# EXPLORING INTERFACES IN A DISTRIBUTED COMPONENT-BASED PROGRAMMING FRAMEWORK FOR ROBOTICS*

A. C. Domínguez-Brito, F. J. Santana-Jorge, J. Cabrera-Gámez, J. D. Hernández-Sosa, J. Isern-González and E. Fernández-Perdomo

*Instituto Universitario SIANI, Dpto. de Informática y Sistemas*
*Universidad de Las Palmas de Gran Canaria, Las Palmas de Gran Canaria, Spain*

Keywords: Robot programming environments and languages, Component-based software engineering.

Abstract: CoolBOT is a C++ distributed component-based programming framework for robotics. A system can be seen as a distributed network of software components interconnected by port connections where system behavior emerges from the interaction and independent execution of the components integrating the system. Recently we have endowed CoolBOT with two new types of software components: *views* and *probes*. On one side, in order to separate and decouple robot control from graphical displays, we have introduced the concept of *view* as an integrable, composite and reusable graphical interface available for CoolBOT system integrators and developers. On the other side, *probes* have been devised as interfaces for interoperability with non CoolBOT software.

## 1 INTRODUCTION

Some of the significant approaches within the robotics community based on the CBSE (Component Based Software Engineering) (George T. Heineman and William T. Councill, 2001) paradigm are G$^{en}$oM/BIP from LAAS Openrobots project (Mallet et al., 2010)(Basu et al., 2006)(Bensalem et al., 2009)(Openrobots, 2011), Smartsoft (Schlegel et al., 2009)(SmartSoft, 2011), OROCOS (Orocos, 2011), ORCA (Brooks et al., 2005), OpenRTM-aist (Ando et al., 2008) and ROS (ROS, 2011).

In this paper we want initially to focus our attention on system graphical interfaces which constitute a specific kind of software component on their own, that we normally find in robotic CBSE developing software approaches. Thence, in ROS (ROS, 2011) we can find the following tools: `rviz` for 3D visualization, `rxbag` for showing "off-line" sensory data collected during runtime, `rxplot` for scalar data visualization in runtime and `rxgraph` to show a system graph also in runtime. Similarly, in ORCA (Orca, 2011) the following utilities are available: `RegistryView` to show a graphical representation of the structure of a component, and `OrcaView2d` and `OrcaView3d` which are respectively 2D and 3D graphical interfaces for sensory data. Equally the Openrobots project (Openrobots, 2011) provides the powerful tool `gdhe` for system 3D visualization.

Recently we have made freely available a distributed CBSE C++ framework for programming robotic systems developed at our laboratory, as an open source initiative we have called The CoolBOT Project (Domínguez-Brito et al., 2007; CoolBOT, 2011). In CoolBOT, graphical interfaces are software components on their own called *views*. On the other side and considering the problem of interoperability of CoolBOT-based systems with non-CoolBOT software, we have also introduced the concept of *probe*.

The rest of this document is organized as follows. In section 2 we will introduce briefly an overview about CoolBOT. Next, we will pass to focus on the support for views and probes provided by the framework respectively in sections 3 and 4. Finally, last section is devoted to present briefly the conclusions we have drawn from the work presented in this paper.

*This work has been partially supported by: research project *TIN2008-06068* funded by the Ministerio de Innovación y Ciencia, Gobierno de España, Spain; research project *ProID20100062* funded by the ACIISI, Gobierno de Canarias, Spain; and by European Union's FEDER funds.



C. Domínguez-Brito A., J. Santana-Jorge F., Cabrera-Gámez J., D. Hernández-Sosa J., Isern-González J. and Fernández-Perdomo E. (2012).
EXPLORING INTERFACES IN A DISTRIBUTED COMPONENT-BASED PROGRAMMING FRAMEWORK FOR ROBOTICS.
In *Proceedings of the 4th International Conference on Agents and Artificial Intelligence*, pages 667-672
DOI: 10.5220/0003881706670672

## 2 CoolBOT OVERVIEW

CoolBOT (Domínguez-Brito et al., 2007) is a C++ distributed component oriented programming framework aimed to robotics developed at our laboratory which follows the CBSE paradigm for software development.

