Sabrina Araújo Cardoso

Controle orientado a eventos de uma fábrica inteligente baseado em árvores de comportamento e numa estrutura de programação distribuída

Campina Grande, Paraíba

Novembro de 2024

Sabrina Araújo Cardoso

Controle orientado a eventos de uma fábrica inteligente baseado em árvores de comportamento e numa estrutura de programação distribuída

Trabalho de Conclusão de Curso submetido à Coordenadoria do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, *Campus* Campina Grande, como parte dos requisitos necessários para obtenção do Grau de Bacharel em Engenharia Elétrica.

Universidade Federal de Campina Grande - UFCG

Centro de Engenharia Elétrica e Informática - CEEI

Departamento de Engenharia Elétrica - DEE

Orientador: Antonio Marcus Nogueira Lima, Dr.

Campina Grande, Paraíba

Novembro de 2024

Sabrina Araújo Cardoso

Controle orientado a eventos de uma fábrica inteligente baseado em árvores de comportamento e numa estrutura de programação distribuída

Trabalho de Conclusão de Curso submetido à Coordenadoria do Curso de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, *Campus* Campina Grande, como parte dos requisitos necessários para obtenção do Grau de Bacharel em Engenharia Elétrica.

Trabalho aprovado em: 04 de Novembro de 2024

Antonio Marcus Nogueira Lima, Dr.
Orientador

Prof. Dr. Gutemberg Gonçalves dos Santos Júnior
Professor Convidado

Campina Grande, Paraíba
Novembro de 2024

Dedico este trabalho a minha família e à todos que acreditaram em mim.

Agradecimentos

Gostaria de agradecer, primeiramente a Deus, por ter me dado força, coragem, foco e muita perseverança para que eu pudesse encarar todos os obstáculos encontrados durante a graduação.

Também quero agradecer imensamente a minha família. Em especial, a minha mãe, Iranilda, que mesmo diante de todas as dificuldades enfrentadas, nunca me deixou desamparada e nunca desacreditou do meu potencial. E a minha irmã, Silvana, que sempre foi minha fonte de inspiração. Elas sempre me apoiaram em todas as minha decisões. E sei que posso contar sempre com elas para o que for preciso. Também quero agradecer a meu sobrinho, Diego Filho, que chegou ao mundo há menos de 1 ano, e foi a luz que me deu forças para não desanimar nessa reta final.

Agradeço a minha companheira Jade, que sempre esteve ao meu lado em todos os momentos de estresse e me deu forças quando nem eu mesma acreditava que conseguiria ter. Sem sua companhia, minha jornada no curso seria mais ardosa. Obrigada por deixar tudo mais leve.

Não posso deixar de agradecer aos meus amigos de curso, em especial Débora e Dhara, que sempre me acompanharam nos momentos de café, estudo e desabafo. Com vocês foi possível descontrair e tentar amenizar toda a carga acompanhada pela graduação. Agradeço também a Marina e Felipe, por estarem sempre me dando força no ambiente de trabalho e tornando tudo mais divertido e leve. Sem vocês, esse trabalho não seria finalizado.

Agradeço ao professor Antonio Marcus por ter me acolhido desde o segundo período do curso e sempre ter me dado oportunidade. O senhor foi essencial para o meu crescimento acadêmico e profissional.

Meus sinceros obrigada à todos!

Resumo

Neste trabalho de conclusão de curso é apresentado o desenvolvimento de um sistema de controle orientado a eventos para uma fábrica inteligente, utilizando árvores de comportamento integradas ao *framework* ROS. A implementação foi realizada em um ambiente experimental que inclui o robô manipulador UR10, o robô móvel TurtleBot2i e sensores de visão, como a câmera Kinect. O foco do projeto está na coordenação autônoma desses dispositivos para tarefas de manipulação e transporte de objetos, com a detecção de marcadores ArUco e a adaptação às condições do ambiente em tempo real. A metodologia empregada inclui a modelagem de um supervisor em árvore de comportamento que gerencia as operações do UR10 com base nas informações de localização do objeto enviadas pelo Kinect e no transporte realizado pelo TurtleBot2i. Os resultados demonstram a capacidade do sistema de executar tarefas de forma sincronizada e eficiente, com flexibilidade para integrar novos dispositivos e adaptar-se a diferentes cenários de produção. Como trabalhos futuros, propõem-se a implementação de um sistema de navegação autônoma para o TurtleBot2i e a setorização dinâmica de objetos com base na identificação visual, bem como o desenvolvimento de um gêmeo digital para simulação e aprimoramento do controle cinemático do UR10.

Palavras-chave: Controle orientado a eventos, fábrica inteligente, ROS, robótica, modelagem, árvores de comportamento.

Abstract

This thesis presents the development of an event-driven control system for a smart factory using behavior trees integrated with the ROS framework. The implementation was carried out in an experimental environment that includes the UR10 robotic manipulator, the TurtleBot2i mobile robot, and vision sensors such as the Kinect camera. The project focuses on the autonomous coordination of these devices for object manipulation and transportation tasks, utilizing ArUco marker detection and real-time adaptation to environmental conditions. The methodology involves designing a behavior tree-based supervisor that manages the UR10 operations based on object location data provided by the Kinect and transportation conducted by the TurtleBot2i. Results demonstrate the system's ability to perform tasks in a synchronized and efficient manner, with flexibility to integrate new devices and adapt to various production scenarios. Future work suggests implementing autonomous navigation for the TurtleBot2i, dynamic object sectorization based on visual identification, and the development of a digital twin for simulation and enhancement of UR10's kinematic control.

Keywords:Event-driven control, Smart factory, ROS, robotics, TurtleBot2i, modeling, behavior trees.

Lista de ilustrações

Lista de Abreviaturas e Acrônimos

CEEI	Centro de Engenharia Elétrica e Informática.
UFCG	Universidade Federal de Campina Grande.
Embedded	Laboratório de Sistemas Embarcados e Computação Pervasiva.
ROS	*Robot Operating System.*
SED	Sistemas a Eventos Discretos.
DH	Denavit-Hartenberg.
UR10	Universal Robots modelo UR10.
DoF	*Degrees of Freedom* (Graus de Liberdade).
GUI	*Graphical User Interface* (Interface Gráfica do Usuário).
ICR	*Instantaneous Center of Rotation* (Centro Instantâneo de Rotação).
AI	*Artificial Intelligence* (Inteligência Artificial).
I4.0	Indústria 4.0.
RViz	*ROS Visualization* (Ferramenta de Visualização do ROS).
3D	Três Dimensões.
IoT	*Internet of Things* (Internet das Coisas).
CPS	*Cyber Phisycs Systems*(Sistemas Ciber-Físico).
GD	Gêmeo Digital.
STL	*Standard Tessellation Language.*
URDF	*Unified Robot Description Format.*

Sumário

1 Introdução

Neste capítulo serão apresentadas as justificativas e os objetivos desenvolvidos neste trabalho. Por fim, será descrita a organização dos capítulos e das seções deste documento.

1.1 Justificativa

A Indústria 4.0 (I4.0), marca uma transformação nos processos produtivos ao promover a fusão entre os mundos físico e digital por meio de tecnologias emergentes, como a Internet das Coisas (IoT), sistemas ciber-físicos (CPS) e inteligência artificial (AI). Com origem na Alemanha, em 2011, esse movimento visa criar fábricas inteligentes capazes de operar de maneira interconectada e adaptativa, respondendo em tempo real às flutuações do mercado e às demandas de personalização. As fábricas inteligentes, ou *smart factories*, representam o ápice da I4.0 ao implementar automação avançada e controle descentralizado, permitindo uma produção flexível e personalizada. Nessas fábricas, dispositivos conectados e sistemas autônomos trabalham em sincronia, otimizando o uso de recursos e a eficiência (ONU; PRADHAN; MBOHWA, 2023; GORI; GORI, 2022).

Para alcançar o potencial das fábricas inteligentes, o *Robot Operating System* (ROS) desempenha um papel importante. Com o ROS, é possível coordenar robôs, sensores e atuadores para que trabalhem de forma integrada, respondendo de maneira eficaz a eventos discretos e adaptando-se em tempo real às variações do ambiente produtivo (Open Robotics, 2024). Além disso, o ROS permite o uso de ferramentas de simulação, como o Gazebo, que viabilizam testes e otimizações antes da implementação no ambiente físico, reduzindo custos e riscos associados à inovação (DELOITTE, 2020; ŽEMLA et al., 2023).

A estrutura de controle orientada a eventos baseada em árvores de comportamento é especialmente adequada para o ROS, pois proporciona a modularidade e a flexibilidade necessárias para ambientes produtivos dinâmicos. Árvores de comportamento são amplamente utilizadas em robótica, permitindo a coordenação de tarefas, e, quando integradas ao ROS, facilitam a programação e a execução de fluxos de trabalho de maneira escalável e adaptável. Ao adotar essa abordagem em uma estrutura de programação distribuída, este estudo explora o potencial do ROS e das árvores de comportamento para otimizar o controle de fábricas inteligentes, promovendo uma operação autônoma e descentralizada, alinhada aos requisitos da I4.0 (ÖGREN; SPRAGUE, 2022; HEPPNER et al., 2023).

A justificativa para este trabalho está na necessidade de desenvolver e avaliar um

modelo de controle distribuído que integre o ROS e tecnologias de árvores de comportamento, explorando a automação orientada a eventos em fábricas inteligentes. A utilização do ROS como base para a implementação de sistemas distribuídos oferece uma solução promissora para o controle em ambientes industriais, facilitando não apenas a gestão eficiente de dispositivos, mas também uma comunicação robusta entre eles. Essa abordagem promove a autonomia e flexibilidade operacional, alinhando-se aos objetivos I4.0, ao mesmo tempo em que traz soluções para desafios de coordenação e controle de sistemas complexos (CSALÓDI et al., 2021; ALVARES et al., 2023).

1.2 Objetivo Geral

O objetivo geral deste trabalho é desenvolver uma arquitetura de controle orientado a eventos para uma fábrica inteligente. Esta arquitetura utilizará árvores de comportamento e uma estrutura de programação distribuída para integrar e gerenciar a comunicação entre diferentes dispositivos, como robôs móveis, manipuladores robóticos e sensores de visão, de forma modular e escalável. A proposta visa criar uma plataforma que permita a automação inteligente das operações industriais, fornecendo suporte para tomadas de decisão autônomas e eficientes.

Para alcançar o objetivo geral, foram estabelecidos os seguintes objetivos específicos:

- Implementar uma árvore de comportamento para gerenciar tarefas e interações entre dispositivos na fábrica inteligente, permitindo a coordenação autônoma de robôs e sensores.
- Integrar o sistema de controle com o ROS, possibilitando uma comunicação eficaz entre os dispositivos e o sistema central.
- Desenvolver simulações para validar a funcionalidade e a eficiência do sistema proposto, incluindo a movimentação e interação entre robôs e sensores.
- Realizar testes de validação do sistema implementado para avaliar a eficácia da arquitetura de controle na execução de tarefas em um ambiente industrial experimental.

1.3 Organização do Documento

O documento está organizado em seis capítulos. O Capítulo 2, aborda os conceitos essenciais para a compreensão do estudo. Também são explorados temas de cinemática

de braço robótico, robôs móveis, e fundamentos do ROS. Este capítulo finaliza com uma discussão sobre árvores de comportamento.

O Capítulo 3, descreve a configuração do ambiente experimental e os dispositivos utilizados. Também aborda a integração desses dispositivos com o ROS, a modelagem cinemática do UR10 e as etapas de simulação, incluindo a estrutura dos pacotes e sua integração com o ROS.

O Capítulo 4, discute o desenvolvimento de um modelo de controle baseado em Sistemas a Eventos Discretos (SED) aplicado ao UR10 e a modelagem do supervisor em árvores de comportamento, adequados para uma fábrica inteligente.

O Capítulo 5, apresenta a implementação prática da árvore de comportamento no ROS, detalhando os testes e validações das funcionalidades desenvolvidas, com foco na interação entre os componentes simulados.

Finalmente, o Capítulo 6, resume os principais achados do trabalho, discute as contribuições e limitações da abordagem adotada, e sugere direções para estudos futuros. Essa estrutura visa guiar o leitor desde a fundamentação até a aplicação prática e as conclusões, oferecendo uma visão completa do desenvolvimento e validação do controle orientado a eventos em uma fábrica inteligente.

2 Fundamentação Teórica

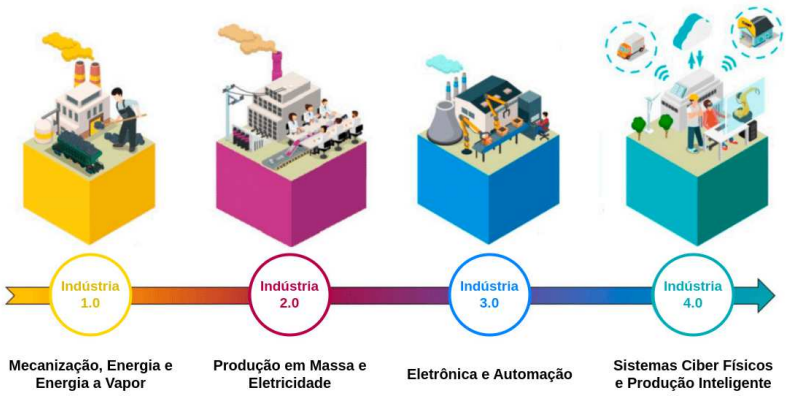
Neste capitulo são apresentados os conceitos-chave que formam a base teórica necessária para o desenvolvimento deste trabalho.

