A COMPONENT-BASED ARCHITECTURE FOR ROBOT CONTROL

Diego Caberlon Santini[*]
diegos@ece.ufrgs.br

Walter Fetter Lages[*]
fetter@ece.ufrgs.br

[*]Federal University of Rio Grande do Sul
Electrical Engineering Department
Av. Osvaldo Aranha, 103
90035-190 Porto Alegre-RS, Brazil

ABSTRACT

This work deals with the specification of an open architecture for control of manipulator robots. The architecture defines policies for the use of the OROCOS framework and is specified for a generic manipulator robot with N joints, through the definition of component models to abstract the hardware and each block of the robot controller. To show its generality, the proposed architecture is used to implement two different controllers: an independent PID for each joint and controller with feedforward compensation. The validation is made through the implementation in real-time on the Janus robot.

KEYWORDS: Open architecture, OROCOS framework, Robot control

RESUMO

Uma Arquitetura Baseada em Componentes para Controle de Robôs
Este trabalho aborda a especificação de uma arquitetura aberta para controle de robôs manipuladores. A arquitetura define políticas para o uso do *framework* OROCOS. A arquitetura é especificada para um robô manipulador genérico com N juntas, através da definição de modelos de componentes para abstrair o *hardware* e cada bloco do controlador do robô. Para mostrar a sua generalidade, a arquitetura proposta é utilizada para implementar dois controladores diferentes: um controlador PID independente para cada junta e um controlador com compensação em *feedforward*. A validação é feita através da implementação em tempo real no robô Janus.

PALAVRAS-CHAVE: Arquitetura aberta, OROCOS, Controle de robôs.

1 INTRODUCTION

The demand for robots with increased performance has prompted the academia to the development of more complex controllers, such as computed torque control or feedforward dynamic compensation. However, their application on industrial robots are restricted because of limitations associated with the architecture of conventional robotic controllers, as each robot manufacturer uses its own proprietary interface and protocols (Kozlowski, 1998).

An open architecture is a proposed solution to such a problem, as all aspects of design can be changed (Ford, 1994). Therefore, benefits such as reduced cost and shorter development time could be achieved by using off-the-shelf hardware and software.

Nowadays, some robot manufacturers (Neuronics, 2010; Barrett, 2010) are selling what they call ”open source software“ robots, whose give access to the lower-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 the manufacturer and it is not possible

Artigo submetido em 10/09/2010 (Id.: 01194)
Revisado em 30/10/2010, 16/02/2011
Aceito sob recomendação do Editor Associado Prof. Geovany Araujo Borges

to use the software of one manufacturer with the robot of another one. A truly open architecture should support the hardware of many robots with minor changes in software.

In academic research, many projects have been carried out to develop open architecture controllers. In Gaspar (2003), an implementation of a control architecture for manipulator robots is presented. This implementation is based on the Robot Control Interface (RCI) (Lloyd, 1992). 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 (Brugali and Scandurra, 2009).

In Liandong and Xinhan (2004), 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) (Object Management Group, 2010) to exchange data. However, that work does not make use of a common system architecture, which would allow 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.

In Smith and Christensen (2009) 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 his own implementation for each robot. Eventually, that would lead to the development a standard library on which robot software could be developed. 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.

The ROS (ROS.org, 2010) and the Microsoft Robot Developers Studio (RDS) (Microsoft, 2008) are popular frameworks among the robotics community. Although both are based on components and offer communication and other mechanisms required for the implementation of robot software, they do not operate in real-time and therefore are not suitable for the implementation of low-level joint control software. Those frameworks are more appropriate for the development of application software for robots.

The OROCOS project (Bruyninckx, 2001) is a component-based framework for robot and machine control. It follows an open source development model, has been successfully used in other projects (Swevers et al., 2007; Tavares et al., 2007) and has had widespread acceptance in the robotic field. However, OROCOS has a slow learning curve. It can be somewhat confusing because it has so many features and therefore it is possible to implement a given functionality in many ways.

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 framework resources, this paper proposes in the following sections an architecture for robot control based on OROCOS. The proposed architecture is implemented by defining new component models to represent the building blocks of a typical robot control system. So this paper approaches a software engineering problem with a focus on control systems.

It is important to observe that the component models are defined in a way to be independent of a specific robot hardware. This way, the user can easily migrate the software from on robot to another one. This is achieved by abstracting the interface with hardware of the robot and defining the remaining of the architecture based on this abstraction.

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 (Bruyninckx, 2001). The framework is based on components. A component model is a concept similar to class in C++, in that component models describe the interface, properties, and behaviors of components, hence every component model is implemented as a C++ class and can be compiled as a dynamic loadable library. Likewise, components are objects that are instantiated from component models with particular characteristic.

Each component model inherits a public interface from the base class (`TaskContext`), which defines primitives for component interactions: Events, Methods, Commands, Properties and Data Port, as seen in Figure 1.

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. Syn-

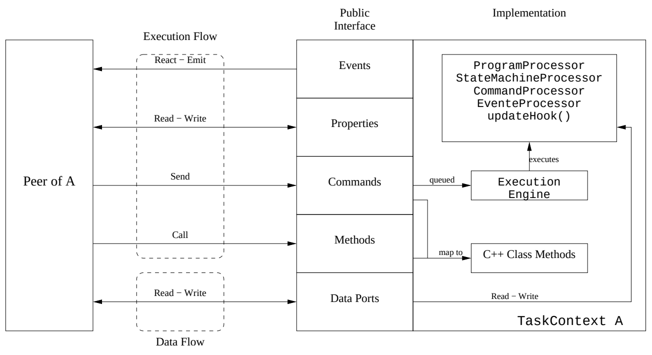


Figure 1: OROCOS component interface.

chronous 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 C++ member functions, but can also be called from scripts. They are executed in a synchronous fashion with respect to the calling component, executing as an usual C++ function.

Commands are similar to Methods, but Commands are sent from a component and executed according to the activity of the receiving component, asynchronously with respect to the sender. Commands are queued for execution in the receiving component. Figure 2 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.

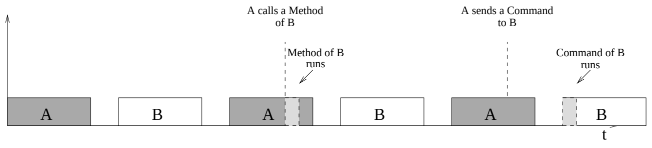


Figure 2: Commands × Methods.

