Service Component Architectures in Robotics: the SCA-Orocos integration

D. Brugali[1] L. Gherardi[1] M. Klotzbücher[2] H. Bruyninckx[2]

[1] University of Bergamo, DIIMM, Italy
{brugali,luca.gherardi}@unibg.it
[2] Katholieke Universiteit Leuven, PMA, Belgium
{markus.klotzbuecher,herman.bruyninckx}@mech.kuleuven.be

Abstract. Recent robotics studies are investigating how robots can exploit the World Wide Web in order to offer their functionality and retrieve information that is useful for completing their tasks. This new trend requires the ability of integrating robotics and information systems technology. On the first side a set of robotics component based frameworks, which are typically data flow oriented, have been developed throughout the last years and Orocos is one of the most mature. On the other side the state of the art is represented by the Service Oriented Architecture, where the Service Component Architecture defines a component-based implementation of this approach.
The paper reports the progress of our work, which aims to promote in the robotics field a cooperation between Service Oriented Architecture and Data Flow Oriented Architecture. To achieve this we propose an integration between SCA and Orocos. We first highlight a set of architectural mismatches that have to be faced for allowing this integration and then we introduce a java-based library, called JOrocos, that represents our solution to these mismatches. Finally we describe a case study in which SCA and Orocos components cooperate for monitoring the status of a robot.

1 Introduction

Recent advances in robotics and mechatronic technologies have stimulated expectations for emergence of a new generation of autonomous robotic devices that interact and cooperate with people in ordinary human environments.

Engineering the control system of autonomous robots with such capabilities demands for technologies that allow the robot to collect information about the human environment, to discover available resources (physical and virtual), and to optimally exploit information and resources in order to interact with people adequately. Common approaches in robotics build on sophisticated techniques for perception and learning, which require accurate calibration and extensive off-line training

Recent approaches investigate how the robot can exploit the World Wide Web to retrieve useful information such as 3D models of furniture [10] and images of

objects commonly available at home [15]. In [16] the *Robotic Information Home Appliance* is illustrated as a home robot interconnected to the home network that offers a friendly interface to information equipment and home appliances.

This new trend poses new challenges in the development of robot software applications since they have to integrate robotic and information systems technologies, which account for quite different non-functional requirements, namely performance and real-time guarantees at one side and scalability, portability, and flexibility at the other side.

Modern robot control systems are typically designed as (logically) distributed component-based systems, where the interactions between components (control, sensing, actuating devices) are usually more complex compared to more traditional business applications. In Robotics, the software developer faces the complexity of event-based and reactive interactions between sensors and motors and between several processing algorithms. For this reason, robotic-specific component-based models and toolkits have been developed, which offer mechanisms for real-time execution, synchronous and asynchronous communication, data flow and control flow management, and system configuration.

In contrast, the most common middleware infrastructures for the World Wide Web and home networks are the Java Platform Enterprise Edition and Service Oriented Architectures. Service Oriented Architectures (SOA) have been proposed as an architectural concept in which all functions, or services, are defined using a description language and where their interfaces are discoverable over a network [8].

Some attempts to develop robotic applications as SOA systems can be found in the literature (a recent survey can be found in [5]). Their main disadvantage is that they give up the typical component-based nature of robotics systems and force a pure service oriented approach.

More recently, Service Component Architectures (SCA) [12] have been proposed as an architectural concept for the creation of applications that are built by assembling loosely coupled and interoperable components, whose interactions are defined in terms of bindings between provided and required services. As such, SCA offer the advantages of both the Component-based engineering approach typically used in robotics and the Service Oriented Architectures.

In order to bridge the gap between current component-based approaches to robotic system development and modern information systems technologies, we have developed the JOrocos library that extends the popular Orocos robotic framework [11] with Java technologies. Thanks to the JOrocos library, a robot control application can be designed as a SCA system, where components encapsulating real-time control functionality are seamlessly integrated with web services and the most common Java toolkits, such as the SWING framework for developing graphical user interfaces.

