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

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. Heinenman and William T. Councill, 2001) paradigm are G ENOM/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-aiT (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.
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.

Fig. 1 gives a global view of 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° 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.

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 1: CoolBOT. Secure Navigation System.

Figure 2: Abstraction layers in CoolBOT.
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 been designed 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-skel). 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 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...
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, con-
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 an 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.

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 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.

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
The CoolBOT Project called we have started recently an open source initiative CoolBOT, constitutes the work around which Biggs and Kotoku, 2010).


The framework we have presented in this document, CoolIBOT, constitutes the work around which we have started recently an open source initiative called The CoolBOT Project (CoolIBOT, 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


