Runtime Self-Adaptation in a Component-Based Robotic Framework*

Daniel Hernández-Sosa, Antonio C. Domínguez-Brito, Cayetano Guerra-Artal, and Jorge Cabrera-Gámez
IUSIANI (Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería),
Universidad de Las Palmas de Gran Canaria, Edificio Central del Parque Científico Tecnológico,
Campus de Tafira, 350017, Las Palmas de Gran Canaria, Spain
{dhernandez, adominguez}@iusiani.ulpgc.es, {cguerra, jcabrera}@dis.ulpgc.es

***Abstract*— The development and maintenance of software for robotic systems is a hard task due to the complexity inherent in these systems. Besides, the resulting applications have to deal with limited resources and variable execution conditions that must be considered in order to keep an acceptable system performance. To address both problems we have integrated a set of dynamic adaptation policies inside CoolBOT, a component oriented framework for programming robotic systems. CoolBOT contributes to reduce the programming effort, promoting robustness and code reuse, while the adaptation scheme provides a dynamic modulation of system performance to meet available computational resources at runtime. In this paper we also present two demonstrators that outline the benefits of using the proposed approach in the development of real robotic applications.**

***Index Terms*— robotic systems, self-adaptive software components**

I. Introduction

Let us imagine we are about to design and build the software for a robot that will operate as a tactical multi-purpose system. In order to reduce the programming effort, we will rely on software reuse, recycling a set of well-tested components (motion control, obstacle avoidance, mapping, path planning, etc.), for our application. Given that the system should operate adequately in a specific hardware equipment, several questions arise. Once the whole system is integrated, which working periods should we assign to the different periodic components?. Could it be possible to impose CPU load quotas to each subsystem in order to guarantee its execution and, at the same time, avoiding processor overload?.

In most situations this is a test-and-error calibration process, for having the system working conveniently, verifying its design requirements, and above all, guarantying that components observe their working periods. Would it be possible to make the system to adapt automatically its configuration and to tune its behavior in order to accommodate itself to the available resources, avoiding this detailed and tedious process of calibration?. Furthermore, due to the tactical conception, the system evolves along their life cycle in a set of execution phases or contexts that demand different topologies and configuration of components. Thus, this calibration problem appears recurrently during system execution.

*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.

So stated, our problem requires two basic contributions: modular software reuse and dynamic adaptive control.

When strict guaranties of bounded reaction times and frequencies of operation are needed, hard real-time techniques are the obvious choice. However, there are many contexts of application in robotics where those strict guaranties can be relaxed to a certain extent. In these contexts, while hard real-time off-line analysis could guarantee system's limits for the most demanding or worst case situations, it may result in under-utilization of available resources during significant periods of time. A soft real-time adaptive control scheme may, in our opinion, represent a more convenient solution.

The management of shared resources at low-level has not received enough attention in robotic and sensor-effector systems research. Some authors [1][2], however, coincide in the necessity of including the adaptive aspect in order to build really robust systems. This is specially important in mobile robotic systems, configured as tactical multi-objective designs which are often affected by the shortage of resources [3].

In the context of software development for robotic applications, the complexity of programming and maintenance [2] has promoted the proposal of architectures [4] and frameworks [5] [6]. Following this tendency we have designed and built **CoolBOT** [7] [8], a component-based programming framework aimed at facilitating the development of robotic systems.

As a consequence of this analysis, we have integrated in CoolBOT a set of mechanisms and policies aimed at obtaining run-time system adaptability. These capabilities are offered to the framework users as a valuable resource in the design of robotic applications. This binomial component-based software/adaptation has also been considered by other authors [9] [10] in other application domains with promising results. There are also some examples of adaptive robotic architectures [11], but not for soft real-time frameworks.

This paper is organized in the following sections: first, in section II, a brief outline of CoolBOT. Then, in section III the integrated adaptation mechanisms are presented. Section IV explains briefly two demonstrators. Finally, section V is devoted to present the conclusions we have drawn from this work.

II. COOLBOT

CoolBOT [7] [8] is a component-oriented framework that allows designing software in terms of composition and integration of software components. The framework provides means to abstract, design and build these components and to compose and integrate them hierarchically and dynamically conforming a whole system.

