UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DARLAN IORIS

INTEGRAÇÃO DO FRAMEWORK OROCOS AO ROBÔ BARRETT WAM

Porto Alegre
2011

DARLAN IORIS

INTEGRAÇÃO DO FRAMEWORK OROCOS AO ROBÔ BARRETT WAM

Projeto de Diplomação apresentado ao Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Engenheiro Eletricista.

ORIENTADOR: Prof. Dr. Walter Fetter Lages

CO-ORIENTADOR: Prof. Diego Caberlon Santini, Msc.

Porto Alegre
2011

DARLAN IORIS

INTEGRAÇÃO DO FRAMEWORK OROCOS AO ROBÔ BARRETT WAM

Este Projeto foi julgado adequado para a obtenção dos créditos da Disciplina Projeto de Diplomação do Departamento de Engenharia Elétrica e aprovado em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Banca Examinadora:

Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Prof. Dr. Marcelo Götz, UFRGS
Doutor pela Universität Paderborn, Alemanha

Prof. Dr. Renato Ventura Bayan Henriques, UFRGS
Doutor pela Universidade Federal de Minas Gerais – Belo Horizonte, Brasil

Chefe do DELET: _____
Prof. Dr. Altamiro Amadeu Susin

Porto Alegre, Dezembro de 2011.

DEDICATÓRIA

Dedico à minha mãe, Ledir Maria Ioris, por todo seu esforço em minha educação.

AGRADECIMENTOS

Agradeço a todos meus familiares e amigos, pelo suporte e companheirismo durante toda minha vida, principalmente no período acadêmico.

Ao meu orientador, Prof. Dr. Walter Fetter Lages, pela orientação e ensinamentos.

Ao meu amigo e co-orientador, Prof. Diego Caberlon Santini, Msc., pela ajuda e orientação.

À UFRGS e em especial ao Departamento de Engenharia Elétrica, pela oportunidade de estudar em uma das melhores instituições do país.

RESUMO

Este documento trata do desenvolvimento de uma interface entre o *framework* OROCOS e o robô Barrett WAM. A interface é desenvolvida através de um componente do OROCOS, o qual é integrado à uma arquitetura aberta para controle de robôs manipuladores, anteriormente desenvolvida. Os componentes desta arquitetura são atualizados para uma versão atual do OROCOS, a fim de manter a compatibilidade com projetos atuais. Para desenvolver a interface, é realizado um estudo do *framework* OROCOS, da arquitetura utilizada e do *hardware* e *software* do robô Barrett WAM.

Palavras-chave: Barrett, WAM, OROCOS, Interface.

ABSTRACT

This document deals with the development of an interface between OROCOS framework and Barret WAM robot. The interface is designed through a OROCOS component, which is integrated to an open architecture for control of manipulator robots, previously developed. The components of this architecture are updated to a new OROCOS version, in order to keep compatibility with current projects. To design the interface, it is done a study about OROCOS framework, the used architecture and Barrett WAM robot hardware and software.

Keywords: Barrett, WAM, OROCOS, Interface.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BFL	*The Orocos Bayesian Filtering Library*
CAN	*Controller Area Network*
CBSE	*Component-Based Software Engineering*
KDL	*The Orocos Kinematics and Dynamics Library*
MH	*Multi-Host*
MP	*Multi-Process*
MT	*Multi-Thread*
OCL	*The Orocos Components Library*
OROCOS	*Open Robot Control Software*
PC	*Personal Computer*
RT	*Real-Time*
RTAI	*Real-Time Application Interface*
RTOS	*Real-Time Operating System*
RTT	*The Orocos Real-Time Toolkit*
XML	*Extensible Markup Language*
WAM	*Wam Arm Manipulator*

SUMÁRIO

1 INTRODUÇÃO

A necessidade de robôs de alta performance requer sistemas robóticos com arquiteturas que exigem *hardware* e *software* cada vez mais complexos. O reaproveitamento de conhecimento e de partes funcionais e existentes, torna a tarefa de projetar sistemas robóticos menos trabalhosa, mais rápida e confiável, uma vez que não é necessário construir todo o sistema do robô. No entanto, a utilização de arquiteturas, em sistemas robóticos, que possibilitem a reutilização de *hardware* e *software* ainda não é amplamente utilizada. Robôs comerciais geralmente possuem uma arquitetura própria do seu fabricante, a qual define sua própria interface, protocolos de comunicação e funcionamento. Isto torna extremamente difícil a sua aplicação em diferentes plataformas de *hardware*, a sua ampliação para sistemas mais complexos e a operabilidade com sistemas de outros fabricantes.

1.1 Arquiteturas Abertas

Para superar tais dificuldades, torna-se necessário a utilização de novas técnicas de desenvolvimento, as quais possibilitem maior integração entre as tecnologias e conhecimentos desenvolvidos por diferentes projetistas.

Em uma arquitetura aberta, todos os detalhes do robô são amplamente documentados e a estrutura de software e hardware é feita de tal forma, que permite a adição de novos sensores, controladores e interfaces, independente dos seus fabricantes. Dessa forma, todos os aspectos de projeto do robô podem ser modificados sem muita dificuldade (SANTINI, 2009). Portanto, busca-se, através de arquiteturas abertas, criar padrões de projeto que facilitem o reúso e a integração de sistemas existentes com sistemas novos.

Arquiteturas abertas devem ser aplicáveis em diferentes sistemas e devem possuir uma implementação que abstraia o *hardware* utilizado, de forma a não serem específicas para um determinado fabricante. Desta forma, toda a parte de controle e gerência pode ser feita independentemente da plataforma de *hardware* utilizada, o que possibilita o reúso de *software* em diferentes aplicações e facilita a ampliação de capacidade em sistemas existentes com mínimas alterações em *software*.

Uma metologia que fornece as ferramentas necessárias para o desenvolvimento de uma arquitetura aberta com tais características é CBSE *(Component-Based Software Engineering)*. A CBSE utiliza o conceito de componente para desenvolver unidades de *software* autônomas, cada uma sendo capaz de abstrair uma determinada parte do *hardware* ou uma funcionalidade, oferecendo suas funções ao restante do sistema através de uma interface padrão. Esta modularidade dada ao sistema através do uso de componentes é a base para uma arquitetura aberta.

Para o desenvolvimento e a utilização de uma arquitetura aberta, é necessário o uso de um *framework* que ofereça as funcionalidades necessárias para a criação de componentes,

tais como um padrão de interface, as formas de comunicação e a troca de dados entre os componentes, através da definição de uma política de criação e uso dos componentes. É importante o uso de um *framework* reconhecido, de forma a manter a compatibilidade com o maior número possível de projetos existentes na área.

1.2 *Component-Based Software Engineering (CBSE)*

A CBSE é uma abordagem que surgiu na comunidade de engenharia de software na última década. Destina-se a mudar a ênfase na construção do sistema de programação tradicional para compor sistemas de *software* com uma mistura de componentes padrões e componentes customizados (BROOKS et al., 2005). A utilização desta abordagem permite a reutilização de componentes de *software* em diversas aplicações, ao invés de se construir um sistema inteiramente novo, o que diminui significativamente a quantidade de código gerado. Além da diminuição do esforço empregado, outros benefícios são evidentes, pois um componente amplamente usado torna-se confiável, robusto, eficiente e bem conhecido quanto as suas funcionalidades, interfaces e limitações.

Tendo-se em vista um projeto complexo que envolva diferentes funcionalidades, pode-se mais facilmente construí-lo utilizando blocos funcionais, a partir do reúso de componentes prontos. Assim, o projetista não necessita conhecer a implementação específica de cada funcionalidade, focando apenas na de seu interesse. Além disso, este sistema torna-se modular, o que ajuda nas dependências de controle e aumenta a flexibilidade do sistema para futuras alterações e manutenções. Os testes no sistema também ficam mais fáceis, uma vez que pode-se testar cada componente separadamente.

Contundo, desenvolvimento de *software* baseado em componentes e a reutilização de código ainda não são práticas amplamente utilizadas na robótica. Hoje em dia a maioria das pesquisas e desenvolvimento de *software* ainda são baseadas em arquiteturas de *software* personalizadas, construídas a partir do zero (BRUGALI; SCANDURRA, 2009). Portanto a maioria das aplicações na robótica são sistemas desenvolvidos com uma finalidade específica, os quais acumulam grande quantidade de *software* que implementam sistemas completos. No entanto, isso não favorece o reúso de *software*, pois este torna-se específico para um determinado *hardware*, sistema operacional ou meio de comunicação, além de que toda funcionalidade e conhecimento está dentro do código, e não exposta de forma clara e organizada em uma interface.

Afim de superar tais dificuldades, a CBSE possuí três principais funções (BROOKS et al., 2005):

- Desenvolver *software* a partir de partes pré produzidas.
- Reutilizar tais partes em diferentes aplicações.
- Ser capaz de oferecer fácil manutenção e customização a estas partes para desenvolver novas funções e funcionalidades.

1.3 Componentes

Segundo COLLIN-COPE (2001), um componente de *software* pode ser identificado como tendo as seguintes propriedades:

- é uma unidade binária (não código fonte) de desenvolvimento.

- implementa uma (ou mais) interface(s) bem definida(s).
- fornece acesso a um conjunto de funcionalidades inter relacionadas.
- deve ter seu comportamento caracterizado de maneira bem definida, sem necessidade de acesso ao código fonte.

Desta forma, componentes podem ser vistos como unidades implementadas de *software* que abstraem os detalhes de implementação e fornecem especificações de funcionalidades e interface bem definidas, o que possibilita o desenvolvimento de variadas aplicações com maior rapidez e confiabilidade do que aplicações sem componentes. A especificação de um componente deve informar quais são as funcionalidades oferecidas aos usuários, as obrigações do usuários ao utilizá-las, o que pode ser apresentado como condições ou modos de uso, e como acessar as funcionalidades e as dependências associadas a elas. Já a implementação de um componente define como ele suporta estes recursos e obrigações, através de objetos (instância de classe) e algoritmos estruturados para implementar as funcionalidades declaradas na especificação do componente.

Separar a especificação da implementação de um componente permite desenvolver um *software* modular e extensível, pois permite a evolução independente entre componentes e o sistema como um todo, sem perda de operabilidade.

Componentes podem ser utilizados em diferentes configurações para formar um sistema. Esta flexibilidade vem do fato de que as interfaces dos componentes são especificadas previamente e cada novo componente deve implementar ou utilizar esta interface. Isto também possibilita os componentes serem desenvolvidos independentemente, pois possuirão interfaces compatíveis. Portanto, em um sistema não há diferença para ele se um componente for substituído ou atualizado, desde que este mantenha o padrão de interface.

