

Integrating Systems in Robotics

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

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. This document presents CoolBOT, a component oriented framework for programming robotic systems, based on a port automata model that fosters controllability and observability of software components. A simple demonstrator illustrates the benefits of using the proposed approach.

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.

All authors are with the 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, 35017, Las Palmas, Spain.

Emails: jfernandez@iusiani.ulpgc.es, adominguez@iusiani.ulpgc.es, dherandez@iusiani.ulpgc.es, and jcabrera@dis.ulpgc.es

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.

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. We have taken for software components the approach given in [5], where 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 its internals, and it

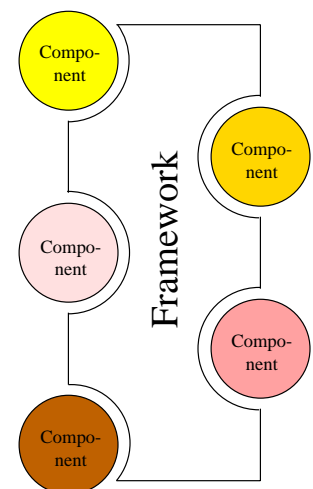


Fig. 1. A component-oriented framework.

is independent of context until instantiation time. In addition, it can be deployed without modifications by third parties. Furthermore, 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 we will introduce briefly a component-oriented framework for programming robotic systems called **CoolBOT** [6] that allows designing systems in terms of composition and integration of software components. CoolBOT is a C++ framework that provides means to design and build components and to compose and integrate them hierarchically and dynamically. Fig. 1 helps to illustrate this concept of component-oriented framework. In CoolBOT 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. Finally, CoolBOT also provides means to make a system and its components able to be externally controlled and observed.

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 final states.

CoolBOT components interact and inter communicate each other by means of *port connections* established among their input and output ports. Data are transmitted through port connections in discrete units called *port packets*. *Port packets* are defined as discrete units of information which can be received through input ports, and/or issued through output

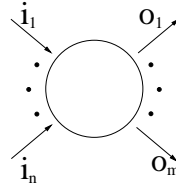


Fig. 2. Component external view.

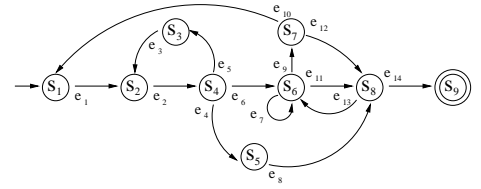


Fig. 3. Component internal view.

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).

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 I. DEFAULT OBSERVABLE VARIABLES

NAME	BRIEF DESCRIPTION
state (<i>s</i>)	Automaton state where the component is situated.
priority (<i>p</i>)	Current component execution priority.
config (<i>c</i>)	Requests a supervised configuration change, or confirms configuration commands.
result (<i>r</i>)	Result of execution.
error description (<i>ed</i>)	Error description indicating a locally unrecoverable exception.

TABLE II. DEFAULT CONTROLLABLE VARIABLES

NAME	BRIEF DESCRIPTION
new state (<i>ns</i>)	Desired automaton state where the component is commanded to go.
new priority (<i>np</i>)	Desired execution priority the component is commanded to have.
new exception (<i>nex</i>)	Externally induced exception.
new config (<i>nc</i>)	Component's configuration 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 *finish*. The remaining labels indicate events provoked by default controllable variable changes: *ns_r*, *ns_{re}*, *ns_s*, *ns_d*, *np*, and *nex* (see tables I and II). 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.

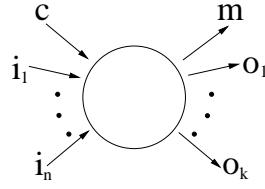


Fig. 4. Control and monitoring ports.

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.

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.

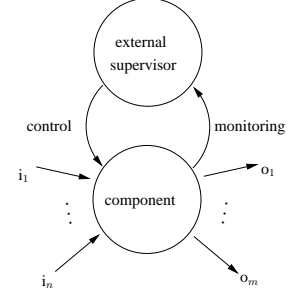


Fig. 5. A typical component control loop.