Properties are variables that can be read from a configuration file in Extensible Markup Language (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 1. Note that peering is not a symmetric relation. If a component A is peer of a component B, then A can use the public interface of B, but B can not use the public interface of A unless B is also a peer of A.

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 1, 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.

As shown above, 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 task. 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.

3 PROPOSED ARCHITECTURE

A typical topology for a computer-based control system is shown in Figure 3.

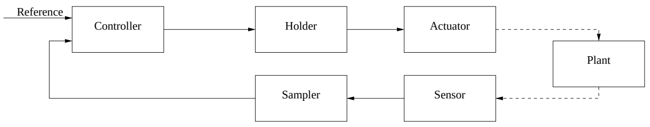


Figure 3: Typical topology for a sampled-data control system.

The dynamic model of a manipulator robot can be represented by Fu et al. (1987):

$$\tau = D(q)\ddot{q} + H(q,\dot{q}) + G(q) \tag{1}$$

where q is the vector of angular joint positions, τ is the vector of torques applied on joint, $D(q)$ is the generalized inertia matrix, $H(q,\dot{q})$ is the vector of centrifugal and Coriolis forces and $G(q)$ is the vector of gravity forces.

For a N-joint robot, expression (1) can be rewritten as:

$$\ddot{q}_i = f_i(q_1,\ldots,q_N,\dot{q}_1,\ldots,\dot{q}_N,\tau_i) \tag{2}$$

where q_i is the position of joint i.

By defining $x = \left[\begin{array}{cc} q^T & \dot{q}^T \end{array} \right]^T$, the model (1) can be represented in a state-space form as:

$$\dot{x} = f(x, \tau) \tag{3}$$

The architecture proposed in this work is based on component models for the blocks of the system shown in Figure 3 and on (3). In other words, the architecture can be used for any system with a diagram block such as Figure 3 and a model such as (3) and not only for robots. Each component model can then be instantiated in a component by using the specific parameters of a given robot or system. Figure 4 shows the models which are used as the base of the architecture. In Section 3.2 those models are instantiated for a generic robot.

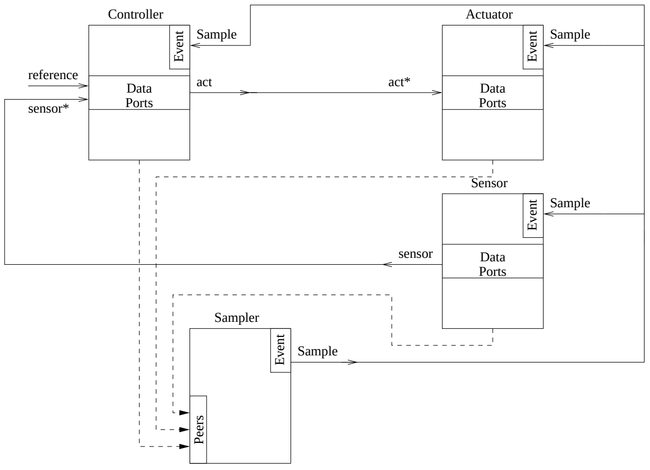


Figure 4: Component models of the proposed architecture.

The component models `Sensor`, `Controller` and `Actuator`, are models for base components and specify interfaces which enforce polices on how to use the services of the OROCOS framework. By using those models as bases 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 to 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 based on the state x of (2) and is represented by a vector which size depends on the number of joints of the robot, given by:

$$\texttt{sensor} = \begin{bmatrix} q_1 & q_2 & \cdots & q_N & \dot{q}_1 & \dot{q}_2 & \cdots & \dot{q}_N \end{bmatrix}^T \tag{4}$$

The `Sensor` component model has a virtual member function called `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 `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 `sample_now()` which is called in response to an Event from the `Sampler`. However, its Data Port has a different behavior. It is a read-only port where the component can read the vector of actuator values from. This Data Port is based on the input τ of (2) and is represented by another vector with variable size given by:

$$\texttt{act} = \begin{bmatrix} \tau_1 & \cdots & \tau_N \end{bmatrix}^T \tag{5}$$

Those values should be applied in each joint of the robot. Furthermore, the Data Port of this component is configured to trigger a virtual member function whenever a data is written (by an external component) to it. This behavior is signaled by an * in the port called `act`*. This virtual 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 virtual member functions 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 3 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 `controller_now()`, which should compute the control signal and write it to the `act` Data Port. Once again, this model would be derived in a specialized controller for the actual robot. Similar to the preceding component models, this component model has a virtual function called `sample_now()` which is connected to the Event from the `Sampler`.

In a typical control system, the cycle begins with a read of sensor and then, this measure is used to compute the control value which is applied in the actuator. The architecture follows the same principle. The `Sample` Event just enables the `Controller` and `Actuator` and does not cause any activity, these components will be activated by their respectively Data Ports: `sensor`* and `act`*. However, the `Sample` Event enables `Sensor` and should then writes the sensor measurements to the `sensor` Data Port. This write will activate the `Controller` which will generate the `act` and will activate the `Actuator`.

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. Hence, the `Joint` component model 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, thus enabling all joints to have a common interface even if their hardware is not the same. Figure 5 shows the interface of this component model.

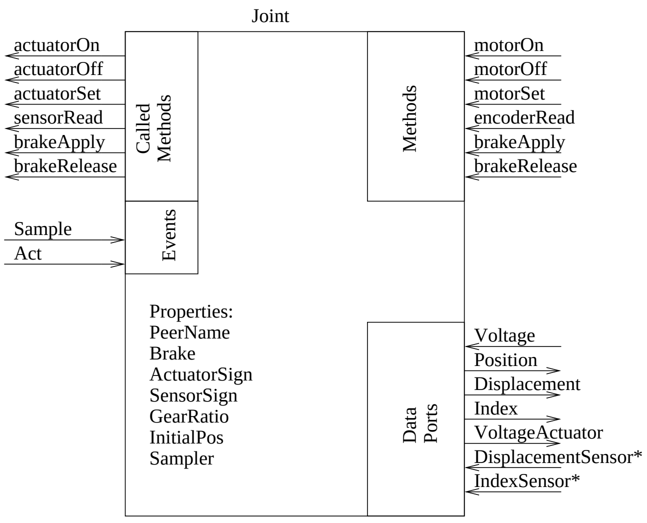


Figure 5: The `Joint` component model.