In CoolBOT, components are active entities that act on their own initiative, carrying out their own specific tasks, running in parallel or concurrently, and are normally weakly coupled. From this point of view, a robotic system might be seen as a network of weakly coupled parallel and/or concurrent components interacting asynchronously in some way. More specifically, components are modelled as **Port Automata** [12][13], a concept that establishes a clear distinction between the internal functionality of a component, an automaton, and its external interface, conformed by input and output ports. Components only interact and inter-communicate externally by means of *port connections* established among their input and output ports.

The framework introduces two kinds of facilities in order to support monitoring and control of components: *observable variables*, which represent features of components that might be of interest from outside in terms of control, or just for observability and monitoring purposes; and *controllable variables*, that represent aspects of components which might be externally controlled so, through them, the internal behavior of a component can be modified from outside. In addition, to guarantee external observation and control, CoolBOT components provide by default two important ports: the *control* port, *c* and the *monitoring* port, *m*, both depicted in Fig. 1. The figure also illustrates graphically the whole external interface of a typical component: these two defaults ports and the rest of its input and output ports (labelled as $i_1, ..., i_n$, and $o_1, ..., o_k$ respectively). Fig. 2 illustrates a typical control loop for a component using another component as an external supervisor.

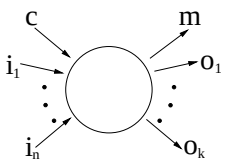

Fig. 1: The *control* port, *c*, and the *monitoring* port, *m*

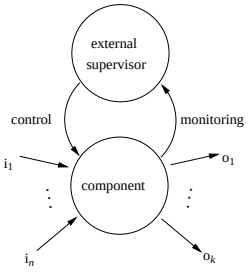

Fig. 2: A typical component control loop.

Internally all components are modelled using the same default state automaton, the *default automaton*, shown in Fig. 3, that contains all possible control paths a component may follow. The *default automaton* can be always brought externally in finite time by means of the *control* port to any of the controllable states of the automaton, which are: **ready**, **running**, **suspended** and **dead**. The rest of states are reachable only internally, and from them, a transition to one of the controllable states can be forced externally. The **running** state, the dashed state in Fig. 3, 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.

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*.

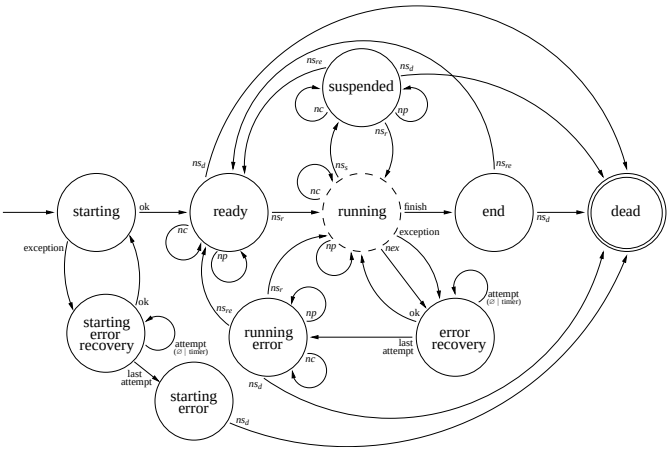

Fig. 3: The Default Automaton.

CoolBOT components are classified into two kinds: *atomic* and *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. 4, 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. 3. 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.

Analogously to modern operating systems that provide IPC (**I**nter **P**rocess **C**ommunications) mechanisms to inter communicate processes, CoolBOT provides ***I**nter **C**omponent **C**ommunications* or *ICC* mechanisms to allow components to interact and communicate among them. CoolBOT *ICC* mechanisms are carried out by means of input ports, output ports, and ports connections. 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.