1.4 Robótica baseada em componentes

Aplicar os conceitos da CBSE à robótica permite a manipulação e o desenvolvimento de sistemas robóticos complexos, trabalhando com o uso de componentes previamente desenvolvidos. BROOKS et al. (2005) cita os seguintes benefícios:

- Complexidade: Mesmo os sistemas robóticos mais simples são complexos, pois possuem variados tipos de elementos como atuadores, sensores e controladores que precisam interagir entre si. A maioria deles requer sua execução em uma *thread* própria, as quais podem se comunicar de forma síncrona ou assíncrona. Um mecanismo que padronize essa comunicação e delimite cada elemento de acordo com suas tarefas e características, torna-se necessário para gerenciar a complexidade deste sistema, através de um *software*.
- Flexibilidade: para o desenvolvimento de projetos complexos, é de extrema importância que se possa desenvolver, alterar e testar partes específicas sem que isso afete o sistema como um todo. A flexibilidade de um sistema de componentes permite focar especificamente em uma tarefa particular, mantendo o restante intacto e funcional.
- Ambientes distribuídos: Sistemas robóticos distribuídos são amplamente utilizados, tipicamente em situações onde um robô móvel é controlado em uma estação de trabalho remota. A modularidade de um sistema baseado em componentes, torna mais simples a tarefa de comunicar um sistema distribuído em dois ou mais ambientes.

- Variedade de *hardware* e sistemas operacionais : Sistemas robóticos frequentemente requerem diferentes plataformas de *hardware* e sistemas operacionais. A modularização através de componentes permite desvincular a implementação da aplicação do *hardware* e do sistema operacional. Assim pode-se construir aplicações genéricas, que não dependem especificamente destes fatores, podendo ser aplicadas em inúmeras situações.

Tais características permitem ao desenvolvedor projetar novas aplicações baseando-se em partes existentes e confiáveis, o que torna o novo sistema robusto e confiável

1.5 Frameworks para Robótica

Várias arquiteturas, *frameworks* e componentes tem sido propostos e desenvolvidos para auxiliar no desenvolvimento de sistemas de controle robóticos. Embora a maioria deles adote uma arquitetura baseada em componentes com o objetivo de reutilizar *software*, o projeto geral das arquiteturas difere, normalmente devido ao desejo de oferecer suporte eficiente a um projeto ou arquitetura em particular. Os *frameworks* usualmente mais utilizados em recentes pesquisas em robótica incluem (JUNG; DEGUET; KAZANZIDES, 2010):

- *Player* (GERKEY; VAUGHAN; HOWARD, 2003): é um conjunto de ferramentas para robôs móveis, incluindo uma gama de *drivers* de dispositivos robóticos. Conceitualmente, é uma camada de abstração de *hardware* para dispositivos robóticos que também inclui mecanismos de comunicação de dados entre *drivers* e programas de controle. As interfaces de comunicação são baseados em uma arquitetura cliente/servidor baseada em *socket* TCP.

- OROCOS (BRUYNINCKX, 2001): *Open RObot COntrol Software* (OROCOS) foi iniciado em 2001 para desenvolver software de código aberto para controle de robôs. Por ser o *framework* utilizado neste trabalho, será visto em maiores detalhes na seção 2.1.

- Orca (BROOKS et al., 2005): Ramificação do projeto OROCOS, o Projeto Orca, visa proporcionar blocos construídos (componentes) que podem ser combinados em conjunto para construir arbitrariamente sistemas robóticos complexos que não sejam de tempo real. Ele usa a *Internet Comunications Engine* (ICE) (HENNING, 2004) como *middleware* de rede.

- ROS (QUIGLEY et al., 2009): *Robot Operating System* (ROS) é um pacote de código aberto que fornece tipos de serviços em sistemas operacionais, tais como abstração de *hardware*, controle de dispositivos em baixo nível e comunicação entre processos, bem como diversas ferramentas para facilitar o desenvolvimento. A intenção é criar uma plataforma comum sobre a qual os pesquisadores podem construir, e compartilhar, algoritmos de robótica de alto nível em áreas como a navegação, localização, planejamento e manipulação.

- Cisst (THE CISST LIBRARIES, 2008): O pacote *cisst* é um conjunto de bibliotecas projetadas para facilitar o desenvolvimento de sistemas de intervenção assistidos por computador.

Tabela 1: Comparação entre para *frameworks* robótica.

Framework	*Windows*	*Linux*	RTOS	MT	MP	MH
Player	parcial	sim	não	não	sim	sim
OROCOS	sim	sim	sim	sim	sim	sim
Orca	parcial	sim	parcial	não	sim	sim
ROS	parcial	sim	parcial	não	sim	sim
Cisst	sim	sim	sim	sim	sim	sim

Uma comparação destes *frameworks* é mostrado na Tabela 1 (JUNG; DEGUET; KAZANZIDES, 2010), que mostra nível de suporte das plataformas *Windows*, *Linux*, RTOS (*Real-Time Operating System*), MT(*Multi-Thread*), MP (*Multi-Process*) e MH (*Multi-Host*) em cada *framework*, podendo ser: suporte total (sim), suporte parcial (parcial) ou não suportada (não). No suporte à RTOS, a classificação refere-se ao suporte a *hard real-time* e não apenas ao *framework* ser executável em RTOS.

Portanto, o OROCOS será utilizado por apresentar mais funcionalidades que os outros e ter uma abrangência maior que o Cisst, que é mais para área médica.

1.6 Objetivos

O presente trabalho apresenta como objetivo integrar o *framework* OROCOS ao robô manipulador Barrett WAM, de forma que seja possível acessar as funcionalidades do robô através de componentes construídos neste *framework* utilizando uma arquitetura aberta proposta em SANTINI (2009).

Tal arquitetura define componentes que abstraem o sistema de controle. Estes são genéricos e independentes do hardware utilizado, portanto deverá ser proposta uma interface entre a biblioteca do robô Barrett WAM e a arquitetura de controle, através de um componente que acesse as funcionalidades específicas da biblioteca do robô e as exporte para o restante da arquitetura.

A arquitetura proposta em SANTINI (2009) utiliza a versão 1.8.2 do OROCOS. Para possibilitar uma integração do trabalho com desenvolvimentos atuais, a arquitetura também será portada para a versão atual do OROCOS (2.5). Apesar de mudança significativa na interface, a atualização é feita de forma a manter o conceito e as funcionalidades inalteradas.

Para alcançar o objetivo proposto, inicialmente é feita uma revisão do estado da arte no desenvolvimento de software robótico, apresentando conceitos de CBSE e os principais *frameworks* utilizados na área, conforme apresentado neste capítulo.

No Capítulo 2, explica-se o funcionamento do *framework* OROCOS, destacando as principais diferenças entre a versão 1.8.2 e 2.5 e apresentado as alterações necessárias na arquitetura de SANTINI (2009). O Capítulo 3 introduz o *hardware* e o software do robô Barrett WAM com os objetivos de apresentar sua arquitetura de *hardware* e as funcionalidades da sua biblioteca e facilitar futuros trabalhos em cima do robô.

A interface da biblioteca no OROCOS, através de um componente, é apresentado no Capítulo 4, enquanto a sua colocação em uma arquitetura aberta é tratada no Capítulo 5, onde resultados também são apresentados. Por fim, concluí-se o texto no Capítulo 6, indicando as possibilidades de futuros desenvolvimentos deste trabalho.

2 PLATAFORMAS UTILIZADAS

2.1 OROCOS

Orocos é um acrônimo para *Open RObot COntrol Software*. O objetivo do projeto é desenvolver um *framework* de propósito geral, modular e de código aberto para controle de máquinas e robôs (OROCOS PROJECT SMARTER CONTROL IN ROBOTICS AND AUTOMATION, 2011).

Ele roda sobre os sistemas operacionais *Linux* e *Windows*, além de RTOS como o RTAI (RTAI, 2011) e o Xenomai, embora tenha-se encontrado dificuldades na sua utilização com o RTAI.

O projeto OROCOS nas versões 2.x possui três bibliotecas escritas em C++:

- ***Orocos Toolchain*** é destinada a criação de aplicações de robótica de tempo real, utilizando componentes de *software* configuráveis e de tempo real. Fornece a infraestrutura e as funcionalidades para construir aplicações de robótica em C++, além de uma biblioteca de componentes de controle, gerência e acesso à *hardware* prontos para uso. Basicamente, reúne as bibliotecas RTT e OCL da versão antiga.

- ***The Orocos Kinematics and Dynamics Library***(KDL) é uma biblioteca para cálculos de modelos cinemáticos e dinâmicos em tempo real.

- ***The Orocos Bayesian Filtering Library*** (BFL) fornece um conjunto de aplicações independentes para processamento de informação de forma recursiva e algoritmos de estimação baseados na regra de Bayes, como filtro de Kalman e filtro de partículas.

O OROCOS foi desenvolvido a partir das seguintes premissas (BRUYNINCKX, 2001):

1. Possuir modularidade e flexibilidade de forma que os usuários construam seus próprios sistemas *à la carte* e que desenvolvedores possam contribuir para os módulos que eles estão interessados, sem precisar desenvolver código para todo o sistema.

2. Ser independente de fabricantes de robôs comerciais.

3. Servir para todos os tipos de dispositivos robóticos e rodar em diversas plataformas.

4. Oferecer componentes de *software* para modelos cinemáticos e dinâmicos de robôs, geração de trajetória, sensoriamento, controle, interface de *hardware*.

5. Despertar motivação nos pesquisadores e engenheiros para contribuírem e utilizarem o seu código.

Um componente no OROCOS é uma unidade básica que executa uma ou mais ações, as quais são controladas pela sua Atividade. O OROCOS permite que essas ações sejam uma função em linguagem C e C++, ou um *script* com sua própria linguagem ou ainda uma máquina de estados hierárquica. Neste documento serão tratadas apenas ações em linguagem C e C++. A Atividade é iniciada pela classe `Activity`, a qual permite que o usuário configure os parâmetros: `Period, Priority and Scheduler`. O parâmetro `Period` permite iniciar uma atividade periódica com o período desejado. Uma atividade não periódica pode ser configurada deixando este parâmetro em zero. A prioridade da atividade é configurada através do parâmetro `Priority`. Já o parâmetro `Scheduler` permite configurar o escalonador da atividade, os quais são: um de tempo real, `ORO_SCHED_RT`, e outro não, `ORO_SCHED_OTHER`.

