Programming with Components in Robotics*

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

Dpto. Informática y Sistemas
Universidad de Las Palmas de Gran Canaria, Spain
Edf. Informática y Matemáticas, Campus de Tafira
35017 Las Palmas de Gran Canaria, Spain
{acdbrito,dhernandez,jcabrera}@dis.ulpgc.es

Abstract

This paper describes a component-oriented programming framework for robotics, **CoolBOT**, which is actually under development at the University of Las Palmas de Gran Canaria (ULPGC). The framework has been designed to assist robotic system developers in building more structured and reusable systems. Components are the basic building blocks used in this framework, modeled as Port Automata, PA [7], that interact through their ports and that can be composed to build up new components from existing ones. Components, whether atomic or compound, are internally modeled as Discrete Event Systems and controlled using the same state control graph. CoolBOT hides any aspects related to communications and provides standard mechanisms for different modes of data exchange between components, exception handling and support for distributed computing environments.

1 Introduction

During the last years we have known about a number of successful projects in robotics in very different fields, ranging from exploration in space and harsh environments on Earth to medical robotics and entertainment. These systems illustrate from different perspectives that there is actually a wealth of well developed solutions to many of the basic problems that need to be solved and integrated when designing a robotic system, even though many of them still remain as very active research fields. This situation is fostering the development of more ambitious systems of increasing complexity to face new challenges, but also makes evident some difficulties.

*This work has been partially supported by the UE/DGES research project **1FD1997-1580-C02-02**, and by the research project **PI/1999/153** funded by the Autonomous Government of Canary Islands - Gobierno de Canarias.

One of them is the lack of a methodology to develop robotic systems in a principled way, a problem that has been identified by several authors, who have recognized that traditional programming and validation techniques are not adequate for intelligent robotic systems [5][1]. Such a methodology should help in designing systems that were more scalable, reusable in new scenarios, more robust and reliable, and easier to debug and profile.

These problems have been often tackled proposing new robot architectures and specification languages (see [1] for a good up-to-date review). Certainly, some architectures seem better suited than others to favor the goals stated above as proved by the fact that majority of current intelligent robot systems use some sort of hybrid architecture [4][**?**][**?**][6], effectively combining the advantages of reactive and deliberative architectures.

Other research has been focused in the design of specification languages for robot systems, with a large variety in objectives and scope. Relevant to the research presented here, are those languages designed for task-level control as RAP [**?**], ESL [3] or TDL [**?**]. Typically, these languages offer primitives for task coordination and control, task communication, and also basic primitives for exception handling. An advantage of ESL and TDL is that they are, respectively, extensions of Lisp and C++. This allows to use the same language for coding the whole system, that is, not only for task control, but also for coding the rest of the system.

Along this paper we will introduce CoolBOT, a component-oriented programming framework being developed at ULPGC, whose main goal is to bridge the gap between these two approaches for tackling the design of complex robot systems. The rest of this paper discusses first the design objectives that are guiding the development of CoolBOT, followed by a description of CoolBOT, then its methodology of use, and finally, some discussion about the proposed framework.

2 Design Objectives

Latest trends in Software Engineering are exploiting the idea of Components as the basic units of deployment when building complex software systems, specially if software reuse, modular composition and third-party software integration are important issues. CoolBOT should be understood as a component framework, in the sense defined in [9].

The following are the most important considerations that have guided the design of CoolBOT:

- **Component-Oriented.** CoolBOT is conceived as a component-oriented programming framework that relies on a specification language to manipulate components as building blocks in order to functionally define a robotic system by integrating components.

- **Component Uniformity.** A component-based approach clearly demands certain level of uniformity among components. Within CoolBOT this uniformity manifests itself in two important aspects. First a uniform interface is defined for all components based on the concept of port automata. Additionally, 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, besides component uniformity sets the real basis for development of debugging and profiling tools.

- **Robustness & Controllability.** A component-oriented robot system will be robust and controllable because its components are also robust and controllable. A component is considered robust when:

  1. It is able to monitor its own performance, adapting to changing operating conditions, and it also implements its own adaptation and recovery mechanisms to deal with all errors that are internally detectable.
  2. Any error detected by a component that cannot be recovered by its own means, should be notified using standard means through its interface, bringing the component to an idle state waiting for external intervention, that either will order the component to restart or to abort.