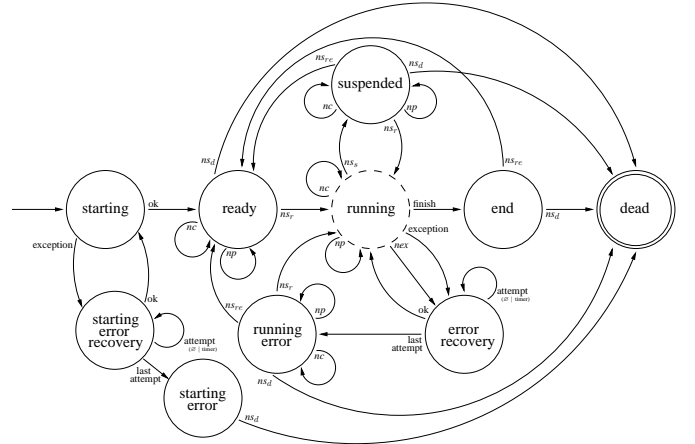


Fig. 6. The Default Automaton.

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. In particular, in CoolBOT we may distinguish two levels of exception handling for which the framework provides facilities:

- **Local Exception Handling**: As a good rule of design any component must deal with any exception first at a local level, running its associated handler in the way the *default automaton* establish (figure 6). If, as a result of the whole process of recovering from a exception, the exception persists, the component gets into **starting error**

or **running error** states, and remains there waiting for external intervention.

- **External Exception Handling:** Errors arriving to an external supervisor in a control loop (Fig. 5) from any of its *local* components must be managed first by this external supervisor. In turn those errors can be either ignored, or propagated to another external supervisor situated in a control loop of higher hierarchy in the system.

E. Inter Component Communications

Analogously to modern operating systems that provide IPC (Inter Process Communications) mechanisms to inter communicate processes, CoolBOT provides *Inter Component Communications* or *ICC* mechanisms to allow components to interact and communicate among them. CoolBOT *ICC* mechanisms are carried out by means of input ports, output ports, and ports connections. 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.

CoolBOT components inter communicate by means of *port connections* formed by *output ports* and *input ports*. Fig. 7 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.

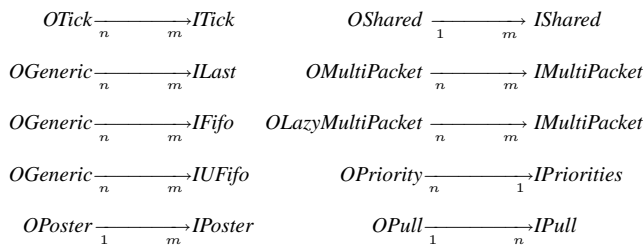


Fig. 7. Port connections ($n, m \in \mathbb{N}; n, m \geq 1$).

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

In CoolBOT there are two basic communication models for port connections. A **push model**, where the initiative for sending a port packet relies on the output port part, the data producer (the sender) sends port packets on its own, completely uncoupled from its consumers (the receivers). And a **pull model**, where 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,

TABLE III. PORT CONNECTIONS

OUTPUT PORT	INPUT PORT	BRIEF DESCRIPTION
<i>OTick</i>	<i>ITick</i>	Implement a protocol to signal events between components (<i>tick connections</i>).
<i>OGeneric</i>	<i>ILast</i> <i>IFifo</i> <i>IUFifo</i>	There is a queue (fifo) of port packets in the input port (<i>last, fifo and unbounded fifo connections</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. Implement a protocol of shared memory (<i>shared connections</i>).
<i>OMultiPacket</i> <i>OLazyMultiPacket</i>	<i>IMultiPacket</i>	Accept multiple type of port packets through the same connection (<i>multi packet connections</i>).
<i>OPriority</i>	<i>IPriorities</i>	Implement a protocol of sending with priority (<i>priority connections</i>).
<i>OPull</i>	<i>IPull</i>	Implement a protocol of request/answer between components (<i>pull connections</i>).

sending a request to the producer (the sender) whenever a new port packet is demanded.

