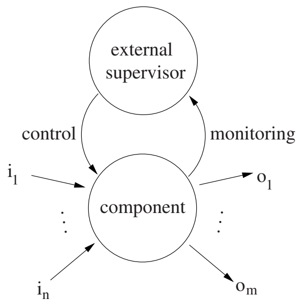


Fig. 4. A typical component control loop.

- The *monitoring port*: This is a public output port by means of which component *observable variables* are published. 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. An external controller or supervisor can control a component by means of it.

CoolBOT provides components with several default observable and controllable variables, in addition to the observable and controllable variables specific to each component. A brief description of these default variables is shown in Tables 1 and 2 (the symbols representing each variable are given beside variable names in parenthesis).

Table 1. Default observable variables.

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

Fig. 4 illustrates graphically a typical execution control loop for a component using another component as *external supervisor*. This is possible thanks to the default *control* and *monitoring* ports CoolBOT imposes on all components.

Table 2. Default controllable variables.

Default Controllable Variables	
Name	Brief Description
new state (ns)	Desired automaton state where the component is commanded to go.
new priority (np)	Desired execution priority the component is 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.

3.3 Default Automaton

Internally all components are modelled using the same default state automaton, the *default automaton*, shown in Fig. 5 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 them correspond to internal events: *ok*, *exception*, *attempt*, *last attempt* and *finish*. The other ones indicate default controllable variable changes: ns_r, ns_re, ns_s, ns_d, np, nc and nex. Subscripts in ns_i indicate which state has been commanded: r (*running* state), re (*ready* state), s (*suspended* state), and d (dead state). Event *np* happens when an external supervisor forces a priority change; event *nc* when a change of configuration is commanded from the external supervisor; and event *nex* occurs when the supervisor injects "artificially the occurrence of an exception.

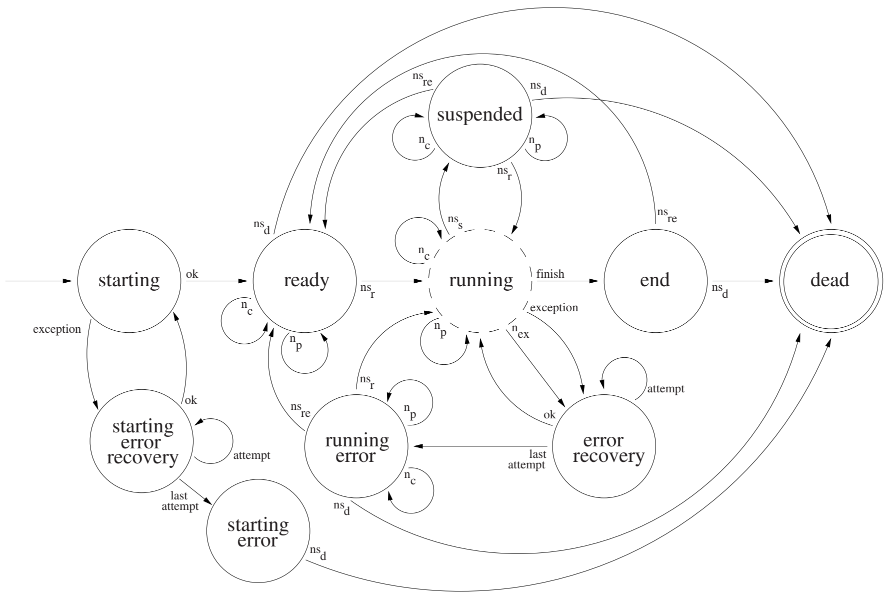


Fig. 5. The Default Automaton.

The *default automaton* is said to be *"controllable* because it can be brought in finite time by an *external supervisor* by using 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 Fig. 5, 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 Fig. 5 we can see how CoolBOT components evolve along their execution time, since they are launched until they finish their execution. Basically, the *default automaton* organize the life of a component in several phases which correspond to different states:

- *starting*: This state is devised to allocate resources for a correct task execution.

- *ready*: In this state the component is ready to execute, and waits for an external command to start (ns_r).
- *running*: In this pseudo-state the component is carrying out its specific task running its *user automaton.*
- *suspended*: In this state the component is suspended and remains idle until ordered to transit to other state.
- *end*: In this state the component has just finished a task execution. Getting to this state the component publishes the result of its execution, if any, through its *monitoring port.*

Furthermore, there are two pair of states conceived for handling faulty situations during execution. One of them devised to face errors during resource allocation (*starting error recovery* and *starting error* states), and the other one though to deal with errors during task execution (*error recovery* and *running error* states). These states are part of the support CoolBOT provides for error and exception handling in components.

In fact, in the same manner that exceptions constitute a useful concept which is present in numerous programming languages (C++, Java, etc.) to separate error handling from the normal flow of instructions in a program., we may define and use exceptions in CoolBOT components. Thus, a component may define a list of *component exceptions* to signal and handle erroneous, exceptional or abnormal situations during execution, where each component exception is defined using the pattern shown in Fig. 6. In the figure `<exceptionId>` is a number identifying the exception; `<description>` is a description of the exception; `<handler>` is an optional handler to try a recovery procedure, optionally `<retries>` indicates the number of recovery attempts, and `<period>` specifies the period in milliseconds between attempts; `<onSuccessHandler>` is an optional handler to be executed in case of a successful recovery; and finally, `<onFailureHandler>` is also an optional handler which is executed when all recovery exception attempts have failed.