  Furthermore a component will be considered controllable when it can be brought with external supervision - by means of a controller or a supervisor - along an established control path. In order to obtain such an external controllability, components will be modeled as automata whose states can be forced by an external supervisor, and where all components will share the same control automaton structure [**?**].

- **Modularity & Hierarchy.** The architecture of a robot system will be defined in CoolBOT using components as elementary functional units. As in almost any component-based framework, there will be atomic and compound units. An atomic component will be indivisible and a compound component will be a component which includes in its definition other components, whether atomic or not, and provides a supervisor for their monitoring and control. A whole system is nothing else but a large compound component including several components, which in turn include another components, and so on. This chain of decompositions finishes when an atomic component is reached. Hence a complete system constitutes a hierarchy of components.

- **Distributed.** CoolBOT components can be distributed along a computer network to integrate a system.

- **Reuse.** Components are units that keep their internals hidden behind a uniform interface. Once they have been defined, implemented and tested they can be used as components inside any other bigger component or system.

- **Completeness & Expressiveness.** The computing model underlying CoolBOT should prove valid to build very different architectures for robotic systems and expressive enough to deal with concurrence, parallelism, distributed and shared resources, real time responsiveness, multiple simultaneous control loops and multiple goals in a principled and stable manner.

3 Framework Description

In CoolBOT, components are modeled as **Port Automata** [7][8][2], because this concept establishes a clear distinction between the internal functionality of an active entity, the automaton, and its external interface, the input and output ports. Components define active entities which carry out a specific tasks, and perform all external communication by means of their input and output ports. Components act on their own initiative, running in parallel or concurrently, and are normally weakly coupled, i.e. no acknowledgements are necessary when they communicate through their ports.

Atomic and compound components are externally equivalent, offering the same uniform external interface and internal control structure. These properties are extremely important in order to attain standard mechanisms that guarantee that any component can be externally monitored and controlled.

3.1 Default Variables, Default Ports and the Default Automaton

In order to be able to build modular systems from reusable units, in CoolBOT all components must be observable and controllable at any time from outside the component itself. W ith this purpose, CoolBOT components may define:

- *observable variables*, variables which are externally observable and permit publishing aspects of the component which are meaningful in terms of control, or just for observability,
- *controllable variables*, variables representing aspects of the component which can be controlled, i.e. modified or updated, externally.

Observable and controllable variables are read and written through two special ports, called the **default ports**, which are present in every component. Figure 1 displays these ports: the control port **c**, through which a component is externally controlled using its controllable variables; and the monitoring port **m**, which exports the components observable variables, and allows the component to be externally observable.

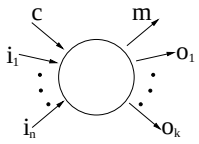

Figure 1: The default ports.

Internally components have a common structure embodied as an automaton, shown on figure 2, the **default automaton**, that contains all possible control paths for a component. In the figure some transitions are labeled as $\mathbf{c}_i$'s denoting that they are provoked by a command through the control port **c** displayed in figure 1. The default automaton is said to be "controllable" because it can be brought externally by means of its control port **c** to any of the controllable states of the automaton: **ready**, **running**, **suspended** and **dead**, in finite time. The rest of states are reachable only internally, and from them

a transition to one of the controllable ones can be forced. The **running** state, the dashed state in figure 2, represents the state or set of states that structures the specific functionality of a certain component. This particular automaton, termed **user automaton**, varies among components and must be defined by the developer/user when the component is implemented.