A partir da versão 2.0 do OROCOS, a interface de um componente consiste das primitivas: Atributos e Propriedades, Operações e Portas de Dados.

- **Propriedades e Atributos** são variáveis destinadas a configurar e ajustar o componente com determinados valores. Propriedades têm a vantagem de poder ser escritas e lidas em um arquivo no formato XML, portanto, pode armazenar um valor de forma persistente e carregá-lo posteriormente. Um exemplo de uso é um parâmetro de controle. Atributos refletem uma variável de classe em C++ na interface do componente e podem ser lidos e escritos durante tempo de execução por um programa. Um componente pode ter qualquer número de atributos ou propriedades, de qualquer tipo. Estas primitivas permanecem inalteradas da versão anterior.

- **Operações** são objetos que definem as funções que um componente oferece na sua interface. Ao utilizar um operação, qualquer método de qualquer classe pode ser adicionado à interface de um componente. Assim, as funções implementadas em C/C++ podem ser usadas em *scripts* ou podem ser chamadas de outro processo ou através de uma rede. As operações recebem argumentos e retornam um valor. O valor de retorno pode ser usado como argumento para outras operações ou armazenados em uma variável. Uma operação pode ser implementada de duas formas, que podem ser `OwnThread` ou `ClientThread`. Isso permite que o desenvolvedor do componente possa escolher se a operação, quando chamada, seja executada na *thread* do próprio componente (`OwnThread`), ou na *thread* do componente que chamou a operação (`ClientThread`). Uma operação `ClientThread` é realizada de forma síncrona com o componente que chamou a operação, pois é executada no mesmo instante pela *thread* deste componente. Já uma operação `OwnThread` é realizada de forma assíncrona com o componente que chamou a operação, pois sua execução irá depender da atividade do componente que recebeu a chamada da operação, ou seja, sua *thread* executará a operação chamada quando possível.

  Do ponto de vista do componente que chama a operação, esta pode ser chamada de duas formas, ambas utilizando um objeto da classe `OperationCaller`. Esta classe disponibiliza as funções de execução `call` e `send`. Na função `call`, o componente interrompe sua execução até o retorno da operação chamada. Já na função `send`, o componente continua sua execução, recebendo da função `send` um objeto chamado `SendHandle`, que permite seguir o *status* da operação e coletar seus resultados. Se utilizado sem a especificação de uso da função `call` ou `send`, o `OperationCaller` utiliza a função `call` por padrão. A utilização de `Call/Send` e `ClientThread/OwnThread` gera quatro combinações. Utilizando como exemplo um componente A que oferece uma operação, portanto recebe

Tabela 2: Combinações entre *Call/Send* e *ClientThread/OwnThread*.

OperationCaller	*ClientThread*	*OwnThread*
Call	*Thread* do componente B	*ExecutionEngine* do componente A
Send	*GlobalExecutionEngine*	*ExecutionEngine* do componente A

um pedido de execução para esta operação, e um componente B que requisita a operação, a Tabela 2 mostra quem executa esta operação, em cada uma das situações. `ExecutionEngine` é o objeto que gerência a execução de tarefas pela *thread* do componente, explicado em seguida.

Esta tabela mostra um caso especial: quando o componente B faz um `send` e o componente A definiu a operação como `ClientThread`, um terceiro precisa executá-lo. Esse é o trabalho do `GlobalExecutionEngine`. Uma vez que nenhuma *thread* deseja executar esta função, o `GlobalExecutionEngine`, que roda com a *thread* de menor prioridade no sistema, o pega.

Relacionando-se os tipos de operação com as primitivas método e comando da versão antiga, tem-se que as operações do tipo `ClientThread` chamadas pela função `call` são equivalentes aos métodos, enquanto as operações `OwnThread` chamadas pela função `send` são equivalentes aos comandos. As outras duas situações são possíveis apenas na versão nova.

- **Portas de dados** são objetos os quais implementam os mecanismos usados por um componente a fim de enviar ou receber um fluxo de dados. A porta é definida por um nome único dentro desse componente, o tipo de dados que quer trocar e o tipo da porta, o qual pode ser de dois tipos, para leitura (`InputPort`) ou escrita (`OutputPort`) de dados. Um porta de entrada de dados, pode ser configurada para disparar a atividade de um componente ou executar uma função quando receber dados. Essa função deve ser implementada através de uma *callback* que é executada de forma assíncrona. Tais portas são criadas como `eventports`, ou seja, portas capazes de reagir a uma escrita. Esta porta substitui a primitiva evendo da versão 1.8.

Componentes construídos no OROCOS são derivados da classe `TaskContext`, a qual define a interface pública, descrita anteriormente através das primitivas, para a iteração de componentes. Para que um componente tenha acesso à interface de outro, ele deve ser conectado como par deste componente, o que pode ser feito através de operações da classe `TaskContext`. Desta forma, ele ganha acesso à interface do componente par. As portas de dados são exceções e não necessitam que o componente seja um par para ter acesso, porém, também devem ser conectadas entre si, o que pode ser feito através de operações da classe `TaskContext` ou da própria porta, ou mesmo através de outro componente (`deployer`) que faz a inicialização através de um arquivo XML.

Enquanto a classe `TaskContext` fornece a interface de controle do componente, a classe `ExecutionEngine` executa a aplicação do usuário, segundo a atividade configurada no componente, conforme o período, a prioridade e o escalonador. É a classe núcleo de um componente, executando operações assíncronas, código do usuário e máquinas de estado. Permite iniciar, pausar, parar programas e máquinas de estado. A Figura 1 mostra a interface de um componente no OROCOS e sua interação com um componente par (OROCOS COMPONENT BUILDER'S MANUAL, 2011).

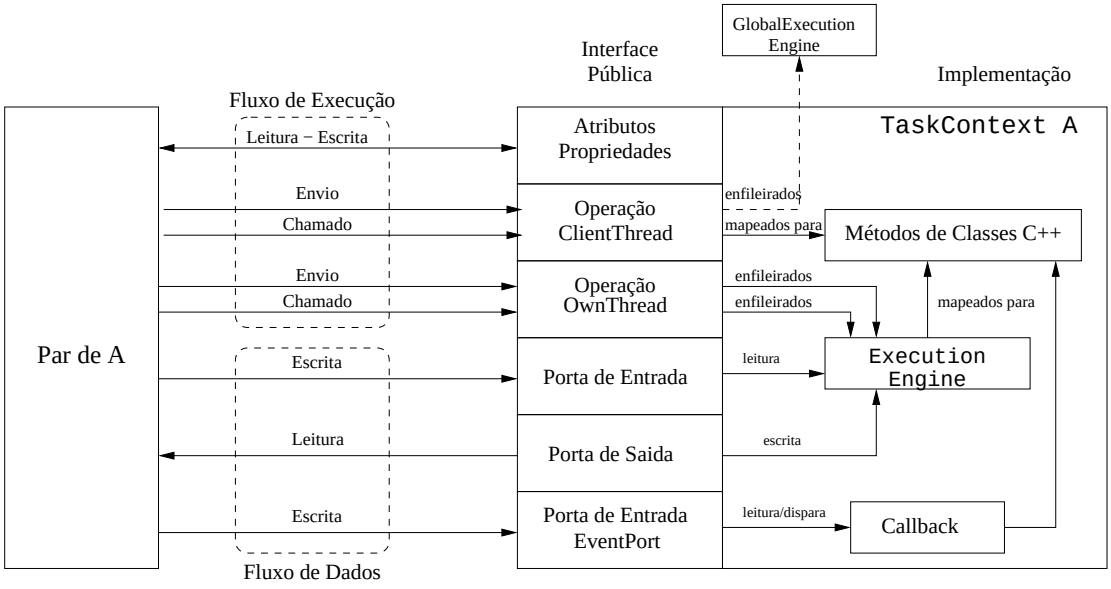

Figura 1: Interface de componentes no OROCOS versão 2.5.

Todo componente derivado da classe `TaskContext` possue funções padrão onde vai o código do usuário. Como apresentado em SANTINI (2009), são as funções:

- `configureHook()`: Função executada para configurar o componente;
- `startHook()`: Função executada para inicializar o componente;
- `updateHook()`: Função executada segundo a atividade do componente;
- `stopHook()`: Função executada para parar o componente;
- `cleanupHook()`: Função executada para finalizar o componente;
- `activeHook()`: Função executada para ativar o componente;
- `errorHook()`: Função que substitui `updateHook` em caso de erro não crítico;
- `resetHook()`: Função para recuperar um erro crítico;

Uma descrição detalhada da interface e operação de componentes do OROCOS na versão 1.8, também pode ser encontrada no referido trabalho. Este trabalho utiliza a versão 2.5 do OROCOS, sendo apresentadas apenas as características mais importantes que sofreram alterações da versão 1 para a 2. Efetivamente, as alterações mais importantes foram a mudança na interface padrão dos componentes do OROCOS, visto que foram retiradas as primitivas Métodos, Comandos e Eventos, dando lugar à primitiva Operações.

2.2 Arquitetura Utilizada

A arquitetura utilizada neste trabalho, bem como os componentes nela utilizados, foram desenvolvidos em SANTINI (2009). Esta arquitetura foi desenvolvida com a finalidade de controlar robôs manipuladores, através da implementação de componentes genéricos, os quais são independentes entre si e com o *hardware* do robô. Desta forma, eles

podem ser manipulados e configurados para controle de diferentes sistemas robóticos, independente do seu *hardware*. A arquitetura é utilizada somente em alto nível, através dos componentes `Sensor`, `Actuator`, `Controller` e `Sampler`, sem necessidade de se trabalhar em um nível mais baixo. As modificações necessárias nestes componentes, bem como suas extensões, são apresentadas a seguir.

2.2.1 Componentes de Controle

Os componentes utilizados são, especificamente, os que abstraem a parte de controle. Aqueles que abstraem os sensores e os atuadores do sistema, são estendidos para utilização no presente trabalho.

- **Sampler** é o amostrador do sistema. Ele tem como função sincronizar os demais componentes. Visto que isto era feito através da primitiva Evento, que foi retirada na versão 2 do OROCOS, este componente foi alterado criando-se uma porta de saída (`SamplePort`) que é utilizada para gerar o *sample* do sistema, através de uma escrita na porta que é realizada de acordo com a atividade do componente. Para ter acesso ao *sample* do sistema, todos os demais componentes devem possuir uma porta de entrada do tipo `EventPort` conectada à porta `SamplePort` do `Sampler`.

