More about CoolBOT[*]

Antonio C. Domínguez-Brito, Cayetano Guerra-Artal,
Josep Isern-González, Angel M. Naranjo-Cabrera
and Jorge Cabrera-Gámez

IUSIANI - Universidad de Las Palmas de Gran Canaria (ULPGC), Spain

{adominguez, cguerra}@iusiani.ulpgc.es, jisern@dfm.ulpgc.es, angel@jqbit.com and jcabrera@dis.ulpgc.es

Abstract

This document presents an operating version of CoolBOT, a component oriented software framework for programming robotic systems, that was already presented in WAF'2002 [3] when it was at the beginning of its development. CoolBOT has been designed having in mind the idea of programming by integrating software components, in order to reduce the developing effort typically invested when programming robots. CoolBOT also fosters some interesting features, such as asynchronous execution, asynchronous inter communication, data-flow-driven processing, and cognizant failure systems. A simple demonstrator illustrates the benefits of using the proposed approach.

1 Introduction

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. 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. Other authors [1] have made already similar considerations identifying the building of software architectures as the way the robotics community has mainly chosen to address the problem. Nowadays, 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, other authors [5] [8] 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.

In the following sections we will introduce this work, a component-oriented software framework aimed to programming robotic systems. Thus, in the next section a brief introduction will be given. Then in section 2 their main concepts and abstractions will be briefly explained. Next, in section 3 a simple demonstrator is commented in some detail, and finally, in section 4 we will comment some of the conclusions we have drawn from this work.

2 CoolBOT

CoolBOT [2] [4] 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* [10] 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 in other systems.
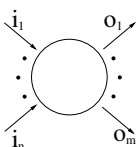

Figure 1: Component external view.

In CoolBOT, components are modelled as *Port Automata* [9]. 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, i_i, by the arrows oriented towards the circle, and output ports, o_i, by arrows oriented outwards.
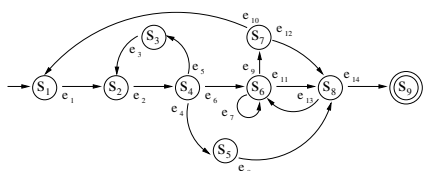

Figure 2: Component internal view.

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, e_i, 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. 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 called *port packets*. *Port packets* are also classified by their type,

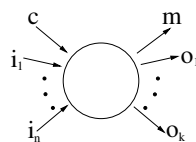and usually each input and output port can only accept a specific set of port packet types.

Figure 3: The *control* port, *c*, and the *monitoring* port, *m*

CoolBOT introduces two kinds of variables as facilities in order to support the monitoring and control of components: *observable variables*, that represent features of components that should be of interest from outside in terms of control, or just for observability and monitoring purposes; and *controllable variables*, which represent aspects of components which can be modified from outside, in order to be able to control the internal behavior of a component. Tables 1 and 2 enumerate respectively the default observable and controllable variables present in all components.

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.

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.

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. The *monitoring* port: which is a public output port by means of which component *observable variables* are published; and the *control* port, that is a public input port through which component *controllable variables* are modified and updated. Fig. 4 illustrates graphically a typical execution control loop for a component using these ports where there is another component as external supervisor.

Internally all components are modelled using the same default state automaton, the *default automaton*, shown in Fig. 5, that contains all possible control paths a component may follow. The *default automaton* can be always brought externally in finite time by means
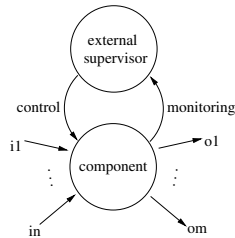

Figure 4: A typical component control loop.

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 Fig. 5, constitutes 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. 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 thought 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.

Exceptions constitute a useful concept present in numerous programming languages (C++, Java, etc.) to separate error handling from the normal flow of instructions in a program. Importing this concept of exception, a CoolBOT component may define a list of *component exceptions* to signal and handle erroneous, exceptional or abnormal situations dur-
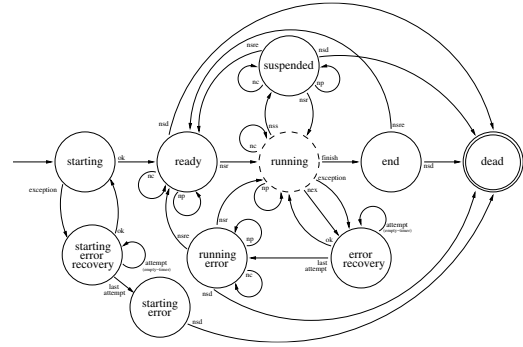

Figure 5: The Default Automaton.

ing execution. In particular, in CoolBOT we may distinguish two levels of exception handling:

- **At local level**. As a good rule of design any component must deal with any exception first at a local level, running its associated handler in the way the *default automaton* establish (figure 5). If, as a result of the whole process of recovering from a exception, the exception persists, the component gets into **starting error** or **running error** states, and remains there waiting for external intervention.