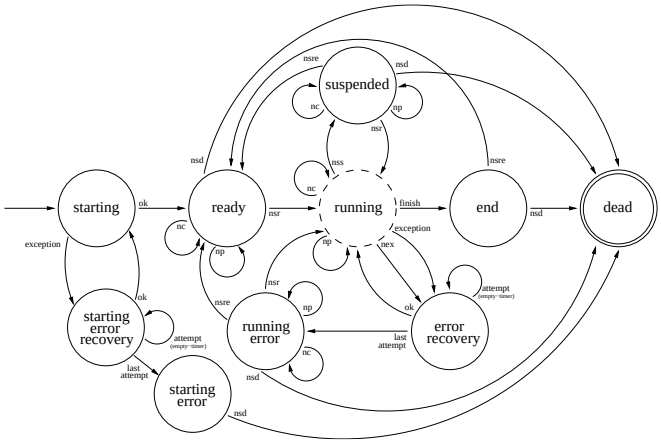

Figure 2: The Default Automaton.

As implied by figure 2, CoolBOT endows all components with several default observable and controllable variables. As to the observable ones: **state** is the default automaton state where the component is at each instant; **user state**, the state where the automaton is when the component is in the user automaton, i.e. the actual state within the **running** pseudostate; **result**, indicates if a task has been finished, and in that case, the results achieved – success or fail– and complementary information about the execution; and **error**, describes the situation when the component gets either into **starting error** state or into **running error** state. As to the controllable variables: **new state**, used to bring the component to one of the controllable states of the default automaton; and **configuration**, the execution and parameters of a component can be modified through this controllable variable.

3.2 Port Connections and Communication Models

A pair output port/input port constitutes a **connection** between two components. Data is transmitted through port connections in discrete units called **port packets**, [2], that can carry information or not. In the last case they are used to signal the occurrence of an event and are called **event packets**; the other ones are termed **data packets**. Each port can only transmit a type of data packet, although it is also possible to define **converters** between different types of port packets.

CoolBOT also provides different typologies of output and input ports that define the type of port connections they can form. Figure 3 depicts all the possibilities, and its graphical representation; the left side represents the types of output ports, the right side the types of input ports, the number at both ends of the arrows indicate the possible cardinality of connections. The possible

types of connections are: a **variable** connection, transports data packets which are variables, whether observable or controllable, mainly used to implement the default ports; a **poster** connection, transport data packets and allows a component to publish data for different consumers; a **tick** connection, only transport event packets, mainly used to implement timers and signal events; a **last** connection, transports data packets and permits a component to publish data for different consumers as well, but in this case local copies are kept on the input end of the connection; a **fifo** connection, transports data packets, the input end of the connection is a queue where packets are queued in a queue of finite length; and finally, a **ufifo** connection is like a fifo connection that grows a bit when it is full (ufifo stands for "unbounded fifo").

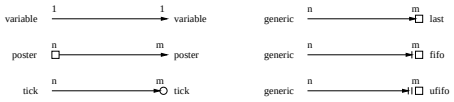

Figure 3: Port Connections

Additionally, there are two basic **communication models** for port connections: the **push** model and the **pull** model. In a push connection the initiative for sending a port packet relies on the output port part, i.e., the data producer sends port packets on its own, completely uncoupled from its consumers. By default all port connections are "pushed". A pull connection implies that packets are emitted when the input part of the communication – the consumer – demands new data to process. In this model the consumer keeps the initiative, and hence it must send a request to the producer whenever a port packet is demanded. Pull connections are implemented using two push connections, for minimum overhead.

3.3 Atomic Components

An atomic component is embodied as a thread and models a port automaton [2]. Atomic components have been devised to abstract hardware like sensors and effectors, and/or other software libraries like third party software.

As any component in CoolBOT, an atomic component is implemented as a port automaton following the model of the default automaton (figure 2). Besides, it is necessary to define the part of the component automaton that is specific to its internal control and functionality, the user automaton, represented by the dashed **running** state in figure 2. Once the user automaton has been defined, to complete the component coding it is necessary to fill in the transitions between automaton states and the states themselves, where several possible sections are provided for each state: an **entry section**, which is a starting code executed each time the automaton gets into the state; an **exit section**, a code executed each time the automaton is about to leave the state; and a **periodic section**, executed periodically when the automaton remains in the state.