2.1 Indústria 4.0

A Indústria 4.0 (I4.0), também conhecida como a Quarta Revolução Industrial, representa uma transformação dos processos produtivos, marcada pela fusão dos mundos físico e digital através de tecnologias emergentes. Esse conceito surgiu na Feira de *Hannover*, Alemanha, em 2011, com o objetivo de impulsionar a competitividade industrial e digitalizar os sistemas de produção. Seu reconhecimento global ocorreu em 2016, quando o Fórum Econômico Mundial adotou o termo (DELOITTE, 2020). De acordo com (ŽEMLA et al., 2023), este foi um marco na transição para um modelo industrial conectado, flexível e autônomo.

A I4.0 propõe uma integração das tecnologias digitais nos sistemas de produção, criando um ambiente onde máquinas, dispositivos e pessoas se comunicam em tempo real. Esse modelo aumenta a eficiência, flexibilidade e personalização dos processos industriais. Segundo (CSALÓDI et al., 2021), a evolução histórica da indústria pode ser dividida em quatro fases principais (como ilustrado na Figura 1), que retratam a evolução das revoluções industriais.

Figura 1 – As revoluções industriais.



Fonte: Adaptado de (BENOTSMANE; KOVÁCS; DUDÁS, 2019).

- Primeira Revolução Industrial: iniciada no final do século XVIII, introduziu a mecanização da produção artesanal através de máquinas a vapor.

- Segunda Revolução Industrial: no início do século XX, a eletrificação das fábricas e a introdução da linha de montagem, idealizada por Henry Ford, permitiram a produção em massa, reduzindo custos e tornando bens de consumo mais acessíveis.
- Terceira Revolução Industrial: a partir da década de 1970, a automação dos processos trouxe maior controle sobre os processos produtivos e introduziu a programação computacional como uma ferramenta na manufatura.
- Quarta Revolução Industrial: de acordo com (ALVARES et al., 2023), a I4.0 representa a convergência entre o mundo físico e o digital, onde tecnologias como Internet das Coisas (IoT), Sistemas Ciber-Físicos (CPS), *Big Data* e Inteligência Artificial (AI) permitem que as fábricas sejam capazes de se adaptar dinamicamente às mudanças de produção, personalizar produtos em massa e operar de forma autônoma.

Os CPS são primordiais na I4.0, integrando sistemas computacionais com processos físicos para que máquinas e dispositivos se comuniquem e atuem de maneira coordenada. Isso cria uma relação contínua entre o ambiente físico e o digital, onde sistemas monitoram e ajustam processos automaticamente. Esse processo é baseado na análise de dados em tempo real, resultando em tomadas de decisão mais rápidas e informadas (ŽEMLA et al., 2023).

A IoT permite a interconexão de máquinas, sensores e dispositivos através da internet, possibilitando a coleta e troca de dados em tempo real. Isso viabiliza, por exemplo, que sensores em uma linha de produção identifiquem uma falha iminente e comuniquem-se com outros dispositivos para ajustar a operação ou acionar manutenção preventiva, evitando paradas inesperadas e perdas de eficiência.

Outro componente importante da I4.0 é o Gêmeo Digital (GD), que se trata de uma réplica virtual de sistemas físicos onde é possível simular, prever e otimizar operações. Na manufatura, os GDs são empregados para antecipar falhas e testar novas configurações de produção, reduzindo custos e minimizando riscos de interrupção. Além disso, os dados gerados pelos GDs ajudam a identificar padrões e tendências que orientam a melhoria contínua da eficiência e da qualidade dos processos (ALVARES et al., 2023).

A robótica é um dos pilares fundamentais da I4.0, desempenhando um papel importante na automação dos processos produtivos. Os avanços em robótica possibilitaram que robôs executem não apenas tarefas repetitivas, mas também trabalhem em conjunto com humanos, colaborando de forma segura e adaptável.

Essas inovações possibilitaram a criação de fábricas inteligentes, onde máquinas e sistemas compartilham dados em tempo real, otimizando operações autônomas e aumentando sua eficiência (IBM, 2020).

Portanto, a I4.0 marca uma transição no modelo produtivo tradicional, levando a um ambiente flexível e conectado, onde todos os elementos, de máquinas a operadores, colaboram para otimizar a produção. Em última análise, a I4.0 estabelece uma base para a *Smart Factory* (Fábrica Inteligente), na qual os sistemas produtivos são capazes de aprender e se auto-otimizar, garantindo uma produção mais eficiente, personalizada e autônoma.

2.1.1 Fábrica Inteligente

A fábrica inteligente é a implementação prática dos princípios da I4.0, criando um ambiente de produção flexível, conectado e autônomo, que utiliza tecnologias digitais para otimizar e adaptar continuamente seus processos. Ou seja, a fábrica inteligente é a manifestação concreta desse conceito, aplicando tecnologias como *IoT*, *CPS*, GD, *AI* e robótica para criar um ambiente produtivo.

Uma diferença importante entre a I4.0 e a fábrica inteligente está no enfoque. Enquanto a primeira busca a integração digital e a conectividade em larga escala, a segunda se preocupa em aplicar essas inovações de forma prática para melhorar a produção. Nesse contexto, os GDs desempenham um papel importante ao simular processos e operações antes de sua implementação no ambiente físico, o que reduz custos e minimiza riscos (CAGGIANO; TETI, 2018).

Tecnologias como a robótica colaborativa também são fundamentais para as fábricas inteligentes. De acordo com (RAHMAN; ARTHUR,), a automação inteligente permite que as fábricas se ajustem rapidamente às demandas dinâmicas, melhorando a capacidade de resposta ao mercado.

Em suma, a fábrica inteligente é a materialização do potencial da I4.0. Ela se caracteriza pela conectividade total, integração de sistemas, flexibilidade produtiva e capacidade de aprendizado e auto-otimização. Essas características permitem que elas respondam de maneira mais eficaz às mudanças no mercado e às demandas dos clientes, proporcionando vantagens competitivas significativas em um ambiente industrial cada vez mais dinâmico.

2.2 Introdução à Sistemas a Eventos Discretos

Os Sistemas a Eventos Discretos (SED) têm ganhado destaque em aplicações práticas, especialmente em áreas como automação industrial, redes de comunicação e controle de tráfego. Esses sistemas respondem a eventos específicos em vez de mudanças contínuas no tempo, sendo importantes para o funcionamento de ambientes que demandam respostas rápidas. Este capítulo apresenta os fundamentos teóricos dos SED, baseando-se nos conceitos explorados por (CASSANDRAS; LAFORTUNE, 2008).

Para entender os SED, é necessário, inicialmente, discutir o conceito de "sistema" e a teoria de controle que o sustenta, uma vez que estes fornecem a base para a modelagem e análise de sistemas dinâmicos. A palavra sistema é utilizado em diversas áreas do conhecimento e sua definição é intuitiva. Diferentes fontes literárias fornecem definições do termo, mas o *IEEE Standard Dictionary of Electrical and Electronic Terms* oferece uma definição voltada para a engenharia, considerando um sistema como: "uma combinação de componentes que atuam em conjunto para realizar uma função que não seria possível com os componentes individuais".

A análise quantitativa de sistemas e o desenvolvimento de técnicas de controle são fundamentais para medir e otimizar o desempenho de sistemas com base em critérios bem definidos. Para isso, é necessário construir modelos matemáticos que representem o comportamento real dos sistemas. Em termos simples, um modelo pode ser entendido como uma "representação" que imita o comportamento do sistema original.

Um dos processo de modelagem mais simples é o de entrada e saída. Nele, é definido um conjunto de variáveis mensuráveis associadas ao sistema, as quais representam valores reais observados ao longo de um intervalo de tempo $[t_0, t_f]$. Dentre essas variáveis, seleciona-se um subconjunto chamado de "variáveis de entrada", que podem ser controladas ao longo do tempo $\mathbf{u}(t) = [u_1(t), \ldots, u_p(t)]^T$. Essas variáveis funcionam como estímulos ao sistema. Em paralelo, é escolhido as "variáveis de saída" $\mathbf{y}(t) = [y_1(t), \ldots, y_m(t)]^T$, que podem ser medidas diretamente enquanto as entradas variam. Essas variáveis representam as respostas do sistema aos estímulos de entrada.

O objetivo de um modelo matemático é encontrar uma função que descreva a relação entre as variáveis de entrada e saída tal como

$$\mathbf{y} = \mathbf{g}(\mathbf{u}) = [g_1(u_1(t), \ldots, u_p(t)), \ldots, g_m(u_1(t), \ldots, u_p(t))]^T, \tag{2.1}$$

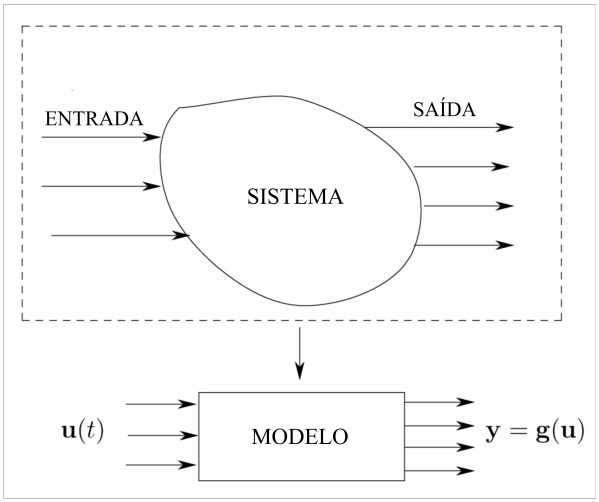
na qual $\mathbf{g}(\cdot)$ é o vetor de funções que mapeia as entradas $\mathbf{u}(t)$ nas saídas $\mathbf{y}(t)$. A Figura 2 ilustra esse processo.

Na modelagem de sistemas, uma distinção importante é feita entre sistemas "estáticos" e "dinâmicos". Em um sistema estático, a saída $y(t)$ depende apenas do valor atual da entrada $u(t)$, ou seja, o sistema não apresenta "memória" de estados passados.

Outra classificação importante para sistemas dinâmicos é a distinção entre sistemas "variantes" e "invariantes" no tempo. Um sistema é invariante no tempo se seu comportamento não mudar ao longo do tempo. Isso significa que, se uma entrada específica produz uma determinada saída, essa relação entre entrada e saída será a mesma independentemente do momento em que a entrada for aplicada.

Por outro lado, um sistema variante no tempo apresenta mudanças em seu comportamento ao longo do tempo. Nesse caso, a relação entre a entrada e a saída depende

Figura 2 – Processo de Modelagem Simples.



Fonte: Adaptado de (CASSANDRAS; LAFORTUNE, 2008).

do instante em que a entrada é aplicada, ou seja, uma mesma entrada pode gerar saídas diferentes em momentos distintos.

O "estado" de um sistema em um dado instante de tempo representa uma coleção de informações necessárias para descrever seu comportamento naquele momento. Em outras palavras, o estado de um sistema em um instante t_0 é o conjunto de informações disponíveis que, combinado com $\mathbf{u}(t)$ para $t \geq t_0$, permite determinar $\mathbf{y}(t)$ de forma única para todos $t \geq t_0$.

O estado de um sistema é geralmente representado por um vetor $\mathbf{x}(t)$, cujas componentes $x_1(t), \ldots, x_n(t)$ são conhecidas como variáveis de estado. Essas variáveis representam diferentes dimensões da "memória" do sistema e capturam informações sobre seu comportamento passado e presente.

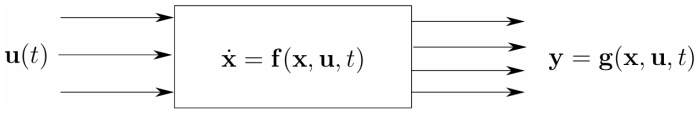
As equações de estado fornecem um conjunto de expressões necessárias para especificar o estado $\mathbf{x}(t)$ para todos os instantes $t \geq t_0$, dado o estado inicial $\mathbf{x}(t_0)$ e a função de entrada $\mathbf{u}(t)$ para $t \geq t_0$. Além disso, o espaço de estados de um sistema é definido como o conjunto de todos os valores possíveis que o estado pode assumir. Para construir um modelo em espaço de estados de um sistema, são definidas as seguintes equações:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2.2}$$

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t) \tag{2.3}$$

A primeira equação representa o conjunto de equações de estado, que descrevem a dinâmica do sistema a partir de suas condições iniciais $\mathbf{x}(t_0) = x_0$. Já a segunda equação define as equações de saída. A Figura 3 ilustra esse processo de modelagem em espaço de estados.

Figura 3 – Diagrama de blocos representando o modelo completo.



Fonte: (CASSANDRAS; LAFORTUNE, 2008).

A escolha das variáveis de estado geralmente depende da intuição, experiência e da presença de quantidades físicas naturais que servem como variáveis de estado.