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

Fig. 6. Exception Pattern.

All in all, if an exception comes up during runtime in a component, the component is driven to one of the recovery error states (*starting error recovery* or *error recovery*) where the handler for the exception is tried several times. Only when all recovery attempts have been unsuccessful the component is took to a situation of unrecovered exception (*starting error* or *running error*). In this situation, an error description is published through the monitoring port of the component, and it remains idle waiting for external supervision.

3.4 Multithreading

CoolBOT 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 in Windows, and POSIX threads in GNU/Linux, which are the two operating systems where CoolBOT is supported in its current version.

```
void Component::kernel()
{
  PortPacket* packet;

  initialization();

  while(true)
  {
    packet=waitForSomething(inputPorts);

    if(!processPortPacket(packet)) break;
  } // end of while

  finalization();
}
```

Fig. 7. A component in execution: A continuous loop processing port packets. Simplified C++ kernel code.

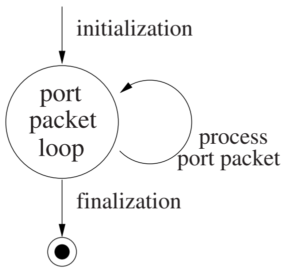


Fig. 8. Component simplified kernel. Graphical representation.

At runtime a CoolBOT component executes a continuous loop processing port packets where 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. In general, incoming port packets trigger the transitions which form the components automaton. Figures 7 and 8 shows respectively a simplified C++ version and a graphical representation of the kernel code which drives a component at runtime. Observe that according to this code, depending on at which rate or frequency a component is receiving port packets, the component would be blocked waiting for incoming port packets 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.

The loop for processing incoming packets which is located at the kernel of a component can be multithreaded or not, as Fig. 9 illustrates. In general, a component needs for its execution at least a thread in the underlying operating system, called the *main thread*. Furthermore, in order to make a component more responsive, it is possible to distribute the attention of a component on different input ports using different threads of execution. These threads are called *port threads*, and they are responsible for disjoint sets of component's input ports, and similarly to the *main thread*, they are also *data-flow driven* through their incoming port packets. The *main thread* executes the automaton of the component, and controls and observes any other components *port threads*, if any. The *main thread* is also responsible for maintaining the consistency of the internal data structures that constitute the internal state of the whole component. As a result, in CoolBOT any component corresponds with at least an underlying thread, concretely the *main thread*, guaranteeing in this way that if the code of a component hangs completely the systems will not collapse.

3.5 Inter Component Communications

Analogously to modern operating systems that provide *IPC* (*Inter Process Communications*) mechanisms to inter communicate processes, CoolBOT provides *Inter Component Communications* 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. 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.

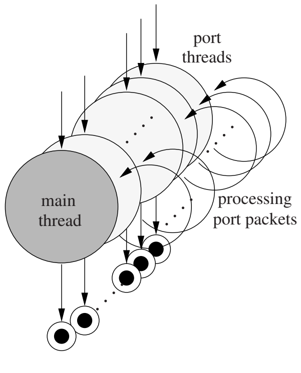


Fig. 9. Multiple threads.

There are several types of output and input ports supported by CoolBOT which combined adequately in the form of port connections implement different protocols of interaction between components. Fig. 10 shows all the different types of output and input ports supported by CoolBOT (on the right and on the left respectively), and all the possible combinations between them to form port connections. Below the arrows, the cardinality of each type of port connection is also indicated. 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 constitute a compatible pair of output and input ports. Each type of port connection implements a different protocol of interaction between components, Table 3 resumes the protocol implemented by each one (the letter in parenthesis beside each port name is its corresponding symbol).

$$\begin{tikzcd}
OTick \arrow[r, "n \qquad\qquad m"'] & ITick & OShared \arrow[r, "1 \qquad\qquad m"'] & IShared \\
OGeneric \arrow[r, "n \qquad\qquad m"'] & ILast & OMultiPacket \arrow[r, "n \qquad\qquad m"'] & IMultiPacket \\
OGeneric \arrow[r, "n \qquad\qquad m"'] & IFifo & OLazyMultiPacket \arrow[r, "n \qquad\qquad m"'] & IMultiPacket \\
OGeneric \arrow[r, "n \qquad\qquad m"'] & IUFifo & OPriority \arrow[r, "n \qquad\qquad 1"'] & IPriorities \\
OPoster \arrow[r, "1 \qquad\qquad m"'] & IPoster & OPull \arrow[r, "1 \qquad\qquad n"'] & IPull
\end{tikzcd}$$

Fig. 10. Port connections ($\forall\ n,m \in \mathbb{N}; n,m \geq 1$).

Table 3. Port connections.