Atomic components are implemented as simple DES [2] and a description language is used to define its observable, controllable and local variables, input and output ports, user automaton and the code sections necessary to implement the different states. The description code should include information about what

third party libraries - hardware drivers or other software libraries - must be linked with the component to achieve an executable component. This component description will be then compiled generating a C++ class embodying the component where all transitions and necessary state sections will be codified as class function members that should be filled in by the developer. Also the necessary makefiles will be created to compile the component with the specified third party libraries.

Figure 4 shows two examples of atomic components: a **motion controller**, a component having direct access to a robotic platform and abstracts the low level vehicle motion interface executing motion commands received from other components at a specified frequency; and an **ir/us/bumper server** that encapsulates the low level interface to the sensor suite typically found in any mobile robot, providing time-stamped sensory data scans at a specific frequency; it also detects emergency situations when the platform is too close to an obstacle. On the figure are also depicted the non default observable and controllable variables provided by each component, besides the configuration of each is detailed.

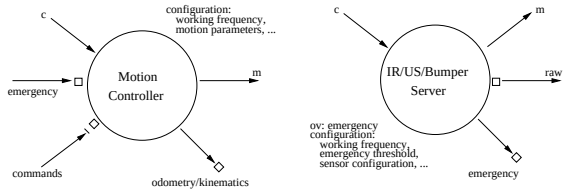

Figure 4: Two Atomic Components

3.4 Compound Components

A compound component, is a composition of instances of another components which, in turn, can be either atomic or compound. Figure 5 graphically illustrates this concept using a compound component that abstracts sensors and motors available in a mobile robot. In other words, a compound component is a component that uses the functionality of instances of another atomic or compound components to implement its own functionality.

The automaton that coordinates and controls the functionality of a compound component is called its **supervisor**, shown also on figure 5, and like atomic components it follows the control graph defined for the default automaton (figure 2). Similarly to atomic components, compound components will be specified by means of a description code. In it, the developer/user completes the rest of the automaton describing the states and state transitions that constitute the user automaton, and also state code sections. Alike atomic components, when this description code is compiled, a C++ class is generated implementing the compound component. The description code of the compound component specifies all its interface including how its local components' input and output ports are mapped as input and output ports of the compound component; its observable and controllable variables, where some of them can be either completely new or mappings of observable and controllable variables of instances of its local components. Automaton transitions between states in the supervisor are triggered by **rules** which are conditions involving observable and controllable

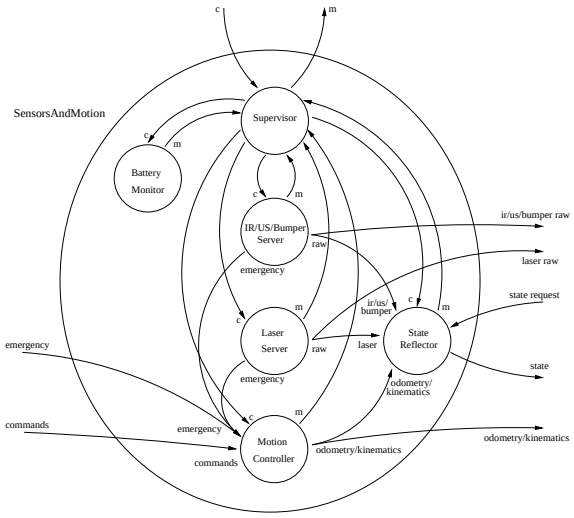

Figure 5: A Compound Component

variables of its local components and/or its own local variables.

3.5 Exception Handling

CoolBOT's exception handling mechanisms exploits two basic ideas. First, all components, no matter if they are atomic or compound share the same exception handling, communication and control schemes. Second, CoolBOT capitalizes on the idea presented in section 2 to build up a reliable system from reliable components.

A component should incorporate the capability to measure its own performance. For example, in case of a periodic task, if it is respecting its frequency of operation, of if another component in a chain of connected components is not keeping its pace or not working at all. The component's definition includes a list of the exceptions that the component can detect, along with specific "continuity plans" whenever they are available. The evolution of the component's state when an exception has occurred is the same for all components. The possible transitions are evident from the control graph of the default automaton.