2.2.0.1 Controle por realimentação

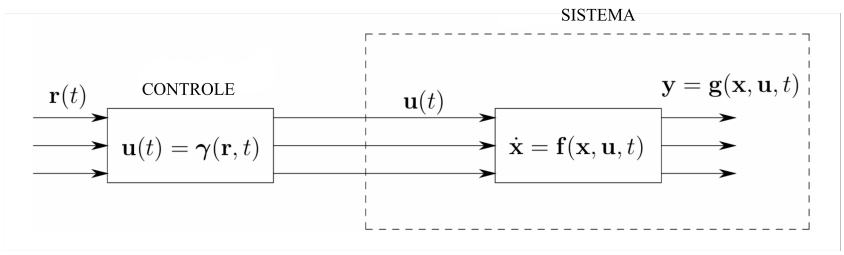
Até aqui, foi discutido como o sistema responde a uma entrada específica. Mas em aplicações práticas, um sistema geralmente existe para realizar uma função específica ou para alcançar um comportamento desejado. Para cumprir essa função, ele precisa ser controlado por "sinais de controle". Por exemplo, ao dirigir um carro, o motorista ajusta o volante, o acelerador e o freio para manter o veículo na estrada e em segurança. Nesse caso, o comportamento desejado é manter o carro em segurança e em movimento suave ao longo da via.

Em termos gerais, esse comportamento é definido por um sinal de referência $r(t)$, e a entrada de controle é ajustada conforme uma lei de controle da forma:

$$\mathbf{u}(t) = \gamma(\mathbf{r}(t), t), \tag{2.4}$$

na qual γ é uma função que determina o sinal de controle $\mathbf{u}(t)$ com base na referência $\mathbf{r}(t)$. Este conceito é ilustrado na Figura 4, que apresenta a função de controle no processo de modelagem de sistemas.

Figura 4 – Diagrama de blocos representando o modelo completo e a lei de controle.



Fonte: Adaptado de (CASSANDRAS; LAFORTUNE, 2008).

O conceito de realimentação (*feedback*) trata de utilizar qualquer informação disponível sobre o comportamento do sistema para ajustar continuamente a entrada de controle. Ele é amplamente utilizado no cotidiano em diversas formas, em uma conversa, por

exemplo, falamos quando a outra pessoa está em silêncio e escutamos quando ela começa a falar, ajustando nossa interação com base nas reações do outro. Sua principal vantagem é permitir correções em tempo real, especialmente em presença de distúrbios inesperados.

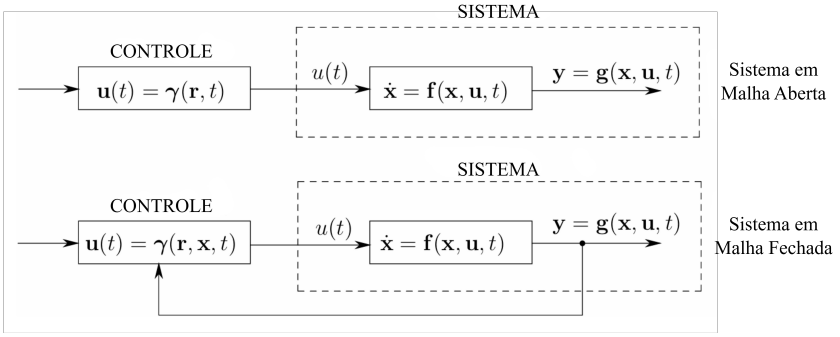
Matematicamente, o uso de realimentação implica estender a lei de controle para incluir, além de $\mathbf{r}(t)$, $\mathbf{y}(t)$ ou, de forma mais geral, o estado $\mathbf{x}(t)$. Assim, uma lei de controle mais completa pode ser escrita como:

$$\mathbf{u}(t) = \gamma(\mathbf{r}(t), \mathbf{x}(t), t) \tag{2.5}$$

A possibilidade de utilizar *feedback* no controle de um sistema leva a uma classificação importante: sistemas em malha aberta e em malha fechada.

Em um sistema de malha aberta, o controle é realizado sem realimentação da saída ou do estado do sistema. Isso significa que a entrada permanece fixa independentemente do impacto (positivo ou negativo) que tem sobre a saída observada. Em contrapartida, em um sistema de malha fechada, $\mathbf{u}(t)$ depende do estado atual ou da saída do sistema, sendo ajustado continuamente com base no *feedback*, formando uma "malha fechada" de controle que permite correções dinâmicas para atingir o comportamento desejado. Esse processo é ilustrado na Figura 5, onde a realimentação é representada pela inclusão de informações sobre o estado no modelo de controle.

Figura 5 – Sistemas em Malha Aberta e em Malha Fechada.



Fonte: Adaptado de (CASSANDRAS; LAFORTUNE, 2008).

2.2.1 Sistemas a Eventos Discretos

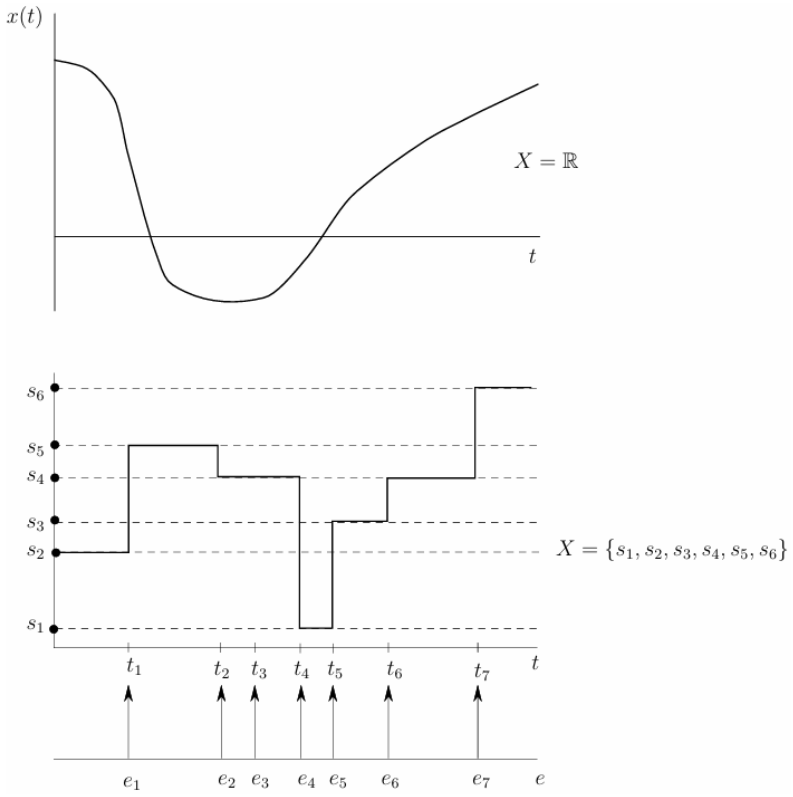
Os SED são uma classe de sistemas dinâmicos onde o espaço de estados é descrito por um conjunto discreto de valores e as transições de estado ocorrem apenas em pontos discretos no tempo. Essas transições estão associadas a "eventos" que desencadeiam mudanças de estado, caracterizando o comportamento do sistema.

O conceito de "evento" pode ser entendido como uma ocorrência instantânea que provoca uma transição de estado de um valor para outro. Por exemplo, um evento pode

representar uma ação específica (como pressionar um botão), uma ocorrência espontânea ditada por fatores externos (como uma falha de computador) ou o resultado de várias condições que são simultaneamente atendidas (como o nível de um fluido em um tanque excedendo um valor crítico).

Para representar os eventos em SED, utiliza-se o símbolo e e define-se um conjunto E, que reúne todas as possíveis ocorrências. A Figura 6 ilustra a diferença entre sistemas dinâmicos com variáveis contínuas e SED. Em um sistema contínuo, o estado $x(t)$ varia continuamente ao longo do tempo e é governado por uma equação diferencial, como na Eq. 2.2. Em contraste, os SED apresentam estados que pertencem a um conjunto discreto (por exemplo, $X = \{s_1, s_2, s_3, s_4, s_5, s_6\}$), e suas transições de estado ocorrem somente em resposta a eventos discretos.

Figura 6 – Comparação de caminhos de amostra para sistemas de variáveis contínuas e SED.



Fonte: (CASSANDRAS; LAFORTUNE, 2008).

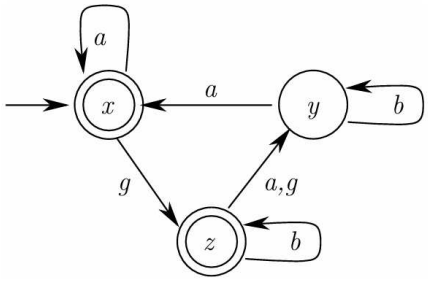
Uma forma de estudar o comportamento lógico dos SED é utilizando teorias de linguagens e autômatos. Nesse contexto, o conjunto de eventos E de um SED é considerado o "alfabeto" de uma "linguagem", onde as sequências de eventos representam as "palavras" dessa linguagem. Uma sequência de eventos do alfabeto forma uma "palavra". Se s é uma palavra, seu comprimento, denotado por $|s|$, é o número de eventos nela contidos, incluindo múltiplas ocorrências de um mesmo evento.

A linguagem é entendida como uma maneira de descrever o comportamento do sistema, especificando todas as sequências de eventos que ele pode processar ou gerar. Algumas linguagens podem ser descritas por enumeração simples, enquanto outras, como linguagens infinitas, exigem descrições mais complexas. Para lidar com essas linguagens, são necessárias estruturas compactas e manipuláveis, como "autômatos".

A forma mais intuitiva de entender um autômato é através de um diagrama de transição de estados, onde cada nó representa um estado, e cada arco, um evento, que indica uma transição entre estados.

Considerando um conjunto de eventos $E = \{a, b, g\}$ e o diagrama de transição de estados mostrado na Figura 7, os nós (x, y, z) representam os estados do autômato e os arcos E representam as transições entre esses estados. Ou seja, cada arco descreve as mudanças de estado do autômato em resposta a um evento específico.

Figura 7 – Diagrama de Transição de Estados.



Fonte: (CASSANDRAS; LAFORTUNE, 2008).

Um "autômato determinístico" é formalmente definido como uma sextupla:

$$G = (X, E, f, \Gamma, x_0, X_m), \tag{2.6}$$

onde X é o conjunto de estados, E é o conjunto finito de eventos (alfabeto), $f : X \times E \to X$ é a função de transição, indicando como o sistema responde a cada evento em um determinado estado, $\Gamma : X \to 2^E$ é a função de eventos ativos, representando os eventos válidos em cada estado, x_0 é o estado inicial do autômato, e $X_m \subseteq X$ é o conjunto de estados marcados (ou finais).

O autômato inicia em seu estado inicial x_0 e responde aos eventos $e \in \Gamma(x_0)$, realizando transições conforme definido por f. Essa sequência de transições continua enquanto novos eventos ocorrem e f está definida para esses eventos.

O conjunto de caminhos que partem do estado inicial e terminam em um estado marcado define as linguagens associadas ao autômato. Essas linguagens são chamadas de linguagem gerada e linguagem marcada do autômato.

- Linguagem gerada: inclui todas as sequências de eventos que correspondem a caminhos possíveis no diagrama de transição de estados, começando no estado inicial x_0.

$$L(G) = \{s \in E^* : f(x_0, s) \text{ está definida}\} \tag{2.7}$$

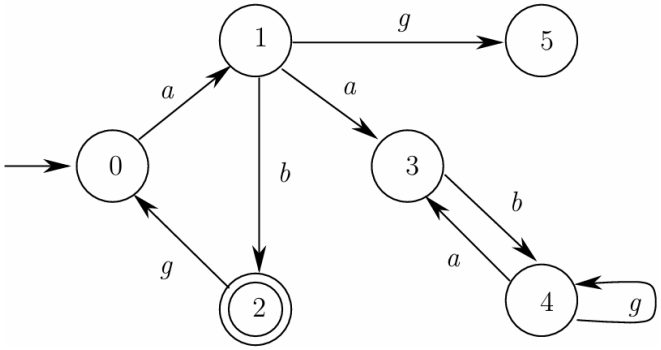
- Linguagem marcada: representa as sequências que levam o sistema a atingir estados específicos, que indicam que uma tarefa foi concluída ou um objetivo foi alcançado.

$$L_m(G) = \{s \in L(G) : f(x_0, s) \in X_m\} \tag{2.8}$$

Um autômato é considerado bloqueante se $L_m(G)$ é um subconjunto próprio de $L(G)$, ou seja, $L_m(G) \subset L(G)$. Esse bloqueio pode ocorrer por meio de:

- *Deadlock* (impasse): o sistema entra em um estado onde nenhum evento adicional pode ocorrer e não há estados marcados para indicar a conclusão de uma tarefa.
- *Livelock*: o sistema entra em um conjunto de estados de onde não consegue sair e onde nenhum estado é marcado, criando um ciclo interminável de eventos sem completar o objetivo.

Figura 8 – Exemplo de um Autômato Bloqueante.



Fonte: (CASSANDRAS; LAFORTUNE, 2008).

Um exemplo de autômato bloqueante é mostrado na Figura 8. Nela, o estado 5 representa um estado de *deadlock*, enquanto os estados 3 e 4 estão envolvidos em um *livelock*.

2.3 Controle Supervisório

Na área de controle de SED, uma técnica importante é o uso de supervisores para modificar o comportamento de um sistema em tempo real, permitindo que ele atenda a

um conjunto específico de requisitos operacionais. Este controle é realizado por meio de um "supervisor" que monitora e interage com o sistema, restringindo seu comportamento para um subconjunto desejado de estados e eventos possíveis. Isso é útil para garantir condições de segurança e evitar bloqueios que podem comprometer o desempenho de sistemas automatizados.

Para modificar o comportamento de um autômato G, introduz-se um supervisor S que interage em um laço de *feedback*, permitindo ou bloqueando eventos controláveis. Assim, o conjunto de eventos E é particionado em:

- E_c: eventos controláveis, que podem ser desabilitados por S.
- E_{uc}: eventos não controláveis, que não podem ser impedidos.

