AN ARCHITECTURE FOR ROBOT CONTROL BASED ON THE OROCOS FRAMEWORK

Diego Caberlon Santini¹, Walter Fetter Lages¹

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

Abstract: This work deals with the specification of an open architecture for control of manipulator robots. The architecture is implemented by using the OROCOS framework. The architecture is specified for a generic manipulator robot with $N$ joints, through definition of components which abstract the hardware of the robot. Three different controllers were implemented: an independent PID for each joint and a computed torque controller. The validation is made through the implementation on the Janus robot.

Keywords: Open architecture, OROCOS, Robot control

1. INTRODUCTION

Industrial robots are currently employed in a large number of applications, and thus there is a wide range of configurations, physical shapes and drive systems. The demand for increased performance has led to development of complex controllers, for example, computed torque control as well as feedforward dynamic compensation controllers. Their application on real robots are very restricted because of limitations associated with the architecture of conventional robotic controllers, since each robot manufacturer uses its own proprietary interface and protocols.

A proposed solution to deal with this problem is to retrofit robots, such as done in [1]. However, this is very costly and demands a long time, since major changes are required, which includes the replacement of working parts and the development of all software features.

An open architecture appears as solution to such a problem, as all aspects of design can be changed or by anyone, rather than only by the original manufacturer [2]. Benefits such as reduced cost and time of development are achieved by reusing off-the-shelf hardware and software.

Nowadays, some robot manufacturers [3, 4] are selling what they call "open source software" robots, giving access to the low-level control of motor torques. Those features can be very useful to build complex controllers. However, those robots can not be said to be open architecture robots because they depend on the specific hardware of each manufacturer. It is not possible to use the software of one manufacturer with the robot of another one.

In academic research, many projects have been carried out to develop open architecture controllers. In [5], an implementation of a control architecture for manipulator robots is presented. This implementation is based on the Robot Control Interface (RCI) [6], which is a substrate for implementing new functionalities. However, this approach lacks interoperability and extensibility since the RCI is not a component-based software.

In a component-based software, components can be connected together to achieve the global functionality of the robot. Each component is an executable building block with defined interface and functionality [7].

In [8], a component-based open architecture is presented. Each device of a robot is modeled by a component that implements its functionality and uses the Common Object Request Broker Architecture (CORBA) [9] to exchange data. However, that work does not make use of a common system architecture, such as the Open Robot Control Software (OROCOS) [10], which allows the researchers to focus on the problem of robot control and do not require them to rewrite code to fit the controllers to a communication framework such as CORBA.

A common framework not only defines how the components can interact to each other by means of communication and synchronization mechanisms but also provides infrastructure and functionality to build the system.

In [11], the design of a high-performance robot manipulator built from off-the-shelf components is shown. It was built to fill the lack of standard systems in robotics, on which comparative research could be done. Even though the design procedure and the hardware implementation are fully described, the software does not use an open architecture.

An open architecture is fundamental to enable researchers to exchange results and port implementations from one robot to another one. A common framework would encourage other researchers to use available implementations instead of developing their own implementation for each robot. Eventually, that would lead to a standard library on which robot software could be developed.

The OROCOS project is a component-based framework for robot and machine control. It follows an open source development model [10]. The OROCOS has been successfully used in other projects [12, 13] and has had widespread acceptance in the robotic field. However, OROCOS has a slow learning curve. It can be a little confusing because it has so many features and therefore, it is possible to implement a functionality in many ways.

This work presents an open architecture for robot control using OROCOS. The architecture aims to fill the absence of a widespread open architecture in robotic research and also the absence of standard way to implement controllers in OROCOS.
Unfortunately, OROCOS does not define a policy on how to use the available mechanisms and due to the complexity of the framework, it is not easy for a new user to understand all the details of each mechanism and to decide which one is the best one for each case.

Hence, by defining policies for using the resources of the framework, this paper proposes in the following sections an architecture for robot control based on OROCOS. The proposed architecture is implemented by defining new components to represent the building blocks of a typical robot control system.

2. THE OROCOS FRAMEWORK

OROCOS is an acronym for Open Robot Control Software. It is an open source framework for control of robots and machines [10]. The OROCOS is divided in four libraries written in C++.

The OROCOS Real-Time Toolkit (RTT): a package that is the base of OROCOS, with the abstraction of the operating system and primitives for task communication.

The OROCOS Components Library (OCL): a library with many component models, ranging from models for robot control to models for system debugging and drivers for the underlying hardware.