- **SensorWAM** tem a função de abstrair os sensores do sistema. Ele é derivado do modelo `Sensor`, que não tem implementação própria, e que possui uma porta de dados de saída onde ele deve escrever o valor lidos pelos sensores. Essa porta é representada por um vetor de tamanho N, onde N é o número de juntas do sistema. Ao modelo base `Sensor` também foi adicionada uma porta de entrada do tipo `EventPort` para receber o *sample* do sistema (*SamplePort*). Sempre que uma escrita é feita nesta porta, é executada uma *callback* de forma assíncrona, responsável pela execução das tarefas do componente. Esta *callback* é virtual e deve ser implementada na extensão do modelo `Sensor`. No caso do `SensorWAM`, esta tarefa é uma requisição de leitura dos sensores. Também é adicionada mais uma porta de entrada de dados do tipo `EventPort` (`InputSensorPort`) para receber os dados dos sensores. Após ser requisitada a leitura dos sensores, quando ocorrer uma escrita na porta `InputSensorPort` ele executa uma *callback*, a qual lê esta porta e escreve os dados na porta de saída.

- **ActuatorWAM** tem a função de abstrair os atuadores do sistema. Ele é derivado do modelo `Actuator`, que não tem implementação própria, e possui uma porta de dados de entrada onde devem ser recebidos os valores de atuação do sistema, a qual é representada por um vetor de tamanho N. O `ActuatorWAM` acrescenta uma porta de saída, aonde são escritos os valores de atuação do sistema, sempre que ocorrer o *sample*. Ao modelo base `Actuador` foi adicionada uma porta de entrada para receber o *sample* do sistema, exatamente da mesma forma descrita para o modelo `Sensor`.

- **ControllerNPID** implementa o controle do sistema, através de um PID para cada junta. É derivado do modelo `Controller`, o qual possui portas de entrada, para referências e sensores, e saída para atuação, adicionando as portas necessárias para se comunicar com N componentes PID. No modelo base `Controller` foi adicionada uma porta de entrada `EventPort` para conectar-se ao *sample* do sistema, assim como aos componentes anteriores.

- **ControllerWAM** implementa o controle do sistema, através do controlador da biblioteca do Barrett WAM. É derivado do modelo `Controller`, sendo adicionadas uma porta de entrada e uma de saída para a comunicação com o componente que acessa a biblioteca do robô. Este componente e o `ControllerNPID` são utilizados em topologias diferentes.

- **PID** implementa a lei de controle de mesmo nome, sendo que cada componente PID é responsável pelo controle de uma única junta do robô.

A Figura 2 mostra o modelo básico dos componentes utilizados, adaptados para a versão 2.5. As extensões destes componentes e as topologias utilizadas neste trabalho, serão apresentadas no Capítulo 5.

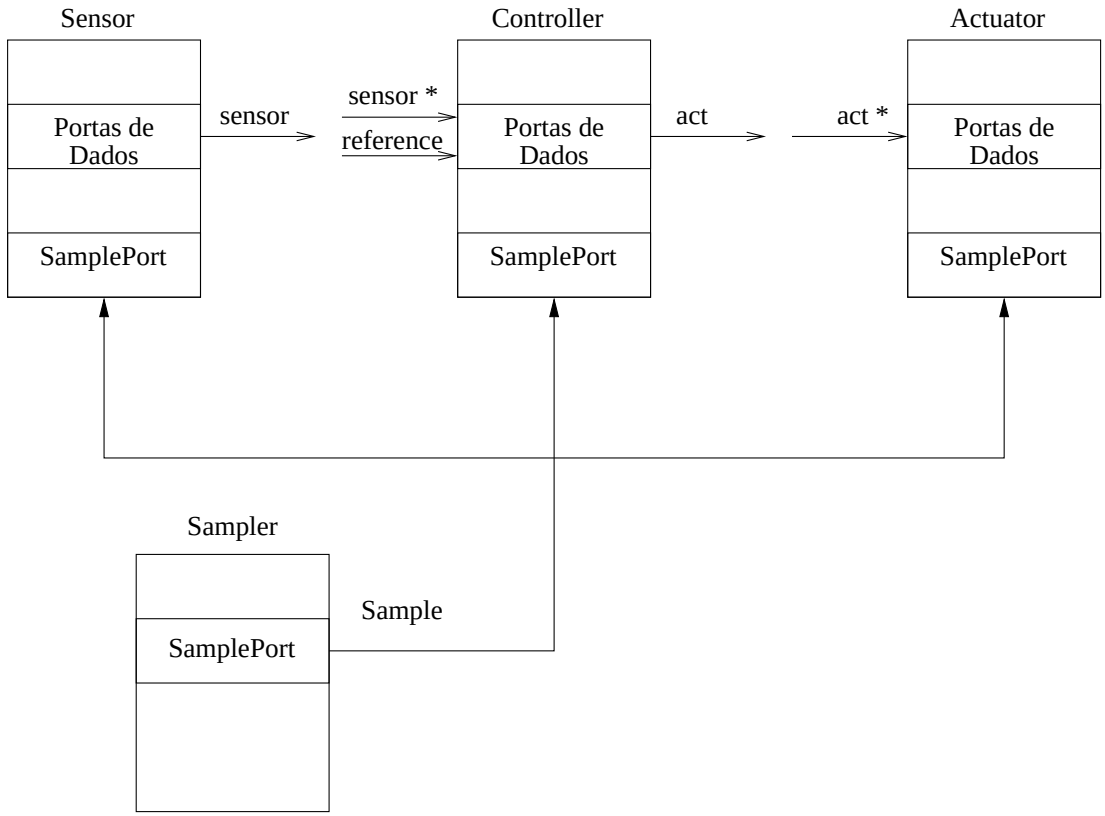

Figura 2: Modelo dos componentes

As funcionalidades e a interface de cada componente, bem como sua forma de operação são apresentadas em detalhes em SANTINI (2009). Aqui, é resumida a função de cada componente, com o objetivo de apresentar as alterações realizadas em cada um, para que fossem utilizados neste trabalho na versão 2.5 do OROCOS.

2.2.2 Componente `TrajectoryGenerator`

Além dos componentes anteriormente citados, também é utilizado um componente existente fornecido pelo OROCOS, chamado `nAxesGenerator`, que é utilizado para gerar uma trajetória para o deslocamento das juntas até a posição de referência desejada. Neste trabalho, será referenciado como `TrajectoryGenerator`.

O `nAxesGenerator` usa um perfil de velocidade trapezoidal com uma aceleração máxima e velocidade máxima para calcular a trajetória a partir da posição atual de N eixos para uma posição desejada em um determinado espaço de tempo. A velocidade e aceleração, final e inicial, são nulas. Os movimentos de todos os eixos são sincronizados, para que todos os eixos demorem o mesmo tempo. A velocidade máxima, aceleração máxima e número de juntas são propriedades: `max_vel`, `max_acc`, `num_axes`, logo são configuráveis. A operação `moveTo` inicia o cálculo da trajetória, recebendo como parâmetros um vetor com as posições desejadas e o tempo de duração da trajetória, e a operação `resetPosition` que para o movimento e mantém a posição do robô, definindo a posição desejada para a posição medida e a velocidade desejada para zero. Por fim, uma porta de entrada, `nAxesSensorPosition`, recebe a posição atual dos eixos e duas portas de saída, `nAxesDesiredPosition` e `nAxesDesiredVelocity`, e escreve as posições e velocidades calculadas, respectivamente.

2.3 *Real-Time* Linux

O escalonador do Linux, como o de outros sistemas operacionais como o Windows ou MacOS, é projetado para melhorar o desempenho médio. Desta forma, ele não garante que qualquer tarefa em particular sempre será executada em um tempo determinado. Uma tarefa pode ser suspensa por um tempo arbitrariamente longo, por exemplo, enquanto um *driver* de dispositivo atende uma interrupção.

Garantias temporais são oferecidas pelos sistemas operacionais de tempo real (RTOS), os quais são normalmente utilizados para aplicações de controle ou de comunicação, não para propósito geral. O Linux tem sido adaptado para suporte em tempo real. Estas adaptações são chamadas de *Real-Time* Linux (RT Linux), dentre as quais pode-se citar versões como o Xenomai, RTAI (*The Real-Time Application Interface*) e o RTLinux. Estes sistemas RT Linux são extensões para o *kernel* do Linux, e para sua sua construção é necessário, basicamente, obter o código fonte do *kernel* e configurá-lo adequadamente para o propósito desejado, aplicar o *patch* e compilar o *kernel*.

A ideia geral do RT Linux é que um *micro-kernel* de tempo real roda sob o Linux, o que significa que o *kernel* de tempo real escalona o *kernel* Linux. Tarefas de tempo real são executadas pelo *kernel* de tempo real, e os programas normais do Linux são executados quando não há tarefas de tempo real a serem executadas. O Linux pode ser considerado como a tarefa ociosa do escalonador de tempo real. Quando essa tarefa ociosa é executada, ela executa seu próprio escalonador e escalona os processos normais do Linux. Visto que o *kernel* de tempo real escalona um processo normal do Linux, este é colocado em espera quando uma tarefa de tempo real necessitar ser executada (XENOMAI GENERAL, 2011).

2.3.1 Xenomai

Xenomai é um *framework* de desenvolvimento de tempo real, cooperando com o *kernel* do Linux para fornecer um suporte generalizado e *hard real-time* para aplicações no espaço de usuário, perfeitamente integrado no ambiente Linux (XENOMAI: REAL-TIME FRAMEWORK FOR LINUX, 2011).

É baseado em um núcleo RTOS abstrato, utilizável para a construção de qualquer tipo de interfaces de tempo real, através de um núcleo que exporta um conjunto de serviços RTOS genéricos. Qualquer número de personalidades RTOS, chamados de *skins*, pode então ser construída sobre o núcleo, fornecendo sua própria interface específica para os

aplicativos, usando os serviços de um único núcleo genérico para implementá-lo.

O Xenomai permite executar *threads* em tempo real estritamente no espaço do *kernel*, ou dentro do espaço de endereçamento de um processo do Linux. Uma tarefa de tempo real no espaço do usuário ainda tem o benefício de proteção de memória, mas é escalonada pelo Xenomai diretamente, e não mais pelo *kernel* do Linux. O pior caso de latência de escalonamento desse tipo de tarefa é sempre perto dos limites de *hardware* e previsível, uma vez que Xenomai não é obrigado a sincronizar com a atividade do *kernel* do Linux em tal contexto, e pode antecipar qualquer atividade regular do Linux sem atraso. Por isso, o ambiente preferencial de execução para aplicações do Xenomai é o espaço do usuário.

