

Programming by Integration in Robotics^{*}

José L. Fernández-Pérez, Antonio C. Domínguez-Brito, Daniel Hernández-Sosa,
and Jorge Cabrera-Gómez

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

Abstract. This document presents the first operating version of CoolBOT, a component oriented software framework for programming robotic systems. 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

Developing and integrating software for controlling robotic systems is costly due to the complexity inherent in these systems. There is a need for tools that permit a reduction in the programming effort, aiming at the generation of modular and robust applications, and promoting software reuse. The techniques which are of common use today in other areas are not adequate to deal with the complexity associated with these systems [1]. Some authors [2] have made already similar considerations identifying the building of software architectures as the way the robotics community has mainly chosen to address the problem. Other authors [3][4] 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 *CoolBOT*, a component-oriented software framework aimed to programming robotic systems based on a *port automata model* [5][6] that fosters controllability and observability of software components. Thus, in the next section, Sect. 2, a short introduction to the framework will be given, where their main concepts and abstractions will be briefly explained. Next, in Sect. 3 a simple demonstrator is commented in some detail, and finally, in Sect. 4 we will comment some of the conclusions we have drawn from this work.

^{*} This work has been partially supported by the research project *PI2003/160* funded by the Autonomous Government of Canary Islands (Gobierno de Canarias - Consejería de Educación, Cultura y Deportes, Spain), and by the ULPGC research projects *UNI2004/11* and *UNI2004/25*.

2 CoolBOT

CoolBOT [7] 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* [8] 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.

In CoolBOT, components are modelled as *Port Automata* [5][6]. 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. 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.

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. 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 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 , and nex . Subscripts in ns_i indicate which state has been commanded: r

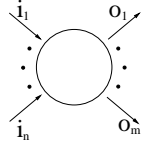


Fig. 1:
Component
external view

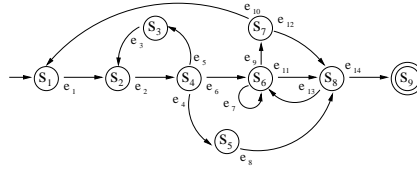


Fig. 2: Component internal view

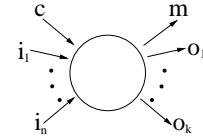


Fig. 3: The *control*
port, *c*, and the
monitoring port, *m*

(*running* state), *re* (*ready* state), *s* (*suspended* state), and *d* (*dead* state). Event *np* happens when an external supervisor forces a priority change, and event *nex* occurs when it provokes the occurrence of an exception.

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. Having a look to Fig. 5 we can see how CoolBOT components evolve along their execution time. Basically, the *default automaton* organize the life of a component in several phases which correspond to different states: *starting*, devised for initial resource allocation; *ready*, the component is ready for a task execution; *running*, here the component executes its specific task; *suspended*, execution has been suspended temporarily; *end*, a task execution has just been completed.

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

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. There are several types of output and input ports supported by CoolBOT which combined adequately implement different protocols of interaction between components. Specifically the framework offers the following protocols: a protocol for event signaling, an active sender/passive receiver protocol, a passive sender/active receiver protocol, a protocol for sharing memory between components, a protocol for connections transporting packets of multiple types, a sending-with-priority protocol and a request/answer protocol.

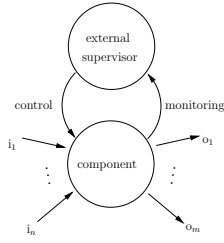


Fig. 4: A typical component control loop

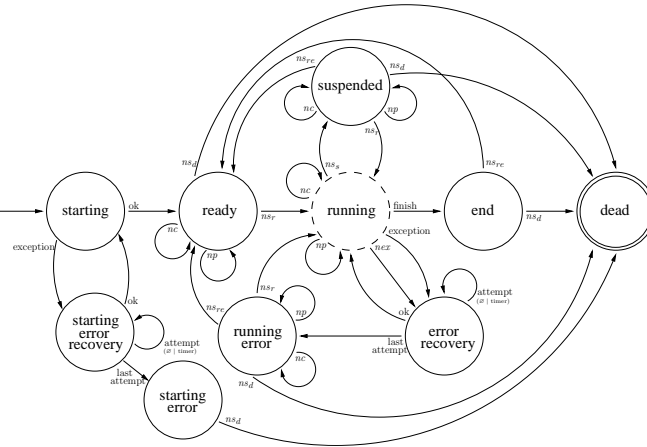


Fig. 5: The Default Automaton.