The OROCOS Kinematics and Dynamics Library (KDL): a library for computing kinematic and dynamic models in real-time. Currently, only kinematic models are implemented and tested. Dynamic models are under development.

The OROCOS Bayesian Filtering Library (BFL): a library for recursive information processing and estimation algorithms based on the Bayes rule, such as the Kalman filter and the particle filter.

In this work, only the RTT and the OCL are directly used. The KDL is indirectly used through OCL and the BFL is not used.

2.1. OROCOS Real-Time Toolkit

The RTT is a middleware that abstracts the operating system and provides facilities for communication and execution of the components of the system. A detailed view of the layers in an OROCOS base application is seen in Figure 1. The operating system abstracts the hardware and the RTT abstracts the operating system, enabling the application to run on different platforms. The RTT provides an interface for creating component models. Note that a component can also use the native libraries of the operating system, although implying a loss in portability.

A component is a basic unit which executes one or more action with an specific activity. Actions can be implemented as a C language function, as a script in its own language or even as a hierarchic state machine. There are four policies for the control of component activities: NonPeriodicActivity, PeriodicActivity, SequentialActivity and SlaveActivity.

Each component inherits a public interface from its base class (TaskContext), which defines primitives for component interactions: Events, Methods, Commands, Properties and Data Port, as seen in figure 2.

An Event can have functions attached to it. Whenever the event is triggered, the attached functions are called. Those functions may be synchronous or asynchronous. Synchronous functions are executed as a part of the Event triggering procedure and hence execute in the same thread that triggered the Event. Asynchronous functions are deferred to be executed as a part of the component activity.

Methods are similar to Methods, but Commands are sent from a component and executed according to the activity of the receiving component, asynchronous with respect to the sender. Commands are queued for execution in the receiving component. Figure 3 explains the difference between Commands and Methods, supposing the interaction between two components with periodic activity with same period. When
A calls a Method of B, it is executed immediately. When A sends a Command to B, the execution is deferred until the scheduling of B.

![Figure 3: Commands × Methods.](image)

Properties are variables that can be read from a configuration file in XML format and therefore can be used to store persistent values such as configuration parameters of components or data that should persist across shutdown and power-up of the system.

In order to use the above primitives, a component should peer with another one. By peering, a component becomes able to access the public interface of its peer, as shown in Figure 2.

The Data Port is a primitive for data exchange and can be configured to use a FIFO buffer or not. Also, they can be write-only, read-only or read-write and are accessed in real-time in a thread-safe fashion. Hence, while reading a Data Port, a mutual exclusion procedure ensures that data will not change until the end of the reading operation. Data Ports can also be configured to trigger the activity of a component or to execute a function upon reception of data. Of course, to exchange data, Data Ports should be connected to each other.

The ExecutionEngine block, shown in Figure 2, is executed according to the activity of the component and implements the processing of the scripts, Commands, Events and State Machines associated to the component.

### 2.2. The OROCOS Components Library

The component models in OCL offer functionalities such as: hardware access, debug and log generation tools. This work uses only basic components of OCL, described in the following.

#### TaskBrowser

The TaskBrowser is a component used to interact with other components. Each OROCOS component inherits a standard interface through the TaskContext base class, which is used by the TaskBrowser to interact. It works like a text console, receiving command lines and executing them.

The TaskBrowser can dynamically connect as a peer of any other component and creates Data Ports similar to the component under visit and connects to them. This way, it can send data and Commands, trig Events and execute Methods of any component in the system.

#### DeploymentComponent

The DeploymentComponent is a component used to load and configure other components from descriptions in XML files, as shown in Figure 4. First of all, the command `import` is used to load the component libraries. Then, the components are instantiated through the `loadComponent` command.

![Figure 4: Initialization of the system by the DeploymentComponent.](image)

The system configuration is performed by connecting the components with their peers and their Data Ports. With all connections done, the Properties of each component are updated from the respective XML file. The component activity scheme can be configured in this phase and new scripts and State Machines can be inserted in the ExecutionEngine of each component. At the end, a user hook to be called at the component activity can be installed.

Finally, the `start()` of the component is called and the component enters the running state.

#### ReportingComponent

The ReportingComponent is used to monitor and capture data exchanged by OROCOS components. It works by connecting as a peer of the components to be monitored and attaching to the specified Data Ports. Nonetheless, the ReportingComponent is a virtual component and therefore can not be used directly. It is used through derived components: FileReporting which implements data logging...
in a file and ConsoleReporting which implements data logging on the console.