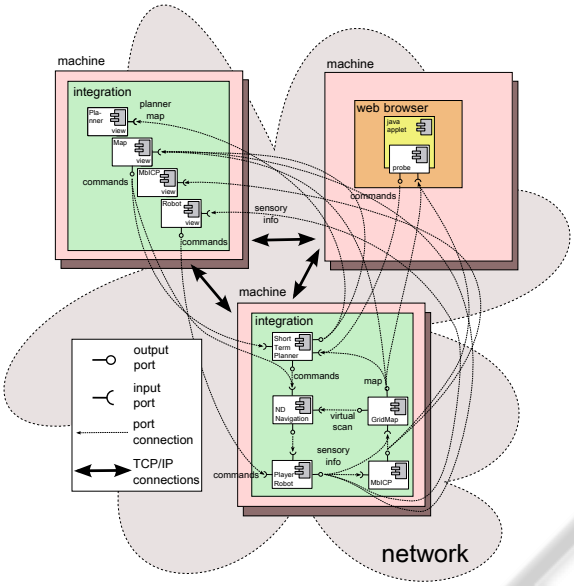


Figure 1: CoolBOT. Secure Navigation System.

Fig. 1 gives a global view of a a real system developed using CoolBOT. The example shows a secure navigation system for an indoor mobile robot based on the ND+ algorithm for obstacle avoidance (Minguez et al., 2004). It has been implemented attending to (Montesano et al., 2006).

The system is organized using two CoolBOT integrations, one only formed by CoolBOT component instances, and the other one containing CoolBOT view instances. The former one is really the integration which implements the navigation system. As we can observe, it consists of five component instances, namely: `PlayerRobot` (this is a wrapper component for hardware abstraction using the Player/Stage project framework (Vaughan et al., 2003)), `MbICP` (this is a component which implements the MbICP laser-based scan matching algorithm (Minguez et al., 2006)), `GridMap` (this component maintains a grid map of the surroundings of the robot built using robot range laser scans, it also generates periodically a $360^o$ virtual scan for the ND+ algorithm), `NDNavigation` (implements the ND+ algorithm) and `ShortTermPlanner` (a planner which uses the grid map for planning paths in the robot surroundings). On other machine another integration is shown hosting four view instances through which we can control and monitor the system remotely. In addition, in another machine, a web browser hosting a Java applet using a CoolBOT *probe* to connect to some of the components of the system.

As we can observe in Fig. 1, CoolBOT provides means for distributed computation. A CoolBOT *integration* is an application (a process) which integrates instances of CoolBOT components and *views*. As mentioned previously, port connections are multiplexed using TCP/IP connections between software components in different integrations, but in the same integration, port connections are supported using thread synchronization resources (locks, conditional variables, mutexes and shared memory). Integrations can be instantiated in any machine and port connections can be established dynamically at any moment, whether local or remote.

In general, in CoolBOT we can find three types of software components in terms of integration, deployment and composition: *components*, *views* and *probes*. All these three types are software components in the whole sense, since we can compose them indistinctly and arbitrarily without changing their implementation to build up a given system. Fig. 2 depicts the software abstraction layers involved when using CoolBOT.

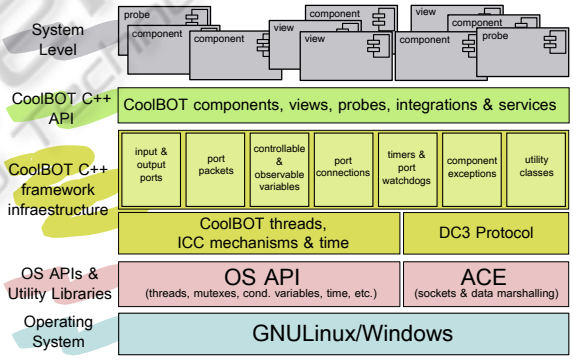


Figure 2: Abstraction layers in CoolBOT.

CoolBOT *Components* have been devised to implement any kind of functionality. When connected, they form *port connections*, as depicted on Fig. 1. Through them, they interchange discrete units of information termed *port packets*. Port connections are established dynamically in runtime, are unidirectional, and follow a *publish/subscribe* paradigm of communication (Brugali and Shakhimardanov, 2010). Output and input ports may be defined out of a set of available typologies (more details in (Domínguez-Brito et al., 2007)).