All in all, CoolBOT provides a wide and rich set of output and input ports for inter component communications that frees developers of an important workload in software design and development, what implies a less error-prone system programming. In addition, the typology of different ports offers a number of communications patterns wide enough to allow a broad variety of component interactions.

IV. A SIMPLE DEMONSTRATOR

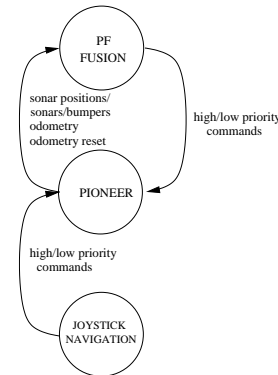


Fig. 8. The avoiding level.

CoolBOT has been conceived to promote integrability, incremental design and robustness of software developments in robotics. In this section, a simple demonstrator will be outlined to illustrate how such principles manifest in systems built using CoolBOT. The example has been implemented and tested using a Pioneer 2DX robot from ActivMedia Robotics and a Sick laser range finder, and it shows how a mobile robot can be endowed, using CoolBOT, with different capabilities by means of integrating components in an incremental way. Initially the robot will be just able to avoid obstacles moving away from them. And finally it will end up with the ability of doing SLAM (Simultaneous Localization and Mapping) [10] [11].

The first level of this simple demonstrator is shown in Fig. 8 and it made up of three components: the *Pioneer* and the *PF Fusion* components. The *Pioneer* component is a component that encapsulates the set of sensors and effectors provided by an ActivMedia Robotics Pioneer robot. The *PF Fusion* component which is a potential field fuser. It is possible to configure several potential fields in this component to make it do several navigation tasks, for example, to go to an specific point, or to get to a docking point following some approaching directions. In the integration shown in Fig. 8 it has been configured to avoid obstacles while going to a destination point.

Another component also shown in Fig. 8 is used for moving the platform by means of a joystick device. When this component is in **running** state to the *Pioneer* component, we command directly the robot movements using the joystick. When this component is used the *PF Fusion* component is kept in **suspended** state. This is because only one component in the system can control the robot directly..

The second and last level of our demonstrator is depicted in Fig. 9. Note that the systems adds two new components, the *Sick Laser* which controls a Sick laser range finder and *Scan Alignment* that performs self-localization using a SLAM (Simultaneous Localization And Mapping) algorithm [10][11].

The *Sick Laser* component works in a continous loop, periodically reading the laser device and publishing the data by means of a poster output port readable. This component receives

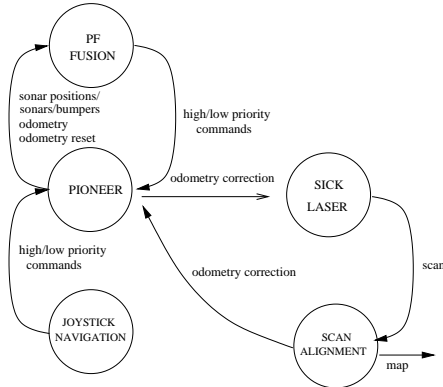


Fig. 9. The whole system.

odometry packets from the *Pioneer* component in order to be able to stamp every scan with the robot pose at the time the scan was taken. The *Scan Alignment* component communicates with the *Pioneer* and *Sick Laser* components, it reads scans and performs a correction which is sent to the *pioneer* component which in turn will correct the pose and its associated covariance. The component does this by means of a simultaneous localization and mapping algorithm which produces a map of the environment [10][11]. This map is also published in an output port to be used by any other component. So, now, the system allows a robot to make self-localization and map building, at the same time that it avoids obstacles.