Essa estrutura possibilita que o supervisor atue dinamicamente, desativando certos eventos conforme o comportamento observado do sistema. O papel do supervisor S é manter o sistema em uma "linguagem segura" que representa comportamentos admissíveis.

2.4 Cinemática de Braço Robótico

Todo o conteúdo abordado nesta seção é baseado no capítulo 7 de (CORKE, 2023), que trata da cinemática de manipuladores robóticos.

Braços robóticos, ou manipuladores, consistem em uma série de segmentos rígidos interligados por articulações que permitem ao robô executar movimentos controlados. A base do braço costuma estar fixa, enquanto cada articulação contribui para o posicionamento e orientação do "efetuador final" (*end effector*), que é a ferramenta localizada na extremidade do braço.

Como pode ser visto na Figura 9, os manipuladores variam em complexidade, desde braços simples com duas juntas até robôs industriais com seis ou mais graus de liberdade (*DoF - Degrees of Freedom*). A cinemática de um braço robótico estuda o movimento do sistema de elos e juntas, sem considerar forças ou massas, para determinar a posição e orientação do efetuador final com base nos ângulos ou posições das juntas. A "cinemática direta" calcula a posição do efetuador final a partir das configurações das juntas, enquanto a "cinemática inversa" busca as configurações de junta necessárias para que o efetuador final alcance uma posição e orientação específicas.

Essas operações tornam-se possíveis devido a uma modelagem matemática, que utiliza parâmetros como os de Denavit-Hartenberg (DH) para descrever as relações geométricas entre os elos e as juntas. Tais parâmetros simplificam o cálculo de posições e

Figura 9 – Exemplos de diferentes tipos de braços robóticos: (a) manipulador industrial com 6 *DoF*, (b) robô SCARA com 2 *DoF*, (c) robô de pórtico que se move ao longo de um trilho suspenso, e (d) um manipulador paralelo.

Fonte: (CORKE, 2023).

permitem o desenvolvimento de soluções tanto analíticas quanto numéricas para a movimentação precisa dos braços robóticos.

2.4.1 Cinemática Direta

A cinemática direta busca determinar a posição e orientação do efetuador final do robô com base nas variáveis das articulações, como ângulos de rotação ou deslocamentos lineares.

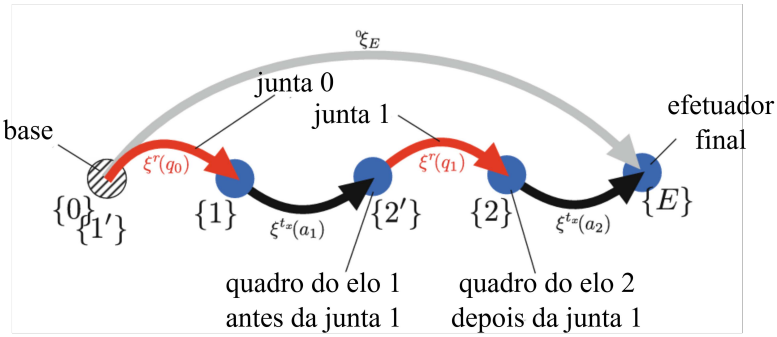
A estrutura física de um robô é composta por elos conectados por juntas, e esses elos definem o formato e a aparência do robô, além de influenciar suas propriedades de massa, inércia e potencial de colisão. Cada elo ℓ é identificado por um quadro de referência $\{\ell\}$, e a pose de cada elo depende das coordenadas das articulações $\{q_j, j \in [0, \ell]\}$, conforme ilustrado na Figura 10.

Considerando que o elo ℓ é o "pai" do elo $\ell + 1$ e o "filho" do elo $\ell - 1$, a base fixa do robô é o elo 0, que não possui pai, enquanto o efetuador final é o último elo, sem filhos. A pose do efetuador final ${}^0\xi_N$, representando a cinemática direta, é dada pela composição das transformações de cada elo e junta:

$$ {}^0\xi_N = {}^0\ \xi_1(q_0) \oplus^1 \xi_2(q_1) \oplus \cdots \oplus^{N-1} \xi_N(q_{N-1}) \tag{2.9} $$

na qual, ${}^j\xi_{j+1}(q_j)$ representa a pose (posição e orientação) do quadro do elo $j+1$ em relação ao quadro do elo j, como função da variável de junta q_j

Figura 10 – Relação entre elos e articulações em uma cadeia cinemática de manipulador robótico serial.



Fonte: Adaptado de (CORKE, 2023).

Essa expressão pode ser simplificada em uma forma funcional, onde $\mathcal{K}(.)$ representa a cinemática direta específica do robô, incorporando os parâmetros de juntas e elos:

$$ {}^0\xi_N = \mathcal{K}(\mathbf{q}) \tag{2.10} $$

2.4.2 Denavit-Hartenberg (DH)

A notação DH, desenvolvida nos anos 1950 por Jacques Denavit e Richard Hartenberg, é uma maneira simplificada e padronizada para representar a relação geométrica entre os elos e juntas de um robô. Cada elo e junta do manipulador são representados em uma tabela de parâmetros, onde cada linha descreve a configuração entre dois elos consecutivos. Essa notação é compacta e prática, pois, ao invés de utilizar seis parâmetros para definir a relação entre dois elos (três para rotação e três para translação), ela utiliza apenas quatro parâmetros, aproveitando duas restrições geométricas específicas.

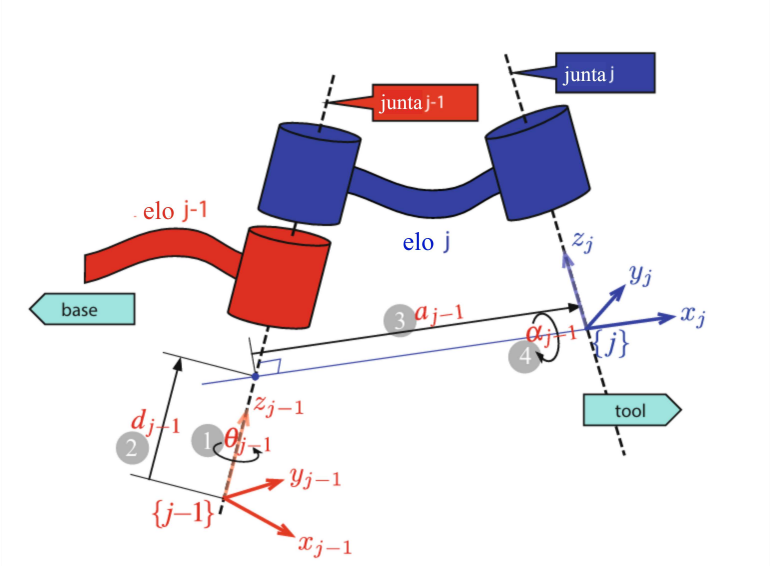
Quatro parâmetros descrevem cada elo do robô, definindo a transformação entre quadros de coordenadas consecutivos, como mostra a Figura 11.

Esses parâmetros são organizados em uma tabela (Tabela 1) para descrever o manipulador. A notação DH reduz a quantidade de dados necessária para representar as transformações entre os elos, ao aplicar duas restrições: o eixo x_j deve intersectar o eixo z_{j-1}, e o eixo x_j deve ser perpendicular ao eixo z_{j-1}. Essas restrições simplificam o cálculo da cinemática direta.

A transformação entre dois elos consecutivos j e $j+1$ pode ser representada pela sequência de transformações elementares, onde cada termo representa uma rotação ou translação ao longo de um dos eixos:

$$ {}^j\xi_{j+1} = \xi^{rz}(\theta_j) \oplus \xi^{tz}(d_j) \oplus \xi^{tx}(a_j) \oplus \xi^{rx}(\alpha_j) \tag{2.11} $$

Figura 11 – Ilustração dos parâmetros DH entre elos consecutivos.



Fonte: Adaptado de (CORKE, 2023).

Tabela 1 – Parâmetros de Denavit-Hartenberg para um elo robótico

Parâmetro	**Símbolo**	**Descrição**
Ângulo de Junta	θ_j	Ângulo entre x_j e x_{j+1} ao redor de z_{j-1}
Deslocamento de Junta	d_j	Distância ao longo de z_j entre j e x_{j+1}
Comprimento do Elo	a_j	Distância ao longo de x_{j+1} entre z_j e z_{j+1}
Torção do Elo	α_j	Ângulo entre z_j e z_{j+1} ao redor de x_{j+1}

Fonte: Adaptado de (CORKE, 2023).

Expandindo a expressão em forma de matriz homogênea, obtemos a transformação entre os quadros de coordenadas j e $j+1$ como uma matriz:

$$
{}^{j}A_{j+1} = \begin{pmatrix} \cos\theta_j & -\sin\theta_j\cos\alpha_j & \sin\theta_j\sin\alpha_j & a_j\cos\theta_j \\ \sin\theta_j & \cos\theta_j\cos\alpha_j & -\cos\theta_j\sin\alpha_j & a_j\sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.12}
$$

A matriz de transformação homogênea 2.12 é composta por duas partes principais: uma submatriz de rotação $\mathbf{R}_j^{j+1}$, que representa a orientação do elo $j+1$ em relação ao elo j, e um vetor de translação $\mathbf{t}_j^{j+1}$, que representa a posição do elo $j+1$ em relação ao elo j.

A decomposição da matriz homogênea é dada por:

$$^{j}A_{j+1} = \begin{pmatrix} \mathbf{R}_j^{j+1} & \mathbf{t}_j^{j+1} \\ \mathbf{0}_{1\times 3} & 1 \end{pmatrix} \tag{2.13}$$

na qual,

$$\mathbf{R}_j^{j+1} = \begin{pmatrix} \cos\theta_j & -\sin\theta_j \cos\alpha_j & \sin\theta_j \sin\alpha_j \\ \sin\theta_j & \cos\theta_j \cos\alpha_j & -\cos\theta_j \sin\alpha_j \\ 0 & \sin\alpha_j & \cos\alpha_j \end{pmatrix} \tag{2.14}$$

define a orientação do novo sistema de coordenadas $(x_{j+1}, y_{j+1}, z_{j+1})$ em relação ao sistema (x_j, y_j, z_j). E,

$$\mathbf{t}_j^{j+1} = \begin{pmatrix} a_j \cos\theta_j \\ a_j \sin\theta_j \\ d_j \end{pmatrix} \tag{2.15}$$

define a posição do novo sistema de coordenadas $(x_{j+1}, y_{j+1}, z_{j+1})$ em relação ao sistema (x_j, y_j, z_j). Com isso, essa notação permite calcular a pose do efetuador final em função das configurações das juntas do robô.

2.4.3 Cinemática Inversa

Diferente da cinemática direta, que determina a posição do final a partir dos ângulos das juntas, a cinemática inversa resolve o problema oposto: dados a posição e a orientação desejadas do efetuador final, ela calcula os ângulos das juntas necessários para atingir essa pose.

Matematicamente, o problema da cinemática inversa pode ser representado pela função inversa da cinemática direta, conforme a Equação 2.16:

$$q = \mathcal{K}^{-1}(\xi_E) \tag{2.16}$$

de modo que, q é o vetor de configurações das juntas, $\mathcal{K}^{-1}$ representa a função de cinemática inversa, e ξ_E é a pose desejada do efetuador final, incluindo posição e orientação.

De modo geral, essa solução não é única. Para uma dada posição e orientação do final, pode haver diversas configurações de juntas que chegam a essa mesma pose. Essa característica exige que se escolha cuidadosamente entre as soluções possíveis, levando em conta fatores como eficiência, segurança e restrições mecânicas do robô.

Entre as soluções comuns para a cinemática inversa de manipuladores robóticos, temos:

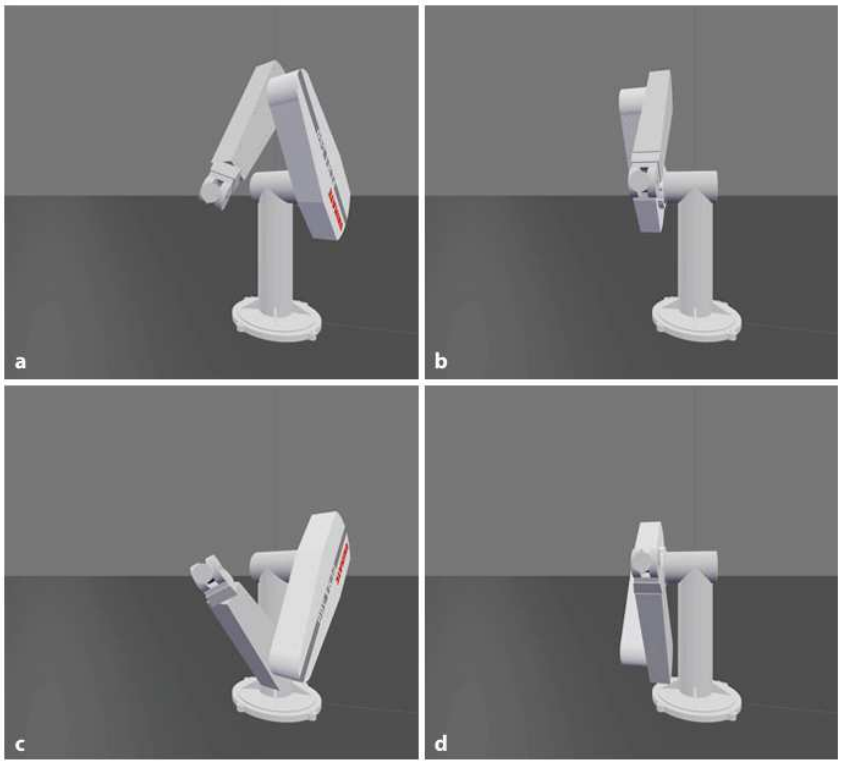
- Ombro para a direita ou para a esquerda: a configuração do ombro pode ser ajustada para que o final atinja a mesma posição com o braço posicionado para o lado direito

ou esquerdo em relação à base do robô. Essas configurações são referidas como soluções *right-handed* (ombro à direita) e *left-handed* (ombro à esquerda).