The paper is organized as follows. Section 2 introduces the Service Oriented Architecture and its main features. Section 3 briefly presents the Orocos concepts that are useful for better understanding the paper. Section 4 describes the architectural mismatches between SCA and Orocos and how JOrocos is de-

signed in order to define a bridge between the two component models. Section 5 presents a simple case study in which JOrocos is used for implementing an application where SCA and Orocos components work together. Finally section 6 draws the relevant conclusion.

2 Service Component Architecture

Robot control applications are increasingly being developed as component-based systems [4]. The reason is that, ideally, components embedding common robot functionality should be reusable in different robot control systems and application scenarios

This is achieved by clearly separating the component specification, which should be stable [3], from its various implementations.

A component specification explicitly declares which functionality (provided interfaces) are offered to its clients (code invoking an operation on some component, not necessarily in a distributed environment), and the dependencies (required interfaces) to the functionality that are delegated to other components.

Separating component specification from its implementation is desirable for achieving modular, interoperable, and extensible software and to allow independent evolution of client and provider components. If client code depends only on the interfaces to a component and not on the components implementation, a different implementation can be substituted without affecting the client code. Furthermore, the client code continues to work correctly if the component implementation is upgraded to support an extended specification [4].

The Service Component Architecture defines a generalized notion of a component, where provided interfaces are called *Services* and required interfaces are called *References*. Services and references are thus typed by interfaces, which describe sets of related operations that can be invoked synchronously or asynchronously.

The communication between pairs of components (i.e. operation invocation) occurs according to the specific binding protocol associated to *services* and *references*. A single service or reference can have multiple bindings, allowing different remote software to communicate with it in different ways, i.e. the WSDL binding to consume/expose web services, the JMS binding to receive/send Java Message Service, the Java RMI binding for classical caller/provider interactions.

The components in a SCA application might be built with Java or other languages, or they might be built using other technologies, such as the Abstract State Machines Language (ASML) [14][2]. Components can be combined into larger structures called composites [13], that are to be deployed together and that can themselves be further combined. Components in a composite might run in the same process, in different processes on a single machine, or in different processes on different machines.

The structure of SCA components and their interconnections are defined using an XML-based metadata language (SCDL), by which the designer specifies: the set of *services* provided and the *references* required by each component; the

implementation of each service as a link to a Java or other programming languages file; the component *properties*, i.e. data for configuring the component functionality, whose values are not hard-coded in the component implementation; the associations between references and services of different components; the bindings that specify access mechanisms used by services and references according to some technology/protocol; the aggregation of components in *Composites*.

SCA is supported by graphical tools and runtime environments. The tools build on the Eclipse Modeling Framework and allow the generation of a SCDL configuration file from a graphical representation of components and systems. SCA runtime environments, like Apache Tuscany and FRAscaTI, parse the configuration file, instantiate the implementation of each component, assign values to component properties, locate component services and references and create bindings with pairs of services and references.

3 The Orocos framework

Orocos is one of the oldest open source framework in robotics, under development since 2001, and with professional industrial applications and products using it since about 2005. The focus of Orocos has always been to provide a hard real-time capable component framework — the so-called *Real-Time Toolkit* (RTT) implemented in C++ — and as independent as possible from any communication middleware and operating system.

In the context of this paper, the following Orocos primitives are most relevant for the development of real-time components[3]:

- *TaskContext*: this is the Orocos version of a component, providing the basic infrastructure to make a system out of pieces of code that can interact with each other via data and events, a default *life cycle state machine*, reporting and timing support, an operating system abstraction layer, etc. The latter is, for example, provided via the primitive of an *Activity*, that is being mapped onto the process or threading system of the underlying operating system.
- *Data Ports*: components provide or request certain data types on their interface, in order to allow data flow based application architectures. Such architectures are very common in real-time control applications, since those typically perform the same kind of operations and interactions all over again, triggered by time or by hardware events. Hence, data flow is used for implementing the business logic of real-time components as it allows the developers to guarantee that a set of operations will be executed in a fixed amount of time: the period of the component. Figure 1 shows a system of Orocos components connected through their data ports according to the data-flow paradigm. It is taken from the Component Builder's Manual[3].

[3] The technical details can be found on the project's website, and more in particular in the *The Orocos Component Builder's Manual* `http://www.orocos.org/wiki/orocos/toolchain/toolchain-reference-manuals`.