A. Control Interface

In CoolBOT, as we have mentioned in section III-C every component presents a uniformity of external interface and internal structure. This is suitable for implementing a control system that allow us to connect to several components and control them. Such a system could be able to control and observe the behavior of a component by means of its *control* and *monitoring* default ports. At the same time it could be able to “sniff” port connections between components. This is very helpful when trying to debug such systems or to check its normal operation.

For the system integration of Fig. 9 was designed a graphical front-end to control and observe the system during operation, a snapshot appears in Fig. 10. As we can see the figure shows the map, the current scan and the result of the corresponding points and tangent calculations, on the left side we can see

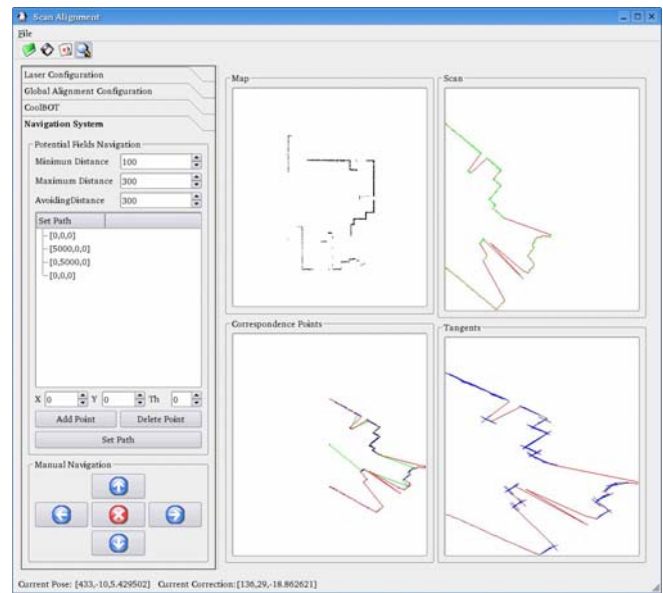


Fig. 10. The control interface.

part of the controlling interface which allow us to change the state of every component, and connects to different ports of the system to have a look to the different information (robot odometry, odometry corrections underdone, laser scans, the map as it gets built, etc.). Moreover, it is possible to change some operational *controllable* variables (frequency of laser scans, *Scan Alignment* component parameters, etc.). Additionally, it is possible to establish different paths the robot should follow during map building. Besides, as we have commented previously it is also possible to control the robot movements directly with a joystick device.

B. Test and Results

Robustness is a key important aspect in robotics, and CoolBOT has aimed some of its infrastructure to facilitate robust system integration. As every component works in an autonomous and asynchronous fashion, the system does not have to stop running whenever any of its components enters into an error unrecoverable state, on the contrary, it can keep working with part of its functionality (if possible), or waiting for the faulty component to restart its normal execution. Focusing on the robustness of the system we prepared several tests that show the behavior of the the system previously presented upon the malfunction of any of its components.

In the system of Fig. 9, the most important components are: the *Pioneer*, *Sick Laser* and the *Scan Alignment* components. Based on these three components, two tests were made. The first of them shows the way the systems works whenever any of the components hangs, and second one is related to the degradation of the system when the *Sick Laser* component stops running:

- **First Test:** The main goal is to overcome a situation provoked by a faulty component without collapsing. It should be able of keeping operation, either running, or waiting for the faulty component to restore the situation. In this

test the system did not collapse when any combination of its component was in a non-recoverable error state. Table IV enumerates every possible combination of error states in any component. In the three first columns *R* stands for *Running* (the **running** pseudo state of Fig. 6) and *E* for *Error* (the **running error** state in Fig. 6). Any of the errors shown in Table IV does not lead the system to a fatal situation, except if we consider the last case. The asynchronicity of execution, initiative and communications between the components which form the systems allows it to keep the system working until the faulty one is recovered, or the whole system is explicitly stopped.