CoolBOT components are distributed *active objects* (Ellis and Gibbs, 1989), in fact they can be seen as "data-flow machines". More formally, CoolBOT

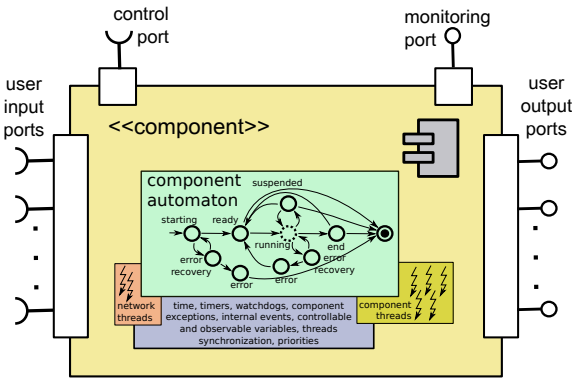


Figure 3: CoolBOT. Component structure.

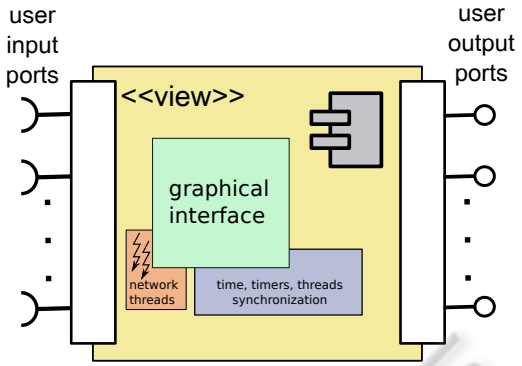


Figure 4: CoolBOT. View structure.

components are modeled as *port automata* (Stewart et al., 1997). Fig. 3 provides a view of the structure of a component in CoolBOT. Its external interface comprises all its input and output ports, its types, and the port packets it accepts through them. There are two special ports (*control* and *monitoring* ports) in any component, the rest are user defined. Internally a component is organized as an automaton, as illustrated in Fig. 3. The part of this automaton which is common to all components is called the *default automaton*, which allows for an external supervisor to control a component's execution using an standard procedure. Depicted by meta-state *running* in Fig. 3 (the dotted circle) each component defines its *user automaton* which is specific for each component and implements its functionality. Transitions among component's automaton states are triggered by incoming port packets, and also by internal events. The user can associate C++ callbacks to transitions. As active objects, CoolBOT components can organize its execution using multiple threads of execution as depicted on Fig. 3. All components are endowed at least with one thread of execution; the rest, if any, are user defined.

We have postponed deliberately the presentation of *views* and *probes* to the following sections, sections 3 and 4 respectively, so we elaborate on them next.

## 3 VIEWS

CoolBOT *views* are decoupled, integrable, composite and reusable graphical interfaces (GUIs) available for CoolBOT system integrators and developers, which, as software components, may be interconnected with any other component, view or *probe*.

In Fig. 4 we can observe the structure of a view in CoolBOT. As we can see views are also endowed with an external interface of input and output ports. Although, contrarily to components, views internally are not modeled using an automaton, as we can observe on the figure.

In fact, views use the same infrastructure provided by the framework to intercommunicate with other components, views and *probes*, residing them either locally in the same integration, or remotely in another one. Contrarily to CoolBOT components, views have been devised to implement no functionality, except the implementation of a GUI in a typical window-based graphical library, in fact, the current views already developed and operational which are available in CoolBOT have been implemented using the GTK graphical library (GTK, 2010).

Thence, to develop a CoolBOT view, we initially create a *workspace* (using a specific tool termed `coolbot-ske`). This workspace is a directory containing a template for generating our future view using `coolbot-c` (a `.coolbot-view` file), CMake template files needed to compile the view, and a *pkg-config* file (pkg-config, 2011) necessary to include and link the view in whatever application we would like to integrate our views (usually a CoolBOT integration), once completely developed.

Once created the workspace, using the description language accepted and using another tool, the CoolBOT compiler, `coolbot-c`, we describe the structure our component will have, mainly its external interface of input and output ports. Then, using `coolbot-c` the view is mapped into a C++ class, in particular CoolBOT generates the GTK *skeleton* of this C++ class, in the sense that the class is not completely functional, the functionality should be completed by the developer/user. Concretely, the part which should be completed by the user is precisely its specific graphical implementation, which it is done using GTK directly or using a GTK GUI graphical programming software like Glade (Glade, 2010). Mind that, the view's external interface of input and output ports is provided transparently by the framework and generated