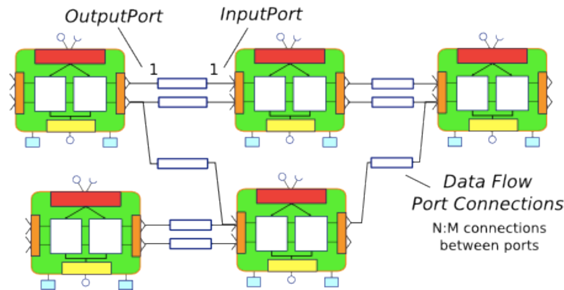


Fig. 1: The Orocos Data Flow Oriented Architecture.

- *Connection policy*: while Orocos wants to be as independent as possible from communication middleware (in order to let users make their choice based on their own criteria), it does provide some "middleware" anyway, for those interactions that take place in-process (for hard real-time data exchange between threads). Ports hide the low-level locking primitives of the underlying operating system, so that users don't have to worry about them, but the design goal of being able to guarantee real-time performance is not well served by a *complete* shielding of the way *Activities* interact. Hence, Orocos provides *Lock free data Ports* as one of its major features: an *Activity* will never have to wait to get or read data when it needs them, because Ports can be configured to copy their data when one or more other Activities are also accessing the data.
- *Services*: Orocos implements the concept of services, which are containers of operations. They are not used for implementing the kind of operations that regard the business logic of the real-time components, but for example for configuring their period or retrieving information about their status (stopped, running, etc).
- *Properties* are part of the Service Configuration interface and are used to load or tune application specific configurations at runtime (e.g. the parameters of a PID).

4 SCA - Orocos Integration

In order to make possible the interaction between SCA and Orocos components we had to face some architectural mismatches presented by the two frameworks. In fact, despite both SCA and Orocos components interact by exchanging messages, the syntax and semantics of these messages is fundamentally different.

In SCA messages are used for invoking services provided by components. Services are defined by explicit interfaces that completely describe the name of each operation, its arguments and the return value (the signature of the method). The message sent by the requester component to the provider component describes which operation has to be executed and provides its parameters. Hence the execution of the component functionality starts when the message is received.

In Orocos instead the communications are based on data flows and the messages are used for exchanging data. Components periodically elaborate data received on the input ports and write their results on the output ports. This means that the components business logic is regularly executed every T milliseconds, where T is the period of the component.

This meaningful difference introduces two main problems:

1. How an invocation of a SCA service can produce an input that will be processed in the next cycle of an Orocos component business logic?
2. How the data published on an Orocos output port can trigger the execution of a SCA service?

Let's introduce how we solved these problems by means of a simple scenario in which two SCA components and an Orocos component cooperate in order to move a Kuka youBot [9] towards a given position. The youBot is a mobile manipulator with an omnidirectional and holonomic base and a five degrees of freedom arm. The components are described below:

– A SCA component, called *Locomotor*, which provides a service for moving a youBot towards a position defined by the client and monitoring its activity. The component is in charge of transforming the given cartesian position in a set of commands (joint positions), forwarding them to the robot and retrieving its status. In order to do that the component requires two services, which are provided by the driver of the robot for sending and receiving these information.
– An Orocos component, called *youBotDriver*, which provides an input port and an output port. The component implements the API of the youBot and is in charge of actuating the axes in order to reach the joints positions specified by the client on the input port. The output port is instead used for periodically publishing the status of the robot, for example the position and the velocity of the joints.
– A SCA component, called *SCAyouBotDriver*, which is implemented by using the JOrcos library and represents a proxy to the youBotDriver component within the SCA system.

The *SCAyouBotDriver* is described by means of the following interfaces:

– Provided interface *sendingCommand*. This interface provides a service for receiving commands from the *Locomotor* and writing these commands on the input port of the *youBot Driver*. The result is the activation of the *youBot Driver* operations that are in charge of moving the robot.
– Provided interface *retrievingStatus*. This interface is invoked by the *Locomotor*. The *SCA youBot Driver* periodically checks and retrieves the new data available on the *youBot Driver* output port. The interface provides the operation for retrieving these values.