**nAxisGeneratorPos**

This component model is a trajectory generator. It generates a trajectory from the current position in joint space to the desired one. It has the following Properties: number of joints of the robot, maximum velocity and acceleration of each joint. The trajectory is generated by using a trapezoidal velocity profile, which considers the maximum velocity and acceleration of each joint. The motion of all joints are normalized to move for the same time.

Three Data Ports are used. A read-only port to receive a vector with the current position and two write-only ports where the position and velocity of the generated trajectory are available.

The `nAxisGeneratorPos` component has a `moveTo(pos,time)` command. This command generates a new profile of position and velocity for the joints. The trajectory starts in the current position of the joints and ends at the position specified by the `pos` argument. The trajectory takes the time specified by the `time` argument.

The component has a method called `resetPosition`, which implements an emergency stop. Contrarywise to a command, a method is always executed in a synchronous fashion to the caller. This method sets the trajectory position to the current position and the trajectory velocity to zero, forcing the robot to a stop.

Figure 5 shows the position profile generated for a robot with two joints for the following sequence of actions: a `moveTo` command is sent in \( t = 15 \) s, another `moveTo` is sent in \( t = 40 \) s, the `resetPosition` method is called in \( t = 50 \) s and finally, another `moveTo` command is sent in \( t = 60 \) s. The maximum velocities are limited to \( 0.6 \text{ rad/s} \).

As shown in the above sections, the OROCOS framework provides many possibilities for interaction and communication among components and therefore, the user has to choose what communication mechanism to use for implementing his system. Many of such mechanisms are similar and could be used to replace the others, if they were not available. However, each of the communication mechanisms is best tailored for some type of communication. Unfortunately, OROCOS does not define a policy on how to use the available mechanisms and due to the complexity of the framework, it is not easy for a new user to understand all the details of each mechanism and to decide which one is best for each communication task.

Hence, by defining policies for using the resources of the framework, this paper proposes an architecture for robot control based on OROCOS. The proposed architecture is implemented by defining new components to represent the building blocks of a typical robot control system.

3. **PROPOSED ARCHITECTURE**

A typical topology for a control system is shown in Figure 6. This system can be generalized by including the error computation (the difference between the reference and the measurement) in the controller, which would then receive the signal from sensors and act on the plant through actuators.

![Figure 6](image)

For a computer based control system two more blocks are needed, as shown in Figure 7. These block work as bridges between the continuos time plant and the discrete time controller, by sampling the sensor signal and by holding the computer generated control signal until the next control cycle. This topology is the base for the proposed architecture.

![Figure 7](image)

The dynamic model of a manipulator robot can be represented by [14]:

\[
\tau = D(q)\ddot{q} + H(q, \dot{q}) + G(q)
\]

where \( D(q) \) is the generalized inertia matrix, \( H(q, \dot{q}) \) is the vector of centrifugal and Coriolis forces, \( G(q) \) is the vector of gravity forces, \( q \) is the vector of angular joint positions and \( \tau \) is the vector of torques allied on joints.
Expression (1) can be rewritten as:

\[ \ddot{q}_i = f_i(q_1, \ldots, q_N, \dot{q}_1, \ldots, \dot{q}_N, \tau_i) \]

or

\[ \dot{x} = f(x, \tau) \]

with

\[ x = \begin{bmatrix} q & \dot{q} \end{bmatrix}^T \]

The architecture proposed in this work is based on component models for the blocks of the system shown in Figure 7. Each component model can then be instantiated in a component by using the specific parameters of a given robot. Figure 8 shows the models which are used as the base of the architecture. In Section 3.2, those models are instantiated for an actual robot.

The component models Sensor, Controller and Actuator, are models of base components and specify interfaces which enforce polices on how to use the services of the OROCOS framework. By using those models as a base for the components representing a specific manipulator, the user does not have to deal with the details of the communication mechanisms of the framework and is free to concentrate in the details on how to specify the parameters of his specific manipulator.

The Sampler is the component that generates the sampling rate of the control loop and synchronizes the other components. It generates an Event which is received by the other components, automatically triggering a control cycle. Generally, the Sampler is configured to generate periodic Events, but it could be configured to generate aperiodic Events as well. Each component of the system is registered as a peer to the Sampler in order access its events interface.