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. 6, 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 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 Development Process

The process of developing CoolBOT components and systems is resumed on Fig. 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

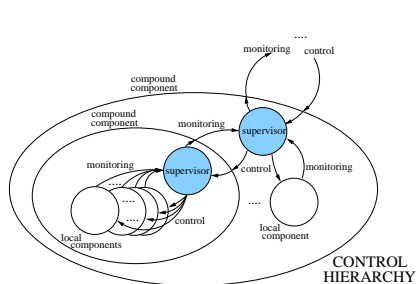


Fig. 6: Compound components.

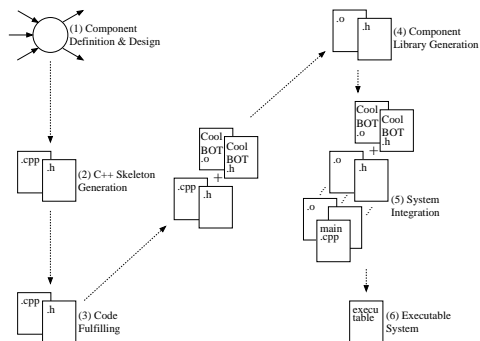


Fig. 7: The development process.

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.

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* which encapsulates the set of sensors and effectors provided by an ActivMedia Robotics Pioneer robot; the *PF Fusion* that 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. The second and last level of our demonstrator is depicted in Fig. 9. Note that the systems adds two new components, the *Sick Laser* which controls a Sick laser range finder and *Scan Alignment* that performs self-localization using a SLAM (Simultaneous Localization And Mapping) algorithm [9][10].

4 Conclusions

This document describes briefly a first operating version of CoolBOT, a component-oriented C++ programming framework supported under

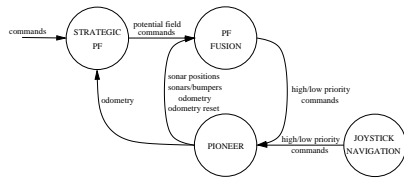


Fig. 8: The avoiding level

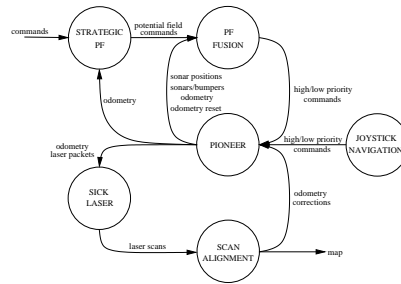


Fig. 9: The whole system

GNU/Linux and Microsoft Windows that favors a programming methodology for robotic systems that fosters software integration, concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing. The framework also promotes a uniform approach for handling faulty situations.

References

1. Kortenkamp, D., Schultz, A.C.: Integrating robotics research. *Autonomous Robots* **6** (1999) 243–245
2. Coste-Maniere, E., Simmons, R.: Architecture, the Backbone of Robotic Systems. *Proc. IEEE International Conference on Robotics and Automation (ICRA'00)*, San Francisco (2000)
3. Fleury, S., Herrb, M., Chatila, R.: $G^{en}oM$: 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)*, Grenoble, Francia (1997) 842–848
4. Schlegel, C., Wörz, R.: 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 (1999)
5. Steenstrup, M., Arbib, M.A., Manes, E.G.: Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences* **27** (1983) 29–50
6. Stewart, D.B., Volpe, R.A., Khosla, P.: Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Transactions on Software Engineering* **23** (1997) 759–776
7. Domínguez-Brito, A.C., Hernández-Sosa, D., Isern-González, J., Cabrera-Gómez, J.: Integrating robotics software. *IEEE International Conference on Robotics and Automation*, New Orleans, USA (2004)
8. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley (1999)
9. Lu, F., Milios, E.: 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)
10. Lu, F., Milios, E.: Globally consistent range scan alignment for environment mapping. *Autonomous Robots* **4** (1997) 333–349