– Required interface *notifying*. This interface is provided by the *Locomotor* and is used by the *SCA youBot Driver* for notifying some events. The possible events are *idle*, *busy*, *refresh*. The component raises the event *busy* when it starts the execution of an operation and the event *idle* when this operation is completed. In this way the *Locomotor* knows whether the operation that it requested is completed or not. The state *refresh* is instead raised when new data are read from the *youBot Driver* output port.

Here is important to consider that in order to notify the *Locomotor* about the availability of new data before the *youBot Driver* deadline, the *SCA youBot Driver* should check the output port with a frequency at least two times greater than the one of the Orocos component.

The components and the interfaces of this scenario are depicted in figure 2.

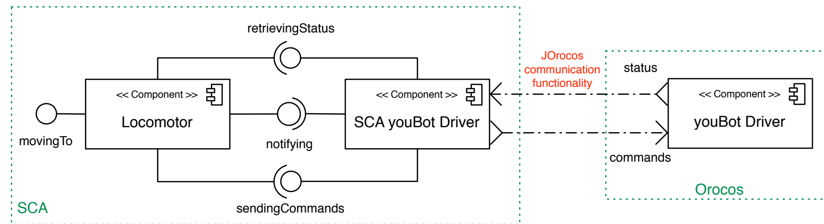


Fig. 2: The youBot Scenario

Another difference between SCA and Orocos regards the synchronization of the component operations after the action of sending a message. In SCA it is possible to define by means of an annotation whether the message is sent in a synchronous or asynchronous way. In the first case the thread that sends the message suspend itself until the result is returned. In the second case instead the thread continues its execution without waiting for the return value. A callback message will notify the component when the return value of the sent message will be computed. In Orocos all the messages are sent in an asynchronous way. The components read the data on the input ports and publish data on the output ports without waiting for other component activities.

Our library face this problem by providing the possibility of reading data from the Orocos output port in an asynchronous way, according to the Publish/Subscribe communication paradigm [6] (more information about the implementation will be described in the subsection 4.2). In this way both SCA and Orocos component don't have to wait after sending a message. However a synchronous communication can always be defined in the implementation of *SCA youBot Driver*.

The last mechanism provided by SCA that is not defined in Orocos is the hierarchical composition of the components. In SCA this functionality is available by using the concept of composite. A composite contains different components

and allows the developer to promote a set of their services in order to make them accessible to the clients of the composite. In this way a composite can be reused as a simple component in a more complex architecture.

Here is possible to leverage on this SCA mechanism and create composites that contains different bridges to Orocos components (like the *SCA youBot Driver*) and promotes their operations as services. This approach is inspired by the facade design pattern, which aims to provide a unified interface to a set of interfaces in a subsystem [7]. In this way a single reusable SCA component, the composite, can provide the functionality defined in several Orocos components to its client.

4.1 The JOrocos library and its architecture

The JOrocos library offers a set of mechanisms that allow the implementation of the proxies of Orocos components mentioned in the previous pages (e.g. *SCA youBot Driver*). These mechanisms provide the functionality for reading and writing on Orocos data ports, reading and writing Orocos properties and invoking operations provided by Orocos components.

Another interesting mechanism offered by JOrocos is the introspection of Orocos running components. It provides the functionality for discovering at runtime which components are available, their ports, their operations and their properties. This mechanism allows the development of systems more complex than the scenario defined in introduction of this section: systems in which the SCA composite doesn't have a priori knowledge of the Orocos components and configures itself at runtime according to the information retrieved through the introspection. For example, with reference to the previous scenario, it will be possible to design a system in which the SCA composite doesn't know at compile time which robot has to be controlled. This information will be retrieved at runtime by introspecting the current *Robot Driver* component and according to its ports the *SCA Robot Driver* will configure itself.

This functionality is realized on the top of Corba, the middleware that Orocos uses for exchanging messages between distributed components. Corba doesn't guarantee the respect of real-time constraints and for this reason when the communication between Orocos components has to be real-time the components have to run on the same machine. In this way the communication between the local components doesn't rely on Corba and so the respect of the real-time constraints is not compromised. In the same way the use of Corba is not a problem for our integration because the real-time components will be implemented by using Orocos and will run on the same machine.