- **At supervisor level**. Errors arriving to an external supervisor in a control loop (Fig. 4) from any of its *local* components must be managed first by this external supervisor. In turn those errors can be either ignored, or propagated to another external supervisor situated in a control loop of higher hierarchy in a system.

Analogously to modern operating systems that provide IPC (**I**nter **P**rocess **C**ommunications) mechanisms to inter communicate processes, Cool-BOT 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 inde-

pendently, offering optimized and reliable communication abstractions. CoolBOT components inter communicate by means of *port connections* formed by *output ports* and *input ports*. Table 3 shows all the different types of output and input ports supported by CoolBOT (on the right and on the left respectively), and all their possible combinations. Each type of port connection implements a different model of interaction between components.

Table 3: Port connections.

Output Port	Input Port	Brief Description
OTick	ITick	Implements a protocol to signal events between components (*tick connections*).
OGeneric	ILast	There is a queue (fifo) of port packets in the input port (*last, fifo and unbounded fifo connections*).
	IFifo	
	IUFifo	
OPoster	IPoster	There is a "master copy" of port packets in the output port, input ports keep local copies (*poster connections*).
OShared	IShared	Components share a "shared memory" residing in the output port. Implements a protocol of shared memory (*shared connections*).
OMultiPacket	IMultiPacket	Accepts multiple type of port packets through the same connection (*multi packet connections*).
OLazyMultiPacket		
OPrioriy	IPriorities	Implements a protocol of sending with priority (*priority connections*).
OPull	IPull	Implements a protocol of request/answer between components (*pull connections*).

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. In general, a component needs for its execution at least a thread in the underlying operating system, called the *main thread*. This is the thread that executes the automaton of the component, and it is responsible for maintaining the consistency of the internal data structures that conform the internal state of the whole component. Additionally, 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 called *port threads*.

CoolBOT components are classified into two kinds: *atomic* and *compound* components.
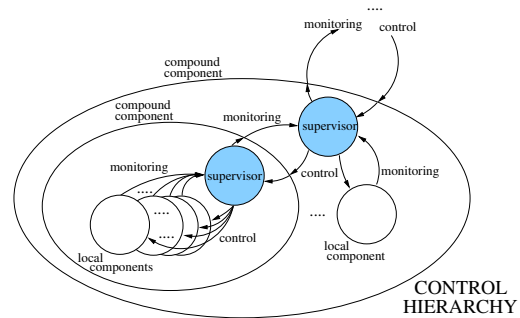

Figure 6: Compound components.

- *Atomic components* that 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. In this way they become isolated pieces of deployable software in the form of CoolBOT components. Thanks to the uniformity of external interface and internal structure the framework imposes on components, they may be used as building blocks that hide their internals behind a public external interface.

- *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. 6, which controls and observes the execution of *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 another 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.

2.1 Developing Components

The process of developing CoolBOT components and systems is resumed on figure 7 in six steps. **(1) Definition and Design**: in this step the component is completely defined and designed. This comprises deciding if it is atomic or not, 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 as skeleton, or generate a new one from a component skeleton description language by means of a compiler. **(3) Code Fulfilling**: Using the component´s 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.
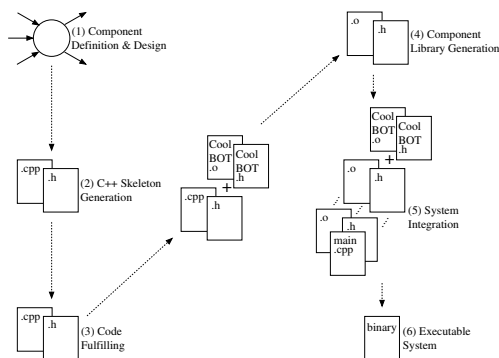

Figure 7: The development process.