- Cotovelo para cima ou para baixo: a posição do cotovelo também oferece opções, permitindo que o braço atinja a posição desejada com o cotovelo apontando para cima (*elbow-up*) ou para baixo (*elbow-down*).

A Figura 12 ilustra as diferentes soluções encontradas para o manipulador PUMA 560.

Figura 12 – a. ombro para a direita e cotovelo para cima, b. ombro para a esquerda e cotovelo para cima, c. ombro para a direita e cotovelo para baixo, d. ombro para a esquerda e cotovelo para cima



Fonte: (CORKE, 2023).

Existem duas abordagens principais para resolver o problema da cinemática inversa:

- Solução analítica: para alguns tipos de manipuladores, especialmente aqueles com 6 *DoF* e uma estrutura de punho esférico, é possível encontrar uma solução analítica para a cinemática inversa. Esse método envolve o uso de técnicas geométricas e algébricas para obter uma expressão fechada para cada ângulo de junta em função da pose desejada do efetuador final.

- Solução numérica: para robôs com configurações mais complexas, onde não é possível obter uma solução analítica, utiliza-se uma abordagem numérica. Essa abordagem é iterativa e busca aproximar as configurações de juntas que levam o efetuador final à pose desejada. Um exemplo comum de método numérico é o algoritmo Levenberg-Marquardt, que utiliza minimização de erros para encontrar uma solução para a cinemática inversa.

2.5 Robôs Móveis

Robôs móveis são dispositivos projetados para navegar em ambientes variados, desempenhando uma ampla gama de tarefas. A mobilidade desses robôs depende de sua configuração cinemática, que determina suas capacidades e limitações de movimento. Esta sessão apresenta dois tipos de robôs móveis (Tipo Carro e Tipo Uniciclo), baseando-se nos conceitos explorados por (CORKE, 2023).

O modelo de robô tipo carro é inspirado em veículos automotores convencionais, onde a direção é controlada pelo ângulo das rodas dianteiras, enquanto as rodas traseiras fornecem a propulsão. Esse modelo é descrito por uma cinemática que impõe restrições de movimento conhecidas como restrições não-holonômicas, as quais limitam o robô a se mover apenas para frente ou para trás, sem a capacidade de realizar movimentos laterais.

As equações de movimento para um robô tipo carro são descritas por:

$$\dot{x} = v \cos\theta \tag{2.17}$$

$$\dot{y} = v \sin\theta \tag{2.18}$$

$$\dot{\theta} = \frac{v}{L} \tan\gamma \tag{2.19}$$

onde v é a velocidade linear do robô, θ é o ângulo de orientação, L é a distância entre os eixos das rodas e γ é o ângulo de direção. O modelo de robô tipo carro é representado graficamente na Figura 13, onde o Centro Instantâneo de Rotação (ICR) e (γ) definem o movimento do robô.

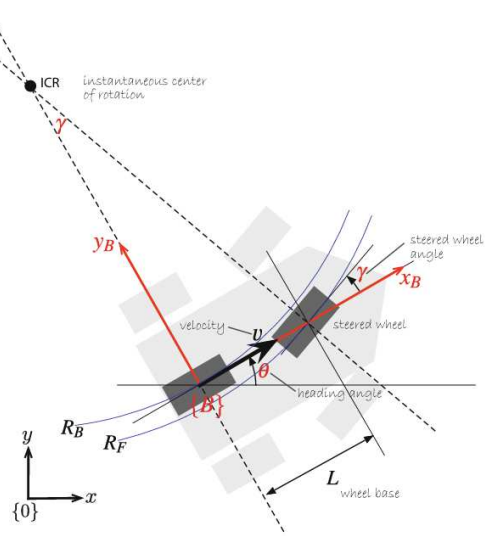
O modelo de robô tipo uniciclo caracteriza-se por uma configuração onde o robô pode controlar independentemente a velocidade linear v e a velocidade angular ω. Isso permite uma maior flexibilidade de movimento, pois o robô pode girar em torno de seu próprio eixo e alterar sua direção com mais facilidade em relação ao modelo tipo carro.

As equações de movimento para o modelo uniciclo são dadas por:

$$\dot{x} = v \cos\theta \tag{2.20}$$

$$\dot{y} = v \sin\theta \tag{2.21}$$

Figura 13 – Modelo de robô do tipo carro, mostrando o *ICR*, velocidade (*velocity*), distância entre eixos (*wheel base*), ângulo de orientação (*heading angle*), e ângulo de direção (*steered wheel angle*).



Fonte: (CORKE, 2023).

$$\dot{\theta} = \omega \tag{2.22}$$

onde v é a velocidade linear e ω é a velocidade angular. A cinemática diferencial implica que as velocidades das rodas direita (v_R) e esquerda (v_L) podem ser calculadas como:

$$v_R = v + \frac{W}{2}\dot{\theta} \tag{2.23}$$

$$v_L = v - \frac{W}{2}\dot{\theta} \tag{2.24}$$

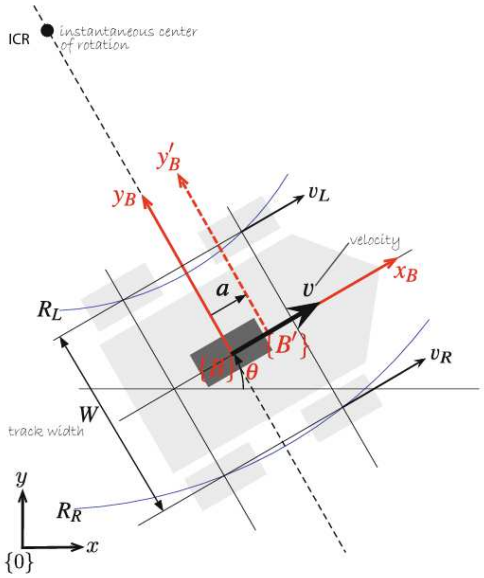
onde W é a distância entre as rodas. Este modelo também está sujeito a restrições não-holonômicas. A Figura 14 representa o modelo de robô tipo uniciclo, também baseado no conceito de *ICR*.

2.6 *Robot Operating System* (ROS)

O *Robot Operating System*[1] (ROS), no português Sistema Operacional de Robôs, é uma coleção de bibliotecas de *software* e ferramentas projetadas para ajudar no desenvolvimento de aplicações robóticas. Composto por *drivers*, algoritmos e instrumentos avançados para desenvolvedores, este *framework* é fundamental para a robótica moderna.

[1] Documentação do ROS: <https://docs.ros.org/en/humble/index.html>

Figura 14 – Modelo de robô do tipo uniciclo, mostrando o ICR, velocidade (*velocity*), distância entre as rodas (*track width*), ângulo de orientação (*heading angle*), e velocidades (v_R e v_L).



Fonte: (CORKE, 2023).

O ROS opera como um metasistema operacional de código aberto, facilitando serviços típicos de um sistema operacional, incluindo abstração de *hardware*, controle de dispositivos de baixo nível, passagem de mensagens entre processos e o gerenciamento de pacotes. Além disso, ele também fornece bibliotecas e ferramentas para adquirir, construir, escrever e executar códigos em diversos computadores.

Embora não seja um sistema operacional real, ele funciona como uma estrutura que organiza e coordena diferentes computadores ou dispositivos com capacidades variadas, ajudando-os a trabalhar juntos. Ele não é usado apenas para controlar robôs, mas também para integrar outros equipamentos e ferramentas, permitindo que esses dispositivos troquem informações e funcionem em sincronia.

Sua principal característica é como o *software* é executado e a maneira como se comunica com o *hardware* sem a necessidade de entendê-lo profundamente. Isso é alcançado por meio da conexão de uma rede de processos, conhecidos como “nós”, ou *nodes* no inglês.

Para criar nós, existem duas abordagens principais: oferecer serviços que podem ser solicitados ou estabelecer conexões de *publicador/assinante* com outros nós. Esses métodos utilizam tipos específicos de mensagens, que podem ser fornecidos pelos pacotes principais ou definidos por pacotes personalizados. Por exemplo, o *driver* de um sensor pode ser implementado como um nó que publica dados em um fluxo de mensagens, que então pode ser consumido por um ou mais nós interessados.

Um pacote é a unidade modular básica dentro do sistema ROS, criada para organizar e fornecer funcionalidades específicas. Ele agrupa nós, arquivos de configuração (*launch*), mensagens, bibliotecas e *scripts* necessários para realizar tarefas, como controle de *hardware*, processamento de dados, simulação e visualização. Seu principal objetivo é facilitar o desenvolvimento, compartilhamento e reutilização de componentes robóticos, permitindo que desenvolvedores combinem diferentes pacotes para criar sistemas de forma modular e eficiente, promovendo a interoperabilidade e a colaboração no ecossistema de robótica.

O arquivo de inicialização no ROS é usado para inicializar e configurar múltiplos nós e parâmetros em uma única execução. Esses arquivos permitem que o usuário especifique quais nós devem ser executados, além de parâmetros, configuração, tópicos e serviços que devem ser definidos para cada um. Com um arquivo *launch*, é possível iniciar uma série de processos e configurações automaticamente, assim, seu principal objetivo é simplificar o gerenciamento e a coordenação de vários nós ao mesmo tempo, garantindo que todos os componentes necessários estejam configurados e sincronizados.

Além disso, o ROS permite a criação de nós em diversas linguagens de programação e possibilita que esses nós se comuniquem de forma eficiente, independentemente da linguagem utilizada. Embora Python e C++ sejam as linguagens mais comuns, outras também são suportadas, permitindo integrações entre diferentes máquinas e linguagens de programação no mesmo sistema.

Os nós no ROS geralmente se comunicam por meio de três tipos principais de interfaces: tópicos, serviços e ações. As próximas seções exploram em mais detalhes o funcionamento de cada uma delas.

2.6.1 *ROS Nodes* (Nós)

No ROS, um nó é um processo computacional que realiza uma tarefa específica dentro de uma aplicação. Cada nó é implementado como um programa executável, localizado dentro de um pacote, e é projetado para funcionar de forma independente, comunicando-se com outros nós para realizar tarefas colaborativas. Eles são organizados em um gráfico e se comunicam entre si por meio de "tópicos", "serviços" ou "ações", a depender da necessidade da interação.

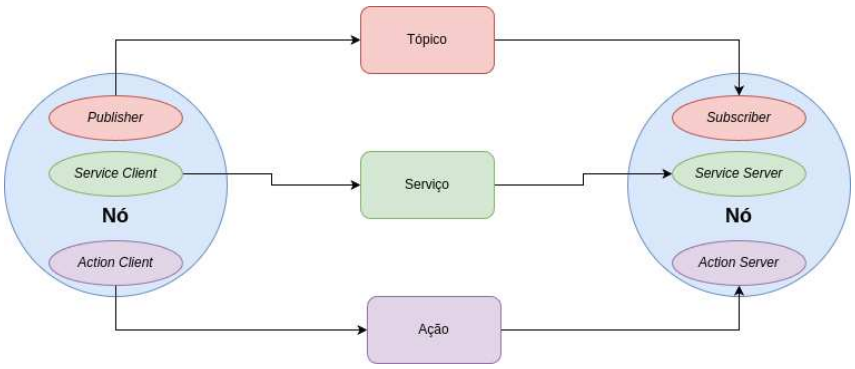
Os nós podem publicar dados em tópicos específicos, tornando-os disponíveis para outros nós, ou se inscrever em tópicos para receber informações publicadas por outros nós. Além disso, os nós podem funcionar como clientes ou servidores de serviços, permitindo que eles solicitem e realizem operações sob demanda. Para tarefas mais longas, os nós podem atuar como clientes ou servidores de "ações", o que permite o andamento de operações prolongadas com atualizações contínuas de progresso (*feedback*). Cada nó é

construído por diferentes componentes, que podem incluir:

- *Publisher* (Publicador): responsável por enviar dados para tópicos específicos.
- *Subscriber* (Assinante): recebe dados de tópicos específicos.
- *Service Client* (Cliente de Serviço): solicita serviços a outros nós.
- *Service Server* (Servidor de Serviço): oferece serviços a outros nós.
- *Action Client* (Cliente de Ação): inicia operações de longa duração.
- *Action Server* (Servidor de Ação): executa operações de longa duração.

A comunicação entre os nós é estabelecida por meio de um processo de descoberta distribuída. Quando um nó é iniciado, ele anuncia sua presença na rede ROS, permitindo que outros nós reconheçam sua existência e estabeleçam conexões para futura comunicação. Este processo é contínuo, garantindo que novas entidades sejam descobertas e integradas ao sistema. A Figura 15, demonstra, por meio de um diagrama, o funcionamento dos nós.

Figura 15 – Representação do funcionamento dos nós no ROS.



Fonte: Autoria própria.