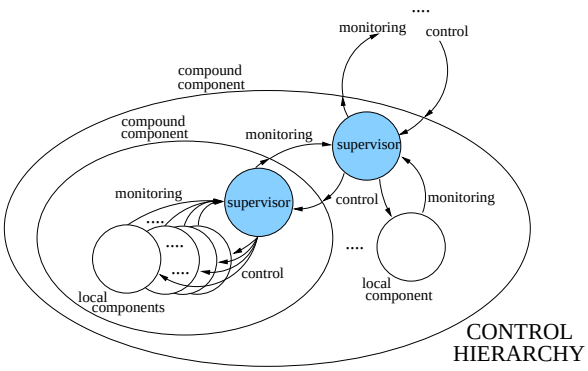

Fig. 4: Compound components.

III. Adaptive Control

A robotic application should be able to adjust its performance as a function of either the available resources or the resources assigned a priori. The objective is to force a smooth degradation when resources are not enough to meet application demands, and to allow a controlled recovery when the system overload disappears. See [14] for a more complete description.

The CoolBOT framework provides mechanisms to support the adaptation of component consumption of computational resources during operation. CoolBOT adaptation mechanisms include a graceful degradation procedure when there are not enough resources available, and a performance status recovery procedure whenever possible. Additional objectives are reactivity, stability and coordination to avoid system imbalances.

A. Elements of System Adaptation

From the point of view of adaptation, a mission may be decomposed into a set of phases, each one of them defined by a configuration of components running concurrently. We will name as *computational context* the configuration of components that defines each one of these phases.

A component, whether atomic or compound, may be declared as *adaptive* or *non-adaptive*. If a component is declared as adaptive, it must publish the set of *performance levels* at which it can operate. A performance level represents a trade-of between resource consumption and quality of results (better quality demands higher consumption). In general, CoolBOT components which are declared as adaptive should be formulated as anytime contract algorithms [15].

The integration of the dynamic adaptation of components inside CoolBOT is designed around two controllable variables: *frequency of operation* and *quality level*. On the frequency axis, a supervisor can modify the period associated to any of the components under its control, for example, increasing their values to face CPU saturation. On the quality axis, the supervisors can command lower qualities (sensor resolution, accuracy of computations, exhaustiveness, etc.) to reduce CPU load and latencies at the cost of increasing uncertainty or decreasing results' quality. A performance level represents a combination of quality and frequency parameters. Externally it is not necessary to know which component configuration corresponds to a certain level of performance. The only condition is that these adaptation levels should be ordered in a monotonic order according to the system load they induce.

The framework will monitor certain operation conditions, named as *adaptive observables*, at run time. These include component level measures such as period, elapsed time or cpu time, and system level measures as computational load, battery level or load profile. Some results from processing can also be used as elements in adaptive control, using the observable variable facility offered by the framework.

Depending on these measures and their reference values some elementary adaptation commands can be triggered on adaptive components through their control ports:

- *Degrade (or Demote)*. Degrades a component to the immediately lower level of performance.
- *Promote*. Promotes a component to its immediately upper level of performance.
- *Operate at a specific level*. Brings a component to a specific level out of the levels of performance the component accepts.

B. Control Strategies

Several control policies have been designed to organize system adaptation. Their objectives include avoiding an unbalanced degradation/promotion in the system, reduce settling

times and fostering stability. Due to the reduction of performance associated to frequency or quality degradation, the system always tries to restore the nominal parameter values as soon as resource limitations disappear.

To support the low level adaptive aspects in CoolBOT we have introduced in all atomic components a special thread called *control thread*, implemented as a port thread. At a higher level of control, adaptive aspects are provided transparently by the supervisors, in compound components and at system level.

The control sequence begins with the activation of one or more control loops detecting adaptive observables that are out-of their desired ranges. Whenever possible, control actions are triggered hierarchically, local actions at component-scope first, and, if problems persist, global actions at system-scope later. The internal control threads are in charge of local scope, while supervisor components operate at higher level.

We will describe now in more detail two adaptive control strategies for controlling timeouts and system load.

1) Timeout control: Timeouts control adapts, on a hierarchical basis, the runtime demands of shared resources in the system in order to guarantee the specified frequencies of operation. Firstly, period violations are detected locally inside the time-pressured component, where the control thread generates the corresponding degradation order. To avoid systems unbalance, however, local control actions are limited to a scope defined by two homogeneity thresholds (minimum and maximum degradation values). If local adaptation resources are not enough, the component notifies the problem to upper levels, the supervisor component, where global actions can be executed.