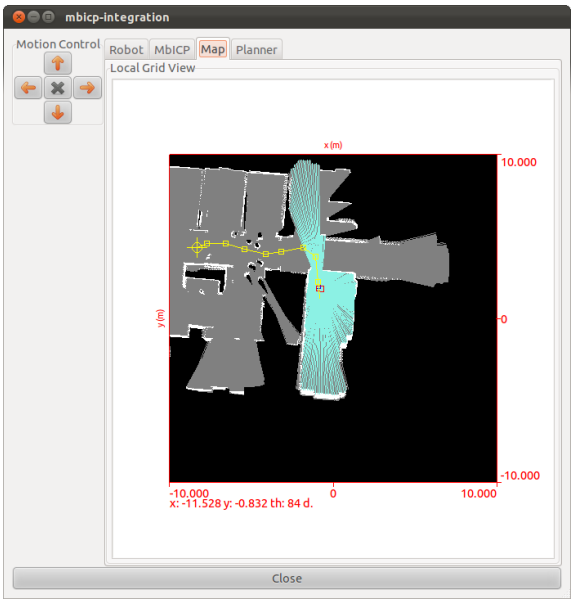


Figure 5: *Map* view's snapshot.

```
/*
 * File: grid-gtk.coolbot-view
 * Description: description file for GridGtk view
 * Date: ...
 * Generated by coolbot-ske
 */

view GridGtk
{
  header
  {
    author "...";
    description "GridGtk View";
    institution "...";
    version "..."
  };

  constants
  {
    private DEFAULT_REFRESHING_PERIOD=500; // milliseconds
    ...
  };

  input port ROBOT_CONFIG
    type poster
    port packet PlayerRobot::ConfigPacket;
  input port GRID_MAP
    type poster
    port packet GridMap::GridMapPacket;
  input port PLANNER_PATH
    type last
    port packet ShortTermPlanner::PlannerPathPacket;

  output port PLANNER_COMMANDS
    type generic
    port packet ShortTermPlanner::CommandPacket;
  output port ND_COMMANDS
    type generic
    port packet NDNavigation::CommandPacket;
};
```

Figure 6: `grid-gtk.coolbot-view`: `GridGtk`'s description file.

by `coolbot-c`, so views use the same infrastructure than component to communicate with other views, or components (and *probes*) whether remote or not.

As already mentioned, Fig. 1 illustrates a real system developed using CoolBOT. In Fig.5 we can see a runtime snapshot of one of the views of Fig. 1, concretely view `Map`, which shows the map the mobile indoor robot is building based on laser scans. In order to illustrate how C++ skeletons for views are generated, we can observe in Fig. 6, the description file for this view.

Once developed CoolBOT views are graphical components we can integrate in a window-based application. In fact, a CoolBOT integration which in-

```
/*
 * File: mbicp-integration.coolbot-integration
 * Description: description file for
 *              mbicp-integration integration.
 * Date: ...
 * Generated by coolbot-ske
 */

integration mbicp_integration
{
  header
  {
    author "...";
    description "MbICP's views integration";
    institution "...";
    version "..."
  };

  machine addresses
  {
    local my_machine: "127.0.0.1";
    the_other_machine: "...";
  };

  listening ports // TCP/IP ports
  {
    ROBOT_PORT: 1950;
    MBICP_PORT: 1965;
    NAVIGATION_MAP_PORT: 1970;
    ND_PORT: 1980;
    NAVIGATION_PLANNER_PORT: 1990;

    ROBOT_VIEW_PORT: 1955;
    MBICP_VIEW_PORT: 1985;
    NAVIGATION_MAP_VIEW_PORT: 1975;
    NAVIGATION_PLANNER_VIEW_PORT: 1995;
  };

  local instances
  {
    view robotView:PlayerRobotGtk
      listening on ROBOT_VIEW_PORT
      with description "Robot";
    view mbicpView:MbICPGtk
      listening on MBICP_VIEW_PORT
      with description "MbICP";
    view mapView:GridGtk
      listening on NAVIGATION_MAP_VIEW_PORT
      with description "Map";
    view navigationPlannerView:PlannerGtk
      listening on NAVIGATION_PLANNER_VIEW_PORT
      with description "Planner";
  };

  remote instances on the_other_machine;
  {
    component robot:PlayerRobot
      listening on ROBOT_VIEW_PORT;
    component mbicpInstance:MbICPCorrector
      listening on MBICP_PORT;
    component navigationMap:GridMap
      listening on NAVIGATION_MAP_PORT;
    component nd:NDNavigation
      listening on ND_PORT;
    component navigationPlanner:ShortTermPlanner
      listening on NAVIGATION_PLANNER_VIEW_PORT;
  };

  port connections // static connections
  {
    connect robot:ODOMETRY to robotView:ODOMETRY;
    ...
  };
};
```

Figure 7: The integration containing `Robot`. `MbICP`, `Map` and `Planner` views of Fig. 1.

cludes views is a GUI application, whose C++ skeleton has been generated using also `coolbot-c` from a description file. In Fig. 7 we can see and example, the four-view integration of Fig. 1. Notice that `coolbot-c` generates C++ skeletons for integrations where the static instantiation and interconnection among components and views are generated automatically. If we want to build a dynamic integration, we must complete the dynamic part of the skeleton generated by `coolbot-c` using the C++ runtime services provided by the framework (Fig. 2).

