

Integrating Robotics Software*

Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, Josep Isern-González, and Jorge Cabrera-Gómez
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 Taura, 350017, Las Palmas de Gran Canaria, Spain
adominguez@iusiani.ulpgc.es, dhernandez@iusiani.ulpgc.es, jiserndfm@ulpgc.es and jcabrera@dis.ulpgc.es

Abstract—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 efforts, 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. In this work we present CoolBOT, a component oriented framework for programming robotic systems, based on the Port Automata model that fosters controllability and observability of software components. A simple demonstrator outlines the benefits of using the proposed approach in the development of a robotic application.

Index Terms—robotic systems, software components, system integration, code reuse, cognizant failures

I. INTRODUCTION

At present, there are no clear programming paradigms for programming robotic systems, and according to some authors [1] the programming techniques which are of common use today are not adequate to deal with the complexity associated with these systems. One aspect where this complexity clearly comes up is software integration. In a given system normally it is necessary to integrate a wide variety of software: software dealing with hardware (sensors, effectors, other hardware), third-party software, software which is not very portable because it is specific to a particular operating system or machine (or both), software done in distinct programming languages, etc. 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. Nowadays it is not only necessary to develop complex algorithms, but also to integrate complex systems that really perform adequately to its own capacities.

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 [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. 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 [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 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 III their main concepts and abstractions will be briefly explained. Next, in section IV a simple demonstrator is commented in some detail, and finally, in section V we will comment some of the conclusions we have drawn from this work.

II. COOLBOT

Latest trends in Software Engineering are exploiting the idea of software components as the basic units of development and deployment when building complex software systems, specially if software reuse, modular composition and third-party software integration are important issues. Out of the multiple definitions for software components we can find (chapter 11 in [5] is a good summary) we have chosen the following definition given in [5] (page 164):

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

Thus, a software component is a piece of software that has been independently developed from where it is going to be used. It should offer a well-defined external interface that hides

*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.

its internals, and it is independent of context until instantiation time. In addition, it can be deployed without modifications by third parties. Also, it needs to come with a clear specification of what it requires and provides in order to be able to be composed with other components.

In this document it will be briefly introduced a component-oriented framework for programming robotic systems called **CoolBOT** [6] that allows designing systems in terms of composition and integration of software components. The framework provides means to design and build components and to compose and integrate them hierarchically and dynamically. The concept of component-oriented framework is illustrated in Fig. 1.

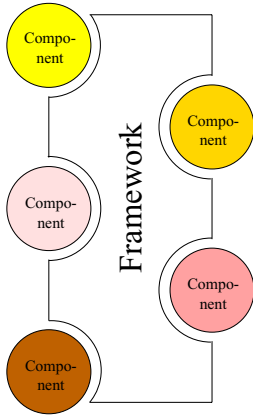


Fig. 1: A component-oriented framework.

All in all, CoolBOT is a component-oriented C++ framework where the software that controls a system is viewed as a dynamic network of units of execution inter-connected by means of data paths. Each one of these units of execution is a *software component* 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. The framework provides the infrastructure necessary to support this concept of component software

and the inter communication between them by means of data paths which can be established and de-established dynamically.

III. COOLBOT FUNDAMENTALS

A. Components as Port Automata

In CoolBOT, components are modelled as **Port Automata** [7][8][9]. This concept establishes a clear distinction between the internal functionality of an active entity, the automaton, and its external interface, its sets of input and output ports. Components define active entities which carry out 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. Fig. 2 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. As shown by the figure, the external interface keeps the component's internals hidden. Fig. 3 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 a port, or by an internal condition, or by a combination of port incoming data and internal conditions. Double circles indicate automaton states.

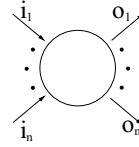


Fig. 2: Component external view.

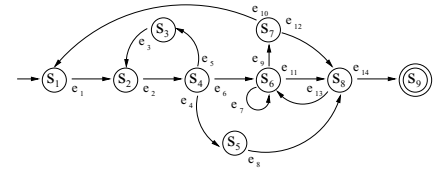


Fig. 3: Component internal view.