Output Port	Input Port	Brief Description
OTick (t)	*ITick (t)*	*Tick Connections*: Implements a protocol to signal events between components
OGeneric (g)	*ILast (l)*	*Last, Fifo and Unbounded Fifo Connections*: There is a queue (fifo) of port packets in the input port
	IFifo (f)	
	IUFifo (uf)	
OPoster (p)	*IPoster (p)*	*Poster Connections*: There is a "master copy" of port packets in the output port, input ports keep local copies
OShared (s)	*IShared (s)*	*Shared Connections*: Components share a "shared memory" residing in the output port. Implements a protocol of shared memory
OMultiPacket (mp)	*IMultiPacket (mp)*	*Multi Packet Connections*: Accepts multiple types of port packets through the same port connection
OLazyMultiPacket (lmp)		
OPrioriy (pr)	*IPriorities (pr)*	*Priority Connections*: Implements a protocol of sending with priority
OPull (pu)	*IPull (pu)*	*Pull Connections*: Implements a protocol of request/answer between components

In CoolBOT there are two basic communication models for port connections. The first one is the *push model*, where the initiative for sending port packets relies on the output port part of a connection, the data producer, which sends port packets on its own, completely uncoupled from its consumers. The second model for communications is the *pull model*, where packets are emitted when the input port part of a connection, the consumer, demands new data to process. In this model the consumer keeps the initiative, sending a request to the producer whenever it need a new incoming port packet. In all cases consumers connect to producers *by subscription*, so if a component connects one of its input ports to the output port of another component it gets subscribed to this port conforming a port connection (this is indicated by the cardinality in Fig. 10). Thus, depending on the nature and typology of the connected ports the cardinality of the relationship between components can be either “many to many, “one to many, “many to one, etc.

All types of port connections in Table 3 observe a push communication model that allows for uncoupled and asynchronous interactions among components. The only one type of port connection that utilizes a pull communication model is the *pull connection* (an *OPull*/*IPull* pair of ports).

Additionally to the connections shown in Fig. 10 there are other possible pairs of port connections which are collectively called *single multi packet* connections. All the possibilities appear in Fig. 11. Their main characteristic is that they combine a type of output or input port that accepts only one type of port packet (a *single packet* port) with a type of input or output port that accepts several types of port packets (a *multi packet* port).

$$\begin{tikzcd}
OMultiPacket \arrow[r, "n"' near start, "m"' near end] & ILast & OLazyMultiPacket \arrow[r, "n"' near start, "m"' near end] & ILast \\
OMultiPacket \arrow[r, "n"' near start, "m"' near end] & IFifo & OLazyMultiPacket \arrow[r, "n"' near start, "m"' near end] & IFifo \\
OMultiPacket \arrow[r, "n"' near start, "m"' near end] & IUFifo & OLazyMultiPacket \arrow[r, "n"' near start, "m"' near end] & IUFifo \\
OGeneric \arrow[rr, "n"' near start, "m"' near end] & & IMultiPacket &
\end{tikzcd}$$

Fig. 11. Simple multi packet connections ($\forall\, n,m \in \mathbb{N}; n,m \geq 1$).

All in all, CoolBOT provides a wide and rich set of output and input ports for inter component communications that frees developers of an important workload in software design and development, what implies a less error-prone system programming. In addition, the typology of different ports offers a number of communications patterns wide enough to allow a broad variety of component interactions.

3.6 Distributed Components

CoolBOT components reside in specific computers when they are instantiated and executed. Frequently, the functionality of a component which runs in one computer may be needed in other machine elsewhere in a computer network. CoolBOT provides a mechanism based on *“mediator”* components [GHJV95] called *proxy components*, and network processes called *CoolBOT servers*, to allow components to be accessible through a computer network.

3.7 Real Time Features

Currently, CoolBOT is supported under the Windows family of operating systems, 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 [RS98] [Gop01].

3.8 Modularity and Hierarchy

CoolBOT components are classified into two kinds: *atomic* and *compound components*. *Atomic components* have been mainly devised in order 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.

Compound components are compositions of instances of several components which can be either atomic or compound. The functionality of a compound component resides in its *supervisor*, depicted in Fig. 12, which controls and observes the execution of its *local components* through the control and monitoring ports present in all of them. The *supervisor* of a compound component concentrates the control flow of a composition of components, and in the same way that in atomic components, it follows the control graph defined by the default automaton of Fig. 5. All in all, *compound components* use the functionality of instances of other atomic or compound components to implement its own functionality. Moreover, they, in turn, can be integrated and composed hierarchically with other components to form new compound components.

3.9 Developing Components

The process of developing CoolBOT components and systems is resumed in six steps on Fig. 13:

(1) *Definition and Design*: In this step the component is completely defined and designed. This comprises deciding whether it is atomic or not, its functionality – *user automaton* –, thread use, resources, output and input ports, port packets, observable and controllable variables, exceptions, timers and watchdogs.

(2) *Skeleton Generation*: There is already a small set of developed components, and component examples in the form of C++ classes illustrating the most common patterns of use. It is possible to start from one of them

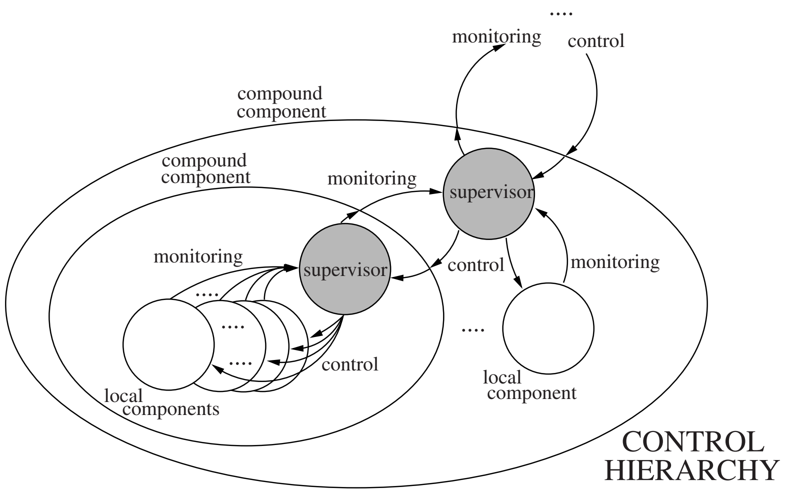


Fig. 12. Compound components.

as skeleton, or generate a new one from a component skeleton description file.