The Sensor model abstracts the sensors of the system. It has a Data Port where it writes the values which are read from the sensors. This Data Port is represented by a vector which size depends on the number of joints of the robot, given by (2).

\[ \text{sensor} = \begin{bmatrix} q_1 & q_2 & \cdots & q_N & \dot{q}_1 & \dot{q}_2 & \cdots & \dot{q}_N \end{bmatrix}^T \] (2)

The Sensor component model has a virtual member function called \text{sample\_now()}, which is called each time a Event from the Sampler is received. This function should read the actual sensors of the robot and write the data on the component Data Port. By default, this function just sets a flag indicating that an Event from the Sampler was received. Of course, given an actual robot, the Sensor component model would be derived in a specialized sensor component model which would overwrite the \text{sample\_now()} function to actually read the sensors.

The Actuator component model abstracts the system actuator. Similar to the Sensor component model it has a virtual member function called \text{sample\_now()} which is called in response to a Event from the Sampler. However, his Data Port has a different behavior. It is a read-only port where the component can read the vector of actuator values from. Those values should be applied to the robot. Furthermore, the Data Port of this component is configured to trigger an Event whenever a data is written (by an external component) to it. This behavior is signaled by an \* in the port called act*. This Event is catch by the Actuator component itself which asynchronously calls a virtual member function called \text{act\_now()}. This function is responsible for actuate the robot with the values just written to the Data Port. Again, given an actual Robot, the Actuator component model would be derived in a specialized actuator component model which would overwrite the \text{act\_now()} function to actually actuate the robot.

The Controller component model abstracts the system controller. It has three Data Ports, similar to the signals that the controller in Figure 7 is connected to. The reference and sensor* Data Ports are used for reading the reference and sensor vectors, respectively. Note that a write to the sensor* Data Port triggers a call to a virtual function called \text{controller\_now()}, which should compute the control signal and write it to the act Data Port. Once again, this model would be derived in an specialized controller for the actual robot. Similar to the preceding component models, this component model has a virtual function called \text{sample\_now()} which is connected to the Event from the Sampler.

### 3.1. The Joint Component Model

In order to represent each joint of a robot there is a Joint component model. A Joint component should be associated with another component representing the hardware of the joint. However it does not represent the hardware of the joint, but just the joint itself. This way, the concept of a joint of the manipulator is kept independent of the supporting hardware, enabling all joints to have a common interface even if their
hardware is not the same. Figure 9 shows the interface of this component model.

![Diagram of Joint component model](image)

When a Joint component is created, it should be connected to a component representing the hardware of the joint. The component representing the hardware should provide the following methods:

- **actuatorOn**: turns the joint actuator on
- **actuatorOff**: turns the joint actuator off
- **actuatorSet**: apply value to the joint
- **sensorRead**: read the sensors of the joint
- **brakeApply**: apply the brakes to the joint
- **brakeRelease**: release the brakes.

Each Joint component has the following properties, which describe the hardware of the joint:

**PeerName**: Name of the component which implements the hardware access.

**Brake**: Indicates if the joint has a brake.

**ActuatorSign**: Used to compensate for wiring difference on the joint hardware, such that when an positive value is applied, the joint moves in the positive direction.

**SensorSign**: Similar to ActuatorSign, but used to compensate for sensor wiring, such that a positive value is read when the joint moves in the positive direction.

**GearRatio**: Ratio between the joint motion and the sensor readings.

**InitialPos**: Initial position of the joint.

**Sampler**: Indicate whether the Joint component should try to connect to the Sample and Act Events.

The Joint component model exports methods which are similar to the ones it imports from a peer called PeerName. This way, it is possible to command the joint of the robot without to command the hardware directly. The motorOn and motorOff methods are directly mapped on the imported actuatorOn and actuatorOff methods. The same occurs with the methods used to handle the brakes. However, they are only exported if the Brake property confirms that the specific joint has a brake. The motorSet method is mapped onto the actuatorSet method by compensating for the wiring according to the ActuatorSign property. Finally, the sensorRead method returns the displacement of the joint of the robot, compensating for the wiring according to the SensorSign property and the reduction between the joint axis and the sensor axis by using the GearRatio property.

There are three Data Ports for the Joint component to connect to the PeerName component and to publish the status of the joint. The VoltageActuator Data Port is used to send a voltage to the component PeerName apply to the actuator, while the Voltage Data Port is used to receive the value the Joint should apply to its actuator. Note that a write to this Data Port should be mapped to a call to the actuatorSet method of the hardware component.