When a `Joint` component is created, it should be connected to a component which represents the hardware of the joint. The component which represents the hardware should provide the following Methods:

`actuatorOn`: turn the joint actuator on

`actuatorOff`: turn 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 whether 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 specified by the `PeerName` Property. 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 occur 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 component which is specified by the `PeerName` Property and to publish the status of the joint. The `VoltageActuator` Data Port is used to send a voltage to the component specified by the `PeerName` Property to 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 component specified by the PeerName Property. 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 component specified by the PeerName Property is used to update the value made available at the Index Data Port. The sensorRead Method imported from the component specified by the PeerName Property should cause the update of IndexSensor* and DisplacementSensor* Data Ports.

The Sample Event triggers a call to the sensorRead 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 the Base Components

Since the Joint component model represents a single joint of the robot and the interfaces of 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 6), extended from Sensor component, multiplexes the $2N$ Data Ports to form a vector as given by (4). Hence, the SensorNMux component has N Data Ports for position (Position1*, ..., PositionN*) and N Data Ports for displacement (Displacement1*, ..., DisplacementN*). 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 (see Figure 7), extended from Actuator component, demultiplexes the act vector, given by (5), and writes the values to the N Voltage1, ..., VoltageN Data Ports. Then, if a Sample

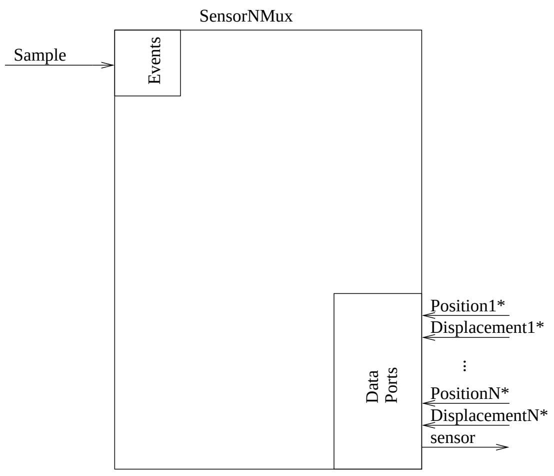


Figure 6: SensorNMux component model

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.

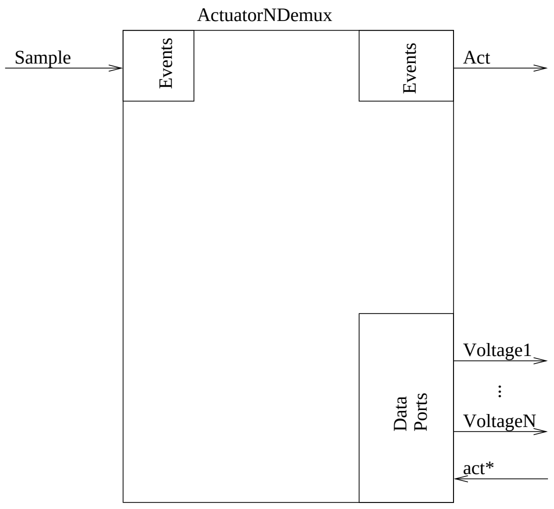


Figure 7: ActuatorNDemux component model.

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 8. CardN 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 system for any robot, independent of the number of joints. Although an electric actuator has been assumed, the architecture is ex-

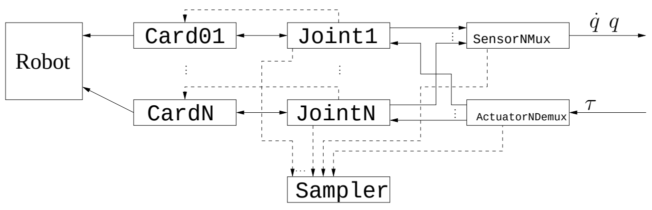


Figure 8: `Actuator`, `Sensor` and `Joint` connections.

tensible to other types of actuators (hydraulic, pneumatic) as long as they can be interfaced with a computer.

An instantiation of the models for a robot with two joints is shown in Figure 9. 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.

The sensor read/actuator write cycle can be described as follows:

1. The `Sample` Event is generated and the `Sensor2Mux` and `Actuator2Demux` components will start to wait for something to be written in their Data Ports: `DisplacementX`* and `Act`* respectively. At the same time, the `Joint1` and `Joint2` components will receive the same Event and will call their own `encoderRead` Methods.

2. Each `Card` component will execute the `encoderRead` Method which will write the appropriate displacement and index status into the Data Ports of each `Joint` component. This will cause the update of `Position`, `Displacement` and `Index` Data Ports, as explained in Section 3.1, which are connected to Data Ports of `Sensor2Mux` component.

3. After all `DisplacementX`* Data Ports of `Sensor2Mux` are updated, the `sensor` is written to and the sensor read cycle is done.

4. At other side, when a data is written to the `Act`* Data Port of `Actuator2Demux`, the `Voltage1` and `Voltage2` Data Ports, which are connected to `Voltage` Data Ports of `JointX` components, are updated and the `Act`* Event is generated.

5. Again, the `JointX` components catch this Event and call their own `motorSet` Methods, which actually actuate its joint, concluding the actuator write cycle.

3.3 An Independent PID for Each Joint

Similar to the `Sensor` and `Actuator` component models in Figure 4, 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. This scheme treats each joint of the robot as a simple joint servomechanism, thus neglecting the coupling among joints.

To implement this control strategy, at first, a generic `PID` component model will be implemented. This component model can be instantiated with appropriate gains for each joint of the robot. Then, the `Controller` will be extended to interact with the N `PID` components.

3.3.1 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 K_d. The saturation is represented by the values $\underline{u}$ and $\overline{u}$, meaning the minimum and maximum value for the controller output. In discrete time form, the PID can be written as (Astrom and Wittenmark, 1984):

$$\begin{cases} u[k] = u[k-1] + k_p(e[k] - e[k-1]) & \\ \qquad +k_i e[k] + k_d(e[k] - 2e[k-1] & \\ \qquad +e[k-2]) & , \underline{u} \leq u[k] \leq \overline{u} \\ u[k] = \overline{u} & , u[k] > \overline{u} \\ u[k] = \underline{u} & , u[k] < \underline{u} \end{cases} \tag{6}$$

where $e[k]$ is is the difference between the reference and the output value.