(3) *Code Fulfilling*: Using the components skeleton obtained in the previous step we complete the component fulfilling its code.

(4) *Library Generation*: Then the component is compiled obtaining a library.

(5) *System Integration*: Next the component may be integrated in a system alone or with other components.

(6) *System Generation*: And finally, the system gets compiled and an executable system is obtained. With it, we can already test the whole integration with our component.

4 Using CoolBOT

CoolBOT is a C++ framework, and every CoolBOT component is a C++ class. Next subsections will introduce briefly how it is the process of building components, and how to integrate them to have a working system following the process illustrated in Fig. 13.

4.1 Building a System

CoolBOT has been conceived to promote integrability, incremental design and robustness of software developments in robotics. In this section, a simple demonstrator will be outlined to illustrate how such principles manifest in systems built using CoolBOT. The first level of this simple demonstrator is shown in Fig. 14 and it is made up of four components: the *Pioneer*

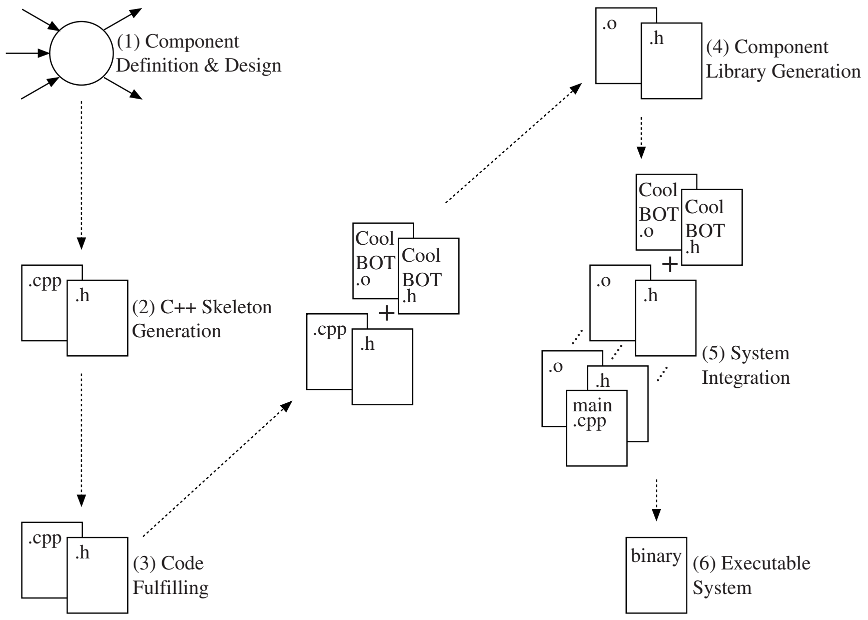


Fig. 13. The development process.

component which encapsulates the set of sensors and effectors provided by an ActivMedia Robotics Pioneer robot; the *PF Fusion* component which is a potential field fuser; the *Strategic PF* component that transforms high level movement commands into combinations of potential fields; and finally, the *Joystick Navigation* component which allows controlling the robot using a joystick. The integration shown in the figure makes the robot able to avoid obstacles while executing a high level movement command like, for example, going to a specific destination point.

CoolBOT maps all components as C++ classes. As an example, Fig. 15 shows the skeleton of the C++ mapping for the component *Pioneer* of Fig. 14. In general, all components inherits a default external interface and a default internal structure through superclass `CoolBOT:Component` as illustrated in Figures 15 and 16. In this manner the framework imposes on components a default minimum external and internal behavior.

As mentioned in subsection 3.9, in the process of developing a component, once we have generated the C++ skeleton for a specific component (step *"(2) C++ Skeleton Generation"* of Fig. 13), we have to start fulfilling its code (step *"(3) Code Fulfilling"* in the same figure). Mind that the C++ skeleton obtained in step *(3)* only has structure but it is empty in terms of functionality. This functionality should be coded in the transitions, and in the entry and exit sections of each one of the states of the component's automaton, which are

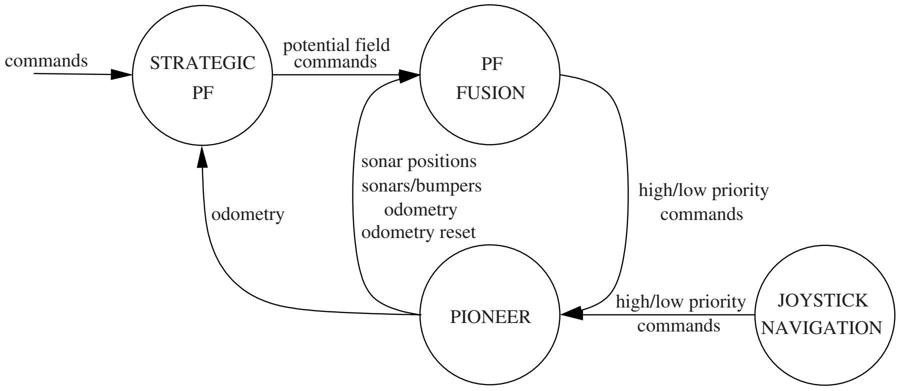


Fig. 14. The avoiding level

```
...
#include "coolbot.h"
using namespace CoolBOT;
...
namespace PioneerSpace
{
    ...
    class Pioneer: public Component
    {
        public:
            ...
        private:
            ...
    };
    ...
}
```

Fig. 15. Component *Pioneer*: class skeleton.