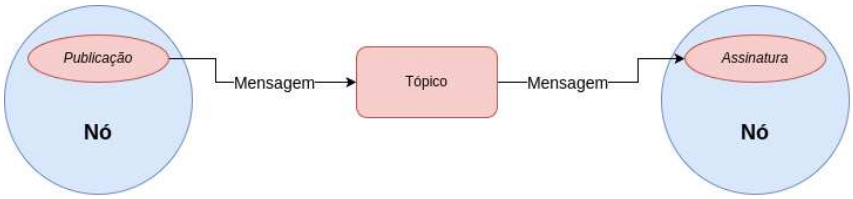
2.6.2 *ROS Topics* (Tópicos)

Os tópicos funcionam como um barramento onde os nós trocam mensagens. Por exemplo, um nó que monitora a temperatura pode publicar essas informações em um tópico. Outros nós podem assinar este tópico para receber atualizações sempre que novos dados forem publicados. Eles permitem que qualquer nó publique e se inscreva em múltiplos tópicos. Assim, eles são ideais para o fluxo contínuo de dados, como leituras de sensores ou estados dinâmicos de robôs.

Para gerenciar tópicos no ROS, são utilizadas várias ferramentas como `ros topic list`, `ros topic info`, e `ros topic pub`, que ajudam os desenvolvedores a listar, obter

informações e publicar em tópicos, respectivamente. A Figura 16, demonstra, por meio de um diagrama, o funcionamento dos tópicos.

Figura 16 – Representação do funcionamento dos tópicos no *ROS*.



Fonte: Autoria própria.

2.6.3 *ROS Services* (Serviços)

Os serviços são um método de comunicação entre nós, utilizando um modelo de chamada (*request*) e resposta (*response*). Diferentemente dos tópicos, que permitem um fluxo contínuo de dados, os serviços respondem a solicitações específicas enviadas por clientes que aguardam uma resposta imediata.

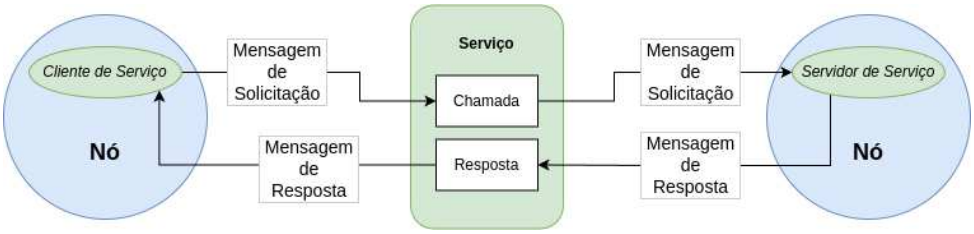
Cada serviço é caracterizado por um tipo específico que define a estrutura das mensagens de solicitação e de resposta. Essencialmente, um tipo de serviço é composto por duas partes principais: uma definição de solicitação e uma definição de resposta. Essa característica é utilizada para operações que requerem uma interação síncrona entre nós, onde um nó cliente envia uma solicitação e espera que o nó servidor processe essa solicitação e retorne um resultado.

Os serviços são identificados por nomes únicos, que são distintos dos nomes de tópicos. A comunicação por meio de serviços é bidirecional e estritamente definida, proporcionando um controle mais direto e imediato sobre as operações realizadas. Eles são compostos por dois elementos principais:

- Servidor de Serviço: é a entidade que processa a solicitação recebida, aceita solicitações para execução de operações e retorna um resultado ao cliente.
- Cliente de Serviço: faz a solicitação de um serviço específico. Pode haver múltiplos clientes para um único servidor de serviço, cada um solicitando a execução de uma operação e aguardando o resultado correspondente.

A Figura 17, demonstra, por meio de um diagrama, o funcionamento dos serviços no ROS.

Figura 17 – Representação do funcionamento dos serviços no *ROS*.



Fonte: Autoria própria.

2.6.4 *ROS Actions* (Ações)

As ações representam um tipo de comunicação destinada a operações de longa duração que requerem interação contínua - *feedback* - durante a execução e a capacidade de cancelamento. Uma ação é composta por três componentes principais: uma meta, um fluxo de realimentação e um resultado final. As ações combinam características tanto dos tópicos quanto dos serviços. Elas são baseadas em um modelo de cliente-servidor, com a capacidade adicional de cancelamento e de fornecimento contínuo de *feedback*, o que as diferencia dos serviços.

Essa estrutura é particularmente útil em aplicações como a navegação robótica, onde uma ação pode ser configurada para mover um robô a uma posição específica. Durante este processo, o robô, através do nó servidor de ação, pode enviar atualizações regulares sobre seu progresso, como distância percorrida ou obstáculos encontrados, antes de finalmente enviar uma resposta quando o destino for alcançado. Este método assegura que as operações sejam monitoradas e ajustadas conforme necessário.

A estrutura de uma ação inclui dois componentes:

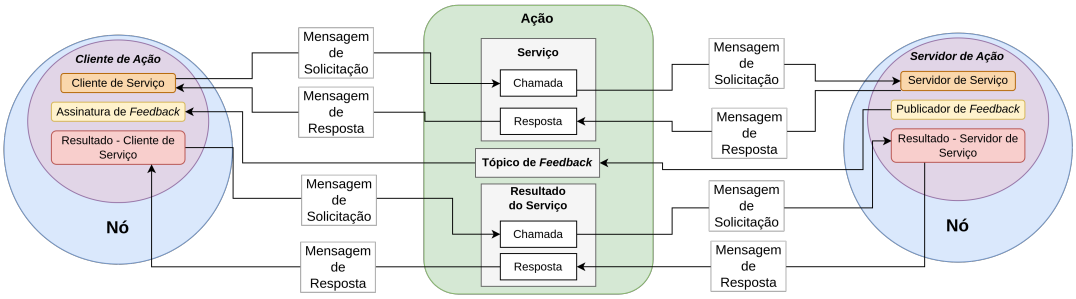
- Servidor de Ação: este é responsável por receber a solicitação de ação, processá-la e fornecer *feedback* regular ao cliente. Além disso, gerencia solicitações de cancelamento ou alterações durante a execução da tarefa.
- Cliente de Ação: o cliente solicita a execução de uma ação e pode continuar a receber *feedback* até a conclusão da tarefa. Os clientes podem iniciar, modificar ou cancelar ações conforme necessário, proporcionando flexibilidade na execução de tarefas complexas.

A Figura 18, ilustra o funcionamento das ações no ROS.

2.7 Gazebo Clássico

O Gazebo é um simulador robótico 2D/3D de código aberto, desenvolvido inicialmente em 2002 na Universidade do Sul da Califórnia por Dr. Andrew Howard e seu aluno

Figura 18 – Representação do funcionamento das ações no ROS.



Fonte: Autoria própria.

Nate Koenig. Ele foi criado para atender à necessidade de testar robôs em ambientes externos sob diversas condições atmosféricas, com um alto grau de fidelidade (ROBOTICS, 2024b).

O simulador oferece ferramentas de desenvolvimento e serviços na nuvem, além de permitir a modelagem de sensores que "observam" o ambiente simulado, como medidores de distância a laser e câmeras. Essas funcionalidades facilitam a simulação de projetos físicos em ambientes realistas, possibilitando testes seguros de estratégias de controle (ROBOTICS, 2024a).

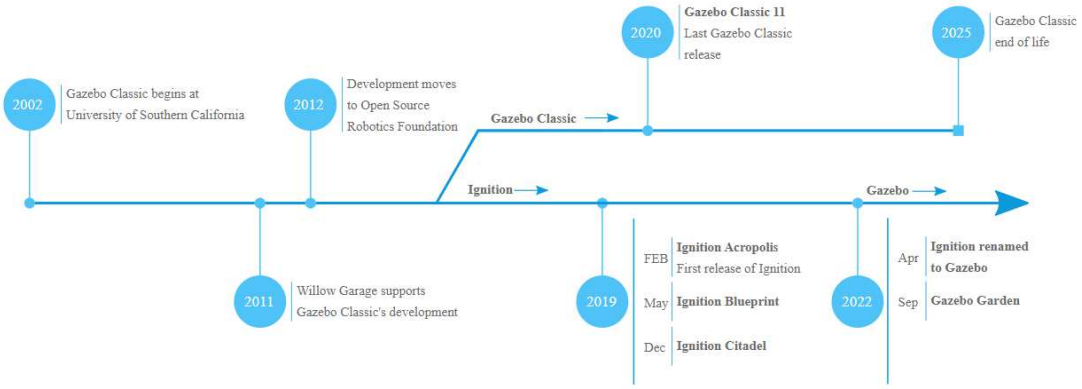
O Gazebo encapsula uma ampla gama de bibliotecas de código aberto, com recursos como tipos de dados matemáticos, registro de atividades, gerenciamento de malhas 3D e passagem de mensagens assíncronas. O *software* também oferece suporte a uma variedade de funcionalidades:

- Simulação: simulador de robôs para pesquisa, *design* e desenvolvimento.
- *Plugins*: possibilidade de desenvolver *plugins* personalizados para controle de robôs e ambientes simulados.
- Transporte: passagem de mensagens assíncrona de forma distribuida.
- Sensores: conjunto de modelos de sensores e ruídos.
- Física: interface baseada em *plugins* para motores de física.
- Renderização: interface baseada em *plugins* para motores de renderização.
- Modelos de Robôs: disponibilidade de diversos modelos de robôs e suporte para construção de modelos customizados usando *Spatial Data File* (SDF).

Em 2017, o desenvolvimento do Gazebo se dividiu em duas vertentes: o "*Gazebo Classic*", que manteve a arquitetura monolítica original, e o "*Ignition*", que representa

uma evolução para uma coleção de bibliotecas pouco acopladas, adaptando-se melhor às novas exigências da robótica (WIKIPEDIA, 2024).

Figura 19 – Linha do tempo da história do gazebo.

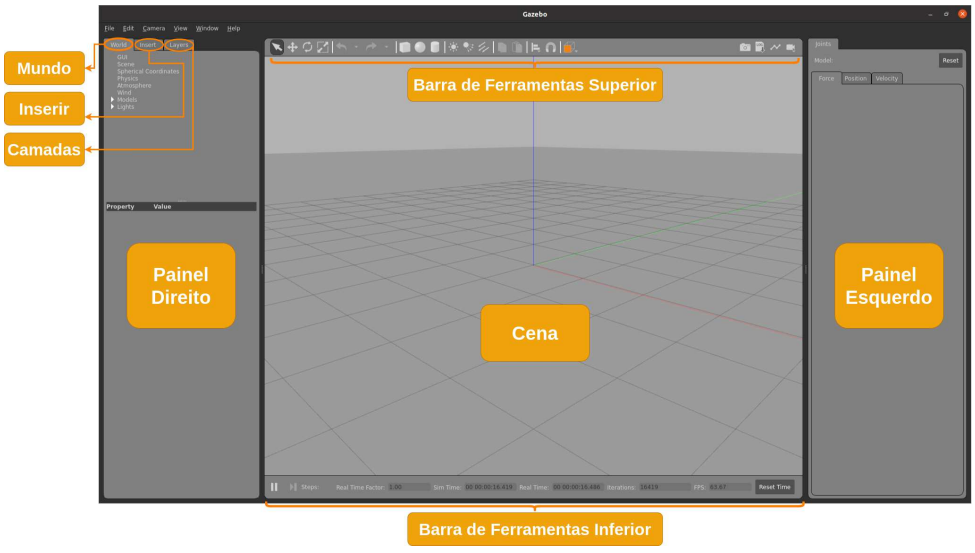


Fonte: (ROBOTICS, 2024a).

O Gazebo se estabeleceu como uma ferramenta essencial para roboticistas, proporcionando um ambiente de simulação que permite a realização de testes de regressão, *design* de robôs e treinamento de sistemas em cenários realistas. Sua natureza de código aberto e a comunidade ativa contribuem significativamente para seu contínuo desenvolvimento e aprimoramento.

De acordo com (ROBOTICS, 2014), a Interface Gráfica do Usuário (GUI) do Gazebo, possui vários componentes, que podem ser vistos na Figura 20. Essas ferramentas permitem uma manipulação detalhada e interativa dos modelos e ambientes dentro do simulador Gazebo.

Figura 20 – GUI do Gazebo.



Fonte: Autoria própria.

2.8 *ROS Visualization* (*Rviz*)

O *RViz* é uma ferramenta de visualização 3D para a representação de robôs, sensores e algoritmos. Sua principal função é permitir a visualização da percepção do estado do robô em seu mundo, tanto real quanto simulado, fornecendo assim uma representação do ambiente operacional do robô (SEARS-COLLINS, 2020).

