Robocontrol 2012
5th WORKSHOP IN APPLIED ROBOTICS AND AUTOMATION
unesp

INTEGRATING THE OROCOS FRAMEWORK AND THE BARRETT WAM ROBOT

Darlan Ioris[1], *Walter Fetter Lages*[1], *Diego Caberlon Santini*[1]

[1]UFRGS, Porto Alegre, Brazil, darlanioris@ece.ufrgs.br, fetter@ece.ufrgs.br, diegos@ece.ufrgs.br

Abstract: This paper deals with the development of an interface between the OROCOS framework and the Barret WAM robot. The interface is designed as an OROCOS component, which integrates the Barrett WAM with a previously developed open architecture for robot control.

Keywords: OROCOS, Control architecture, Open architecture

1. INTRODUCTION

The need for high performance robots imposes a growing complexity in hardware, software and control architectures. The reuse of existing knowledge and functional blocks makes the task of designing a new robot system easier, faster and more reliable. However, such a reuse is not yet the mainstream in the robot industry. Usually, commercial robots are based on proprietary architectures, which precludes its integration in more complex systems or even the software reuse.

In order to overcome such problems, new development methodologies are necessary. Those methodologies should enable the integration of technologies and knowledge of all designers involved in such projects.

In an open architecture, all the details of the robot are documented and the software and hardware structures are such that new sensors, controllers and interfaces can be added. All aspects of the robot design can be easily modified [1]. Therefore, by using open architectures, one intents to create design standards that makes the system integration and hardware and software reuse easier.

Open architectures can be used with many systems and should have an implementation abstracting the hardware and basic software, to avoid dependence on any specific producer. Hence, it should be independent of the supporting platform.

A methodology that provides the tools for the development open architectures with such characteristics is the Component-Based Software Engineering (CBSE). That methodology uses the concept of component to develop software autonomous units, each one able to abstract some hardware part or functionality and exposing an standard interface to the remaining system. That modularity, enabled by components, is the base for an open architecture.

However, the development of component based software and code reuse is not yet a common practice in robotics, Nowadays, most research and software development are based on custom software architectures, built from scratch [2]. Hence, most robot applications are developed for a specific purpose, which accumulates a huge amount of software implementing complex systems. Nonetheless, that does not favor the software reuse, as that software is specific for a hardware, operating system or communication media. Furthermore, all the functionality and knowledge is hard-coded in the program and not exposed in a clear and consistent interface.

This paper proposes to integrate the Barrett WAM manipulator robot [3] to the OROCOS framework [4], thus enabling the use of the robot through components built for this framework and the integration of the robot in an open control architecture proposed in [1].

The state of the art in the development of robotics software and the main frameworks are presented in section 2., along with some concepts of CBSE. In section 3., the OROCOS framework is discussed in more detail, while the section 4. introduces the hardware and software of the Barrett WAM robot, discussing its functionalities and its proprietary library. The architecture proposed in [5] and its use with the most recent version of the OROCOS framework are detailed in section 5.. The interfacing of the `libbarrett` library with the OROCOS framework, through a component is presented in section 6., while its use in an open architecture is dealt with in section 7., where the implementation of two types of controllers are presented. Finally, conclusions and future development directions are presented in section 8..

2. COMPONENT-BASED ROBOTICS

The application of CBSE concepts to robotics enables the development and handling of complex robotics systems through the use of previously developed components [6], yielding the following benefits:

Complexity management: even the simpler robotic systems are complex due to existence of many elements such as actuators, sensors and controllers that interact with each other. Most of them require a proper execution thread synchronously or asynchronously communicating with other threads. A method to standardize the communication and delimit each element according to its tasks and characteristics is necessary to manage the system complexity.

Flexibility: for the development of complex projects, it is very important to be able to develop, change and test specific modules without side-effects in the whole system. The flexibility of a component based system enables to focus on a particular task, retaining the functionality of the remaining system.

Distributed environments: distributed robotic systems are widely used, typically where mobile robots are controlled by a remote station. The modularity of a component based system makes it simpler to communicate with a swarm of robots.

Variety of hardware and operating systems: robotic systems are implemented in a variety of hardware platforms and operating systems. The modularization achieved through components enables decoupling the application implementation from the underlying hardware and operating systems. Hence, it becomes possible to build generic applications, without dependencies on those factors, that would be useful in many situations.

Those properties enables the development of new applications based on existing and reliable components, thus ensuring more robustness to the new system.