3 A Simple Demonstrator

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. 8 and it is made up of four components: the *Pioneer* 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 to avoid obstacles while executing a high level movement command like, for example, going to a specific destination point.
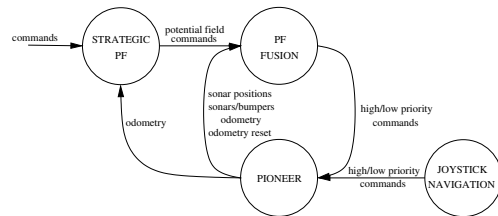

Figure 8: The avoiding level

The second and last level of our demonstrator is depicted in Fig. 9. Note that the systems adds 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 [6][7].

In particular, the integration depicted in Fig. 9 works as follows. The *Sick Laser* component works in 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*
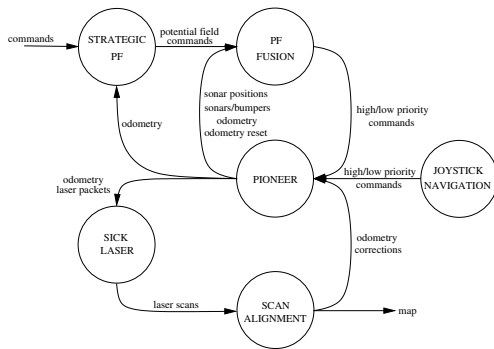
Figure 9: The whole system

Figure 10: The control interface.

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 [6][7]. 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.

3.1 Control Interface

In CoolBOT, as we have mentioned in section 2 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.

For the system integration of Fig. 9 was designed a graphical front-end to control and observe the system during operation, a snapshot appears in Fig. 10. 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 connects
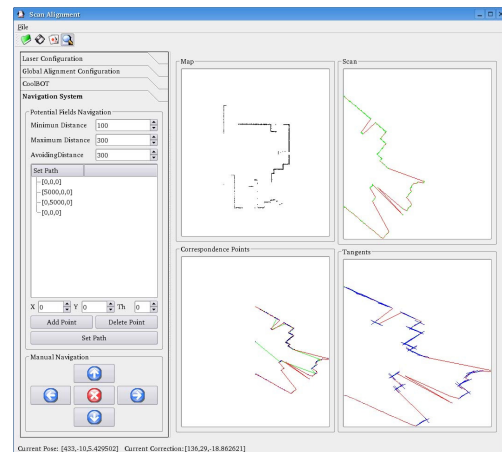
to different ports of the system to have a look to the different information (robot odometry, odometry corrections underdone, laser scans, the map as it gets built, etc.). Moreover, it is possible to change some operational *controllable* variables (frequency of laser scans, *Scan Alignment* component parameters, etc.). Additionally, it is possible establish different paths the robot should follow during map building. Besides, as we have commented previously it is also possible to control the robot movements directly with a joystick device.

3.2 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 error unrecoverable state, on the contrary, it 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. 9, 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 is to overcome a situation provoked by a faulty component without collapsing. It should be able of keeping operation, either running, or waiting for the faulty component to restore the situation. In this test the system did not collapse when any combination of its component was in an non-recoverable error state. Table 4 enumerates every possible combination of error states in any component. In the three first columns *R* stands for *Running* and *E* for *Error*. Any of the errors shown in Table 4 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 4 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, if the serial connection is repaired, the component will restart its normal operation. Once the laser is working again, the *Scan Alignment* component will receive againg new scans and will produce new corrections which, in turn, will be sent to the *Pioneer* component, finishing in this way,

odometry degradation.

Table 4: 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.

4 Conclusions

This document 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 inter-connected 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.

References

[1] E. Coste-Maniere and R. Simmons. Architecture, the Backbone of Robotic Systems. Proc. IEEE International Conference on Robotics and Automation (ICRA'00), San Francisco, 2000.

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

[3] A. C. Domínguez-Brito, D. Hernández-Sosa, and J. Cabrera-Gámez. Programming with Components in Robotics. Waf 2002 - III Workshop Hispano-Luso en Agentes Físicos, Murcia, Mars 2002.

[4] 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.

[5] 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. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 842–848, Grenoble, Francia, September 1997.

[6] 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.

[7] F. Lu and E. Milios. Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4(4):333–349, 1997.

[8] 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 - Eurobot´99. Zürich, Switzerland, September 1999.

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

[10] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.