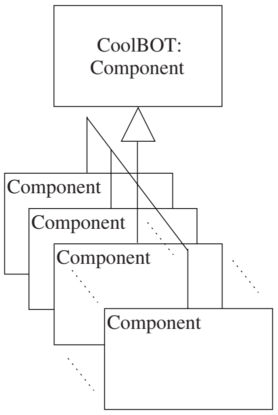


Fig. 16. The superclass `CoolBOT:Component`.

mapped as function members in the C++ skeleton class. An example appears in Fig. 17, where a transition for the component*Pioneer* of Fig. 14 is shown, concretely it maps a transition in a state called `main` which is triggered when an incoming packet reaches the input port called `commands`.

In order to clarify a bit more the code, in Fig. 18 we show the external public interface of input and output ports for this component. Output and input port types are indicated by their symbols in parentheses (consult Table 3 for the symbols associated with each type). Tables 5 and 4 describes respectively the public interface of output and input ports for the component *Pioneer*.

```
...

// state main: begin

...

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

    ...

    return pThis->_currentState_;
}

...

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

Fig. 17. Component *Pioneer*: a transition.

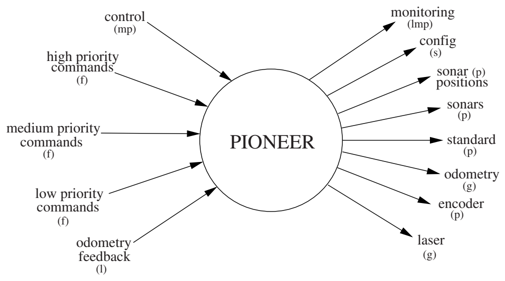


Fig. 18. Component *Pioneer*: external interface.

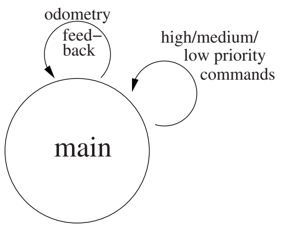


Fig. 19. Component *Pioneer*: *user autoamaton.*

Besides, in Fig. 19 appears the internal *user automaton* for this component which corresponds to the super-state *running* in Fig. 5. The behavior of the *Pioneer* component is not complex. As it is shown in the figure, the components user automaton has only one 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 18). At the same time, in this 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* 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.

Table 4. Component *Pioneer*: public output ports.

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 robots 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 robots 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 5. Component *Pioneer*: public input ports.

Public Input Ports	
Name	Brief Description
control	This is the components 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.

In Fig. 14 all components are *atomic*, the system is simple enough to have no necessity of using a hierarchy of *compound* components. To continue with our small system. We have added new components as a second control level to our little demonstrator of Fig. 14, this is depicted in Fig. 20. Observe that the system has now two new components, the *Sick Laser* component which controls a Sick laser range finder and the *Scan Alignment* component that performs map building and self localization using a SLAM (Simultaneous Localization And Mapping) algorithm.

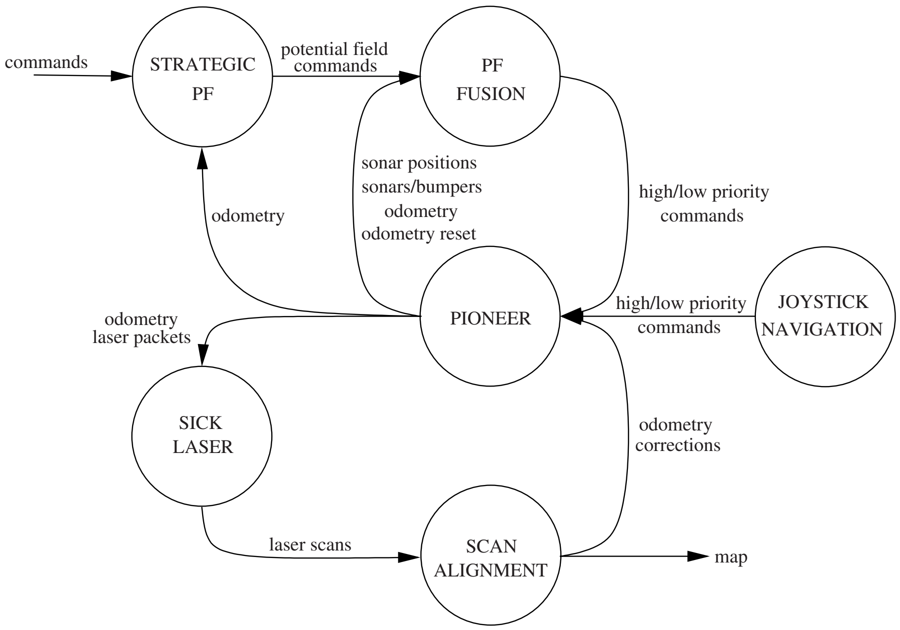
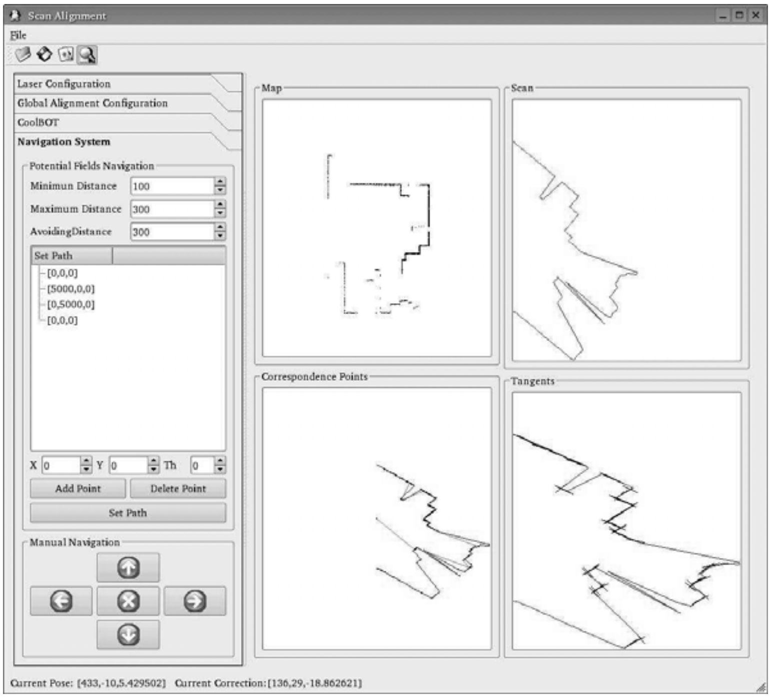


Fig. 20. The whole system

In particular, the integration depicted in Fig. 20 works as follows. The *Sick Laser* component executes a continuous loop, periodically reading the laser device and publishing the data by means of a poster output port. This component receives odometry packets from the *Pioneer* component in order to be able to stamp every scan with the robot pose at the time the scan was taken. The *Scan Alignment* component communicates with the *Pioneer* and *Sick Laser* components. It reads scans and preforms pose corrections which are sent to the *Pioneer* component that in turn will correct the pose and its associated covariance. The component does this by means of a simultaneous localization and mapping algorithm which produces a map of the environment [LM94][LM97]. This map is also published in an output port to be used by any other component. So, now, the system allows a robot to make self-localization and map building, at the same time that it avoids obstacles.

4.2 Control Interface

In CoolBOT, as we have mentioned in section 3, every component presents a uniformity of external interface and internal structure. This is suitable for implementing a control system that allow us to connect to several components and control them. Such a system could be able to control and observe the behavior of a component by means of its *control* and *monitoring* default ports. At the same time it could be able to "sniff" port connections between components. This is very helpful when trying to debug such systems or to check its normal operation.



Minimun Distance 100
Maximum Distance 300
AvoidingDistance 300

Set Path
[0,0,0]
[5000,0,0]
[0,5000,0]
[0,0,0]

X 0 Y 0 Th 0

Current Pose: [433,-10,5.429502] Current Correction: [136,29,-18.862621]

Fig. 21. The control interface.