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 defined as discrete units of information which can be received through input ports, and/or issued through output ports. *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. To establish a *port connection* both input and output ports must be compatible, i.e. both ports must match exactly the type of port packets they can accept.

B. Observability and Controllability

Components should be observable enough to know whether they are working correctly or not, and in that case, they should be controllable enough to make some adjustment in their internal behavior to regulate and adjust their operation. CoolBOT introduces two kinds of variables as facilities in order to support monitoring and control of components.

- **Observable variables:** Represent features of components that should be of interest from outside, they are externally observable and permit publishing aspects of components which are meaningful in terms of control, or just for observability and monitoring purposes.
- **Controllable variables:** Represent aspects of components which can be externally controlled, i.e., modified or updated. Thence, through them the internal behavior of a component can be controlled.

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. 4.

- The **monitoring** port: This is a public output port by means of which component *observable variables* are published. Therefore, through this port, an external observer or supervisor can observe and monitor a component.
- The **control** port: This is a public input port through which component *controllable variables* are modified and updated, consequently an external controller or supervisor can control a component by means of this port.

CoolBOT provides components with several default observable and controllable variables, in addition to the observable and controllable variables specific to each component. A brief description of these default observable and controllable variables are shown in tables I and II (the symbols representing each variable are given beside variable names in parenthesis).

TABLE I: Default observable variables.

Default Observable Variables	
Name	Brief Description
<i>state (s)</i>	Automaton state where the component is situated.
<i>priority (p)</i>	Current component execution priority.
<i>con g (c)</i>	Requests a supervised con guration change, or con rms con guration commands.
<i>result (r)</i>	Result of execution.
<i>error description (ed)</i>	Error description indicating a locally unrecoverable exception.

Fig. 5 illustrates graphically a typical execution control loop for a component using another component as external supervisor. This is possible thanks to the default *control* and *monitoring* ports CoolBOT imposes on all components.

TABLE II: Default controllable variables.

Default Controllable Variables	
Name	Brief Description
<i>new state (ns)</i>	Desired automaton state where the component is commanded to go.
<i>new priority (np)</i>	Desired execution priority the component is commanded to have.
<i>new exception (nex)</i>	Externally induced exception.
<i>new con g (nc)</i>	Component's con guration can be modified and updated during execution through this controllable variable.

C. Default Automaton

Internally all components are modelled using the same default state automaton, the *default automaton*, shown in Fig. 6, 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 the labels corresponds to internal events: *ok*, *exception*, *attempt*, *last attempt* and *nish*. The remaining labels indicate events provoked by default controllable variable changes: ns_r , ns_{re} , ns_s , ns_d , np , and nex (see table II for the symbols). Subscripts in ns_i indicate which state has been commanded, where subscript i can be any of the followings: r (**running** state), re (**ready** state), s (**suspended** state), and d (**dead** state).

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.

The **running** state, the dashed state in Fig. 6, constitutes or represents 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. The initial state of a *user automaton* constitutes its *entry state*.

Having a look to Fig. 6 we can see how CoolBOT components evolve along their execution time, since they are launched until they finish their execution. Basically, the *default automaton* organize the life of a component in several phases which correspond to different states:

- **starting**: This state is devised to allocate resources for a correct task execution.
- **ready**: In this state the component is ready to execute, and waits for an external command to start (ns_r).
- **running**: In this pseudo-state the component is carrying out its specific task running its *user automaton*.
- **suspended**: In this state the component is suspended and remains idle until ordered to transit to other state.
- **end**: In this state the component has just finished a task execution. Getting to this state the component publishes the result of its execution, if any, through its *monitoring* port.

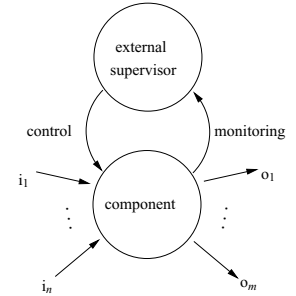


Fig. 5: A typical component control loop.

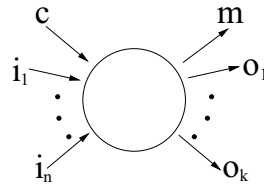


Fig. 4: Control and monitoring ports.