Many architectures, frameworks and components have been proposed and developed to help on the process of building robotic control systems. Although most systems adopt a component based architecture with the purpose of software reuse, in general, the architecture design differs, usually due to the desire to efficiently support a specific project or architecture. The frameworks most used in robotic research include [7]:

***Player* [8]:** it is a set of tools for mobile robots, including drivers for robotic devices. Conceptually, it is a hardware abstraction layer for robotic devices, which also includes data communication and control programs. The communication interfaces are based on a client/server architecture that uses TCP sockets.

OROCOS [4]: Open RObot COntrol Software (OROCOS) was started in 2001 to develop open source code for robot control. As it is the framework used in this work, it is described in further detail in section 3..

Orca [6]: a fork of the OROCOS project, the Orca project, aims to provide construction blocks (components) that could be combined to build arbitrarily complex robotic systems without real-time requirements. It uses the Internet Communications Engine (ICE) [9] as the network middleware.

ROS [10]: the Robot Operating System is a open source package that provides operating system services such as hardware abstraction, low level device control and inter-process communication, as well as development tools. The purpose is to create a common platform upon which researchers could build and share higher level robotics algorithms in areas such as navigation, localization, planning and manipulation.

CISST [11]: a set of libraries designed to ease the development of computer assisted intervention systems.

Those frameworks are compared in Table 1 [7], which shows the level of support for the Windows, Linux, RTOS (Real-Time Operating System), MT (Multi-Thread), MP (Multi-Process) e MH (Multi-Host) platforms for each framework. Regarding the RTOS support, it is important to note that what is considered here is the support for hard real-time execution and not just the possibility to execute the framework in a RTOS.

Table 1: Common frameworks used in robotics.

Framework	Windows	Linux	RTOS	MT	MP	MH
Player	partial	yes	no	no	yes	yes
OROCOS	yes	yes	yes	yes	yes	yes
Orca	partial	yes	partial	no	yes	yes
ROS	partial	yes	partial	no	yes	yes
CISS	yes	yes	yes	yes	yes	yes

The OROCOS framework will be used here as it provides more functionalities and has a scope broader than CISST, which is more devoted to the medical area.

3. OROCOS

The objective of the OROCOS project is to develop a general purpose, modular, open source framework for controlling machines and robots [12]. It executes on the Linux and Windows operating systems and supports real time kernels such as RTAI [13] and Xenomai [14].

An OROCOS component is a basic unit that executes one or more actions, which are determined by its activity. Those actions can be a function in the C or C++ language, a script in its own language or even a hierarchical state machine. Here, only actions in the C and C+ language are considered. The component activity is started by the `Activity` class, which has the parameters `Period`, `Priority` and `Scheduler`. The `Period` parameter is used to define an periodic activity with priority defined by the `Priority` parameter and scheduled with the policy defined by the `Scheduler` parameter, which can be a real time scheduler represented by the constant `ORO_SCHED_RT` or a non real time scheduler represented by `ORO_SCHED_OTHER`.

The interface of an OROCOS component is comprised of the following:

Properties and Attributes: are variables used to configure and adjust the component. Properties can be written to and read from a file in XML format, hence, they can store persistent values. Attributes reflect a member variable of a C++ class and can be read and written to for the execution time of a program, but do not persist across the program end.

Operations: are objects that define the functions that a component exposes at its interface. When configured as an

operation, any method of any class can be added to the interface of a component. This way, functions implemented in C/C++ can be used by scripts or can be called from another process or remotely, through the network. Operations receive arguments and return a value. An operation can be implemented in its `OwnThread` or in the `ClientThread`. A `ClientThread` operation is performed synchronously with the caller component, as it is executed in the thread of the caller. On the other hand, an `OwnThread` operation is performed asynchronously with the caller component, as it is executed in the thread of the called component, whose execution depends on the called component activity.

Form the point of view of the component that calls an operation, there are two behaviors as well, defined by the `OperationCaller` class. When the operation is invoked though the `call()` function, the calling component blocks waiting for the execution of the operation. However, if the operation is invoked through the `send()` function, the caller component continues its execution, and receives from the `send()` function an object of the `SendHandle` class, which is used to track the status of the operation and collect its results. By default, operations are invoked through a `call()` function.