The architecture of the library is depicted in the UML class diagram reported in figure 3. As showed in the diagram the classes of the library are organized in two main packages: *core* and *corba*.

- The core package contains the classes that store data structures and offer operations that are middleware independent. These classes define the core of the library and represent the main entities of an Orocos system.

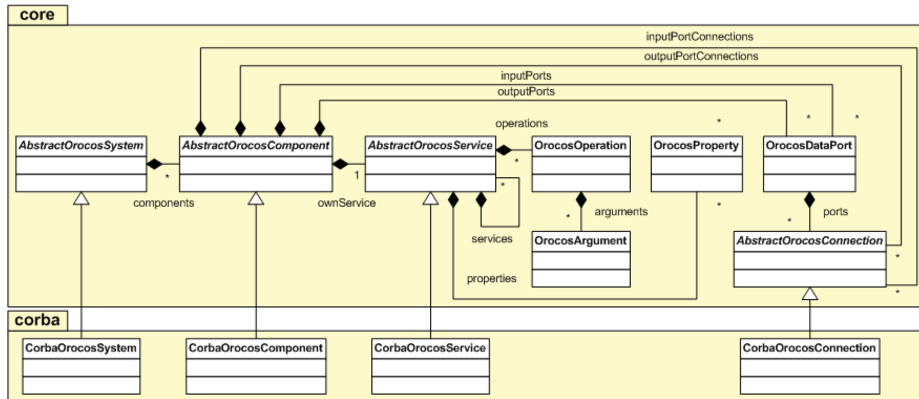


Fig. 3: The JOrocos Architecture

– The corba package contains instead the classes whose methods provide a set of operations that are corba specific.

The classes of the core package whose name starts with the word *Abstract* are abstract classes and have to be extended in order to provide the functionality that are middleware specific. They represent proxies of Orocos entities and offer methods for introspecting them and interacting with them. The other classes of the package are instead completely middleware independent.

The idea is that the separation of the middleware-independent parts (core package) from the middleware-specific parts (corba package) will allow in future an easier extension of the library in order to provide a support for other middlewares.

The main class of the library is named *AbstractOrocosSystem*. It offers the functionality that allows a developer to connect his software to a running Orocos system, introspect its components and retrieve references to them.

An *AbstractOrocosComponent* is a proxy to an Orocos component and allows the clients to introspect its data ports and its own service. The class offers the operations for creating connections to Orocos ports and writing and reading data on these ports.

The data port of an Orocos component are represented by means of the class *OrocosDataPort*. The interaction with these ports is made available by the class *AbstractOrocosConnection*, which provides the channel that allows the operations of writing data on the output ports and reading data from the input ports.

An *AbstractOrocosService* is a proxy to an Orocos service and offers the functionality for introspecting and invoking its operations and introspecting, reading and writing its properties.

The operations of an Orocos service are represented by means of the class *OrocosOperation*. The properties are instead described by means of the class *OrocosProperty*.

4.2 The SCA-OROCOS component

In this subsection we will explain how the interaction between Java and Orocos works and how the component *SCA youBot Driver* is implemented. The code reported in the listing 1.1 shows the interfaces of the services provided by the component. The annotation *@Callback* defines the interface that will be used for notifying the events to the *Locomotor*. The annotations *@OneWay* instead means that the invocation of method will be asynchronous.

```
public interface retrievingStatus{
  public double[] getJointsPositions();
}
@Callback(Notifying.class)
public interface SendingCommands {
  @OneWay
  public void setJointsPositions(double[] values);
}
```

Listing 1.1: The interfaces of the *SCA youBot Driver* services

The listing 1.2 reports the variables declared in the implementation of the *SCA youBot Driver* component.

```
@Service(interfaces={retrievingStatus.class,SendingCommands.class})
public class SCAYouBotDriver implements retrievingStatus,SendingCommands,Observer{
  @Property
  protected String orocosIP;
  @Property
  protected String orocosPort;
  @Callback
  protected Notifying locomotor;
  private AbstractOrocosSystem orocosSystem;
  private AbstractOrodocComponent youBotDriver;
  private double[] jointsPosition;
```

Listing 1.2: Part of the implementation of the *SCA youBot Driver* component

The class implements the interface *java.util.Obsever* (Observer design pattern [7]), which defines a method for being notified when a new data is available on an Orocos output port. Furthermore the class implements the two interfaces that describe the services.