Figure 10 shows the interface of the PID component model. Note that, by defining the PID parameters as Properties, different PID components can be instantiated from the same model. The component is activated when a value is write to its `Sen`* Data Port. Then, the error value $e[k]$ is computed as difference between `Ref` and `Sen`* Data Ports, and by using (6), u is computed and written into the `Out` Data Port. In the end, the values of $e[k-2]$ and $e[k-1]$ are updated to $e[k-1]$ and $e[k]$, and the component will be waiting for another write to its `Sen`* Data Port.

3.3.2 `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 11 shows the interface of the `ControllerNPID` component model.

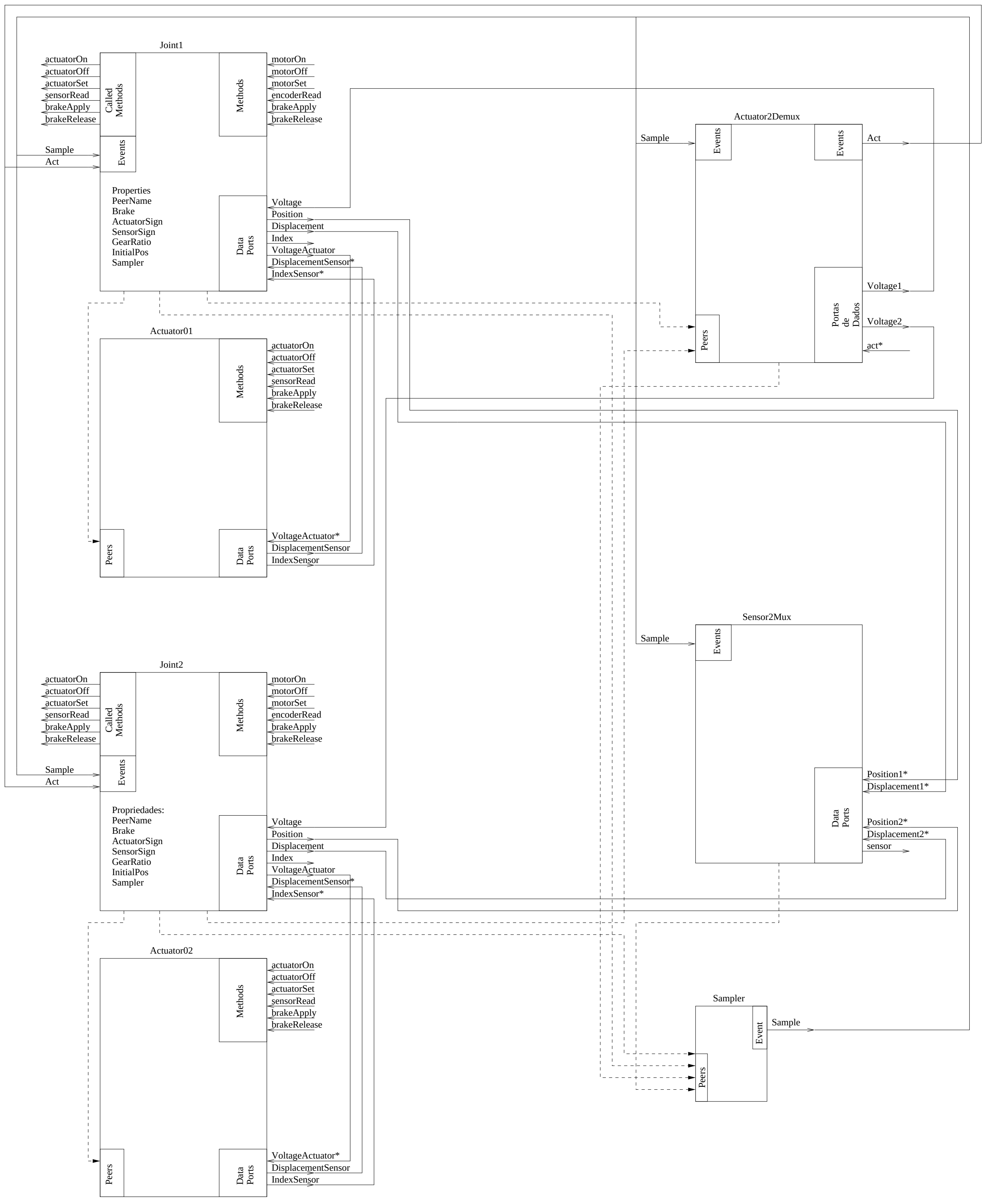


Figure 9: Instantiation for a robot with two joints.

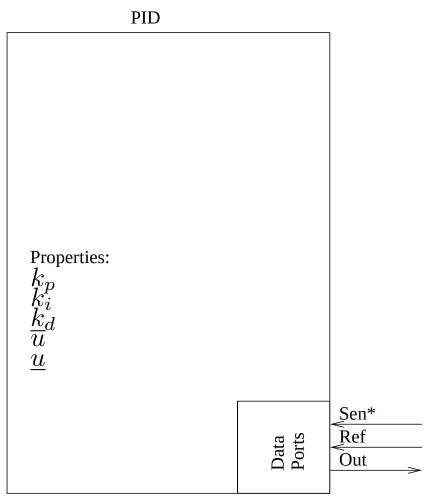


Figure 10: `PID` component model.

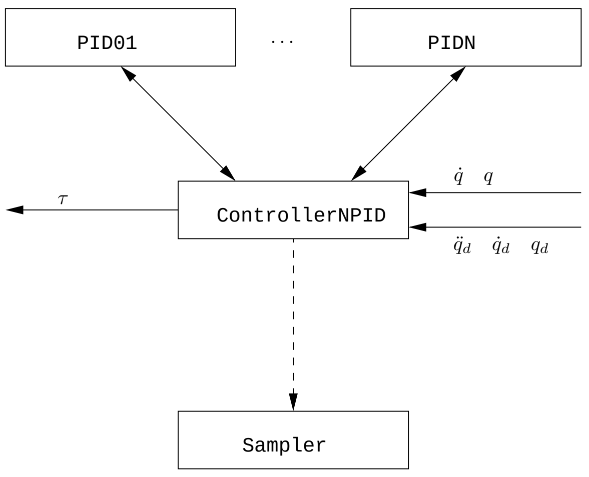


Figure 12: Block diagram of the `ControllerNPID` extension.