Ele se trata de uma plataforma integrada ao ROS que permite tanto a visualização de dados externos quanto o envio de comandos de controle a objetos. Em seu ambiente, o usuário pode interagir com a visualização 3D através de um conjunto diversificado de ferramentas e opções. Sua *GUI* possui vários componentes:

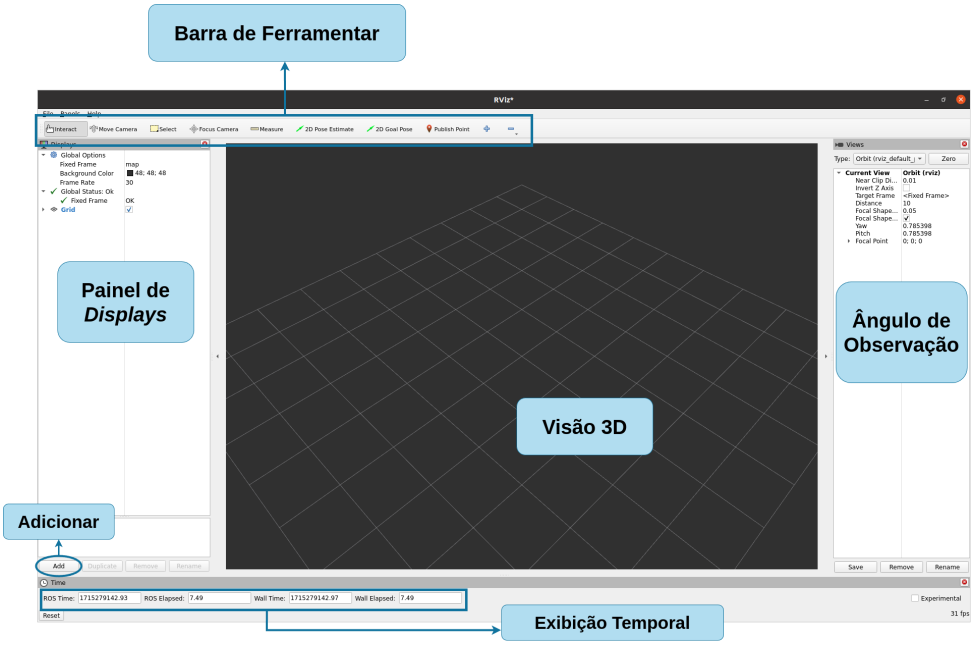
- Painel de *displays*: inclui uma lista de *plugins* para visualizar dados de sensores e informações de estado do robô. *Plugins* adicionais podem ser incorporados usando o botão "Adicionar".
- Barra de ferramentas: oferece acesso a ferramentas multifuncionais que facilitam a manipulação dos dados visualizados.
- Visão 3D: onde os dados são exibidos em três dimensões. Configurações como cor de fundo, quadro fixo e parâmetros da grade são ajustáveis nas opções globais.
- Área de exibição temporal: mostra informações temporais relevantes, incluindo o tempo do sistema e do ROS.
- Configuração do ângulo de observação: permite ao usuário ajustar o ângulo de visualização para otimizar a perspectiva observada.

2.9 Árvore de Comportamento (*Behavior Trees*)

As árvores de comportamento são uma arquitetura de controle amplamente adotada em robótica, principalmente em ambientes dinâmicos e de alta variabilidade. Inicialmente desenvolvidas para a indústria de jogos com o objetivo de controlar personagens virtuais, elas foram incorporadas à robótica por suas características de modularidade, reatividade e escalabilidade (COLLEDANCHISE; ÖGREN, 2018). Essa estrutura hierárquica permite organizar e gerenciar o comportamento de agentes, facilitando adaptações a mudanças de ambiente e respostas rápidas a eventos (ÖGREN; SPRAGUE, 2022).

Estruturalmente, uma árvore de comportamento é composta por nós organizados de forma hierárquica, representando ações ou condições específicas. Esses nós são divididos em dois grupos principais: nós de controle e nós de execução. Os nós de controle orientam o fluxo de execução, estabelecendo a ordem e as condições para transições entre estados,

Figura 21 – *GUI* do *Rviz*.



Fonte: Autoria própria.

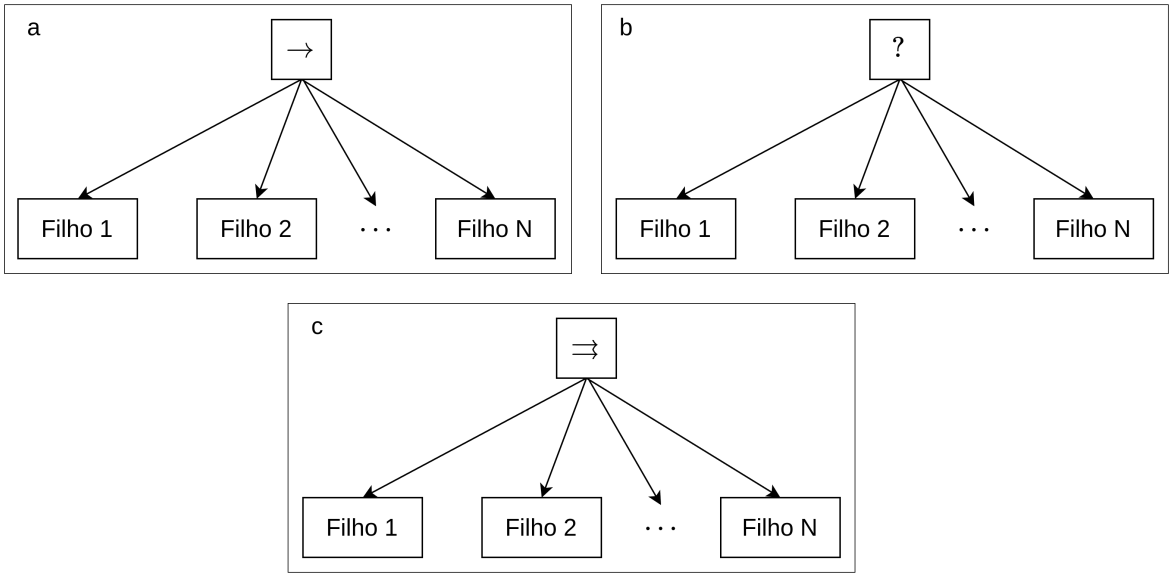
como sucesso, falha ou execução contínua. Três tipos comuns de nós de controle incluem: sequência, *fallback* e paralelo.

O nó de sequência é responsável por encaminhar "ticks" — sinais periódicos de ativação — aos seus filhos da esquerda para a direita, até encontrar um filho que retorne "Falha" (*Failure*) ou "Executando" (*Running*). Quando um filho retorna "Falha" ou "Executando", o nó de sequência cessa a propagação de "ticks" aos próximos filhos e retorna o respectivo estado para o nó pai. O nó de sequência só retorna "Sucesso" (*Success*) se todos os seus filhos retornarem sucesso. A representação simbólica do nó de sequência é um retângulo contendo a seta "$\rightarrow$", conforme ilustrado na Figura 22.

O nó de *fallback* funciona de maneira similar, mas com uma lógica inversa. Esse nó encaminha "ticks" para os filhos da esquerda para a direita, até encontrar um filho que retorne "Sucesso" ou "Executando" (*Running*). Assim, o nó de *fallback* retorna sucesso para o nó pai se algum de seus filhos obtiver sucesso. Ele só retorna falha se todos os filhos falharem. A representação simbólica do nó de *fallback* é um retângulo contendo o símbolo "?".

O nó paralelo (*Parallel*) executa "ticks" em todos os seus filhos simultaneamente. Ele retorna sucesso se pelo menos um número mínimo de filhos, definido pelo usuário como M, retornar "Sucesso". Caso o número de filhos que retornam "Falha" atinja $N - M + 1$, onde N é o número total de filhos, o nó paralelo retorna falha. Se a condição de sucesso ou falha não for atingida, o nó permanece em execução. A representação simbólica do nó paralelo é um retângulo contendo o símbolo "$\rightrightarrows$".

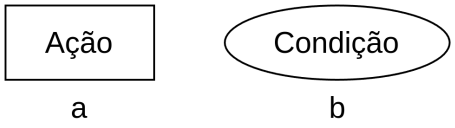
Figura 22 – Representação dos diferentes tipos de nós de controle em uma árvore de comportamento. a) Nó de sequencia, b) Nó de *fallback* e c) Nó paralelo.



Fonte: Adaptado de (COLLEDANCHISE; ÖGREN, 2018).

Os nós de execução, por sua vez, subdividem-se em nós de ação e nós de condição. Os nós de ação representam tarefas específicas, como mover-se até um ponto, capturar um objeto ou interagir com outros dispositivos. Já os nós de condição verificam estados do ambiente ou do próprio sistema, controlando a sequência de execução conforme as condições configuradas. Essa arquitetura modular possibilita a reutilização de blocos de comportamento e facilita adaptações a novos objetivos e situações (GUGLIERMO et al., 2024). O nó de ação é representado por um retângulo e o nó de condição é representado por uma elipse, conforme ilustrado na Figura 23.

Figura 23 – Representação dos diferentes tipos de nós de execução em uma árvore de comportamento. a) Nó de ação e b) Nó de condição.



Fonte: Adaptado de (COLLEDANCHISE; ÖGREN, 2018).

A modularidade e a capacidade de resposta em tempo real das árvores de comportamento oferecem vantagens para robôs que operam em ambientes dinâmicos e compartilhados, como fábricas. Nesses contextos, elas permitem que os robôs reajam a mudanças, como a presença de obstáculos inesperados ou alterações nas prioridades das tarefas (Quasi AI, 2023). A estrutura hierárquica dessa arquitetura facilita o encapsulamento de comportamentos em subárvores, promovendo clareza e simplicidade na programação e no

controle. Além disso, a execução paralela de tarefas permite coordenar ações em equipes de robôs heterogêneos, onde habilidades diferentes devem ser combinadas de maneira integrada e eficiente (HEPPNER et al., 2023).

3 Metodologia Adotada

A metodologia deste trabalho foi desenvolvida para implementar um sistema de controle orientado a eventos em uma fábrica inteligente, utilizando árvores de comportamento e uma estrutura de programação distribuída. Essa abordagem possibilita uma adaptação dinâmica do sistema a alterações no ambiente produtivo. A aplicação foi desenvolvida e validada no Laboratório *Smart Factoring*, no Laboratório de Sistemas Embarcados e Computação Pervasiva (Embedded) da Universidade Federal de Campina Grande (UFCG), um espaço experimental voltado para pesquisas em automação avançada no contexto da Indústria 4.0 (I4.0).

A metodologia foi organizada em quatro fases principais, cada uma com objetivos específicos para assegurar que a arquitetura distribuída e o controle baseado em árvores de comportamento funcionem tanto no ambiente simulado quanto no ambiente físico:

1. Modelagem e representação digital do ambiente: na primeira fase, foi criada uma representação digital da configuração física do laboratório *Smart Factoring* no simulador Gazebo. Essa etapa incluiu a definição do layout do ambiente e o posicionamento dos dispositivos, de forma a replicar o ambiente físico com o maior nível de fidelidade possível. Essa representação permitiu uma simulação dos eventos e das interações entre os dispositivos.

2. Desenvolvimento e integração do controle orientado a eventos: a segunda fase envolveu a criação de um modelo teórico de controle orientado a eventos, com base em uma estrutura distribuída e na teoria de Sistema a Eventos Discretos (SED). Esse modelo foi implementado usando árvores de comportamento (*Behavior Tree*), que definem as transições de estado e as ações dos dispositivos em resposta a eventos específicos. A arquitetura distribuída foi implementada no ROS, facilitando a comunicação e a troca de informações entre os diferentes dispositivos.

3. Configuração de programação distribuída e comunicação: a terceira fase concentrou-se na implementação da estrutura de programação distribuída. Utilizando ROS, cada dispositivo pôde operar de maneira modular, com processos independentes, mas coordenados. Essa estrutura distribuiu o controle entre os dispositivos, garantindo flexibilidade, ao mesmo tempo que assegurava uma comunicação em tempo real.

4. Validação em ambiente físico e ajustes finais: por fim, o sistema foi transferido para o ambiente físico do laboratório *Smart Factoring*, onde foram realizados testes em condições reais. Esta etapa foi importante para validar a adaptabilidade do controle

orientado a eventos, ajustando as árvores de comportamento e a configuração do ROS para garantir uma execução das tarefas em um ambiente de fábrica inteligente.

3.1 Configuração do Ambiente

Este trabalho foi desenvolvido com o objetivo de simular e validar cenários de produção em uma fábrica inteligente. Para isso, foi necessário configurar tanto o ambiente físico, com dispositivos robóticos e sensores, quanto o ambiente de *software*, garantindo a integração distribuída dos dispositivos em uma rede controlada por um sistema orientado a eventos.

Para a implementação do sistema, foram utilizadas as seguintes configurações de *software*:

- Sistema Operacional: Ubuntu 22.04, garantindo compatibilidade com as ferramentas de programação distribuída.
- ROS 2 *Humble*[1]: responsável por gerenciar a troca de mensagens, controle e coordenação das ações distribuídas entre o manipulador UR10, o TurtleBot2i e as câmeras.
- *Docker*[2] e *Dev Container*[3]: para facilitar a modularidade e garantir a portabilidade do sistema, o desenvolvimento foi conduzido em contêineres *Docker*, configurados com um ambiente *Dev Container* dentro do editor de código-fonte gratuito *Visual Studio Code* (*VS Code*). Essa abordagem assegura que as dependências e versões dos pacotes estejam alinhadas em diferentes fases de desenvolvimento e implementação.
- Linguagens e Bibliotecas: Python e C++ para programação dos nós ROS e integração com os dispositivos e `pytree` para implementação de árvores de comportamento.

Essa configuração permitiu a criação de um sistema modular, onde cada componente é gerenciado de forma independente, mas interconectada, através do ROS.