Pode haver alguns casos em que a execução de códigos em tempo real incorporados em módulos do *kernel* é necessário. Por esta razão, a API nativa do Xenomai oferece o mesmo conjunto de serviços de tempo real, de forma transparente para os aplicativos, independentemente do seu espaço de execução. Além disso, alguns aplicativos podem precisar de atividades em tempo real de ambos os espaços em cooperação, portanto, cuidados especiais foram tomados, no desenvolvimento do Xenomai, para que estas possam trabalhar exatamente no mesmo conjunto de objetos API (XENOMAI GENERAL, 2011).

2.4 Conclusão

Neste capítulo nota-se que o OROCOS atende a todas as necessidades para o desenvolvimento de uma interface com o robô Barrett WAM, através de componentes. Pode-se construir um componente que faça as inicializações necessárias do robô através das funções padrão da classe `TaskContext` e utilizar as primitivas para fornecer uma interface de acesso as funcionalidades do robô ao restante do sistema.

A arquitetura utilizada se mostra importante para implementar o laço de controle do sistema através de componentes existentes que abstraem os elementos como sensores, atuador e controlador, em conjunto ao componente de acesso ao robô. É importante que estes componentes possuem uma implementação genérica e extensível, o que permite utilizá-los e adaptá-los a esta implementação de forma simples.

Por fim, o *Linux* de tempo real, utilizando o Xenomai, fornece as garantias temporais necessárias para a execução de tarefas em tempo real, requisito comum a arquiteturas de controle.

3 ROBÔ MANIPULADOR BARRETT WAM

O WAM (*Whole Arm Manipulator*) é um braço robótico que possui modelos de quatro ou sete graus de liberdade, podendo ter uma mão (`BarrettHand`) anexada. A Figura 3 mostra um WAM com sete graus de liberdade, conhecido como WAM 7-DOF (*degrees of freedom*) e com uma `BarrettHand` anexada. Ambos os modelos são quase idênticos, sendo que o WAM 7-DOF inclui um módulo de pulso 3-DOF, enquanto o braço WAM 4-DOF só tem um punho vazio.

Figura 3: Juntas do Robô Barrett WAM.

Cada WAM vem com seu próprio computador interno (referenciado aqui como PC WAM), o qual roda um sistema operacional Linux de tempo real, utilizando o Xenomai. A utilização deste computador é opcional, visto que pode se utilizar um computador externo para rodar a aplicação do usuário.

3.1 Visão Geral do *hardware*

O motores do WAM são movidos por controladores do motor, desenvolvidos e patenteados pela Barrett, denominados Pucks. Os Pucks são pequenos e são colocados ao longo do braço WAM, montados diretamente nos motores que controlam. Um Puck serve como uma fonte de energia para os motores e comanda seu torque de forma suave e contínua, baseado em um comando de torque digital. Também serve como *encoder* para o ângulo do motor, amplificador de potência, e contém fonte de corrente precisa internamente e sensor de temperatura, além de comunicação de alta velocidade através de um barramento CAN. O barramento CAN é um barramento serial, de dois fios, diferencial que oferece comunicação digital a 1 Mbps, com alta imunidade a ruídos.

Os Pucks enviam dados de posição dos motores para o PC WAM e recebem dados de torques a serem aplicados nos motores de volta do PC WAM em uma malha de controle simples. A malha de controle pode ser executada a qualquer taxa de até 1 kHz, sendo a taxa padrão de 500 Hz.

Toda a comunicação é monitorada por uma placa de segurança (*Safety Board*). A *Safety Board* fica na base do WAM e monitora a velocidade do braço, a magnitude dos comandos de torque a serem enviados para os motores, a taxa de comunicação entre o PC WAM e os Pucks, e o status do sistema (WAM USER MANUAL, 2011).

A Figura 4 mostra um diagrama de blocos do *hardware* do WAM, para o hardware de uma junta, a (*Safety Board*), o WAM PC e o barramento CAN. Todas as juntas se conectam ao barramento CAN desta mesma maneira.

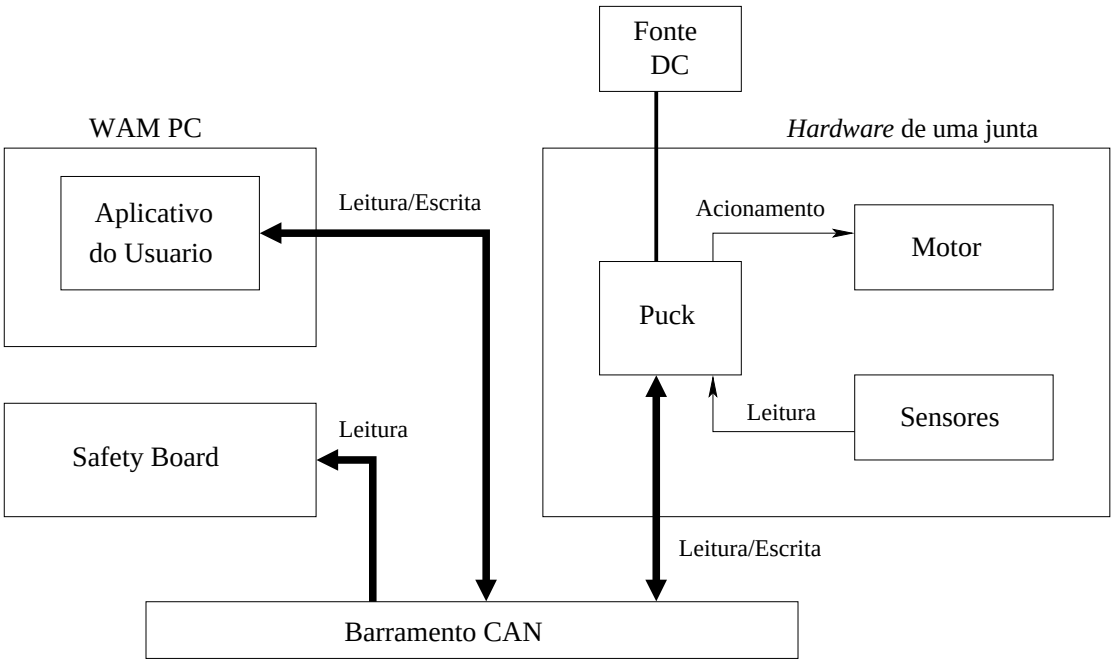

Figura 4: Diagrama de blocos do *hardware* do WAM.

O WAM tem três estados de segurança: E-STOP, IDLE, e ACTIVATED.

1. E-STOP significa que não há tensão no barramento do motor. Na verdade, a alimentação do barramento do motor e as linhas de terra são conectadas, resultando em um efeito de "quebra de resistência"sobre as juntas do braço WAM. Isto significa que efetivamente ele não exerce qualquer força, mas ao invés disso resiste levemente a

qualquer força aplicadas a ele, o que resulta no WAM cair lentamente até encontrar um impedimento físico. Também permite que o WAM possa ser facilmente movido para uma posição desejada, pelo operador. Neste estado, não é possível acessar os Pucks e eles não mantém o controle de posição dos seus motores, portando não há como obter dados dos sensores e nem enviar sinais de controle.

2. IDLE mostra que há tensão aplicada no barramento do motor e os Pucks estão acessíveis. Eles mantém o controle de posição dos seus motores, e os mantém em um estado de frenagem, ignorando qualquer comando de torque enviados a eles. Portanto, neste estado é possível obter dados dos sensores, mas não é possível atuar sobre o WAM.

3. ACTIVATED é o estado em que os Pucks estão efetivamente aplicando qualquer comando de torque que eles recebem do PC de controle, ou seja, é o estado no qual o WAM está pronto para uso. Este estado só pode ser atingido quando não há nenhuma falha detectada pela *Safety Board*.

Neste trabalho, é utilizado o WAM PC, pois utilizar um computador externo traria a necessidade de instalação de uma placa PCI CAN, para comunicação com o robô, além da compilação de um *kernel* de tempo real para este computador, utilizando o Xenomai.

3.2 Visão Geral do *software*

Para a criação de *software* para o WAM, existe uma biblioteca que permite trabalhar com o braço e a mão do WAM, chamada `Libbarret`. A `Libbarrett` é uma biblioteca de código em C++ que permite aos usuários controlar e operar o WAM. Ela contém diversas classes, dentre as quais são apresentadas as mais importantes para o usuário, bem como algumas de suas principais funções (LIBBARRETT PROGRAMMING MANUAL, 2011).

`ExecutionManager` é a classe que supervisiona todas as operações em tempo real. É responsável pelo ciclo de execução do programa, definido geralmente em uma taxa de 500 Hz.

A classe `wam` tem por objetivo permitir ao usuário acessar as funcionalidades do braço do WAM diretamente. Contém funções de *get* para a leitura de posição, velocidade e torque aplicado nas juntas, além da posição da extremidade do braço. Estas funções retornam vetores dos tipos `jp_type, jv_type, jt_type, cp_type`, que são vetores do tipo *double*, utilizados para definir os dados com clareza, evitando erros na sua manipulação. A Tabela 3 mostra estes tipos, e as unidades a eles associadas.

Tabela 3: Tipos de dados da `Libbarret`.

Tipo	Nome do tipo	Unidade
Posição Cartesiana	`cp_type`	Metros
Posição da Junta	`jp_type`	Radianos
Velocidade da Junta	`jv_type`	Radianos por segundo
Torque da Junta	`jt_type`	Newton Metros

Também possui funções de movimentação do braço, como `moveHome` e `moveTo` que permitem movimentar o braço para a posição de origem padrão, ou para um dado ponto, determinado em coordenadas cartesianas utilizando-se o tipo `cp_type`, respectivamente.

Esta função não permite que seja determinado o movimento específico de cada junta, sendo isto calculado pela `Libbarret`.

Já a função `trackReferenceSignal` permite receber um sinal de referência dos tipos `jp_type, jv_type` e segui-lo. Assim, é possível aplicar uma trajetória de referência para cada junta, que será seguido através do controle de posição implementado pela `Libbarret`. Outra função útil é a `gravityCompensate`, que possibilita a compensação da gravidade, ao criar um objeto da classe `gravityCompensator`, o qual calcula a força da gravidade no WAM e compensa enviando torques apropriados para as juntas.

Para que as funções de movimentação funcionem de maneira correta, a classe `wam` possui controladores PID, que são objetos da classe `PIDController`. Eles recebem os sinais de referência e a os valores dos sensores, para calcular o torque a ser aplicado.