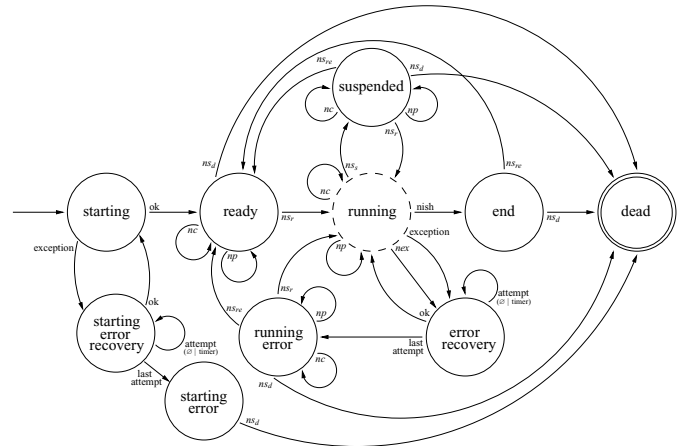


Fig. 6: The Default Automaton.

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 though 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 that we will explain with more detail in next section.

D. Exception Handling

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, where each component exception is defined using the following pattern:

```
On Exception: <exceptionId>
  <description >
  [<handler> [<retries > <period >]]
  [<onSuccessHandler >]
  [<onFailureHandler >]
```

where $\langle \text{exceptionId} \rangle$ is a number identifying the exception; $\langle \text{description} \rangle$ is a description of the exception; $\langle \text{handler} \rangle$ is an optional handler to try a recovery procedure, optionally $\langle \text{retries} \rangle$ indicates the number of recovery attempts, and $\langle \text{period} \rangle$ specifies the period in milliseconds between attempts; $\langle \text{onSuccessHandler} \rangle$ is an optional handler to be executed in case of a successful recovery; and finally, $\langle \text{onFailureHandler} \rangle$ is also an optional handler which is executed when all recovery exception attempts have failed. As commented previously, CoolBOT provides in the default automaton (see Fig. 6) two pairs of states to manage exceptions during resource allocation and task execution. Thus if an exception comes up during runtime in a component, the component is driven to one of the recovery error states (**starting error recovery** or **error recovery**) where the handler for the exception is tried several times. Only when all recovery attempts have been unsuccessful the component is took to a situation of unrecovered exception (**starting error** or **running error**). In this situation, an error description is published through the monitoring port of the component, and it remains idle waiting for external supervision.

E. Multithreading

CoolBOT components are *weakly coupled entities* that execute concurrently or in parallel, on their own initiative, in order to achieve their own independent objectives. Thence, 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, which are the two operating systems where CoolBOT is supported in its current version.

At runtime a CoolBOT component executes a continuous loop processing port packets where the component carries out different actions depending on which input port has received each port packet, and in which state of its automaton the component is. This loop, illustrated on Fig. 7, can be multithreaded

or not. 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, its possible to distribute the attention of a component on different input ports using different threads of execution. These threads are called *port threads*, and they can only execute port transitions. They are responsible for disjoint sets of input ports, and are observed and controlled by the *main thread* of the component.

F. Inter Component Communications

Analogously to modern operating systems that provide IPC (**I**nter **P**rocess **C**ommunications) 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. 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.

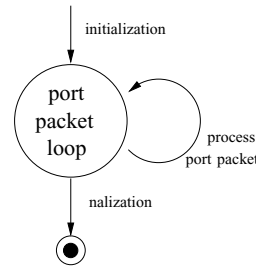


Fig. 7: A running component.

CoolBOT components inter communicate by means of *port connections* formed by *output ports* and *input ports*. Fig. 8 shows all the different types of output and input ports supported by CoolBOT (on the right and on the left respectively), and all the possible combinations between them to form port connections. Below the arrows, the cardinality of each type of port connection is also indicated. Remember that a *port connection* between an output port and an input port is only possible whether both ports match the type of port packets they accept, besides, it is necessary that they also constitute a compatible pair of output and input ports.