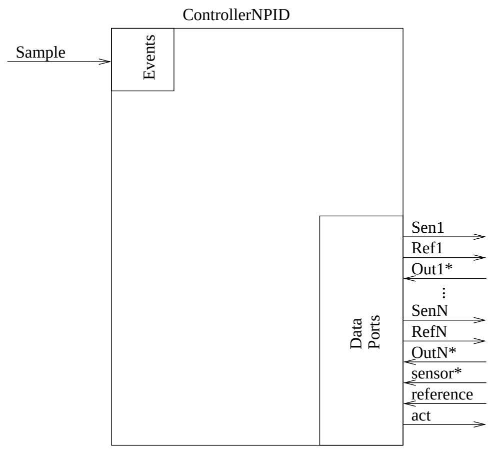


Figure 11: `ControllerNPID` component model.

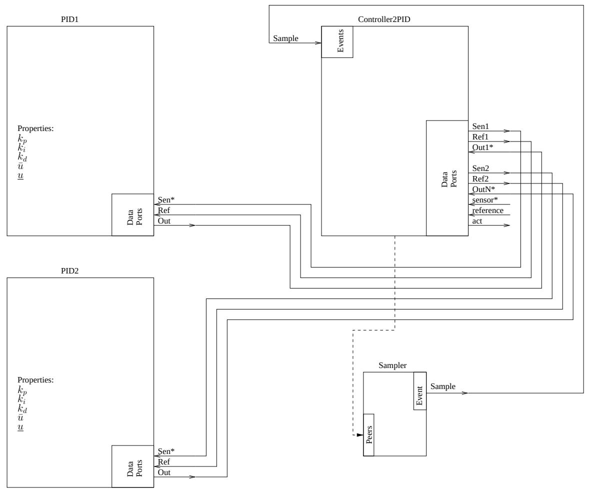


Figure 13: Complete system for a two joint robot with independent PID controllers.

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 12.

Figure 13 shows the connections for a system with two PID controllers. The control cycle of Figure 13 can be described as follows:

1. The `Sample` Event is generated and the `Controller2PID` component starts to wait for something to be written to its `sensor`$*$ Data Port.
2. When a write to `sensor`$*$ occurs, the `Controller2PID` reads its `reference` and `sensor`$*$ Data Ports, demultiplexes its into `Ref1`, `Ref2`, `Sen1` and `Sen2`. Then, it waits for all of its `OutX`$*$ Data Ports to be updated.
3. Each PID component react to the write of each `Sen`$*$ and update each `Out` Data Port.
4. When `Out1`$*$ and `Out2`$*$ are updated, the `Controller2PID` multiplexes this two Data Ports and writes the `act` vector.

3.4 PID with Feedforward Compensation

An independent PID for each joint strategy may work at low speeds, but generally present a poor performance. These problems arise because the coupling among joints and gravity forces are neglected. Better results can be achieved by explicit exploring the knowledge about the dynamic model of the robot.

In this section the `Controller` component model is extended to implement a PID control with feedforward compensation, similar to the PD with feedforward compensation presented in Craig (1989). However, here an integral term is included to reduce steady-state and modeling errors. This control strategy is represented by the block diagram shown in Figure 14. This strategy consists of a PID feedback plus a feedforward computation of the the nominal robot dynamics (1) along the desired joint position trajectory.

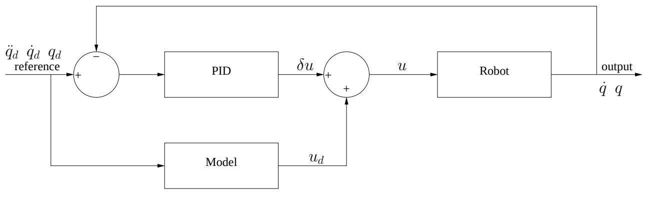


Figure 14: Controller with feedforward compensation.

It is convenient to rewrite the robot model (1) as:

$$\dot{x}(t) = f(x(t), u(t)) \tag{7}$$

If the desired states $x_d(t)$ are known along the desired trajectory, then the torque $u_d(t)$, are also known, can be computed by (1) and satisfy:

$$\dot{x}_d(t) = f_a(x_d(t), u_d(t)) \tag{8}$$

where f_a is the robot model.

The result of (7) minus (8) is

$$\delta\dot{x}(t) = h(\delta x(t), \delta u(t)) \tag{9}$$

where $\delta x(t) = x(t) - x_d(t)$, $\delta u(t) = u(t) - u_d(t)$ and $h(\cdot,\cdot)$ represents the difference between the robot $f(\cdot,\cdot)$ and the robot model $f_a(\cdot,\cdot)$.

If the state $x(t)$ is close to desired state $x_d(t)$, then a PID control law, as defined by (6), can be used to make $\delta x(t)$ converge towards zero.

3.4.1 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 shown in Figure 15 and its internal working is similar to the `PID` component model, however it computes (1) instead of (6). The Data Port `In`* receives a vector with the desired position, velocity and acceleration of the robot and computes, by using (1), the torque, which is written to the `Out` Data Port.

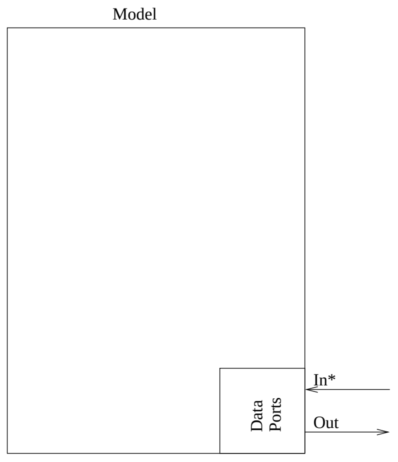


Figure 15: Robot `Model` component.

3.4.2 `ControllerNPIDFF` Component Model

The `ControllerNPIDFF` extends the `Controller` component model to implement a PID with feedforward compensation. Therefore, in order to implement the PID controller with feedforward compensation it is only necessary to change the component model that `Controller` is instantiated from to `ControllerNPIDFF`. Figure 16 shows the interface of the `ControllerNPIDFF` component model.