A classe `wam` possui um objeto de entrada de dados, o qual permite receber dados de torque que serão aplicados diretamente no robô. O torque efetivamente aplicado ao robô é um somatório dos torques calculados individualmente pelos controladores para cada referência, mais o torque recebido nesta entrada, mais o torque calculado para compensar a gravidade.

Em conjunto à *Safety Board* é utilizada a classe `SafetyModule` que permite monitorar os estados do WAM, permitindo obter no *software* a identificação de qual estado de segurança o WAM está em um determinado momento.

Por fim, a classe `ProductManager` possibilita o uso e o gerência do WAM, através da inicialização de componentes de *hardware*, como os Pucks, o barramento CAN e a *Safety Board*, e componentes de *software*, como a criação dos objetos `ExecutionManager`, `Wam` e `SafetyModule`. Também faz a leitura do arquivo de configuração associado ao modelo de *hardware* identificado, o qual possui parâmetros do hardware, como parâmetros cinemáticos e inerciais. Portanto, esta classe é responsável por disparar a execução do programa e inicializar os objetos para a manipulação do *hardware*. (LIBBARRETT DOXYGEN DOCUMENTATION, 2011)

Utilizando estas classes, é possível inicializar o WAM, verificar seu estado e, se nenhuma falha for detectada, manipular o robô em alto nível, através das funções de cada classe.

4 INTERFACE OROCOS-WAM

Com o objetivo de utilizar o robô WAM através de componentes desenvolvidos no OROCOS, tornou-se necessário, primeiramente, desenvolver uma interface entre o robô, através da `Libbarret`, e o OROCOS, através de um componente que pudesse interagir com o robô.

4.1 Componente `OrocosWam`

Para trabalhar com o robô WAM através de componentes do OROCOS, foi desenvolvido um componente com o objetivo de fazer a interface de acesso à `Libbarret`, de forma a inicializar e acessar as funcionalidade do robô. De fato, este componente é o único a interagir diretamente com a `Libbarret`, e portanto ser vinculado especificamente a esta aplicação. Os demais componentes não possuem ligação direta com a `Libbarret` nem com o WAM, o que demonstra a sua generalidade para diversas aplicações.

Este componente, chamado de `OrocosWam`, é derivado da classe de mesmo nome e possui objetos das classes da `Libbarret` apresentadas anteriormente, com os quais realiza a verificação de *hardware*, a inicialização do robô, checagem de *status* e manipulação do WAM. Mais especificamente, possui objetos das classes: `ProductManager`, `SafetyModule` e `WAM`.

Quanto a sua interface, este componente possui quatro portas de dados, as quais são vetores do tipo *double*, duas de entrada e duas de saída, além de uma operação do tipo `ClientThread`. A porta de saída `jointsDataPort` é responsável por passar os dados de posição, velocidade e torque para o sistema. A porta de entrada `posRefPort` é onde devem chegar as posições de referência a serem aplicadas, ao controlador da libbarret e a porta de entrada `torRefPort` é onde devem chegar os valores de torques a serem aplicados em cada junta. Já a porta de saída `ControlDataPort` é onde são escritos os valores calculados pelo controlador da `Libbarrett`. Ambas portas de entrada são do tipo `EventPort`. Por fim, a operação `getJointSensorsOperation` inicia uma leitura dos sensores do WAM.

4.1.1 Inicialização do Componente

Na inicialização do componente, o objeto `ProductManager` é inicializado juntamente com a classe `OrocosWam` e, como seu construtor verifica e inicializa o *hardware* do WAM, é necessário que este componente seja executado em um sistema com acesso ao WAM, caso contrário o componente apresentará mensagem de erro na sua construção e irá abortar. Caso a inicialização ocorra com sucesso, o `OrocosWam` irá executar sua função de configuração (`configureHook()`), onde o objeto `ProductManager` irá buscar

acesso à *Safety Board* e irá retornar um ponteiro para para classe `SafetyModule`. Se ocorrer alguma falha no acesso à *Safety Board* a função `configureHook()` retorna `false` e o componente não é configurado. Sendo o objeto `SafetyModule` inicializado com sucesso, ele busca o status do WAM. Este status é informado ao usuário, via *log* de informação. Caso for E-STOP é retornado `false`, e o componente não é configurado, sendo IDLE o status esperado para prosseguir (o status ACTIVE não é possível ainda neste momento). Feita a inicialização de segurança, o objeto `ProductManager` busca a identificação do WAM. Novamente, caso ocorra algum erro, é retornado `false` e o componente não é configurado. Se ocorrer sem erros, a função `configureHook()` retorna `true` e o componente é configurado com sucesso. Ao final da configuração, ainda é verificado se o WAM se encontra em sua posição inicial, sendo dado um aviso de segurança (*Warning*) ao usuário caso não se encontre. No entanto, isto não impede a configuração do componente.

Após estar configurado, o componente executa a função `startHook()`, onde o `ProductManager` inicializa os Pucks e também um objeto da classe `Wam`, através do arquivo de configuração do *hardware*, e retorna um ponteiro para esta classe. Na função que inicializa um objeto `wam`, o *software* se mantém em espera até que o usuário ative manualmente o robô, para que ele entre em status ACTIVATED. Por último é chamada a função `gravityCompensate` da classe `wam`, para que o robô já inicialize com a gravidade compensada. Se nenhum erro ocorrer, o componente é inicializado, e o WAM fica pronto para uso.

4.1.2 Operação do Componente

Visto que o componente `OrocosWam` é aperiódico e não possui implementação na função `updateHook`, ele não realiza nenhuma ação, enquanto não for solicitado. Assim, sua execução é feita somente pela atividade de sua operação e porta de entrada. A figura 5 mostra a interface do componente e o diagrama do fluxo de execução interno das atividades.

Quando chamada a operação `getJointSensorsOperation`, o componente executa a função `getJointSensors`, a qual utiliza as funções de *get* da classe `Wam` para obter os dados de posição, velocidade e torque de cada junta. De posse desses dados, eles são escritos na porta `jointsDataPort`, em formato de vetor, seguindo da junta 1 a 7, na ordem seguinte: todas as posição, depois todas as velocidade e por último todos os torques (Pos1, Pos2, ..., Pos7, Vel1, Vel2, ..., Vel7, Tor1, Tor2, ..., Tor7).

Quanto ao controle do componente, ele permite que seja utilizado o contralor da `Libabarret` ou não. Para utilizar o controlador da `Libbarrett`, a porta `posRefPort` deve receber referências de posições a serem seguidas por cada junta, esses valores são aplicados a um controlador PID da `libbarret`, para realizar o cálculo de controle. Estes valores são exportados pela porta `ControlDataPort`. Os valores de atuação, devem ser escritos na porta `torJointPort`, que deve receber valores de torques a serem aplicados diretamente em cada junta.

Se utilizada a porta `posRefPort`, quando uma escrita é feita nela é chamada uma *callback* que lê o valor escrito na porta e chama a função `trackReferenceSignal` da classe `Wam`, passando os valores de posição a serem seguidos. Esta função aciona o controlador interno da `libbarret`, o qual calcula os torques para cada junta, a partir do vetor de posições de referência passado, e escreve estes valores na porta de saída `ControlDataPort`.

Quando uma escrita é feita na porta `torJointPort` é chamada uma *callback* que lê

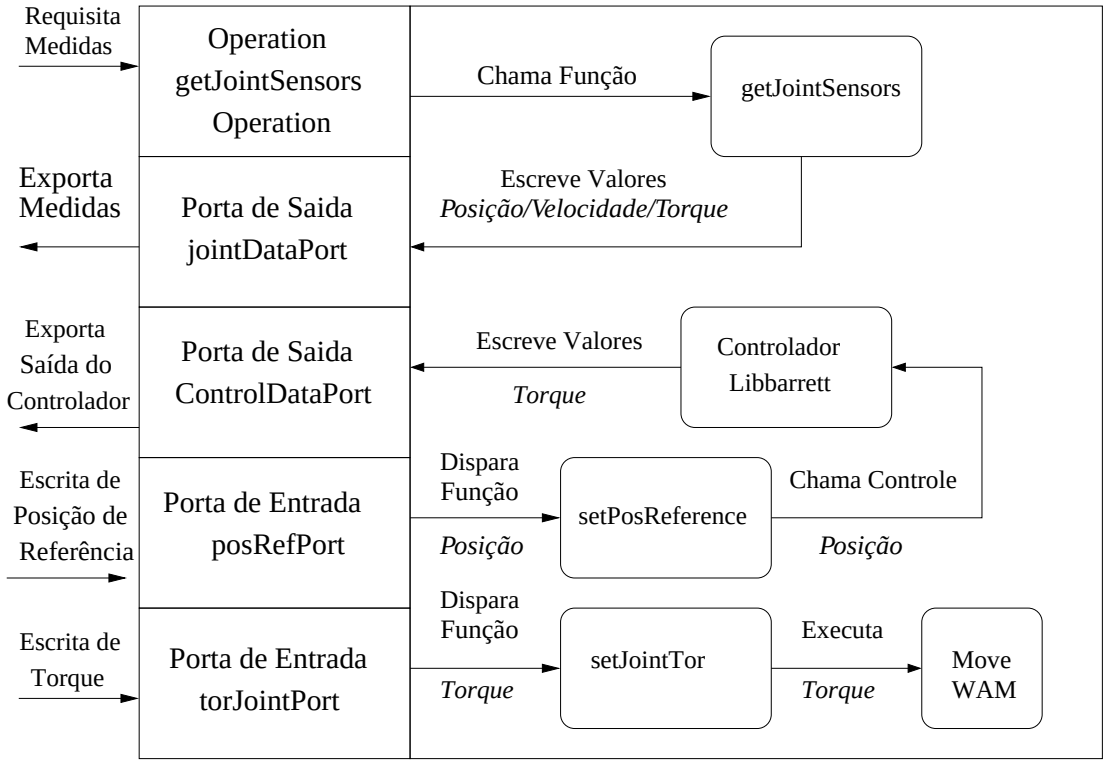

Figura 5: Interface de operação do componente `OrocosWam`.

o valor escrito na porta e verifica se nenhum valor de torque está acima do valor de torque máximo permitido para cada junta, evitando movimentos bruscos e travamento do robô. Se todos valores de torque forem válidos, a função utiliza uma entrada de dados da classe `wam` para passar estes dados, os quais são enviados aos Pucks, que aplicam o torque aos motores, ocasionando o movimento das juntas.