Data ports: are objects used to implement data flow. A port is defined by a unique name in a component, its data type and its port type, which can be a read-only port, with respect to the component where it is defined, represented by the `InputPort` class or write-only port, represented by the `OutputPort` class. An data input port can be configured to trig the activity of a component or call a function when data is received. Those ports are created as `eventports` and can react to the reception of data.

OROCOS components are derived from the `TaskContext`, which defines the public interface of the component. For a component to have access to the interface of another one, it should be configured as a peer of such a component. Data ports are an exception and can be accessed without peering. However, data ports should be connected to each other. That can be done through member functions of the `TaskContext` class, member function of the port itself or even through the `deployer` component, which can perform an initialization though an XML file.

The `TaskContext` class provides the control interface for the component, while the `ExecutionEngine` class executes the user application, accordingly to the configured component activity, period, priority and scheduler. Figure 1 shows the interface of an OROCOS component and its interaction with a peer component [15].

Every component has hook functions where the user can attach its code to define the details of the operation of the component. Those functions are [5]:

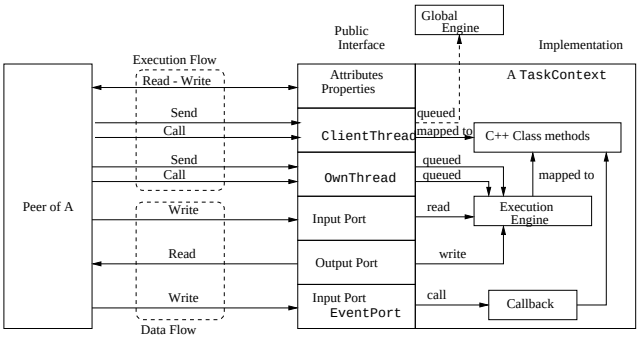

Figure 1: OROCOS component interface.

`configureHook()`: configures the component;

`startHook()`: initializes the component;

`updateHook()`: component activity;

`stopHook()`: stops the component activity;

`cleanupHook()`: finalizes the component;

`activeHook()`: activates the component;

`errorHook()`: called in place of `updateHook()` in case of non critical errors;

`resetHook()`: recovers from a critical error.

4. The Barrett WAM Robot

The Barrett WAM (Whole Arm Manipulator) is a robotic arm with four or seven degrees of freedom, optionally including a hand with three fingers. Figure 2 shows a 7-DOF WAM equipped with a BarrettHand, that was used in this work.

The Barrett WAM has its own internal computer (WAM PC), which executes a real time Linux operating system based on Xenomai. However, the WAM control system can be executed in an external computer, as well, as the internal CAN bus is exposed for external use.

4.1. Hardware overview

The WAM actuators are drived by Barrett patented power modules, named Pucks. A Puck is a digital torque controller, encoder, temperature and current sensor which is mounted directly on each WAM joint. All Pucks and the WAM PC are connected though a CAN bus at 1 Mbps.

The Pucks send the joint position data to the WAM PC and receive the torque to be applied to the joints in a simple control loop. The sampling rate of this control loop can be adjusted up to 1 kHz, but the standard rate is 500 Hz. All communications are monitored by a Safety Board, which checks for the arm speed, torque command values and the overall system status [3].

Figure 2: Barrett WAM.

Figure 3 shows a block diagram of the hardware of each joint, including the Safety Board, the WAM PC and the CAN bus. All joints are connected to the CAN bus in the same way.

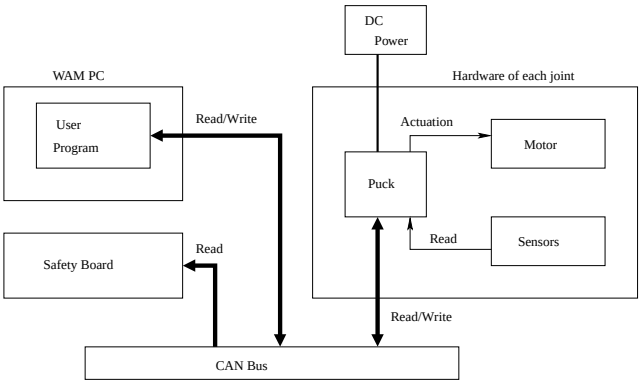

Figure 3: Block diagram of the WAM hardware.

There are three safety states:

E-STOP: there is no voltage applied to the motors. Actually, the motor power lines and the ground lines are connected together, which results in a resistive brake on the joints. Effectively, the arm do not exercises any force, but slightly resists to any force applied to it, resulting in a slow fall down until a physical opposition is reached. Also, it the WAM can be easily moved by the operator to any desired position. In this state it is no possible to communicate with Pucks and they do not control the associate actuators, hence there is no way to obtain data from sensors or send control signals.