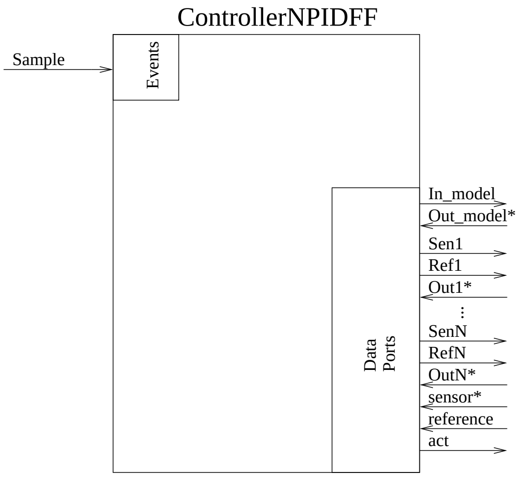


Figure 16: `ControllerNPIDFF` component model.

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 ControllerNPIDFF through the Out_model* Data Port. Then it writes the N values of sensor to its SenX Data Ports, which triggers the PID component. In the end, the torque values are them 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.

4 EXPERIMENTAL RESULTS

The architecture presented in Section 3 was implemented in the vision system of Janus robot. It consists of two links connected in series by two rotational joints. Two cameras are attached to the far-end of the chain. Each joint is driven by a DC motor and has an incremental quadrature encoder and a reference index inductive sensor.

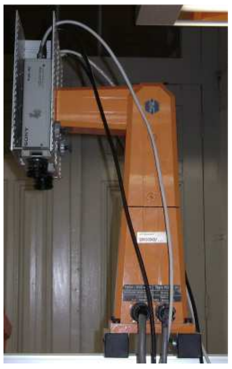


Figure 17: The vision system of Janus.

Each joint also has an actuator card which uses a CANbus with 1Mbit/s for real-time data transfer to a host PC. More details about the hardware and software for this card can be found in Santini and Lages (2008). This card, named as AIC, is modeled as a component which has an interface compatible with the one defined by Joint component model, as shown in Section 3.1.

4.1 Independent PID for Each Joint

Figure 18 shows the block diagram for an independent PID controller for each joint. Each component of Figure 18 is instantiated from a component model as shown in Table 1 and all peering and Data Port connection as shown in Figures 9 and 13 are performed. The hatched components have their execution in real-time and the arrows represent the information flow between components.

Table 1: Characteristics of system.

Component	Model	Activity	Priority	Period
Sampler	Sampler	PeriodicActivity	1	$10\ ms$
aic01	AIC	SequentialActivity	—	—
aic02	AIC	SequentialActivity	—	—
Joint1	Joint	SequentialActivity	—	—
Joint2	Joint	SequentialActivity	—	—
Sensor	Sensor2Mux	NonPeriodicActivity	2	—
BridgeRef	BridgeRef	NonPeriodicActivity	97	—
Generator	nAxisGenPos	PeriodicActivity	98	$10\ ms$
Controller	Controller2PID	NonPeriodicActivity	3	—
pid1	PID	SequentialActivity	—	—
pid2	PID	SequentialActivity	—	—
Actuator	Actuator2Demux	NonPeriodicActivity	3	—
Reporter	FileReporting	PeriodicActivity	100	$10\ ms$
TaskBrowser	TaskBrowser	NonPeriodicActivity	255	—

In order to test the proposed architecture, a sequence of commands, given by Table 2, is executed. Each command is sent at its respective time. The desired and measured position for each joint, are shown in Figure 19.

Table 2: Sequence of test commands.

Time	Command
5s	moveTo(array(1.0,1.0),10)
17s	moveTo(array(2.0,2.0),9)
28s	moveTo(array(3.0,1.0),8)
38s	moveTo(array(1.0,3.0),7)
47s	moveTo(array(4.0,4.0),5)
54s	moveTo(array(0.0,0.0),5)

To evaluate the computing performance of the proposed architecture, the activity of each component in a full control cycle is measured. The Figure 20 shows the timeline of actions started by the Sample Event. As can be seen, the computing of the control loop takes less than $1\ ms$, which is adequate for most robots.

At $t_0 = 0$, the Sample Event is generated by Sampler and the components which are connected to this Event are called in a synchronous way. Figures 9 and 13 shows that Joint1, Joint2, Sensor, Controller and Actuator are connected to Sample Event.

First, the Joint1 executes and calls the SensorRead Method from aic1 card. This process takes about $259\ \mu s$ which is the time of sensing the system using the AIC cards through CANBus. Then, at f_1, the Displacement and Position Data Ports of Joint1 are updated.

After that, the process is done again for the second joint until f_2. So Sensor, Controller and Actuator enable their respective Data Ports: Position1*, Position2*, Displacement1*, Displacement2*, sensor* and act*.

As Joint1 and Joint2 update their Data Ports, which are connected to Sensor Data Ports, the Sensor com-

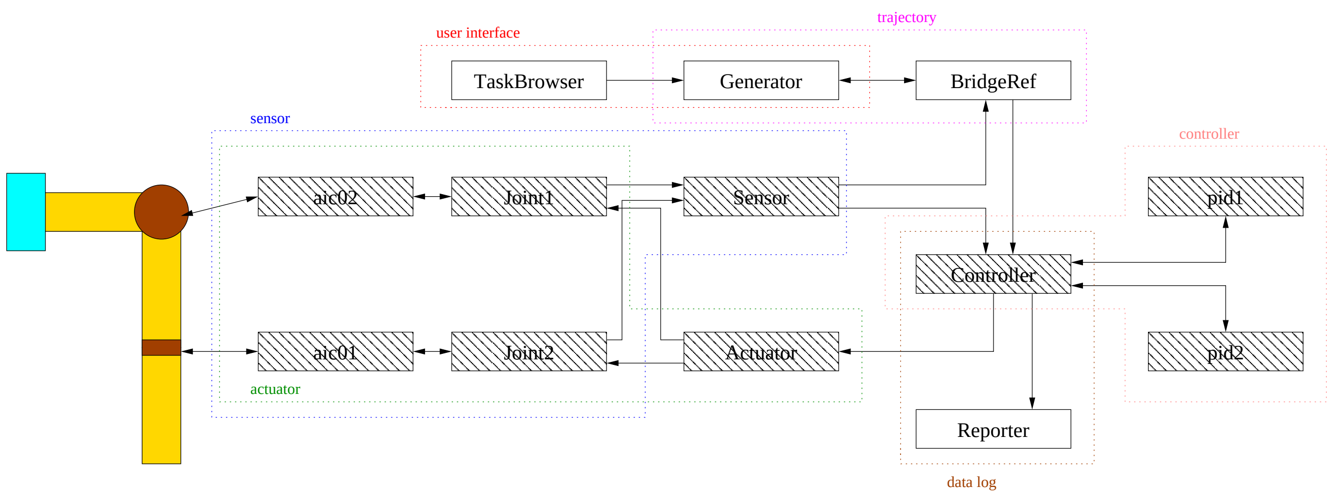


Figure 18: Block diagram for an independent PID for each joint.

ponent is activated and joins Position1*, Position2*, Displacement1*, Displacement2* into sensor Data Port. This process takes about 20 μs and is done at f_4 when Controller is activated.