Por fim, observar-se que ao utilizar o controlador da `Libabarret`, pode-se permitir que ele atue diretamente sobre o robô, sem que seja necessário exportar os valores do cálculo de controle. Para fazer isso, é necessário compilar o componente com a *flag* `DIRECTCONTROLL` habilitada. Isto permite que uma utilização mais simples do robô seja feita, sem exportar a parte de controle e atuação dos componentes. Isso tira grande parte da modularidade do sistema, no entanto, permite uma utilização direta do robô, necessária em testes iniciais.

4.1.3 Parada do Componente

Quando finalizado, o componente `OrocosWam`, na sua função `stopHook`, chama a função `moveHome` da classe `wam`. Esta função movimenta o braço para sua posição de origem, evitando quedas bruscas e colisões quando o sistema de controle for desligado e não existir mais compensação de gravidade.

4.2 Componente `Reference`

Este componente foi criado com o objetivo de fornecer uma interface para o usuário entrar com a posição de referência desejada para cada junta. Para uso do usuário, possui as propriedades `jointPosition`, `moveTime` e a operação `moveToRefOperation`.

A propriedade `jointPosition` é um vetor do tipo *double* que guarda as posições de referência de cada junta. Quando este componente é inicializado, este vetor é escrito com as posições atuais de cada junta. O usuário pode então, modificar a posição desejada para cada junta e chamar a operação `moveToRefOperation`. Esta irá chamar uma operação do componente `TrajectoryGenerator` e enviar os dados de referência armazenados em `jointPosition` e na propriedade `moveTime`, a qual guarda o valor do tempo de execução do movimento das juntas, utilizado para interpolar a curva de referência. Por padrão, `moveTime` é 5 segundos, mas o usuário pode alterar esta propriedade quando desejar.

5 RESULTADOS

5.1 Topologias Trabalhadas

De posse do componente `OrocosWam` para realizar a interface entre os componentes do OROCOS e a `Libbarret`, tendo assim acesso ao *hardware*, foram montadas duas topologias para atuar sobre o robô, a primeira, utiliza o controlador de posição da própria `Libbarret`, o qual é abstraído pelo componente `ControllerWAM`, e a segunda utilizando o controlador original da arquitetura utilizada, através do componente `ControllerNPID`.

5.1.1 Controle com `ControllerWAM`

A Figura 6 mostra a disposição dos componentes nesta arquitetura. As flechas com linhas tracejadas indicam a chamada de operação de um componente para outro, enquanto as flechas com linhas cheias indicam o fluxo de dados entre os componentes. O `Sample` do sistema é mostrado apenas com pequenas flechas nos componentes que o recebem, para não poluir a figura.

Sempre que ocorrer o `Sample` do sistema, o `Sensor` irá fazer um requisição de leitura dos sensores, através da operação indicada, fará a leitura da sua porta de entrada e, após receber os valores, atualizar sua porta de saída com as posições lidas.

Quando o usuário desejar mover o WAM, deverá acessar o vetor de posições `jointPosition` do componente `Reference`. Este vetor, possui a posição atual de todas as juntas do WAM. Basta apenas alterar a posição do vetor associada a cada junta para o valor de referência desejado para a movimentação da junta e executar a operação `moveToRefOperation`. O `Reference` irá passar este vetor de posições de referência e o tempo de execução do movimento para o `TrajectoryGenerator`, o qual irá interpolar uma curva de trajetória para cada junta, passando estes valores ao `ControllerWAM`.

Este componente recebe a trajetória e envia ao componente `OrocosWam`, o qual utiliza o controlador interno da biblioteca do robô para calcular o valor de atuação. Este valor é retornado ao `ControllerWAM`, que envia o valor de controle ao `Actuator`, que por sua vez passa o valor de atuação ao `OrocosWam`, que aplica ao robô.

Desta forma, o controlador da biblioteca do robô é abstraído dentro do componente `ControllerWAM` e a atuação é feita pelo `Actuator`. Portanto, os elementos de controle são abstraídos dentro dos componentes da arquitetura e o `OrocosWam` cumpre seu papel apenas como interface às funcionalidades da biblioteca e do robô.

Assim, será fechado o laço de controle, e o WAM só irá se movimentar quando uma nova trajetória de referência for gerada.

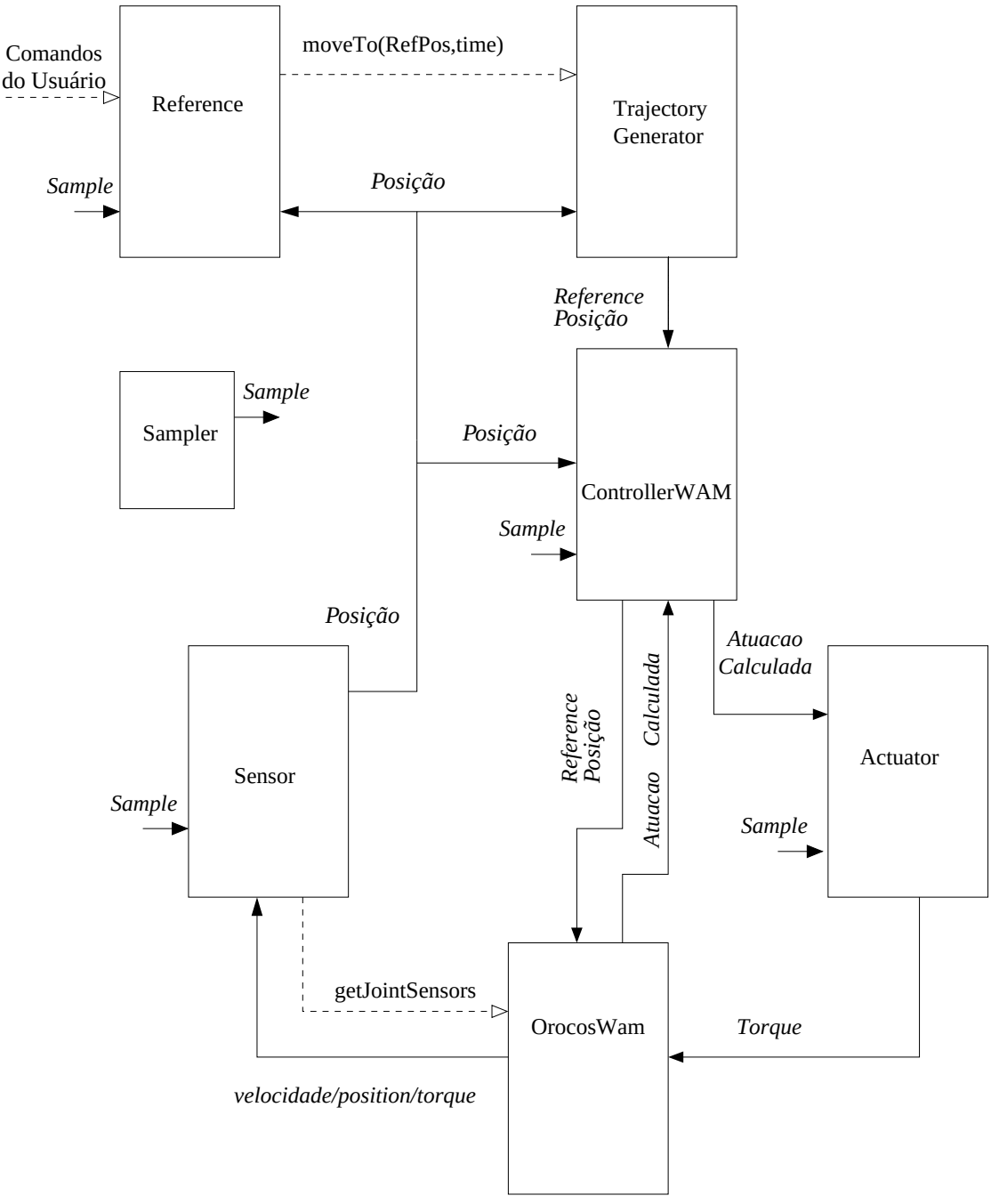

Figura 6: Controle com `ControllerWAM`

5.1.2 Controle com `ControllerNPID`

A Figura 7 mostra a disposição dos componentes nesta arquitetura. Do ponto de vista do usuário, não há alteração, pois ele continua manipulando as posições de referência desejadas da mesma forma, através do componente `Reference`.

Nesta topologia, a trajetória gerada pelo `TrajectoryGenerator` é passada ao componente `ControllerNPID`, que irá quebrar este vetor de referência, e o vetor de posições lidas, recebido do `Sensor`. Os valores de posição e referência de 1 até N, sendo N igual a 7 neste trabalho, serão passados para seus respectivos PID's. Cada PID calcula o valor de atuação da sua junta correspondente, com base nos parâmetros desta junta. O `ControllerNPID` agrupa os valores de atuação calculados em um novo vetor, que é então passado para o `Actuator`, que os repassa como um vetor de torque ao `OrocosWam`, onde estes valores são efetivamentes aplicados ao *hardware*. A forma de leitura dos sensores não é alterada em relação à topologia anterior.

5.2 Conclusão

Através do componente `OrocosWam`, pode-se fazer uma interface entre a biblioteca do robô e a arquitetura aberta utilizada. Esta interface permitiu inicializar o robô dentro de um componente e disponibilizar suas funcionalidades através de uma interface acessível a outros componentes. A utilização desta interface em uma arquitetura externa, permitiu a abstração dos elementos de controle internos à biblioteca do robô dentro de componentes. As duas topologias apresentadas mostram esta abstração, sendo possível abstrair o controlador da biblioteca dentro do componente `ControllerWAM`, ou utilizar um controle implementado diretamente nos componentes, através do componente `ControllerNPID`. É importante observar que há um ganho de modularidade muito grande, uma vez que as tarefas de controle se tornam explícitas através de cada componente, onde cada um tem uma tarefa claramente definida. Um exemplo da vantagem desta modularidade está nos dois componentes de controle apresentados, que permitem realizar o controle de duas formas distintas. O componente PID também exemplifica esta modularidade, uma vez que, se quisermos alterar a lei de controle utilizada no sistema, basta apenas substituir os componentes PID por componentes novos, que implementem a lei de controle desejada. A interface com o usuário também é modular e facilmente alterável, uma vez que basta trocar o componente `Reference` por um outro componente ou mesmo um *software* gráfico, que receba os comandos do usuário e execute os envie para o sistema.