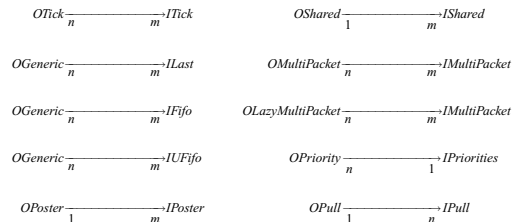


Fig. 8: Port connections ($n, m \in \mathbb{N}; n, m \geq 1$).

Each type of port connection implements a different model of interaction between components, table III resumes the model implemented by each one.

TABLE III: Port connections.

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

In CoolBOT there are two basic communication models for port connections:

- *Push Model*. In a push connection the initiative for sending a port packet relies on the output port part; that is, the data producer (the sender) sends port packets on its own, completely uncoupled from its consumers (the receivers).
- *Pull Model*. A pull connection implies that packets are emitted when the input part of the communication, the consumer (the receiver), demands new data to process. In this model the consumer keeps the initiative, sending a request to the producer (the sender) whenever a new port packet is demanded.

All types of port connections in table III observe a push communication model that allows for uncoupled and asynchronous interactions among components. The only one type of port connection that utilizes a pull communication model is the *pull connection* (an *OPull/IPull* pair of ports).

Additionally to the connections shown in Fig. 8 there are other possible pairs of port connections which are collectively called *single multi packet* connections. All the possibilities appear in Fig. 9. Their main characteristic is that they combine a type of output or input port that accepts only one type of port packet (a *single packet* port) with a type of input or output port that accepts several types of port packets (a *multi packet* port).

G. Atomic and Compound Components

CoolBOT components are classified into two kinds: *atomic* and *compound* components. *Atomic* components have been mainly devised to be used: 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. All that, in order to make them isolated pieces of deployable software in the form of CoolBOT

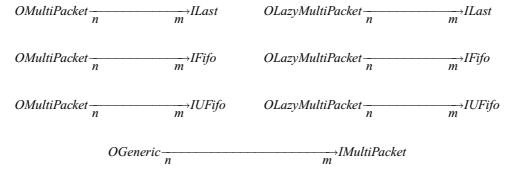


Fig. 9: Simple multi packet connections ($n, m \in \mathbb{N}; n, m \geq 1$).

components. Thanks to the uniformity of external interface and internal structure that CoolBOT imposes on components, CoolBOT components may be used as building blocks that hide their internals behind a public external interface. A *compound* component, is a composition of instances of several components which can be either atomic or compound. *Compound* components use the functionality of instances of another atomic or compound components to implement its own functionality, and in turn, can be integrated and composed hierarchically with other components to form new *compound* components.

IV. 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 shown to illustrate how such principles manifest in systems built using CoolBOT. The example has been inspired from [10] (chap. 4, pag. 107) in order to use a well-know example as a reference, and has been implemented and tested using a Pioneer 2DX robot from Activ Media Robotics. The objective of this section is to illustrate how a mobile robot can be endowed with different capabilities by means of integrating components in an incremental way using CoolBOT. Thus, initially the robot will only be able to avoid obstacles moving away from them, and nally it will end up with a "wanderer" behavior with obstacle avoidance.

The rst level of this simple demonstrator is shown in Fig. 10 and it consists of two components: the *Pioneer* and the *PF Avoiding* components. The *Pioneer* component is a component that encapsulates the set of sensors and effectors provided by an Activ Media Robotics Pioneer robot. The *PF Avoiding* component makes use of a potential

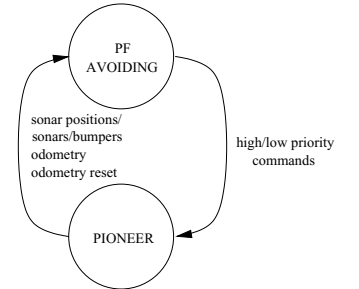


Fig. 10: The avoiding level.

field approach (using repulsive potential fields) to perform obstacle avoidance using robot sonar readings as sensory information. Thus, when this component configuration is executed in our robot the observable behavior is that the robot remains still if there are no obstacles close enough. In the event that an obstacle gets closer, the robot will move away from it.

Consequently, this behavior could allow a person to herd the robot, as the robot behaves in order to keep itself far enough from obstacles.