The first line is a SCA annotation that defines the interfaces of the services of the component. The lines from 3 to 6 declare two SCA properties used for configuring the IP address and the port number of the Corba name service, lines 7 and 8 instead declare a reference to the SCA callback interfaces.

Read and write data on Orocos data ports The JOrocos library allows both the operations of reading and writing on an Orocos data port. In order to be executed these operations require a connection between the java client and the Orocos port. Two types of connections are available: data and buffer. On a data connection the reader has access only to the last written value whereas on a buffer connection a predefined number of values can be stored.

The listing 1.3 reports the constructor of the class *SCA youBot Driver* in which the connections to the port are created.

```
1  public SCAYouBotDriver(){
2      orocosSystem = CorbaOrocosSystem.getInstance(orocosIP,orocosPort);
3      orocosSystem.connect();
4      youBotDriver = orocosSystem.getComponent("youBotDriver", false);
5      youBotDriver.createDataConnectionToInputPort("commands", LockPolicy.LOCK_FREE, this);
6      youBotDriver.subscribeToDataOutputPort("joinstStatus", LockPolicy.LOCK_FREE, this, 500);
7  }
```

Listing 1.3: The implementation of the service *SubscribingTojoinstStatusPort*

- Lines 2-3 retrieve a reference to an Orocos running system and create a connection to it.
- Lines 4 retrieves a reference to the *youBot Driver* component.
- Line 5 creates a data connection to the input port *commands* of the Orocos component *youBot Driver*.
- Line 6 creates a data connection to the output port *status* and starts a thread that periodically check if new data are available on the port. This functionality is implemented in JOrocos (methods *subscribeToDataOutputPort* and *subscribeToBufferOutputPort*). In this case a data connection with a lock free policy is created. The third parameter specifies the Observer object that will be notified when new data will be available on the port (in this case it is the component). Finally the last parameter defines the frequency with which the availability of new data on the port will be checked (it is expressed as period in milliseconds).

Once the component is subscribed to the output port it will be notified as soon as a new data will be available by means of the method *update* (inherited from the Observer interface). The implementation of this method is reported in the listing 1.4. It simply stores the new data on the variable *jointsPosition* and notifies the *Locomotor* that new data are available.

```
1  public void update(Observable arg0, Object arg1) {
2    OrocosPortEvent event = ((OrocosPortEvent)arg1);
3    jointsPosition = ((YouBotStatus)event.getValue()).getJointsPosition;
4    locomotor.notify("refresh");
5  }
```

Listing 1.4: The implementation of the method *update*

From this moment the *Locomotor* can retrieve the new data through the operation provided by the interface *retrievingStatus*. Its implementation is reported in the listing 1.5. It simply returns the position of the joints.

```
1  public double[] getJointsPositions() {
2    returns joinstPosition();
3  }
```

Listing 1.5: The implementation of the service *SubscribingTojoinstStatusPort*

The listing 1.6 reports instead the implementation of the operation defined in the service *sendingCommands*. The purpose of this operation is writing the data received from the *Locomotor* to the Orocos output port. The component first notifies the *Locomotor* that the operation is started, then writes the values on the *commands* port and finally notifies the *Locomotor* that the operation is completed.

```
1 public void setJointsPositions(double[] values) {
2   locomotor.notify("busy");
3   youBotDriver.writeOnPort("commands", values, this);
4   locomotor.notify("idle");
5 }
```

Listing 1.6: The implementation of the service *sendingCommands*

The operations of writing and reading data support both simple and complex data types and respectively receive as parameter and return as result instances of the class *Object*. In this context corba introduced two issues:

1. Corba returns references to the requested objects as instances of the class *Any*. Hence the result of a read operation is an *Any* object, whereas we want to return a more general *Object* instance.
2. The cast from *Any* to the right type is possible only by means of the "*Helper*" classes that are automatically generated from the IDL-to-Java compiler. However we cannot know every possible data type a priori and consequently implement all the possible cast in the code of our library.