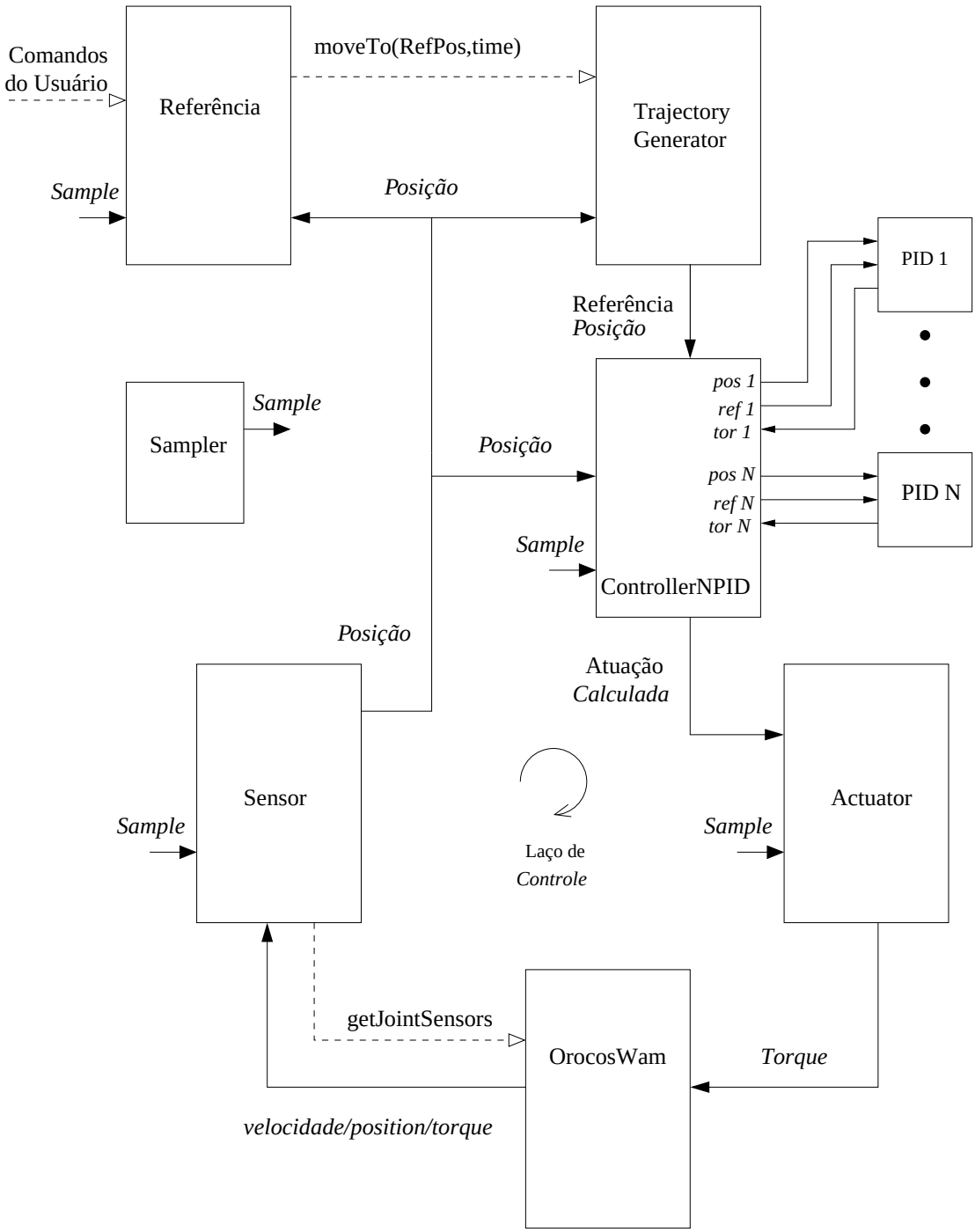

Figura 7: Controle com `ControllerNPID`

6 CONCLUSÕES

Foi proposto neste trabalho, uma interface de integração entre o *framework* OROCOS ao robô manipulador Barrett WAM. Um componente foi criado para exportar algumas funcionalidades da biblioteca `Libbarrett` através da interface padrão do OROCOS.

Para acessar as funcionalidades providas por esta interface, ela foi integrada a uma arquitetura aberta, a qual define os componentes que abstraem o sistema de controle.

A revisão do estado da arte no desenvolvimento de *software* robótico, permitiu conhecer modernas técnicas utilizadas, como a CBSE e os *frameworks* de robótica, e usufruir de suas vantagens para facilitar e dar confiabilidade ao trabalho realizado.

A interface proposta foi implementada através da criação do componente `OrocosWam` que permite a inicialização, acesso à leitura dos sensores e ao controlador da biblioteca e escrita de valores de atuação do robô. Estas funcionalidades, permitiram integrar o componente à arquitetura aberta proposta em SANTINI (2009), através de duas topologias: uma utilizando o controlador interno da biblioteca do robô e outra utilizando o controlador PID utilizado na arquitetura, como foi apresentado no Capítulo 5.

Como desejava-se deixar o presente trabalho compatível com desenvolvimentos atuais, a arquitetura utilizada também foi portada para versão atual do OROCOS (2.5). Isto permite utilizá-la na nova versão não somente neste trabalho, mas também na sua proposta original e em trabalhos futuros. O Capítulo 2 mostra as modificações básicas no OROCOS 2.5 em relação ao 1.8, além das adaptações necessárias aos componentes da arquitetura utilizada, para que eles fossem adaptados à nova versão.

Os desafios encontrados na execução do projeto, podem ser citados a aprendizagem e a utilização do *framework* OROCOS e o estudo da biblioteca do robô. O OROCOS necessita de um bom conhecimento de linguagem C++, leitura de manuais de utilização e o desenvolvimento de vários exemplos, para poder utilizá-lo. Apesar de estar na documentação que o OROCOS suporta o RTAI, encontrou-se dificuldades na utilização conjunta destes sistemas. Em contato com desenvolvedores do OROCOS, constatou-se que o OROCOS 2.0 ainda não está completamente funcional com o RTAI. Quanto a biblioteca do robô, o maior desafio foi entender a sua construção, em virtude de sua pobre documentação. As classes e funções da biblioteca não possuem comentários, em sua maioria, a respeito de sua utilização, funcionalidade, parâmetros e valores de retorno. Isto coloca-se como uma grande dificuldade para o programador que irá utilizá-la, uma vez que ele precisa ler e entender os códigos diretamente, o que não seria necessário, caso fosse bem documentado.

O Capítulo 3 apresenta uma documentação básica sobre o Barrett WAM, criada ao longo deste trabalho, e espera-se que em conjunto ao Capítulo 2 possam servir como uma introdução ao OROCOS e à `Libbarrett`, a fim de encorajar e ajudar novos usuários na suas utilizações.

Para futuros trabalhos, pode-se trabalhar em diferentes níveis. Em baixo nível, espera-se que seja feita uma interface de comunicação diretamente com os Pucks, através do barramento CAN, o que poderia eliminar a necessidade de utilização da biblioteca do robô, permitindo assim que todo *software* fosse modularizado em componentes. Em alto nível, deve-se a criar uma interface gráfica com o usuário, de forma que a utilização do robô fosse mais simples e intuitiva. Aproveitando a modularidade da arquitetura, pode-se também trabalhar diferentes leis de controle aplicadas ao robô Barrett WAM.

REFERÊNCIAS

BROOKS, A.; KAUPP, T.; MAKARENKO, A.; OREBACK, A.; WILLIAM, S. Towards Component-based Robotics. In: IEEE/RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, 2005, Edmonton, Canadá. **Proceedings...** IEEE, 2005. p.163–168.

BRUGALI, D.; SCANDURRA, P. Component-Based Robotic Engineering, Part I: reusable building blocks. **IEEE Robotics and Automation Magazine**, Piscataway, NJ, USA, v.16, n.4, p.84–96, Dec. 2009.

BRUYNINCKX, H. Open robot control software: the orocos project. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2001., 2001, Seoul, Coréia do Sul. **Proceedings...** Piscataway: NJ: USA: IEEE Press, 2001. v.3, p.2523–2528.

COLLIN-COPE, M. **Component based development and advanced OO design**. 2001.

GERKEY, B. P.; VAUGHAN, R. T.; HOWARD, A. The Player/Stage Project: tools for multi-robot and distributed sensor systems. In: INTERNATIONAL CONFERENCE ON ADVANCED ROBOTICS (ICAR'03), 11., 2003, Coimbra, Portugal. **Proceedings...** IEEE, 2003. p.317–323.

HENNING, M. A new approach to object-oriented middleware. **IEEE Internet Computing**, Piscataway, NJ, USA, v.8, p.66–75, Jan. 2004.

JUNG, M. Y.; DEGUET, A.; KAZANZIDES, P. A component-based architecture for flexible integration of robotic systems. In: IEEE/RSJ INTERNATIONAL CONFERENCE ON INTELLIGENT ROBOTS AND SYSTEMS, 2010, Taipei, Taiwan. **Proceedings...** IEEE, 2010. p.6107–6112.

LIBBARRETT Doxygen Documentation. Disponível em: `http://web.barrett.com/libbarrett/`. Acesso em: outubro 2011.

LIBBARRETT Programming Manual. Cambridge, Massachusetts: Barrett Technology, Inc., 2011.

OROCOS Component Builder's Manual. Disponível em: `http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos-components-manual.html`. Acesso em: outubro 2011.

OROCOS Project Smarter Control in Robotics and Automation. Disponível em: `http://www.orocos.org/`. Acesso em: outubro 2011.

QUIGLEY, M.; GERKEY, B.; CONLEY, K.; FAUST, J.; FOOTE, T.; LEIBS, J.; BERGER, E.; WHEELER, R.; NG, A. ROS: an open-source robot operating system. In: IEEE INTL. CONF. ON ROBOTICS AND AUTOMATION (ICRA) WORKSHOP ON OPEN SOURCE ROBOTICS, 2009, Kobe, Japão. **Proceedings...** IEEE, 2009.

RTAI. **RealTime Application Interface**. Disponível em: `https://www.rtai.org/`. Acesso em: Novembro 2011.

SANTINI, D. C. **Arquitetura Aberta para Controle de Robôs Manipuladores**. 2009. Mestrado em Engenharia Elétrica — Escola de Engenharia, Departamento de Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre.

THE cisst libraries. Disponível em: `http://www.cisst.org/cisst`. Acesso em: outubro 2011.

WAM User Manual. Cambridge, Massachusetts: Barrett Technology, Inc., 2011.

XENOMAI General. Disponível em: `http://www.cs.ru.nl/lab/xenomai/`. Acesso em: novembro 2011.

XENOMAI: real-time framework for linux. Disponível em: `http://www.xenomai.org/`. Acesso em: novembro 2011.