## 4 PROBES

As depicted in Fig. 8 a CoolBOT *probe* is provided with an external interface of input and output ports, and likewise component and views, as software components. Equally they implement the same network decoupled support of threads for transparent network communications. In CoolBOT, *probes* are devised as interfaces for interoperability with non CoolBOT software, as illustrated graphically in Fig. 1.

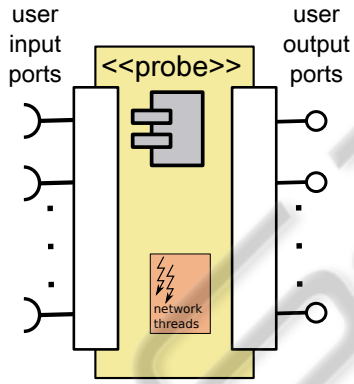


Figure 8: CoolBOT. Probe structure.

A complete functional C++ class implementing a probe is generated when a component is compiled by `coolbot-c`. The probe implements the complementary external interface of its corresponding component, and provides and API (Application Programming Interface) to use the probe to connect and communicate with instances of its corresponding component. Thus, a given probe instance provides means to access a component instance. Those probes generated automatically can be seen as automatic refactorings of external component interfaces in order to support interoperability of CoolBOT components with non CoolBOT software.

As of now we have used CoolBOT probes to interoperate with Java applets inserted in a web browser or used independently, as shown in Fig. 1. More specifically we have used SWIG (SWIG, 2011) in order to have access to the probe C++ classes in Java with the aim of implementing several Java GUI interfaces in Java equivalent to some of the CoolBOT views we have already developed. In Fig. 9 we can see a snapshot of the Java equivalent of a robot view to represent the range sensor information of the mobile robot.

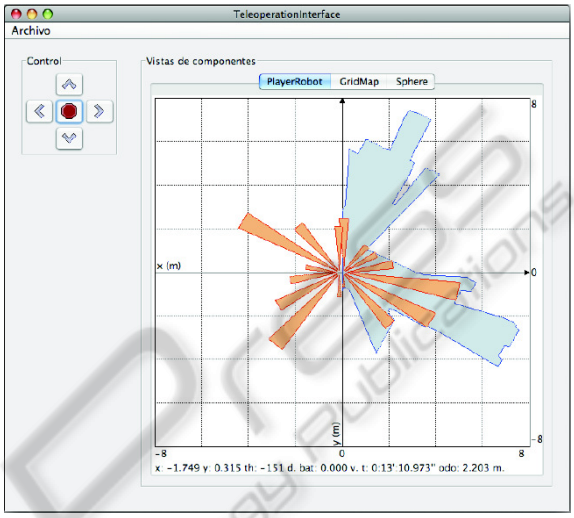


Figure 9: Snapshot of a Java view implemented using a probe.

## 5 CONCLUSIONS

In this paper we have explored the concept of interface in a distributed CBSE framework specifically aimed to build robotic control software. In fact we have put into practice the concept of interface as a software component on its own with its own structure and functionality, in opposition to a more wide concept of generic software component. Thus, we have presented the framework, CoolBOT, where we have available three types of software components to integrate systems: *components*, *views* and *probes*; being the last two, interfaces for two clear purposes, respectively: the implementation of graphical user interfaces in a component-based software system, and the interoperability of components with other different software (whether component-based or not).

GUIs implementing graphical views of a given system are a clear demand in distributed component based robotic software, and it is because of that, that we have integrated in our framework infrastructure, facilities and tools to facilitate its development. In this manner, GUIs can also be seen as integrable parts like any other software component, and at the same time, means are provided to systematize its development, in opposition to other approaches present in the

area. In addition we have also specifically put into practice another kind of software component which is a deployable interface allowing interoperation with other CBSE approaches, or just not CBSE software, this is a problem present in the area, as other authors have also indicated (Makarenko et al., 2007)(Geoffrey Biggs and Kotoku, 2010).