IDLE: there is voltage applied to the motors and the Pucks are accessible. They control the position of motors and sustain a brake state, ignoring any torque command. Hence, in this state it is possible to obtain data from sensors, but it is not possible to actuate the WAM.

ACTIVATED: the Pucks are effectively applying any received torque command to the motors. The WAM is ready for motion. This state can only be reached if there is not any fault detected by the Safety Board.

4.2. Software overview

There is a library, `libbarrett` [16], for creating software for the WAM. That library is written in C++ and is available in source code. The main classes and functions are presented here.

The `ExecutionManager` class supervises all real time operations. It is responsible for the program execution cycle, usually defined as 500 Hz.

Through the `wam` class the user can access the functionalities of the WAM. It implements `get()` function for the reading of position, velocity and torque applied to each joint, as well as the Cartesian position of the hand. Those functions return vectors of `jptype`, `jv_type`, `jt_type` and `cp_type` types, respectively. Member functions to move the arm in Cartesian space are `moveHome()` and `moveTo()`, that move to arm to the home position or to a specific point, respectively. The `trackReferenceSignal()` member receives joint references in position and velocity and tracks them by using a controller implemented in the `libbarret` library. Another very useful member function is `gravityCompensate()`, which computes the torque needed for gravity compensation. The controller used by the member functions of the `wam` class is the PID controller implemented by the `PIDController` class.

The `wam` class has a torque input object. The torque effectively applied to the robot is a sum of the torques individually computed by the `PIDController` of each joint for each reference, the torque computed to compensate for the gravity and the torque received by the input object.

The Safety Board is used by the `SafetyModule` class, which monitors the WAM state and exposes the security state of the WAM at any given time.

Finally, there is the `ProductManager` class, which manages the WAM, by initializing hardware components, such as the Pucks, the CAN bus and the Safety Board, and the software components, by creating the `ExecutionManager`, `Wam` and `SafetyModule` objects. It also reads the configuration file associated to the specific robot in use, thus obtain-

ing kinematic and inertial parameters. Therefore, this class is used for starting the control program and initialize the objects for accessing the WAM hardware [17].

5. CONTROL ARCHITECTURE

An architecture for controlling manipulator robots was developed in [1], with the implementation of generic components. Those components are independent from each other and from the hardware of a specific robot. Thus, they can be configured and specialized for the control of any robot system. That architecture is based on generic `Sensor`, `Actuator`, `Controller` and `Sampler` components. In the current work, some of those components are then specialized for the Barrett WAM robot:

`Sampler`: generates the sampling tick for the system, which synchronizes the other components. This component exposes an output port (`SamplePort`), which is used to generate the sampling for the system through a write to this port accordingly to the component activity. All other components requiring synchronization with the system sampling should implement an input port of the `EventPort` type, that should be connected to the `SamplePort` port of the `Sampler` component. This component is not specialized for the Barrett WAN, as the generic one has enough functionalities.

`SensorWAM`: abstracts the sensors of the Barrett WAM robot. It is specialized from the `Sensor` component model, which does not has an implementation, but specifies an interface with an output data port where the values read from sensors should be written. This port is represented by a vector of size N, the number of degrees of freedom. The base model, `Sensor`, has a input port of the `EventPort` type, *SamplePort*, where it receives the sampling of the system. Each write to this port generates an asynchronous call to a callback function, which executes the task of the component. That callback function is a virtual member function of the `Sensor` component, which is implemented in the `SensorWAM` component. In the case of the `SensorWam` component, this task is a request for read the sensors. This component has another data input port of `EventPort` type, `InputSensorPort`, to receive sensor data from lower level hardware interface. After a request for sensor readings, a write to the `InputSensorPort` port forces the execution of a callback, that gets the sensor data and write them to the output data port.

`ActuatorWAM`: abstracts the system actuators. It is specialized from the `Actuator` component model, which has a input data port to receive the values to be applied to the actuators, represented by a vector of size N and a input port to receive the sampling of the system, as done for the `Sensor` component model. The `ActuatorWAM` component adds an output port where, the values for actuation are written in response to a system sampling event.