2) CPU load control: The load control loop operates only at global level. The system load is estimated and compared with a certain reference level fixed externally. Promotion and degradation actions are generated accordingly to maintain the desired load level.

Candidate selection for targeting control actions plays an important role in adaptation performance. In general, an agreement between reactivity and stability must be reached. The most intense reactions are obtained when one or more of the following conditions are met:

- High frequency components.
- CPU demanding components.
- Multiple destination source components.
- High-resolution sensor components.

The supervisors evaluate these parameters as well as priority to select target components for adaptation commands. An example of conservative policy consist in selecting always the least degraded component among the lowest priority ones for degradation, and the most degraded among the highest priority ones for promotion, applying only a minimum-step control (one performance level jump on each command).

The clear separation of control, processing and communication areas inside facilitates the implementation promoving and preserving modularity in robotic applications.

IV. Demonstrators

In order to illustrate the operation of the adaptation mechanisms on real-world applications, we have implemented two demonstrators: a visual tracking system and a mobile robotic application. As it will explained next each one of them has been integrated using a set of CoolBOT components.

A. First Demonstrator: A Tracking System

The first demonstrator consists in a correlation-based tracking system for a robotic head. A USB web-cam (3Com HomeConnect) has been mounted on a pan-tilt capable neck (DirectedPerception PTU) controlled via serial port.

The goal of the application is to detect first, and keep centered on the image then, a certain pattern. A correlation-based measure [16] is used to localize the target on the image. Simultaneously, the system must perform additional feature extraction (grey-level variance, color detection) on images. The application is organized in the following four components:

- **Vision server**. Atomic component to control the robotic head and serve images to the consumer components.
- **Tracking**. Compound component that processes image data using a correlation algorithm to generate commands for pan/tilt unit. It includes three atomic components that endow the component with three different behaviors: “Normal Tracking”, “Active Searching” and “Passive Searching”.
- **Color-Based Object Detection**. Atomic component that executes a color-based object detection algorithm on image data.
- **Variance**. Atomic component that computes the variance feature from image data.

During execution all components are active, with the tracking component switching between three operation modes associated to its internal atomic components: “Normal Tracking”, “Active Searching” and “Passive Searching”. For explanation purposes we will assign these modes to global system phases or computational contexts. The Fig. 5 shows the configuration of components and interconnections for this demonstrator.

We will show here only the adaptation results associated to the “Normal Tracking” computational context. Fig. 6 illustrates the evolution of the adaptive observable *system load* when different reference levels are commanded. Initially there is no limit so all components execute at their maximum performance level (this represents a 60% of system load). Around 60 seconds the system load reference is set to 30%, and degradation orders are issued from the supervisor to components until this level is reached. Around 130 seconds the reference level is increased to 40% and some components are allowed to promote. After that (200 seconds approx.) a zero reference level is commanded, which drive all components to its maximum degradation levels (this results in about a 15% of system load). Finally some performance recovery is tested again moving the reference level to 30% and 40%.

This result is obtained by applying several computational adaptation mechanisms. For example, in the case of the tracking component they are all quality-oriented: multiple image

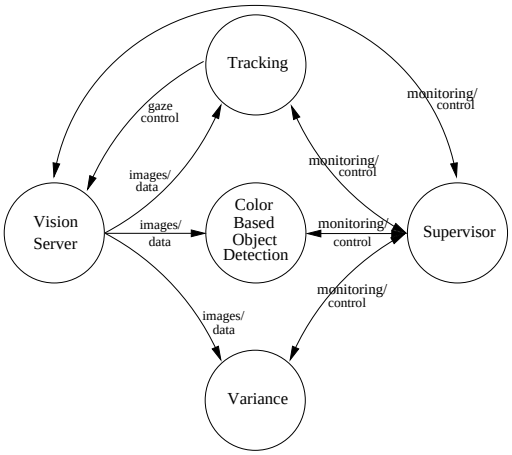

Fig. 5: Components making up the first demonstrator.

resolutions, different correlation pattern sizes and multiple searching areas. Additionally, variable resolution and frequency of operation can be modified in the feature extraction components, allowing for the modification of computational demands. Sensor component can also be configured to produce variable resolution image data to affect system load.