We solved these two problems by means of the Java reflection. Indeed, in the code of the write and read operations we retrieve the class name from the object that has to be written (in the case of the write operations) or from the *Any* object (in the case of the read operations). Then we use the name of the class for loading at runtime the right "*Helper*" class and using its static method for casting *Any* to *Object* or vice versa. The listing 1.7 shows how our library casts an *Any* to an *Object*.

```
1 // value is the object that has to be written
2 String className = value.getClass().getName(); + "Helper";
3 Class<?> helper = Class.forName(className);
4 Method castMethod = helper.getMethod("insert", Any.class, value.getClass());
5 // the insert method inserts value in the any object received as second parameter
6 castMethod.invoke(null, any,value);
```

Listing 1.7: The *Any* to *Object* cast

5 The case study

In order to test the functionality provided by JOrocos we have implemented a simple case study application. It is similar to the scenario introduced in the section 4 but the *Locomotor* component is replaced by a graphical interface

(*youBot Monitor* component), which is in charge of plotting the current state of the joints and allowing the user to set the period (inverse of frequency with which the operations of the component is executed) of the *youBot Driver* component.

The *youBot Driver* is currently a dummy component. Indeed we are working on its implementation in the context of the European project BRICS (Best of Robotics [1]), but unluckily it is not ready yet. The dummy component publishes on the output port a set of random values that describe for each joint position, velocity, current, temperature and error flag (10 bits that provide information about a set of possible errors). For the purpose of the test it doesn't matter whether the values published on the port a real or random. In fact we are only interested in testing the communication between SCA and the Orocos components.

The *youBot Driver* component has a new input port called *period*. When a new data is written on this port the component set its period according to the value of this new data. Due to this new port also the *SCA youBot Driver* has a new provided interface named *settingPeriod*. It provides a service for receiving a new period value from the *youBot Monitor* and writing it on the input port of the *youBot Driver*.

The *youBot Monitor* component has two required interfaces that correspond to the provided interfaces of the *SCA youBot Driver*. It also provides the *notifying* interface, which is used by the *SCA youBot Driver* for notifying its events.

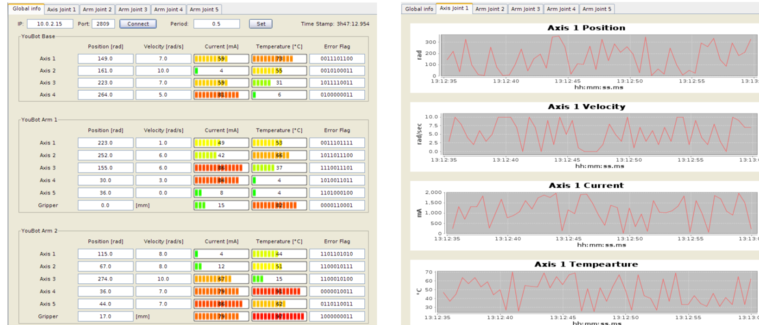
The SCA components and the Orocos component run on two different machines: the last one on board of the robot whereas the other two on the supervisor workstation.

The implementation of the *SCA youBot Driver* component is very similar to the one reported in section 4. The *youBot Monitor* component is instead implemented by using the Java SWING and provides several tabs. In the main tab a set of global information about the state of the joints is showed. This tab also allows the configuration of the *youBot Driver* period. The other tabs instead provide information about a specific joint and plot on a set of charts the trend of the joint values. The graphical interface is depicted in figure 4.

This case study demonstrated how JOrocos makes possible and simple the communication and the cooperation between SCA and Orocos. By writing few lines of Java code we were able to retrieve data from the Orocos output port and set the period of the *youBot Driver* component. Furthermore we didn't have to care about the location of the components on the network (except for setting the IP address and the port of the name service) and the different programming language used for implementing the Orocos component.