`ControllerNPID`: implements the system controller, through an independent controller for each joint. It is specialized from the `Controller` component model, which has input ports for references and sensor values, an output for actuation values, and a port of `EventPort` to receive the system sampling event. The `ControllerNPID` concatenates the ports for communication with the N `PID` components.

`ControllerWAM`: implements the lower level control of the system, through the controller available in the `libbarrett` library. This components is specialized from the `Controller` component model and is similar in functionality to the `ControllerNPID` component. However, the `ControllerNPID` component implements a independent joint control with PID controller, while the `ControllerWAM` uses the original WAM controller available through the `libbarrett` library.

`PID`: implements a PID controller for a single joint.

Figure 4 shows how the base components interact with each other to implement a control loop. Note that this architecture is general for any robot. The function and interface of each component as well as details of its operation is presented in details in [1]. Those components are specialized to the components for the Barrett WAM robot and their topologies are discussed in section 7..

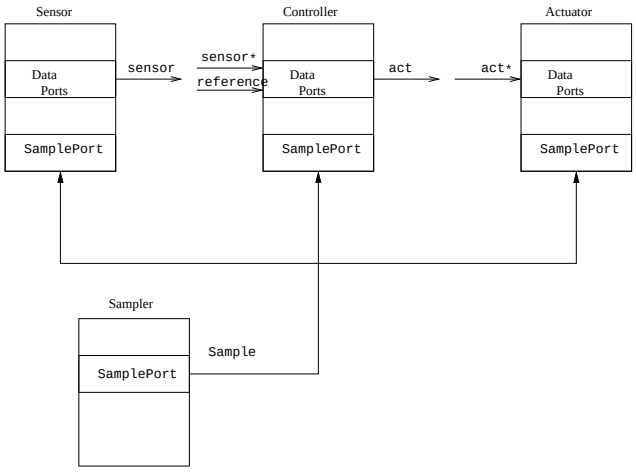

Figure 4: Component Interactions.

A standard OROCOS component, `nAxesGenerator`, is used to generate a reference trajectory for the motion of the joints to the desired point. The `nAxesGenerator` component uses a trapezoidal velocity profile with maximum acceleration and velocity to compute the trajectory from the current N axis position to a desired position in a given time interval. Initial and final velocities and accelerations are null. The motion of all axis are normalized

for simultaneous start and stop. The maximum velocity, maximum acceleration and number of joints are properties of the component, `max_vel`, `max_acc` and `num_axes`, hence configurable. The `moveTo` operation starts the trajectory, receiving as parameters a vector with desired position and the trajectory time and the `resetPosition` operation stops the motion and maintains the current robot position setting the desired position to the current position and the desired velocity to zero. Finally, an input port, `nAxesSensorPosition`, receives the current axis position and two output ports, `nAxesDesiredPosition` and `nAxesDesiredVelocity`, expose the computed position and velocity, respectively.

6. INTERFACING THE BARRETT WAM TO OROCOS

In order to use the WAM robot with OROCOS, there is the need to develop an interface between the robot, or more precisely, the `libbarrett` library and the OROCOS system. That interface assumes the form of OROCOS components whose activities call the functions of the `libbarrett` library.

6.1. `OrocosWam` component

This component is the only one to interact directly with the `libbarrett` library, hence it is the only one specific to the WAM robot. The other components described here do not interact directly with the `libbarrett` library or with the WAM, which shows the generality of the approach and the ease to support other models of robots.

The `OrocosWam` component is composed with objects created from classes defined in the `libbarrett` library, which are used for hardware verification, robot initialization, status verification and motion of the WAM robot. More specifically, objects of classes `ProductManager`, `SafetyModule` and `WAM` are members of the `OrocosWam` component.

Regarding its interface, the `OrocosWam` component has four data ports, which are vectors and a `ClientThread` operation. Two are input ports and two are output ports. The `jointsDataPort` output port exposes the position, velocity and torque data from WAM sensors. The `posRefPort` input port receives the reference position to be used by the controller implemented in the `libbarrett` library, while the `torRefPort` input port receives the torque values to be applied to each joint actuator. The `ControlDataPort` output port exposes the values computed by the controller implemented in the `libbarrett` library. Both input ports of the `EventPort` type. Finally, the `getJointSensorsOperation` operation starts a reading in the WAM sensors.