Regarding strategies, in case of overload, the adaptation try to degrade first color detection and variance components, and later, the tracking component. When conditions for system performance recovery are met, the higher priority tracking component promotes first and secondary feature extraction components later.

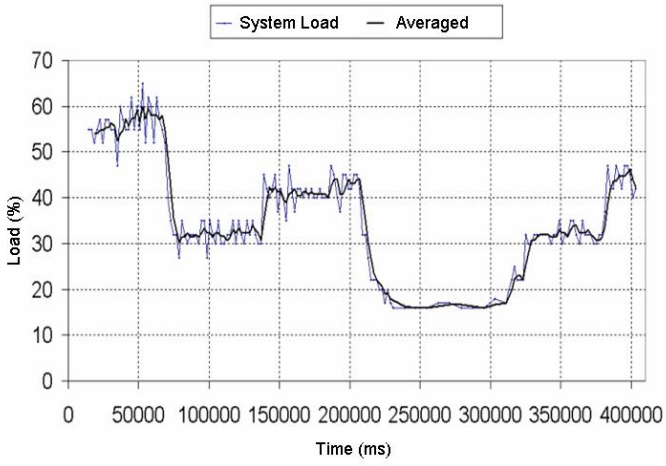

Fig. 6: System load with CPU global control activated

B. Second Demonstrator: Mobile Searching Robot

The second demonstrator mounts the mechanical head described on the previous application on a mobile robot (Pioneer). A notebook has been added for running the application, being connected via USB and serial ports to the head and the robot. Fig. 7 shows the hardware platform used for this experiment.

Fig. 7: Mobile robot and active camera.

A minimal multi-purpose system has been designed combining two main objectives, line following and object detection, that can be prioritized alternatively. In the configuration used for this demonstrator, the robot must follow, as tight as possible, a trajectory defined by a line traced on the floor. At the same time, the robot must look at both sides of the route trying to detect some colored balls. The first task is considered to have a higher priority than the second one, so the adaptation strategy operates modifying the frequency of execution and quality level of the object detection task.

Four CoolBOT components have been used in the integration of the system corresponding to this second demonstrator: one for controlling the Pioneer robot, other one for the PTU unit and the camera, another one for line following, and the last one for object detection. Note that some of the components has been also used in the previous demonstrator.

As configured, on straight-line trajectory segments both tasks can perform alternatively at a pre-defined frequency. On curved segments, however, the risk of loosing the track increases. To avoid this, the movement amplitude and activation period of the object detection component is modulated according to an adaptive observable computed from the curvature of the line that the robot must follow. The modification of the scanning amplitude can be considered a quality-based adaptive control, as processing times are shortened at the cost of reducing the probability of finding color objects. The modification of the period, however, corresponds to a frequency-based adaptive control.

The Fig. 9 represents the executions of the color detection component task along the trajectory. On curved segments, both frequency and amplitude of scanning take lower values. On straight segments both parameters can increase their values.

In this experiment different priorities configurations can be used, leading to alternative robot behaviors. For example, if object color searching becomes the priority task, it is the robot velocity the variable that is conditioned by the amplitude of the gaze scanning movement.

V. Conclusions

In this paper, the runtime adaptation mechanisms available in CoolBOT have been presented. In CoolBOT, the control of shared resources has been integrated in the facilities offered by the framework. If this capacity, is to be used by the programmer, components must be declared adaptive and designed with adaptation capabilities. Adaptive components

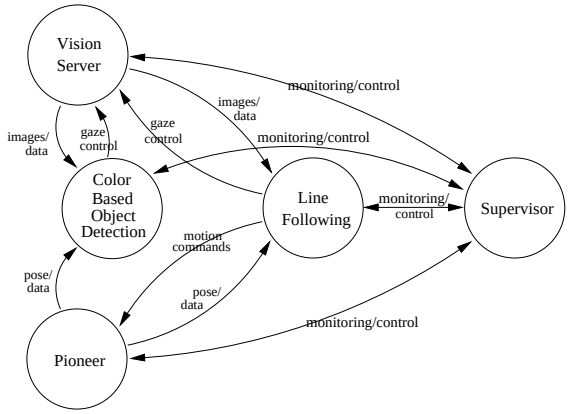

Fig. 8: Components making up the second demonstrator.