TABLE IV. ERROR STATES

PIONEER	SICK		DESCRIPTION
	LASER	SCAN ALIGNMENT	
<i>R</i>	<i>R</i>	<i>R</i>	Normal state, everything is right.
<i>E</i>	<i>R</i>	<i>R</i>	The <i>Pioneer</i> component does not work. The <i>Sick Laser</i> component publishes every scan with the last received pose. Notice that if the <i>Pioneer</i> component does not run, the robot also stops because of a watchdog event in such a way that the scans are published with the actual pose of the robot. The <i>Scan Alignment</i> component keep reducing the error of the map built so far.
<i>R</i>	<i>E</i>	<i>R</i>	The <i>Scan Alignment</i> component reduces errors in the current map, this is done by means of an iterative process to achieve consistent maps. But it stops integrating new scans to the map as the <i>Sick Laser</i> component is not working. In this state the robot pose degrades (explained in more detail in the second test).
<i>E</i>	<i>E</i>	<i>R</i>	The <i>Scan Alignment</i> component reduces errors of the current map built so far, but it does not add new scans to the map.
<i>R</i>	<i>R</i>	<i>E</i>	In this case the <i>Scan Alignment</i> component does not work. This leads to a degradation of the pose of the robot, but the rest of the system still works. This case is also the basis of the second test.
<i>E</i>	<i>R</i>	<i>E</i>	The <i>Sick Laser</i> component publishes scans with the last robot pose received.
<i>R</i>	<i>E</i>	<i>E</i>	The odometry of the robot degrades.
<i>E</i>	<i>E</i>	<i>E</i>	The system here does not collapse itself, but it can do nothing.

- **Second test:** In Table IV there are some situations during system operation that lead to odometry degradation. If the *Sick Laser* or the *Scan Alignment* component gets into erroneous operation, the SLAM algorithm responsible for the pose error, will not work. In this situation, the *Pioneer* component will keep working but as the robot moves, its pose will degrade. For showing this degradation and its subsequent recovering we have made a real experiment where the *Sick Laser* is provided with a mechanism for reinitializing the connection with the laser device upon abnormal disconnection. Thus, for example if the serial connection to the laser is physically broken the component will keep trying to recover communications with the device by restarting the connection periodically. So, if the serial connection is repaired, the component will restart its normal operation. Once the laser is working again, the *Scan Alignment* component will receive again new scans and will produce new corrections which, in turn, will be sent to the *Pioneer* component, finishing in this way, odometry degradation.

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

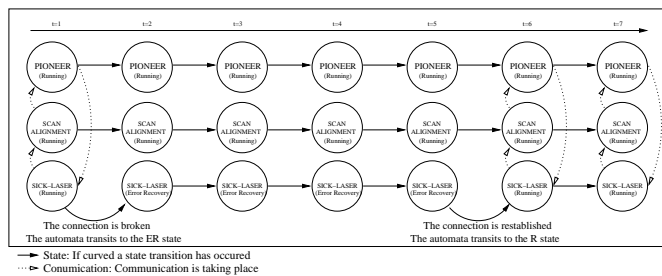


Fig. 11. Pose degradation.

connections. CoolBOT is a tool that favors a programming methodology that fosters software integration, concurrency and parallelism, asynchronous execution, asynchronous inter communication and data-flow-driven processing.

CoolBOT imposes some uniformity on the units of execution it defines, CoolBOT components. This uniformity makes components externally observable and controllable, and treatable by the framework in a uniform and consistent way. CoolBOT also promotes a uniform approach for handling faulty situations, establishing a local and an external level of exception handling. Exceptions are first handled locally in the components where the exceptions come out. If they can not be handled at this local level, they are deferred to an external supervisor, in turn, this handling may scale going up in a hierarchy of control loops.

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/fleury97genom.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] F. Lu and E. Milios, "Robot pose estimation in unknown environments by matching 2d range scans," Proc. IEEE Comp. Soc. Conf. on Computer Vision and Pattern Recognition, Seattle, USA, 1994.
- [11] —, "Globally consistent range scan alignment for environment mapping," *Autonomous Robots*, vol. 4, no. 4, pp. 333–349, 1997.