6 Conclusions

In this paper we have discussed the problem of making possible the cooperation between Service Oriented Architectures (SOA) and Data Flow Oriented Architectures in the robotics field. In particular we have focused our attention on SCA and Orocos, the first a component based SOA and the second an hard real-time



(a) Global status (b) Joint 1 charts

Fig. 4: The graphical interface

component based robotics framework. We have presented a set of architectural mismatches between the two component models and a java-based library, named JOrocos, which allows the developers to bridge these differences by defining proxies components. We have also provided a guideline for the development of these proxies and we have applied it in a case study.

The first mismatch regarded the syntax and the semantics of the messages exchanged between the components in the two frameworks. Here JOrocos provides to the developers the mechanisms for allowing the communication between SCA and Orocos components and translating SCA messages to Orocos messages and vice-versa. However JOrocos doesn't provide the possibility of directly connecting a SCA Service (or Reference) to an Orocos Port. The developer has to define, according to our guideline, a proxy component which provides input to the Orocos component when one of its services is invoked and invoke a service of its client (the SCA component) when the Orocos component produces data on the output port. In this direction a possible improvement will consist of (a) using JOrocos for extending the SCA runtime in order to define a new binding for Orocos and (b) extending the SCA composite designer for supporting this new binding. These extensions will replace the role of the proxy components and will allow the developer to directly connect SCA and Orocos components.

The second mismatch was about the synchronization of the component operations after the action of sending a message. Here JOrocos doesn't provide the possibility of choosing a specific synchronization mechanism. In order to permit both synchronous and asynchronous way, a specific synchronization mechanism has to be implemented in the proxy components. For example, in our scenario we have demonstrated how it is possible to send messages synchronously and asynchronously. Indeed the operation of retrieving the robot status is executed by the *SCA youBot Driver* in a synchronous way, whereas the operation of sending commands in an asynchronous way.

Finally the last mismatch concerns the absence of an hierarchical composition mechanism in Orocos. Here JOrocos allows the developers to leverage on the

SCA composition mechanism for encapsulating several Orocos proxies in SCA composites and reusing them in complex and hierarchical systems.

7 ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. FP7-ICT-231940-BRICS (Best Practice in Robotics).

The authors would like to thank all the partners of the BRICS project for their valuable comments.

References

1. BRICS - Best of Robotics. http://www.best-of-robotics.org/.
2. D. Brugali, L. Gherardi, E. Riccobene, and P. Scandurra. A formal framework for coordinated simulation of heterogeneous service-oriented applications. In *8th International Symposium on Formal Aspects of Component Software (FACS)*, 2011.
3. D. Brugali and P. Salvaneschi. Stable aspects in robot software development. *International Journal on Advanced Robotic Systems*, 3(1):17 –22, march 2006.
4. D. Brugali and P. Scandurra. Component-based robotic engineering, part I: Reusable building block. *Robotics & Automation Magazine, IEEE*, 16:84–96, 2009.
5. R. de Molengraft, van, M. Beetz, and T. Fukuda. A special issue toward a www for robots. *Robotics Automation Magazine, IEEE*, 18(2):20, june 2011.
6. P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.
7. E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
8. E. Marks and M. Bell. *Service-Oriented Architecture (SOA): A planning and implementation guide for business and technology*. John Wiley & Sons, 2006.
9. KUKA youBot store. http://youbot-store.com/.
10. O. Mozos, Z.-C. Marton, and M. Beetz. Furniture models learned from the www. *Robotics Automation Magazine, IEEE*, 18(2):22 –32, june 2011.
11. Open Robot Control Software. http://www.orocos.org.
12. Service Component Architecture (SCA). http://www.osoa.org.
13. SCA Specifications - SCA Assembly Model. http://www.osoa.org/display/Main/The+Assembly+Model.
14. P. Scandurra and E. Riccobene. A modeling and executable language for designing and prototyping service-oriented applications. In *EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011)*, 2011.
15. M. Tenorth, U. Klank, D. Pangercic, and M. Beetz. Web-enabled robots. *Robotics Automation Magazine, IEEE*, 18(2):58 –68, june 2011.
16. T. Yoshimi, N. Matsuhira, K. Suzuki, D. Yamamoto, F. Ozaki, J. Hirokawa, and H. Ogawa. Development of a concept model of a robotic information home appliance, aprialpha. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 205 – 211, 2004.