For the system integration of Fig. 20 a graphical front-end was designed to control and observe the system during operation, a snapshot appears in Fig. 21. As we can see the figure shows the map, the current scan and the result of the corresponding points and tangent calculations. On the left side we can see part of the controlling interface which allow us to change the state of every component, and to connect to different ports of the system to have a look to the different data and results available (robot odometry, odometry corrections underdone, laser scans, the map as it gets built, etc.). Moreover, it is possible to change some operational *controllable* variables (frequency of laser scans, *Scan Alignment* component parameters, etc.). Additionally, it is possible to establish different paths the robot should follow during map building, or, as we have commented previously, to control the robot movements directly with a joystick device.

4.3 Test and Results

Robustness is an important aspect in robotics, and CoolBOT has aimed some of its infrastructure to facilitate robust system integration. As every component works in an autonomous and asynchronous fashion, the system does not have to stop running whenever any of its components enters into an non-recoverable error state. If this occurs the system can keep working with part of its functionality (if possible), or waiting for the faulty component to restart its normal execution.

Focusing on the robustness of the system we prepared several tests that show the behavior of the the system previously presented upon the malfunction of any of its components.

In the system of Fig. 20, the most important components are: the *Pioneer*, *Sick Laser* and the *Scan Alignment* components. Based on these three components, two tests were made. The first of them shows the way the system works whenever any of the components hangs, and the second one is related to the degradation of the system when the *Sick Laser* component stops running.

- **First Test**. The main goal of this test is to overcome a situation provoked by a faulty component without collapsing. The system should be able of keeping operation, either running, or waiting for the faulty component to restore the erroneous situation. As result of this test we found that the system did not collapse for any combination of its components in an non-recoverable error state. Table 6 summarizes the results derived from the test for every possible combination of error states in the components. The first three columns indicate which component is in a faulty situation (R stands for *Running* and E for *Error*), and the fourth one describes briefly the system behavior for each case. Any of the errors shown in Table 6 does not lead the system to a fatal situation, except if we consider the last case. The asynchronicity of execution, initiative and communications between components allow them to keep the system working until the faulty one is recovered, or the whole system is explicitly stopped.
- **Second test**. In Table 6 there are some situations during system operation that lead to odometry degradation. If the *Sick Laser* or the *Scan Alignment* component gets into erroneous operation, the SLAM algorithm responsible for the pose error, will not work. In this situation, the *Pioneer* component will keep working but as the robot moves, its pose will

degradate. For showing this degradation and its subsequent recovering we have made a real experiment where the *Sick Laser* is provided with a mechanism for reinitializing the connection with the laser device upon abnormal disconnection. Thus, for example if the serial connection to the laser is physically broken the component will keep trying to recover communications with the device by restarting the connection periodically. Therefore, when serial connection is eventually reestablished, the component restarts its normal operation. Once the laser is working properly, the *Scan Alignment* component will receive again new scans and will produce new corrections which, in turn, will be sent to the *Pioneer* component, finishing in this way, odometry degradation.

Table 6. Error States

Pioneer	Sick Laser	Scan Alignment	Description
R	*R*	*R*	Normal state, everything is right.
E	*R*	*R*	The *Pioneer* component does not work. The *Sick Laser* component publishes every scan with the last received pose. Notice that if the *Pioneer* component does not run, the robot also stops because of a watchdog event in such a way that the scans are published with the actual pose of the robot. The *Scan Alignment* component keep reducing the error of the map built so far.
