

Programming by Integration in Robotics^{*}

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

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (IUSIANI), Universidad de Las Palmas de Gran Canaria, Spain
jfernandez@iusiani.ulpgc.es, adominguez@iusiani.ulpgc.es,
dhernandez@iusiani.ulpgc.es and jcabrera@dis.ulpgc.es

1 Introduction

Developing 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]. This document presents *CoolBOT* [2], a component oriented framework for programming robotic systems, based on a *port automata model* [3] that fosters controllability and observability of software components.

CoolBOT [2] 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* [4] 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* [3]. 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

^{*} This work has been 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.

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.

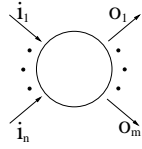


Fig. 1:
Component
external view

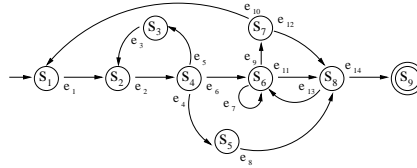


Fig. 2: Component internal view

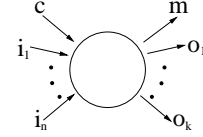


Fig. 3: The *control*
port, *c*, and the
monitoring port, *m*

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* (*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* organizes the life of a component in several phases which correspond to different states: *starting*, devised for initial resource allocation;

tuion of any of its components. In the system of Fig. 7, the key components are: the *Pioneer*, *Sick Laser* and the *Scan Alignment* components. Based on these three components, two tests were made. The first of them showed 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 stopped running.

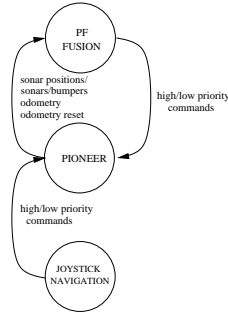


Fig. 6: The avoiding level

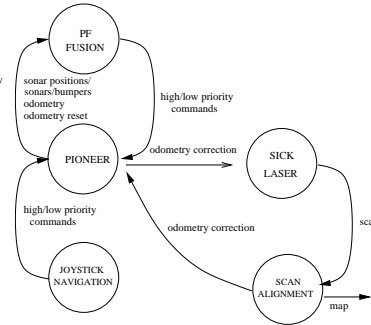


Fig. 7: The whole system

3 Conclusions

This document describes briefly CoolBOT, a component-oriented C++ programming framework 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. Domínguez-Brito, A.C.: CoolBOT: a Component-Oriented Programming Framework for Robotics. PhD thesis, Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria (2003)
3. 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
4. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley (1999)
5. 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)
6. Lu, F., Milios, E.: Globally consistent range scan alignment for environment mapping. *Autonomous Robots* **4** (1997) 333–349