The DisplacementSensor Data Port receives the displacement of the motor axis from the PeerName component. A write to this port triggers an event which computes the displacement of the joint, by using the GearRatio and SensorSign properties and updates it by writing to the Displacement Data Port. The displacement of the joint is also integrated with the initial condition given by the InitialPos property and updated by writing to the Position Data Port. In a similar way, the value in the IndexSensor Data Port received from the PeerName component is used to update the value made available at the Index Data Port. The sensorRead method imported from PeerName should update the IndexSensor and DisplacementSensor Data Ports.

The Sample Event triggers a call to the encoderRead method, which updates the Displacement, Position and Index Data Ports. The Act Event triggers a read from the Voltage Data Port and a write to the VoltageActuator Data Port. These Events enables the synchronized and parallel reading and actuation of all joints.

### 3.2. Connecting Sensor, Actuator and Joint Components

Since the Joint component represents a single joint of the robot and the interfaces Sensor and Actuator components are based on vectors of variables for the whole robot, there is a need for multiplexing the signals from the
$N$ Joints of the robot to form the vector required by the Sensor. In a similar way, there is need for demultiplexing the vector at the output of the Actuator to form the actuation signal to be applied to each one of the Joints.

The SensorNMux component model (see Figure 10) multiplexes the $2N$ Data Ports to form a vector as given by (2). Hence, the SensorNMux component has $N$ Data Ports for position ($\text{Position}_1^*, \ldots, \text{Position}_N^*$) and $N$ Data Ports for displacement ($\text{Displacement}_1^*, \ldots, \text{Displacement}_N^*$). Again, the * is used to indicate that a write to those Data Ports triggers an Event which stores the written value. After all $2N$ Data Ports have been written to following a Sample Event, the sensor Data Port is updated with the corresponding vector.

The ActuatorNDemux component model, Figure 11, demultiplexes the act vector and writes the values to the $N$ Voltage1, ..., Voltage$N$ Data Ports. Then, if a Sample Event has occurred since the last Act Event, a new Act is generated. This event should force all actuators to drive the hardware at the same time.

The SensorNMux and ActuatorNDemux are generic component models to multiplex and demultiplex data from sensors and actuators, respectively. In the proposed architecture they are connected to with $N$ Joint components as shown in Figure 12. Card$N$ is the component which implements the actual access to the hardware of the robot. The dashed lines indicate the peering of the components, while the solid lines indicate the data flow through the Data Ports.

It is important to note that the component models described above are generic and can be used to implement a control system for any robot, independent of the number of joints.

An instantiation of the models for a robot with two joints is shown in Figure 13. The following components are instantiated from the corresponding component models defined above: Card01, Card02, Joint1, Joint2, Sensor2Mux, Actuator2Demux and Sampler. Note that the controller is not included yet.
3.3. Extending the Controller as an Independent PID for Each Joint

Similar to the Sensor and Actuator component models in Figure 8, the Controller component model should be extended to implement the desired control law. In this section, the Controller component model is extended to implement an independent PID controller for each joint of the robot.

First, a generic PID component model will be implemented. Then, this model is instantiated with different gains for each joint of the robot.

Component Model for a PID Controller

A PID controller with saturation has three gains: the proportional gain $K_p$, the integral gain $K_i$ and the differential gain. The saturation is represented by the values $u$ and $\overline{u}$, meaning the minimum and maximum value for the controller output.

The error value $e(t)$ is the difference between the reference and the output value. The control law is then written as:

$$u(t) = K_p e(t) + K_i \int_0^t e(t) \, dt + K_d \dot{e}(t) \quad \text{for} \quad u \leq u(t) \leq \overline{u}$$
$$u(t) = \overline{u} \quad \text{for} \quad u(t) > \overline{u}$$
$$u(t) = u \quad \text{for} \quad u(t) < u$$

In discrete time form, the PID can be written as:

$$u[k] = u[k-1] + k_p(e[k] - e[k-1]) + k_i(e[k] - 2e[k-1]) + k_d(e[k] - 2e[k-1]) \quad \text{for} \quad u \leq u[k] \leq \overline{u}$$
$$u[k] = \overline{u} \quad \text{for} \quad u[k] > \overline{u}$$
$$u[k] = u \quad \text{for} \quad u[k] < u$$

Figure 14 shows the interface of the PID component model.