Until f_5, the Controller splits the data in the sensor* Data Port in position and displacement of each joint, interacts with two PID components and writes the act Data Port to the Actuator. This is done in approximately 30 μs.

Then, the Actuator is activated and splits the act* into the Voltage1 and Voltage2 Data Ports which are connected to Joint1 and Joint2. In the end, the Actuator emits the Act Event at t_5 taking approximately 10 μs.

The Act Event activates Joint1 and Joint2 to effectively actuate over each joint of robot. This takes about 150 μs. Note that the aic01 cards takes less time than aic02 by hardware issues. After f_6, all the real-time component have executed and the other components like, Generator and Reporter can execute.

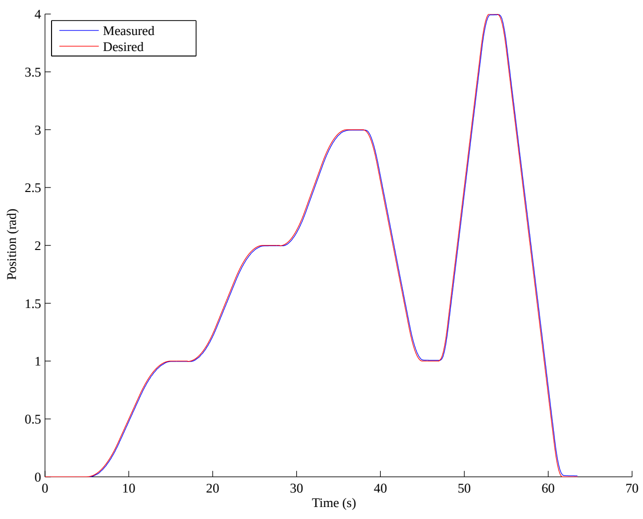
Note that, by using the proposed architecture, the control cycle is performed in 724 μs for two joints, which means about 362 μs for each joint. In (Santini and Lages, 2008) a similar controller was implemented by using a monolithic structure, with a single loop that reads the sensor, computes the control law and drives the actuator. The control loop was performed in 331 μs for a single joint. Hence, the proposed architecture imposes a small overhead on the system. However, to be adapted to another robot, the monolithic implementation requires a rewrite of the program source code, while the architecture proposed here requires just the setting of the joint parameters on its configuration files. There is no need to change the program source code, with the associated complexity and the possibility of introduction of bugs.

4.2 PID with Feedforward Compensation

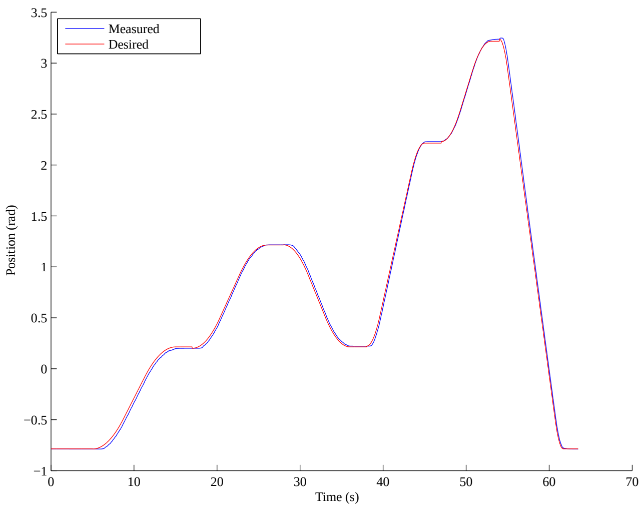
In order to implement the PID controller with Feedforward Compensation, just a minor change is needed on the basic independent PID controller for each joint. It is only necessary to change the instantiation of the Controller component from ControllerNPID to ControllerNPIDFF. Since ControllerNPIDFF uses the model of the robot, as detailed in section 3.4.2, two more components are added to the system, as shown in Table 3.

Table 3: Characteristics of the new components for the PID controller with feedforward compensation.

Component	Model	Activity	Priority	Period
Controller	ControllerNPIDFF	NonPeriodicActivity	3	—
Model	Model	SequentialActivity	—	—

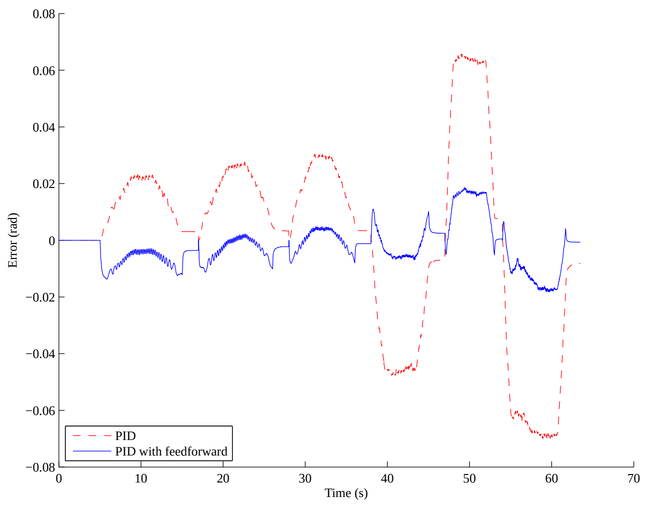


(a) Joint 1.