As it was explained previously, when a component detects an error that it can not deal with, either because there is not any possible recovery mechanism at this level or because the error recovery plan has failed, it communicates the error to its supervisor and goes into a running error state where it waits for external intervention to restart or die. Errors arriving to a supervisor from its local components must be managed first by this supervisor. They can be either ignored, propagated to higher levels in the hierarchy or handled as explained above. However, when handling exceptions within compound components some standard recovery mechanisms are possible, aside from the obvious re-instantiation of the faulty component. Let's suppose, for example, that we have several components that constitute equivalent alternatives for developing the same task, possibly using different resources, but offering the same external interface. Such components could be used alternatively to carry out a specific

task and hence, a general strategy to cope with components in running error might be just substitution of one component with another one providing an equivalent interface and functionality. A complementary strategy may also be useful to avoid suspending a compound component whenever a member of the composition gets into running error. Equivalent components can be declared as **redundant** and executed concurrently or in parallel (i.e. if redundant components execute on different processors), so that if one of then fails, the others will keep the whole component running.

Thus, when a local component instance gets into running error, if a substitute exists, the supervisor will create an instance of it to carry out a substitution and keep the compound component working, and the erroneous instance will be put in a queue of instances to be recovered. Instances in that recovery queue are restarted periodically to check out if the running error persists. There is a deadline for each instance in this recovery queue, if the deadline expires the instance is deleted from the queue and destroyed. Otherwise, if any of them is recovered, the previous situation before its substitution is restored. If a local instance in a running error can not be substituted, it will be added to the recovery queue previously mentioned. If its deadline in the queue is reached then the instance is retired from the queue and destroyed. This may provoke the whole compound component to go to running error, or not, depending on its functionality.

3.6 Adaptation

In an environment like CoolBOT, with multiple periodic component executing concurrently, interactions can lead to low performance situations. If this problem receives no attention, the whole system reactivity/security could be threatened.

We will focus here in system capability to deal with computational resource lack. In a hard real-time system an off-line worst-case analysis can guarantee a correct execution. However, for soft real-time systems this is not a good solution. CoolBOT offers a set of resources and policies aimed at obtaining run-time system adaptability.

This adaptability includes a graceful degradation when there are not enough resources and a performance recovery strategy when some resources are freed. Additional objectives are reactivity, stability and coordination to avoid system imbalance.

3.6.1 Control Actions

Control actions are associated with several component variables. Related controllable variables include **period** or operating frequency for periodic components, **maximum response time** that indicates for the maximum delay for input event processing in aperiodic components, and **priority**. Observable variables are **CPU** or **processing time** as execution time in zero load context (best possible performance), and **elapsed time** as total execution time in operating conditions.

CoolBOT generates three different types of control actions: **degradation** for reducing computational load, **promotion** for recovering from a degraded situation and **equalization** for uniforming load distribution. Each action type

activates in response to different events. Degradation operates on temporal limits violations, promotion on the detection of spare CPU intervals, and equalization on peak-valley load profiles. Diverse techniques apply for event detection: atomic output monitoring for timeout, temporal labeling of input data for response time verification, CPU time versus elapsed time for load profiling, and lowest priority test component for idle CPU intervals.

From the programmers point of view, equalization constitutes an automatic low level mechanism that operates transparently and will not be discussed here. Focusing then on degradation and promotion, they must be taken into account at several application development phases.

- Atomic component design: the designer must define degraded operating modes, if any.
- Compound component design: the designer integrates already existing components adding, in some cases, relative priority levels.
- Application design: the designer selects components and connects them. Priorities, periods and maximum response times are then assigned from high level tasks specifications, e.g. from a task planner reflecting system goals. Periods can be expressed as intervals, indicating maximum and minimum valid operating frequencies.

Available control action are processing time reduction (quality degradation) and period increase (frequency degradation). Control policies coordinate these actions in order to get a fast and balanced reaction to system state.

Degradation follows a hierarchical scheme. At local level and after a time violation detection, the corresponding supervisor enters into error recovery state. If local actions are not sufficient to correct the problem, the supervisor at next level is notified. To avoid system imbalance, local action is limited to degradation levels inside a global homogeneity interval. If, even at whole system level, degradation is not enough, component suspension or system reconfiguration must be considered.