The initialization of the `OrocosWam` components calls the initialization of the `ProductManager` object, which checks and initializes the WAM hardware. If the hardware initialization succeeds, then the `OrocosWam` component calls its configuration function, `configureHook()`, which forces the `ProductManager` to check for the Safety Board and receive a pointer to the `SafetyModule` object. The `SafetyModule` checks the WAM status, which is logged to the user, and waits for an IDLE status. After the safety initialization, the `ProductManager` checks for the WAM identification and the `configureHook()` function returns.

After the configuration, the `OrocosWam` component executes the `startHook()` function, where the `ProductManager` object initializes the Pucks and the `wam` object by reading the hardware configuration file. The `wam` initialization includes a prompt for the user to manually activate the robot through the control pad, which changes its status to ACTIVATED. Finally, the `gravityCompensate()` function from the `wam` object is called, initiating the gravity compensation. Then, the WAM is ready for use.

Given that the `OrocosWam` component is aperiodic and does not implements the `updateHook()` function, it does not perform any action if not called. Hence, its execution is execution is only due to activity in its operation and its input port. Figure 5 shows the component interface and its internal activity execution.

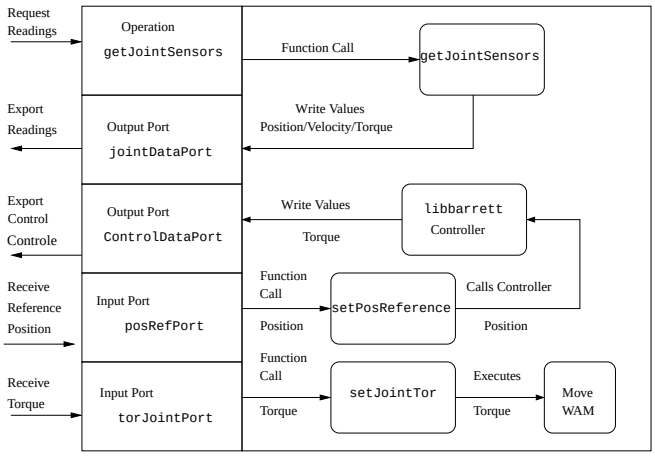

Figure 5: `OrocosWam` component interface and operation.

When the `getJointSensorsOperation` operation is called, the component executes the `getJointSensors()` function, which uses the `get()` function from the `wam` object to obtain position, velocity and torque data for each joint. Those data are written as a vector in the `jointsDataPort` port.

The `OrocosWam` component can use the controller implemented in the `libbarrett` library or not. To use the `libbarrett` controller, the `posRefPort` port should receive position references for each joint. Those values are used by a PID controller implemented in the `libbarrett` library to obtain the control signal and are exported by the `ControlDataPort` port. The values for the actuators

should be written in the `torJointPort` port, which receives the values to be directly applied to each joint.

When the `posRefPort` port is written to, a callback function reads the value from the port and calls the function `trackReferenceSignal()` from the `wam` object with the reference values as parameters. That function uses the internal `libbarrett` controller to compute the torques for each joint, which are then written to the `ControlDataPort` output port.

A write to the `torJointPort` port forces the call of a callback function that checks if the torque values are under the limits for each joint, avoiding dangerous motions. If all values are under the limits, data are sent to the `wam` object, that sends them to the Pucks, which apply the torques to each joint actuator.

When finalizing, the `OrocosWam` component, though its `stopHook()` function, calls the `moveHome()` function from the `wam` object. This function moves the WAM to its home position, avoiding dangerous falls and collisions when the control system is shutdown and no gravity compensation exists any more.

6.2. *`Reference` Component*

This component was created as an interface for specification of the reference position for each joint. The component has the `jointPosition` and `moveTime` properties and the `moveToRefOperation` operation. The `jointPosition` property stores the reference position for each joint and is initialized with the actual position of each joint. The user can them change the desired position an call the `moveToRefOperation` operation to move to robot to the reference position in a time given by the `moveTime` property. The`moveToRefOperation` operation uses the `nAxesGenerator` component to generate a reference trajectory. The default `moveTime` is 5 s, but that can be changed by the user.

7. Implementation of Controllers

Two control topologies were implemented to show the generality of the proposed approach. The first one uses the original position controller implemented in the `libbarrett` library, which is abstracted by the `ControllerWAM` component, while the second one bypasses the `libbarrett` controller and uses the `ControllerNPID` component proposed in [5].

7.1. *Control with the `ControllerWAM` Component*

Figure 6 shows the topology for this controller. The arrows with dashed lines indicate a call operation, while the arrows with solid lines indicate a data flow through data ports. The system sampling is shown by small arrows, to avoid unnecessary detail.