R	*E*	*R*	The *Scan Alignment* component reduces errors in the current map, this is done by means of an iterative proccess to achieve consistent maps. But it stops integrating new scans to the map as the *Sick Laser* component is not working. In this state the robot pose degradates (explained in more detail in the second test).
E	*E*	*R*	The *Scan Alignment* component reduces errors of the current map built so far, but it does not add new scans to the map.
R	*R*	*E*	In this case the *Scan Alignment* component does not work. This leads to a degradation of the pose of the robot, but the rest of the system still works. This case is also the basis of the second test.
E	*R*	*E*	The *Sick Laser* component publishes scans with the last robot pose received.
R	*E*	*E*	The odometry of the robot degradates.
E	*E*	*E*	The system here does not collapse itself, but it can do nothing.

5 Conclusions

This chapter outlines a first operating version of CoolBOT, a component-oriented C++ framework where the software that controls a system is viewed as a dynamic network of units of execution modeled as port automata interconnected by means of port connections. CoolBOT is a tool that favors a programming methodology that fosters software integration, concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing.

CoolBOT imposes some uniformity on the units of execution it defines, CoolBOT components. This uniformity makes components externally observable and controllable, and treatable by the framework in an uniform and consistent way. CoolBOT also promotes a 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 an external supervisor, in turn, this handling may scale going up in a hierarchy of control loops.

In developing CoolBOT we have collected and reused many design ideas which can be found in the literature. Out of the bibliography we can find in the field, CoolBOT owes a big deal of inspiration from mainly three works in the area:

- **G^{en}oM**: a work developed at CNRS-LAAS (Laboratoire d'Analyse et d'Architecture des Systmes) in Toulouse. G^{en}oM is a software framework that was built to support the software architecture for the mobile robots available at LASS. For a general introduction consult [FHC97], [MFB02] and for a deeper look there is a more detailed user manual [FHM]. Recently G^{en}oM has become an open source project available at `softs.laas.fr/openrobots` being part of a bigger initiative from this labaratory called *"LAAS Open Software for Autonomous Systems.* G^{en}oM has been a strong source of inspiration for modelling CoolBOT's components and, not surprisingly, they share many features. The main 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.
- **Smartsoft**: [Sch06] developed by Schlegel and Wrst at FAW (Research Institute for Applied Knowledge) at the University of Ulm in Germany. For more details you can consult [SW99], [Sch03], and [Sch04]. Like G^{en}oM, SmartSoft is a software framework developed to support a software architecture for mobile robots. 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, that essentially implement a basic set of communication protocols with an

emphasis on asynchronicity. Both frameworks also hide the difficult issues of concurrency and synchronization to the programmer.
- **Chimera**: [Ste94] [SVK97] 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 main inspiration we used from Chimera was the concept of *port automaton* given by Steenstrup in [SAM83]. In Chimera, the units of execution handled by the operating system are defined as software modules called *port-based objects* which are units of execution defined formally as *port automata*.

For us, CoolBOT is a long-term experimental tool of which this work is only a first design and development effort. Contrarily to other approaches in the literature, it needs still further experimentation. Only the development of complex demonstrators out of a wide-enough set of robust ready-to-use components will allow us to determine how long CoolBOT scale, and in general to validate this programming tool in the long-term.

References

[CGDBHS02] J. Cabrera-Gámez, A. C. Domínguez-Brito, and D. Hernández-Sosa, *Coolbot: A component-oriented programming framework for robotics*, Lecture Notes in Computer Science, Sensor Based Intelligent Robots: International-Workshop Dagstuhl Castle, Germany, October 15-20, 2000 Selected Revised Papers, vol. 2238, pp. 282–304, Springer Berlin / Heidelberg, January 2002.

[DBHTCG00] A. C. Domínguez-Brito, F. M. Hernández-Tejera, and J. Cabrera-Gámez, *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., 2000.