can coexist with non-adaptive components without problems. These adaptation mechanisms allows the system regulate the load that a computational context may provoke on the system or can be used to make room for new components when the computational context changes. The objective has been to introduce mechanisms that must avoid uncontrolled degradation of the system in high load situations, paving the road to achieve more robust systems.

CoolBOT's adaptation mechanisms can't guarantee real-time performance nor have been conceived with that objective in mind. Instead, the goal was to guarantee the stability of a tactical system, avoiding uncontrolled degradation of system's performance in situations of overloading. In accomplishing this goal, the adaptation mechanisms must allow the system to respect tasks' deadlines or frequency of operation, while at the same time trying to make the best usage of heterogeneous shared resources without demanding specialized tools like real-time systems. Finally, two demonstrators illustrate the effectiveness of the proposed mechanisms on different real-world applications.

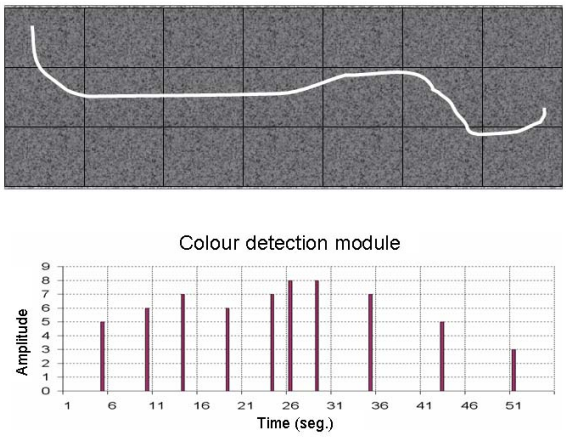

Fig. 9: Color detection component execution chronogram.

REFERENCES

[1] R. R. Murphy, *Introduction to AI Robotics*. The MIT Press, 2000.

[2] D. Kortenkamp and A. C. Schultz, "Integrating robotics research," *Autonomous Robots*, vol. 6, pp. 243–245, 1999.

[3] S. D. Jones, "Robust task achievement," Ph.D. dissertation, Institut National Polytechnique de Grenoble, 1997.

[4] 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.

[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)*, Grenoble, Francia, September 1997, pp. 842–848. [Online]. Available: citeseer.nj.nec.com/fleury97genom.html

[6] 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.

[7] A. C. Domínguez-Brito, D. Hernández-Sosa, I.-G. Josep, and J. Cabrera-Gámez, "Integrating robotics software," IEEE International Conference on Robotics and Automation, New Orleans, USA, April 2004.

[8] A. C. Domínguez-Brito, "CoolBOT: a Component-Oriented Programming Framework for Robotics," Ph.D. dissertation, Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria, September 2003.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 9162, no. 10, pp. 46–54, October 2004.

[10] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, May/Jun 1999.

[11] D. J. Musliner, R. P. Goldman, M. J. Pelican, and K. D. Krebsbach, "Self adaptive software for hard real-time environments," *IEEE Intelligent Systems*, vol. 14, no. 4, pp. 23–29, July/August 1999.

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

[13] 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*, vol. 23, no. 12, pp. 759–776, December 1997.

[14] D. Hernández-Sosa, "Adaptación computacional en sistemas percepto-efectores. Propuesta de arquitectura y políticas de control," Ph.D. dissertation, Universidad de Las Palmas de Gran Canaria, 2003.

[15] S. Zilberstein, "Using anytime algorithms in intelligent systems," *AI Magazine*, vol. 17, no. 3, pp. 73–83, 1996.

[16] C. Guerra-Artal, "Contribuciones al seguimiento visual precategórico," Ph.D. dissertation, Universidad de Las Palmas de Gran Canaria, 2002.