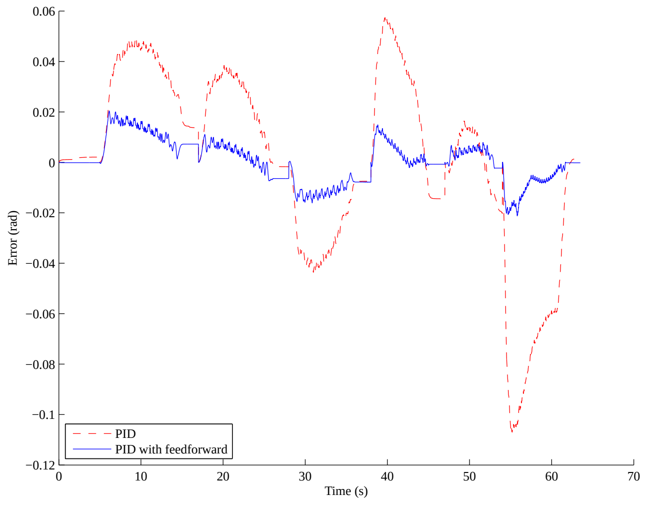


(b) Joint 2.

Figure 19: Position with PID controller.



(a) Joint 1.



(b) Joint 2.

Figure 21: Position error.

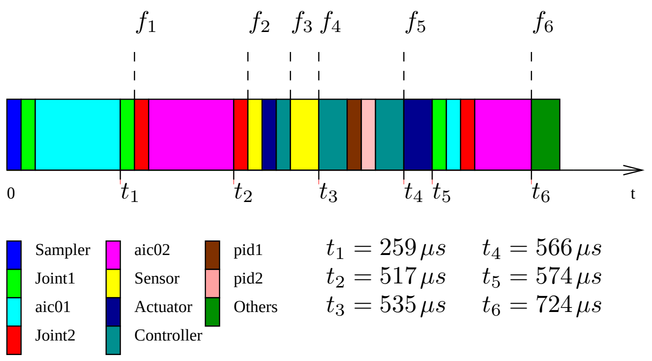


Figure 20: Timeline of action during a control cycle.

The same sequence of commands shown in Table 2 were executed with this new controller. The curves for the desired and measured position can not be visually distinguished from those for the PID controller shown in Figure 19. However, the curves for the position error, shown in Figure 21, reveal that the trajectory tracking performance for the PID with feedforward compensation is better than an independent PID controller for each joint. Nonetheless, since the point here is the architecture for implementation of controllers and not their tuning, it is important to note that no effort was made in order to optimally tune the controllers. Therefore, the relative performance of each controller is valid for the settings used in this particular experiment and should not be regarded as not as representative of that type of controller.

Regarding the timing of the control loop computation, there is a $10\ \mu s$ increase in the computing time, due to the need to compute the robot model. However, a monolithic implementation is subject to the same increase in computing time. In this particular case, that happens because the `model` component has a `SequentialActivity`, which means that it executes in the context of its caller (the `ControllerNPIDFF`), just like a function call in a monolithic implementation.

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 polices 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 with minor changes: Independent PID controllers and a controller with feedforward compensation. The architecture is also independent of robot and can be easily extended to other systems.

The measured times show that the architecture does not introduce significant computing time in the control loop. This allows for the use of the proposed architecture in real-time.

REFERENCES

Astrom, K. J. and Wittenmark, B. (1984). *Computer Controlled Systems: Theory and Design*, Prentice Hall Professional Technical Reference, Indianapolis.

Barrett (2010). Barrett technology, inc. Available: `<http://www.barrett.com/robot/products-arm.htm>`.

Brugali, D. and Scandurra, P. (2009). Component-based robotic engineering (part i) [tutorial], *Robotics & Automation Magazine, IEEE* **16**(4): 84–96.

Bruyninckx, H. (2001). Open robot control software: the orocos project, *Proceedings of the IEEE International Conference on Robotics and Automation*, Vol. 3, Piscataway: IEEE Press, Seoul, pp. 2523–2528.

Craig, J. J. (1989). *Introduction to Robotics Mechanics and Control*, second edn, Addison-Wesley.

Ford, W. (1994). What is an open architecture robot controller?, *Proceedings of the IEEE International Symposium on Intelligent Control*, Piscataway: IEEE Press, Columbus, pp. 27–32.

Fu, K. S., Gonzales, R. C. and Lee, C. S. G. (1987). *Robotics Control, Sensing, Vision and Intelligence*, Industrial Engineering Series, McGraw-Hill, New York.

Gaspar, M. D. (2003). *Uma biblioteca configurável para controle tempo real de robôs manipuladores*, Dissertação (mestrado), Universidade Federal de Minas Gerais, Belo Horizonte.

Kozlowski, K. (1998). *Modelling and Identification in Robotics*, Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Liandong, P. and Xinhan, H. (2004). Implementation of a pc-based robot controller with open architecture, *Proceedings of the IEEE International Conference on Robotics and Biomimetics*, Piscataway: IEEE Press, Shenyang, pp. 790–794.

Lloyd, J. E. (1992). RCI reference manual, *Technical report*, McGill Research Centre for intelligent Machines, McGill University, Montréal.

Microsoft (2008). Robotics developer studio. Available: `<http://www.microsoft.com/robotics>`.

Neuronics (2010). Neuronics AG - linux robot. Available: `<http://www.neuronics.ch/cms_de/web/index.php?id=364>`.

Object Management Group (2010). The corba website. Available: `<http://www.corba.org/>`.

ROS.org (2010). ROS. Available: `<http://www.ros.org>`.

Santini, D. C. and Lages, W. F. (2008). A distributed robot control architecture using RTAI, *Proccedings of the Tenth Real-Time Linux Workshop*, Centro Universitario del Norte, Universidad de Guadalajara, Colotlán, Mexico, pp. 1–5.

Smith, C. and Christensen, H. I. (2009). Constructing a high performance robot from commercially available parts, *Robotics and Automation Magazine* **16**: 75–83.

Swevers, J., Verdonck, W. and De Schutter, J. (2007). Dynamic model identification for industrial robots, *Control Systems Magazine* **27**(5): 58–71.

Tavares, D. M., Aroca, R. V. and de Paula Caurin, G. A. (2007). Upgrade of a scara robot using orocos, *Proceedings of the 13th IASTED International Conference on Robotics and Applications*, Calgary: ACTA Press, Würzburg, pp. 252–257.