Fig. 11 shows the second and last level of our demonstrator which adds two new components on top of the avoiding level presented in Fig. 10: the *Strategic PF* and *Wander* components. The *Strategic PF* is a component that uses also, like the *PF Avoiding*, a potential field approach to implement several behaviors, namely: a “move” behavior using an uniform potential field, a “goto” behavior making use of an attractive potential field, and a “docking” behavior utilizing a “docking” potential field [10]. This component periodically feeds the *PF Avoiding* with strategic potential field vectors. The *PF Avoiding* adds these strategic vectors to the repulsive potential field vectors originated by the different obstacles detected by the sonar sensors, leading the robot accordingly. As a result our robot can perform several behaviors doing obstacle avoidance simultaneously. Finally, there is a *Wander* component which periodically makes the *Strategic PF* move the component randomly in a specific direction using the “move” behavior. The observable behavior of this configuration of components is a wanderer robot that avoids obstacles.

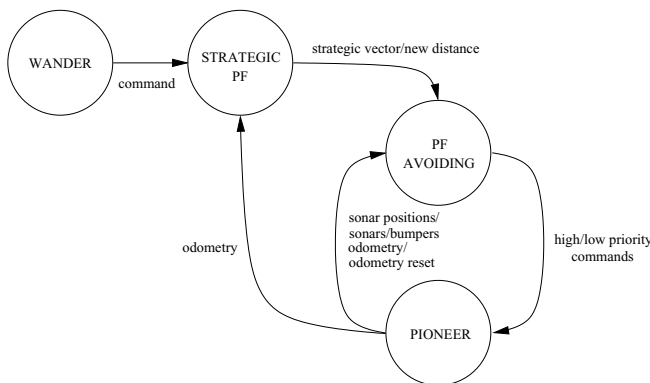


Fig. 11: The wandering level.

Using CoolBOT it is possible to inject externally by means of the component’s *monitoring* port any exception that has been defined for that component, using the *new exception* controllable variable (see table II). There are several faulty situations we have mapped as exceptions in the components of Fig. 10 and 11. The most significant ones are: the lost of connection with the physical robot (defined in the *Pioneer* component); the stuck of the robot due to the activation of the robot’s front and rear bumpers simultaneously (defined in the *PF Avoiding* component); and the forcing of only the avoiding obstacle behavior in the *PF Avoiding* component due to that no strategic vectors are received from the *Strategic PF*. In the systems shown in Fig. 10 and 11 this feature has been used in order to test the robustness of each component, and the robustness of the whole integration of components facing the occurrence of the different exceptions defined in the system.

V. CONCLUSIONS

This document describes briefly 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 concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing.

Certain level of uniformity in structure in software components is critical to allow a basic uniform treatment of components in spite of their individual functionality. CoolBOT imposes some uniformity on the units of execution it defines in order to make them components externally observable and controllable, and treatable by the framework in an uniform and consistent way.

SmartSoft [4] and $G^{en}oM$ [3] are systems that have been proposed with a similar scope than CoolBOT. SmartSoft is a framework that mainly provides patterns for communications, but treats components as opaque processes, not imposing any common internal structure. On the other hand, $G^{en}oM$ focuses on a modular organization in which components are controlled according to the same state automaton and share the same internal architecture. However $G^{en}oM$ only provides with poster-like communications facilities. CoolBOT owes a good deal of inspiration to these systems, and in fact, combines what we think are the most interesting aspects of both, providing a rich set of communications patterns and imposing a common internal structure for components.

REFERENCES

- [1] D. Kortenkamp and A. C. Schultz, “Integrating robotics research,” *Autonomous Robots*, vol. 6, pp. 243–245, 1999.
- [2] 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.
- [3] 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/eury97genom.html
- [4] C. Schlegel and R. Wörz, “Interfacing Different Layers of a Multi-layer Architecture for Sensorimotor Systems using the Object Oriented Framework SmartSoft,” Third European Workshop on Advanced Mobile Robots - Eurobot’99. Zürich, Switzerland, September 1999.
- [5] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.
- [6] 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.
- [7] 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.
- [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*, vol. 23, no. 12, pp. 759–776, December 1997.
- [9] 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.
- [10] R. R. Murphy, *Introduction to AI Robotics*. The MIT Press, 2000.