[DBAC00] A. C. Domínguez-Brito, M. Andersson, and H. I. Christensen, *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, September 2000.

[DBCGHS$^+$01] A. C. Domínguez-Brito, J. Cabrera-Gámez, D. Hernández-Sosa, M. Castrillón-Santana, J. Lorenzo-Navarro, J. Isern-González, C. Guerra-Artal, I. Pérez-Pérez, A. Falcón-Martel, M. Hernández-Tejera, and J. Méndez-Rodríguez, *Eldi: An Agent Based Museum Robot*, Systems Science, ISSN 0137-1223 **27** (2001), no. 4, 119–128.

[DB03] A. C. Domínguez-Brito, *CoolBOT: a Component-Oriented Programming Framework for Robotics*, Ph.D. thesis, Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria (mozart.dis.ulpgc.es/Publications/publications.html), September 2003.

[DBHSIGCG04] A. C. Domínguez-Brito, D. Hernández-Sosa, J. Isern-González, and J. Cabrera-Gámez, *Integrating Robotics Software*, IEEE International Conference on Robotics and Automation, New Orleans, USA, April 2004.

[FHC97] S. Fleury, M. Herrb, and R. Chatila, *$G^{en}oM$: A Tool for the Specification and the Implementation of Operating Modules in a Distributed Robot Architecture*, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Grenoble, Francia), September 1997, pp. 842–848.

[FHM] S. Fleury, M. Herrb, and A. Mallet, *$G^{en}oM$: User's Guide*, softs.laas.fr/openrobots/distfiles/genom-manual.pdf, -.

[Gat92] E. Gat, *Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots*, Proceedings of the Tenth National Conference on Artificial Intelligence (San Jose, CA, USA), July 1992, pp. 809–815.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional Computing Series, Addison-Wesley, 1995.

[Gop01] K. Gopalan, *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, 2001.

[HTCGCS$^+$99] M. Hernández-Tejera, J. Cabrera-Gámez, M. Castrillón-Santana, A. C. Domínguez-Brito, C. Guerra-Artal, D. Hernández-Sosa, and J. Isern-González, *DESEO: an Active Vision System for Detection, Tracking and Recognition*, vol. 1542, pp. 376–391, International Conference on Vision Systems, Las Palmas de Gran Canaria, Spain. Springer-Verlag Lecture Notes on Computer Science, January 1999, ISBN 3-540-65459-3.

[HSDBGACG05] D. Hernández-Sosa, A. C. Domínguez-Brito, C. Guerra-Artal, and J. Cabrera-Gámez, *Runtime self-adaptation in a component-based robotic framework*, IEEE/RSJ International Conference on Inteligent Robots and Systems (IROS 2005), August 2-6, Edmonton, Alberta, Canada, 2005.

[LM94] F. Lu and E. Milios, *Robot pose estimation in unknown environments by matching 2d range scans*, Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, Seattle, USA, 1994.

[LM97] F. Lu and E. Milios, *Globally consistent range scan alignment for environment mapping*, Autonomous Robots **4** (1997), no. 4, 333–349.

[MFB02] A. Mallet, S. Fleury, and H. Bruyninckx, *A specification of generic robotics software components: future evolutions of $G^{en}oM$ in the orocos context*, IROS 2002 Conference, 2002.

[MH01] R. Monson-Haefal, *Enterprise JavaBeans*, O'Reilly, September 2001.

[Rof01] Jason T. Roff, *Ado: Activex data objects*, O'Reilly, 2001.

[RS98] K. Ramamritham and C. Shen, *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), 1998.

[SAM83] M. Steenstrup, M. A. Arbib, and E. G. Manes, *Port automata and the algebra of concurrent processes*, Journal of Computer and System Sciences **27** (1983), 29–50.

[Sch06] C. Schlegel, *Communication Patterns as Key Towards Component Interoperability*, In D.Brugali (Ed.) *Software Engineering for Experimental Robotics*, Spinger STAR series, 2006

[Sch03] C. Schlegel, *Overview of the OROCOS::SmartSoft Approach*, http://www1.faw.uni-ulm.de/orocos/, 2003.

[Sch04] C. Schlegel, *Navigation and execution for mobile robots in dynamic environments: An integrated approach*, Ph.D. thesis, University of Ulm, 2004.

[Ste94] D. B. Stewart, *Real-Time Software Design and Analysis of Reconfigurable Multi-Sensor Based Systems*, Ph.D. thesis, Carnegie Mellon University, Dept. Electrical and Computing Engineering, Pittsburgh, 1994.

[SVK97] D. B. Stewart, R. A. Volpe, and P. Khosla, *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects*, IEEE Transactions on Software Engineering **23** (1997), no. 12, 759–776.

[SW99] C. Schlegel and R. Wörz, *Interfacing Different Layers of a Multilayer Architecture for Sensorimotor Systems using the Object Oriented Framework SmartSoft*, Third European Workshop on Advanced Mobile Robots - Eurobot99. Zürich, Switzerland, September 1999.

[Szy99] C. Szyperski, *Component software: Beyond object-oriented programming*, Addison-Wesley, 1999.