The framework we have presented in this document, CoolBOT, constitutes the work around which we have started recently an open source initiative called *The CoolBOT Project* (CoolBOT, 2011), where the framework itself, its developing tools, and the components, views and probes composing the system presented in this paper are freely available.

## REFERENCES


Ando, N., Suehiro, T., and Kotoku, T. (2008). A Software Platform for Component Based RT-System Development: OpenRTM-Aist. In Carpin, S., Noda, I., Pagello, E., Reggiani, M., and von Stryk, O., editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 5325 of *Lecture Notes in Computer Science*, pages 87–98. Springer Berlin / Heidelberg.

Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling heterogeneous real-time components in BIP. In Fourth IEEE International Conference on Software Engineering and Formal Methods, pages 3-12, Pune (India).

Bensalem, S., Gallien, M., Ingrand, F., Kahloul, I., and Thanh-Hung, N. (2009). Designing autonomous robots. *IEEE Robotics and Automation Magazine*, 16(1):67–77.

Brooks, A., Kaupp, T., Makarenko, A., S.Williams, and Oreback, A. (2005). Towards component-based robotics. In *In IEEE International Conference on Intelligent Robots and Systems*, pages 163–168, Tsukuba, Japan.

Brugali, D. and Shakhimardanov, A. (2010). Component-based robotic engineering (part ii). *Robotics Automation Magazine, IEEE*, 17(1):100 –112.

CoolBOT (2011). The CoolBOT Project. www.coolbotproject.org.

Domínguez-Brito, A. C., Hernández-Sosa, D., Isern-González, J., and Cabrera-Gámez, J. (2007). *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics Series*, chapter CoolBOT: a Component Model and Software Infrastructure for Robotics, pages 143–168. Springer.

Ellis, C. and Gibbs, S. (1989). *Object-Oriented Concepts, Databases, and Applications*, chapter Active Objects: Realities and Possibilities. ACM Press, Addison-Wesley.

Geoffrey Biggs, N. A. and Kotoku, T. (2010). Native robot software framework inter-operation. 2010 International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), November 15-18, Darmstadt, Germany.

George T. Heineman and William T. Councill (2001). *Component-Based Software Engineering*. Addison-Wesley.

Glade (2010). Glade - A User Interface Designer. glade.gnome.org.

GTK (2010). The GTK+ Project. www.gtk.org.

Makarenko, A., Brooks, A., and Kaupp, T. (2007). On the benefits of making robotic software frameworks thin. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07), San Diego CA, USA.

Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., and Ingrand, F. (2010). GenoM3: Building middleware-independent robotic components. IEEE International Conference on Robotics and Automation.

Minguez, J., Montesano, L., and Lamiraux, F. (2006). Metric-based iterative closest point scan matching for sensor displacement estimation. *Robotics, IEEE Transactions on*, 22(5):1047 –1054.

Minguez, J., Osuna, J., and Montano, L. (2004). A "Divide and Conquer" Strategy based on Situations to achieve Reactive Collision Avoidance in Troublesome Scenarios. IEEE International Conference on Robotics and Automation, New Orleans, USA.

Montesano, L., Minguez, J., and Montano, L. (2006). Lessons Learned in Integration for Sensor-Based Robot Navigation Systems. *International Journal of Advanced Robotic Systems*, 3(1):85–91.

Openrobots (2011). LAAS Openrobots. http://softs.laas.fr/openrobots/wiki.

Orca (2011). Orca: Components for Robotics. http://orca-robotics.sourceforge.net.

Orocos (2011). The Orocos Project. http://www.orocos.org.

pkg-config (2011). pkg-config Wiki. http://pkg-config.freedesktop.org.

ROS (2011). ROS: Robot Operating System. http://www.ros.org.

Schlegel, C., Haßler, T., Lotz, A., and Steck, A. (2009). Robotic Software Systems: From Code-Driven to Model-Driven Designs. In Proc. 14th Int. Conf. on Advanced Robotics (ICAR), Munich.

SmartSoft (2011). SmartSoft. http://smart-robotics.sourceforge.net.

Stewart, D. B., Volpe, R. A., and Khosla, P. (1997). Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects. *IEEE Transactions on Software Engineering*, 23(12):759–776.

SWIG (2011). SWIG. http://www.swig.org/.

Vaughan, R. T., Gerkey, B., and Howard, A. (2003). On Device Abstractions For Portable, Reusable Robot Code. In *IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003), Las Vegas, USA, October 2003*, pages 2121–2427.