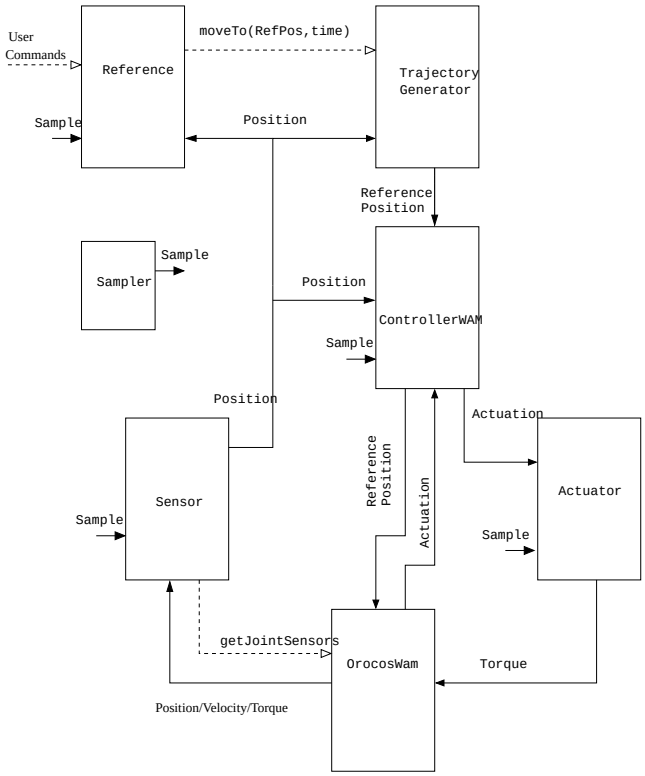

Figure 6: Control with the `ControllerWAM` component.

In every occurrence of a `Sample` event, the `Sensor` component request a reading from the sensors and updates its output port with the values.

To move the WAM, the user should change the `jointPosition` property of the `Reference` component and execute the `moveToRefOperation` operation. The `Reference` component then, uses `nAxesGenerator` component to interpolate a trajectory, sending the values to the `ControllerWAM` component.

The `OrocosWam` component receives the trajectory from the `ControllerWam` component, and uses the `libbarrett` internal controller to compute the actuator values. Those values are sent to the `Actuator` component, which sends them to the `OrocosWam` component, that applies the values to the robot actuators.

The `libbarrett` controller is abstracted in the `ControllerWAM` component and the `Actuator` component abstracts the actuators, while the `OrocosWam` component interfaces with the lower level functions of the `libbarrett` library.

7.2. *Control with the `ControllerNPID` Component*

Figure 7 shows the topology for this controller. From the user point of view, there is no change, as the reference position is manipulated through the `Reference` component, as in section 7.1..

The trajectory generated by the `nAxesGenerator` component is sent to the `ControllerNPID` component, which splits the reference position vector and the vector of current positions received from the `Sensor` component in the reference position and the current position for the PID controller of each joint. Each PID controller computes the actuator value and send it to the `ControllerNPID` component, which concatenates them in a new actuator values vector and sends it to the `Actuator` component. The `Actuator` component sends the actuator values to the `OrocosWam`, where they are effectively applied to the *hardware*. Note that the sensor reading is performed in the same way as in the section 7.1..

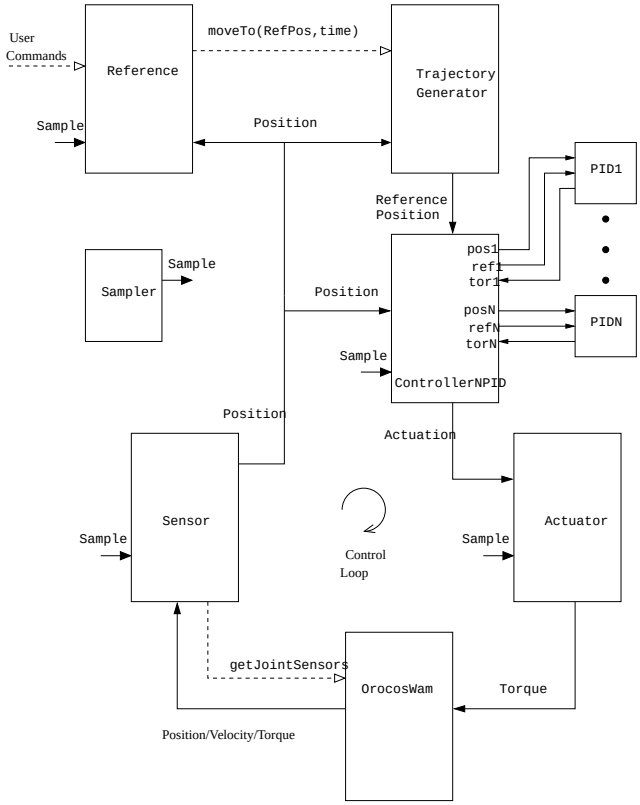

Figure 7: Control with the `ControllerNPID` component.