Whenever it is activated, this component reads the value from the Ref and Sen* Data Ports, computes the value of $u$, by using (4), and writes it to the Out Data Port.

ControllerNPID Component Model

This component model extends the Controller component model, which is a MIMO controller, to use $N$ SISO PID controllers as implemented by the PID component model. Figure 15 shows the interface of the ControllerNPID component model.

This component model has Data Ports to communicate with $N$ PID components. It demultiplexes the reference and sensor vectors and write the values to the $2N$ Data Ports (Ref1,...,RefN) and (Sen1,...,SenN). Also, it acts as a multiplexer, by collecting the outputs of the $N$ PID components written to the Out1,...,OutN Data Ports, and writing them as a vector to the act Data Port after a Sample Event. The block diagram of this extension is shown in Figure 16.
Figure 17 shows the connections for a system with two PID controllers.

![Diagram of a system with two PID controllers](image)

**Figure 17**: Complete system for a two joint robot with independent PID controllers.

### 3.4. Extending the Controller as a Computed Torque Controller

In this section the Controller component model is extended to implement a computed torque controller [14]. This control strategy is represented by the block diagram shown in Figure 18. The basic idea is to use the robot model to cancel the nonlinearities of the plant, such as gravity and Coriolis and centrifugal forces. By canceling the nonlinearities it is possible to impose the desired dynamic (given the relative degree of the system). Since an exact cancellation is not possible in practice, a PID is included to compensate for the differences between the model and the actual robot and for the disturbances of the system.

For implementing this control strategy it is convenient to rewrite the robot model (1) as:

$$\dot{x} = f(x) + g(x)u$$  \hspace{1cm} (5)

where $f(x) = -D(q)^{-1}(H(q,\dot{q}) + G(q))$ and $g(x) = D(q)^{-1}$.

Then, it is possible to linearize the plant by the following state feedback:

$$u = g^{-1}(x)(v - f(x))$$  \hspace{1cm} (6)

where $v$ is an arbitrary input.

Hence, by replacing (6) in (5), the model becomes linear:

$$\dot{x} = v$$  \hspace{1cm} (7)

It is easy to note that for manipulator robots, $g^{-1}(x)$ is equivalent to the matrix $D(q)$ in (1), which is always non-singular [14].

From Figure 18 it can be seen that the robot input is composed by a component relative to the model and a component relative to the PID, computed by (6). The closed-loop system equation can then be written as

$$D(q)\ddot{e} + K_d\dot{e} + K_p\dot{e} + K_i e = 0$$  \hspace{1cm} (8)

where $e = q - q_d$ and the PID gains can be chosen so that the error converges asymptotically to zero, given that $D(q)$ is positive definite [16].

Figure 19 shows the implementation of the computed torque controller. The PID is the same discussed in section 3.3., which is reused.

![Diagram of a computed torque controller](image)

**Figure 19**: Computed torque controller.

### The Robot Model Component

This component implements the inverse model of the robot (1), given the position, velocity and acceleration of the joints it computes the desired torque of the robot. This component model is similar to Controller, as shown in Figure 20. The Data Port In* receives a vector with the desired position, velocity and acceleration of the robot (generated a nAxisGeneratorPos component) and computes the torque, which is written to the Out Data Port.

### ControllerNPIDCT Component Model

The ControllerNPIDCT extends the Controller component model to implement a computed torque controller with $N$ PID controllers for model mismatch and disturbances compensation. Figure 21 shows the interface of the ControllerNPIDCT component model.
An Architecture for Robot Control Based on the OROCOS Framework
Diego Caberlon Santini, Walter Fetter Lages

The implementation is similar to the ControllerNPID component model. It writes a vector with the robot position, velocity and acceleration to the In_model Data Port, which triggers the robot Model component to compute the robot torque, which is received by the ControllerNPIDCT through the Out_model* Data Port. The torque values are then added to the values received from the PID of each joint through the OutN* Data Port and written to the Act Data Port, to be used by the system actuators.

3.5. Conclusion

This paper presented an architecture for control of manipulator robots based on components. The approach is modular and enables the reuse of components developed earlier, thus shortening the development time and enabling the replacement of system components without the need to understand the whole system. Furthermore, the proposed architecture defines some policies on the use of the resources available from the OROCOS framework, therefore making it easier for beginners to start to use the system.

The flexibility of the architecture was demonstrated by implementing two control strategies: Independent PID controllers and Computed Torque. A feedforward controller was implemented as well, but was not described here due to space limitations.

References