Degradation is organized in two phases. If period intervals have been defined, the system degrades first in frequency, moving into less demanding configurations. After that, if time errors persist, quality degradation applies.

Promotion starts at higher levels on the detection of spare computational resources. The Supervisor coordinates recovery, sending promotion signals to lower level supervisor. As with degradation, homogeneity limits must be observed to guarantee a uniform progress.

3.6.2 Control Policy Algorithms

The implementation of control policies is based on supervisor algorithms and component event detection. Figure 6 ilustrates a simplified version of such a control algorithm.

The selection of component for degradation must consider several factors as priority level associated with time violation, estimated repetition period of errors, components topology, or component's computational demand. Fastest correction actions are obtained when degrading high priority, high load, short

```
While( 1 )
    If( Time violation )
        If( All components at Current_Max_Deg_Level )
            If( Highest level Supervisor )
                If( Current_Max_Deg_Level equals Max_Deg )
                    Reconfiguration
                Else
                    Increase Current_Max_Deg_Level
            Else
                Notify higher level
        Else
            Select component for degradation
```

Figure 6: Degradation Algorithm

```
While( 1 )
    If( Available resources and not Time violation )
        If( All components at Current_Min_Deg_Level )
            If( Current_Min_Deg_Level > 0 )
                Decrease Current_Min_Deg_Level
        Else
            Select component for promotion
```

Figure 7: Promotion Algorithm

period components. Actual selection criteria is defined by the component's designer according to its special characteristics.

Component selection for promotion is organized first by priority levels and then by degradation level.

4 Methodology of Use

The process of generating full working components is resumed on figure 8. This process starts with a description code for the component, whether atomic or compound, which is compiled to generate makefiles and C++ files implementing the component, which in the case of atomic components should be completed filling in the skeleton of C++ code generated by the compiler. Once a component has been completely implemented it can be integrated in other compound components or systems, by local or remote instantiation.

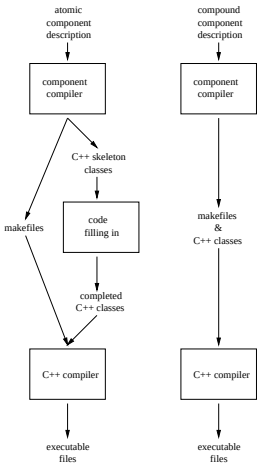

Figure 8: Generating components

5 Final Discussion

Along this paper it has been presented a component-oriented programming framework for robotics, **CoolBOT**, which is actually a research initiative under development at ULPGC. CoolBOT shouldn't be understood as a new architecture for perception-action systems but as an alternative design methodology and its associated set of development tools, that should assist the robotics researcher in the process of conceiving and validating different architecture proposals. We think it belongs to a group of recent proposals that are aimed at defining new languages (e.g. ESL [3] or TDL [**?**]) not new "architectures".

Components developed with CoolBOT will share the same communication abstractions and inner control organization making possible to monitor and debug any of these components using a standard set of tools. We expect that theses features will reveal as essentials to achieve reliable, modular and easy to extend systems. Besides, CoolBOT will provide mechanisms for adaptation to run-time available resources. Obviously, it is too premature to make any claims about the validity of the approach described in this paper over others addressing the same or similar goals. Only a posteriori, that is, through extensive experimentation and cross-validation it will be possible to validate CoolBOT's approach if the systems so built proved to be more reliable, extensible and easier to maintain or adapt.

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, 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.

[3] E. Gat. ESL: A language for supporting robust plan execution in embedded autonomous agents. Proc. of the AAAI Fall Symposium on Plan Execution, AAAI Press, 1996.

[4] D. Kortenkamp, R. P. Bonasso, and R. (Eds) Murphy. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, 1998.

[5] D. Kortenkamp and A. C. Schultz. Integrating robotics research. *Autonomous Robots*, 6:243–245, 1999.

[6] B. Pell, D. E. Bernard, S. A. Chien, E. Gat, N. Muscettola, P. Nayak, M.D. Wagner, and B.C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robots*, 5:29–52, 1998.

[7] 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.

[8] 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(12):759–776, December 1997.

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