8. CONCLUSION

This work proposed an interfacing integrating the Barrett WAM robot to the OROCOS framework. A component was created to export some functionalities of the `libbarrett` library through an standard OROCOS interface. That component was integrated in a open architecture defining components for a control system as proposed in [1].

Two control topologies were presented to show the generality of the proposed abstraction of the Barrett WAM robot through the proposed OROCOS component:the first one using the original controller implemented in the `libbarrett` library and the other implementing and independent PID controller for each joint. The same methodology could be used to implement more sophisticated controllers, such as a compute torque controller, just by replacing the controller component, as shown in [5].

The user interface is very modular, as well. It is enough to replace the `Reference` component by another one, with the same standard interface.

REFERENCES

[1] D. C. Santini and W. F. Lages, "An architecture for robot control based on the OROCOS framework," in *Proceedings of the 4th Workshop on Applied Robotics and Automation*, (Bauru, SP, Brazil), Sociedade Brasileira de Automática, 2010.

[2] D. Brugali and P. Scandurra, "Component-based robotic engineering, part i: Reusable building blocks," *IEEE Robotics and Automation Magazine*, vol. 16, pp. 84–96, Dec. 2009.

[3] Barrett Technology, Inc., Cambridge, MA, *WAM User Manual*, 2011.

[4] H. Bruyninckx, "Open robot control software: The orocos project," in *Proceedings of the 2001 IEEE International Conference on Robotics and Automation*, (Seoul, South Korea), pp. 2523–2528, IEEE Press, 2001.

[5] D. C. Santini and W. F. Lages, "An open control system for manipulator robots," in *ABCM Symposium Series in Mechatronics* (V. J. de Negri, E. A. Perondi, M. A. B. Cunha, and O. Hirikawa, eds.), vol. 4, pp. 490–498, Rio de Janeiro, RJ, Brazil: Associação Brasileira de Engenharia e Ciências Mecânicas, 2010.

[6] A. Brooks, T. Kaupp, A. Makarenko, A. Oreback, and S. William, "Towards component-based robotics," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Edmonton, Canada), pp. 163–168, IEEE Press, Aug. 2005.

[7] M. Y. Jung, A. Deguet, and P. Kazanzides, "A component-based architecture for flexible integration of robotic systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Taipei, Taiwan), pp. 6107–6112, IEEE Press, Oct. 2010.

[8] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics (ICAR'03)*, (Coimbra, Portugal), pp. 317–323, IEEE Press, Jun./Jul. 2003.

[9] M. Henning, "A new approach to object-oriented middleware," *IEEE Internet Computing*, vol. 8, pp. 66–75, Jan. 2004.

[10] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *Proceedings of the IEEE Intermatonal Conferencd on Robotics and Automation Workshop on Open Source Robotics*, (Kobe, Japan), IEEE Press, May 2009.

[11] "The CISST libraries," 2008. `http://www.cisst.org/cisst`.

[12] OROCOS, "Open robot control software," 2002. `<http://www.orocos.org>`.

[13] P. Mantegazza, "DIAPM RTAI for linux: Whys, whats and hows," in *Proceedings of the Real Time Linux Workshop*, (Vienna, Austria), Vienna University of Technology, 1999. `<https://www.rtai.org/index.php?module=documents&JAS_DocumentManager_op=downloadFile&JAS_File_id=31>`.

[14] Xenomai, "Xenomai: Real-time framework for linux," Jan 2012. `<http://www.xenomai.org>`.

[15] "Orocos component builder's manual," 2011. `http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-components-manual.html`.

[16] Barrett Technology, Inc., Cambridge, MA, *Libbarrett Programming Manual*, 2011.

[17] "Libbarrett doxygen documentation," 2011. `http://web.barrett.com/libbarrett/`.