3.1.1 Dispositivos Utilizados

O ambiente físico conta com os seguintes dispositivos, cada um desempenhando uma função específica nas cenas definidas para simulação de produção de fábricas inteligentes:

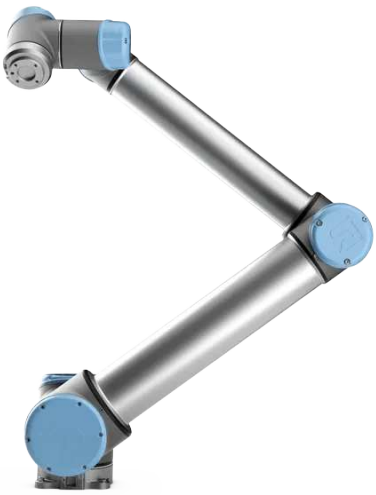
[1] Página principal do ROS *Humble*: <https://docs.ros.org/en/humble/index.html>
[2] Página principal do *Docker*: <https://www.docker.com/>
[3] Página principal do *Dev Container*: <https://code.visualstudio.com/docs/devcontainers/containers>

3.1.1.1 *Universal Robots* UR10

O UR10, é um manipulador robótico colaborativo da *Universal Robots*, projetado para aplicações industriais que demandam precisão e segurança em ambientes onde robôs e humanos aperam em conjunto. Suas juntas são nomeadas de acordo com sua posição no braço robótico, semelhante a nomenclatura de um braço humano.

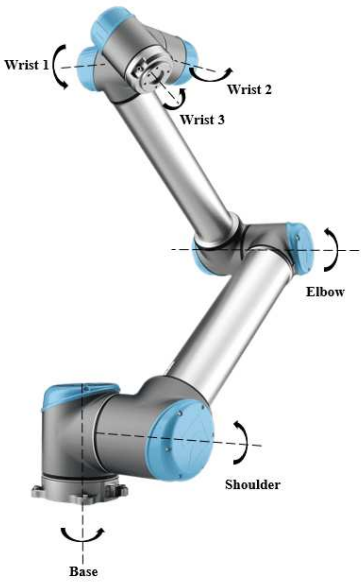
A primeira articulação, chamada base, é responsável pela rotação do robô ao longo de seu eixo principal. A segunda junta é o *shoulder* (ombro), que permite movimentos verticais. A terceira, é o *elbow* (cotovelo), possibilita movimentos adicionais para alcançar e retrair o braço. E , por fim, as três últimas juntas, *wrist 1*, *wrist 2* e *wrist 3*, correspondem às articulações do punho. Nas Figura 24 e Figura 25, é possível ver a estrutura física desse robô, bem como a disposição das juntas.

Figura 24 – Estrutura do UR10.



Fonte: (Universal Robots, 2023b)

Figura 25 – Nomenclatura das juntas do UR10.

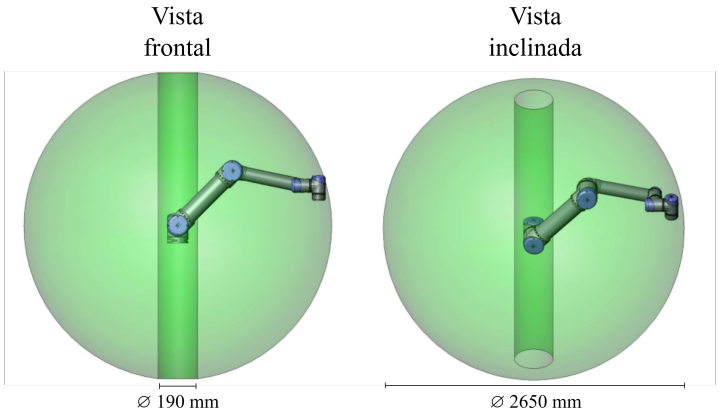


Fonte: Adaptado de (Universal Robots, 2023b)

Todas as juntas do manipulador são rotativas, podendo rotacionar 360°. Desse modo, ele oferece 6 graus de liberdade (*DoF - Degrees of Freedom*). Além disso, de acordo com (Universal Robots, 2023c), ele possui uma carga útil de até 10 kg e um alcance de 1300 mm. Segundo (Universal Robots, 2023d), ele permite a manipulação em uma área de trabalho delimitada, abrangendo um diâmetro de 2600 mm, com um limite máximo de operação de até 2650 mm, conforme pode ser visto na Figura 26.

Para integração com sistemas externos, o robô dispõe de várias portas de entrada e saída. Ele possui possui um total de 16 entradas digitais, 16 saídas digitais, duas entradas analógicas e duas saídas analógicas. A comunicação é facilitada por conexões TCP/IP de

Figura 26 – Área de trabalho do manipulador UR10.

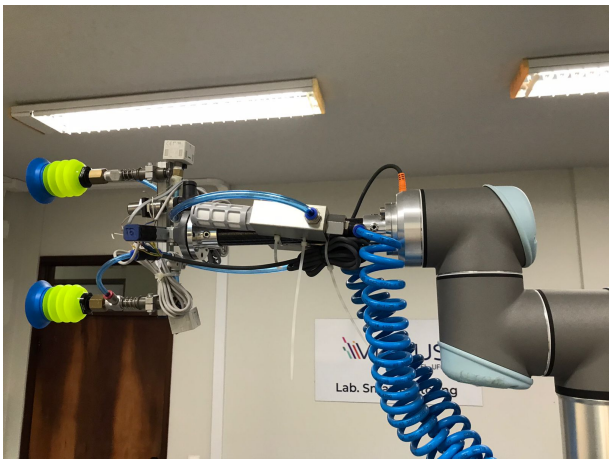


Fonte: Adaptado de (Universal Robots, 2023e).

100 Mbit, Modbus TCP e Profinet.

O efetuador final do UR10 possui uma estrutura equipada com duas ventosas acionadas por um sistema pneumático, controlado diretamente por entradas digitais do sistema de controle do robô. A Figura 27 ilustra a configuração completa do efetuador final com as ventosas.

Figura 27 – Efetuador final do UR10.



Fonte: Autoria própria.

3.1.1.1.1 Integração do UR10 com o ROS

A integração do UR10 com o ROS é realizada por meio de uma interface chamada *Universal Robots ROS2 Driver* (`Universal_Robots_ROS2_Driver`)[4], desenvolvida especificamente para permitir a comunicação e controle de manipuladores robóticos da

[4] Repositório do *driver* no GitHub: <https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver>

Universal Robots no ambiente ROS. Este *driver* oferece suporte a funcionalidades colaborativas, como pausa automática em caso de parada de emergência, parada de segurança, e ajuste automático de velocidade para evitar a violação das configurações de segurança. Adicionalmente, ele possibilita a inclusão de comportamentos ROS diretamente no programa do robô.

A integração completa do UR10 ao ROS exige três passos principais:

1. Instalação do driver: é necessário instalar o `Universal_Robots_ROS2_Driver` em uma máquina com sistema operacional Ubuntu. As instruções detalhadas para realizar essa instalação estão disponíveis no manual de instalação[5].

2. Instalação do controlador externo: no sistema operacional do robô, é necessário instalar o programa *External Control*[6], que permite estabelecer uma conexão entre o robô e o *driver*, possibilitando o controle externo do manipulador via *ROS*.

3. Calibração do robô: para que o manipulador execute os movimentos com precisão, é necessário realizar a calibração de suas juntas. Ela é feita seguindo os passos disponíveis na documentação do pacote[7].

Concluídas as instalações necessárias, é preciso realizar a configuração de rede e a execução do programa de controle externo para estabelecer a comunicação entre o robô e o *driver*. Os passos para essa configuração são descritos a seguir:

1. Primeiramente, tanto o UR10 quanto o computador onde o *driver* do ROS está instalado devem ser inseridos na mesma rede local via TCP/IP, permitindo que o robô e o *driver* possam se comunicar.

2. No UR10, é necessário criar um novo programa no qual será inserido o comando de controle externo. Durante a criação do programa, o endereço IP do computador onde o *driver* está instalado deve ser especificado, permitindo que o robô se conecte ao sistema de controle externo.

3. No computador, o próximo passo é inicializar o arquivo de lançamento (*launch*) do *driver*, que estabelecerá a interface de comunicação com o UR10. Esse arquivo deve receber dois argumentos essenciais: o tipo de robô e o endereço IP[8] do UR10.

[5] Manual de instalação do *driver*: <https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/installation/installation.html>

[6] Manual de instalação do *External Control*: <https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/installation/install_urcap_cb3.html>

[7] Calibração do UR10: <https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/installation/robot_setup.html>

[8] Configuração de Rede do UR10: <https://docs.ros.org/en/ros2_packages/humble/api/ur_robot_driver/installation/robot_setup.html>

4. No UR10, deve ser iniciado o programa que contém o controle externo.

Após a execução desses passos, a integração do manipulador UR10 com o ROS está concluída, permitindo a troca de informações com o robô.

3.1.1.2 Câmera *RBD-D Microsoft* Kinect 360

O Kinect, ilustrado na Figura 28, é um sensor que combina uma câmera RGB e um sensor de profundidade, permitindo a captura simultânea de imagens coloridas e a detecção da distância de objetos. Muito utilizado em robótica e visão computacional, o Kinect oferece um sistema acessível para aquisição de dados espaciais em tempo real, o que possibilita a análise de ambientes e a interação com objetos. Inicialmente criado para o console de jogos *Xbox*, ele ganhou popularidade em pesquisas de robótica e monitoramento ambiental devido ao seu custo-benefício e versatilidade (EL-IAITHY; HUANG; YEH, 2012).

Figura 28 – Sensor *Microsoft* Kinect 360.

Fonte: (Wikimedia Commons, 2024)

A Tabela 2 apresenta as especificações técnicas principais do Kinect, incluindo a resolução do sensor RGB, a faixa de alcance do sensor de profundidade, o campo de visão e a precisão de medição.

Tabela 2 – Especificações técnicas do *Microsoft* kinect

Características	**Especificações**
Resolução do sensor RGB	$640 \times 480\ px$
Faixa de alcance do sensor de profundidade	0.8 à $4.0\ m$
Campo de visão horizontal	$57°$
Campo de visão vertical	$43°$
Frequência de captura	30 FPS
Precisão de medição de profundidade	± 1 cm

Adaptado de (EL-IAITHY; HUANG; YEH, 2012)

No ambiente de desenvolvimento deste trabalho, o pacote do Kinect 360 já estava configurado e funcional no ROS, o que permitiu a recepção das informações de imagem RGB e profundidade diretamente por meio de tópicos, sem a necessidade de instalação

ou configuração adicional. Essa configuração pré-existente facilitou a integração do sensor ao sistema, fornecendo dados em tempo real para processamento de imagem.

3.1.1.3 Câmera Basler Pylon a2A1920-51gcPRO

A câmera Basler Pylon a2A1920-51gcPRO da linha ace 2 R Pro, ilustrada na Figura 29, é projetada para aplicações de visão computacional industrial. Equipado com um sensor Sony IMX392LQR CMOS, o dispositivo oferece captura de imagem sem distorção e em alta velocidade. Suas especificações completas são apresentadas na Tabela 3.

Figura 29 – Câmera Basler Pylon a2A1920-51gcPRO da linha Ace 2 Pro.

Fonte: (Basler AG, 2023)

Tabela 3 – Especificações da Câmera Basler Pylon

Especificação	**Descrição**
Resolução	$1920 \times 1200\ px$
Taxa de Quadros	Até 51 FPS
Sensor	Sony IMX392LQR CMOS
Tamanho do Pixel	$3.45 \times 3.45\ \mu m$
Interface de Dados	Gigabit Ethernet (1000 $Mbit/s$) e Fast Ethernet (100 $Mbit/s$)
Alimentação	via I/O com entrada de 12-24 VDC
Consumo de Energia	3.4 W (12-24 VDC)
Dimensões	48.9 mm × 29 mm × 29 mm (sem lente)
Peso	menor que 105 g

Fonte: (Basler AG, 2023)

No ambiente de desenvolvimento deste trabalho, o pacote da câmera Basler Pylon já estava configurada e funcional no ROS, o que permitiu a recepção das informações de imagem diretamente por meio de tópicos, sem a necessidade de instalação ou configuração adicional. Essa configuração pré-existente facilitou a integração do sensor ao sistema.

3.1.1.4 Robô Móvel Turtlebot2i

O Turtlebot2i é um robô móvel desenvolvido para aplicações de pesquisa, educação e prototipagem na área de robótica (TurtleBot, 2023b). É amplamente utilizado em

projetos de navegação, mapeamento e manipulação, sendo uma plataforma flexível e modular que permite a adição de diversos sensores, o que facilita seu uso em experimentos de robótica autônoma e colaborativa (TurtleBot, 2023c). A Figura 30 ilustra o Turtlebot2i, evidenciando sua estrutura modular.

Figura 30 – Robô Móvel Turtlebot2i.

Fonte: (TurtleBot, 2023a)

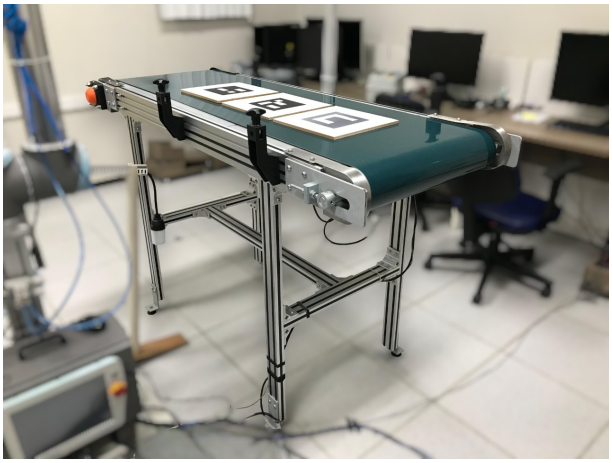
Como o Turtlebot2i já estava integrado ao ROS no ambiente de desenvolvimento, não foi necessário realizar configurações adicionais para o uso deste robô no contexto deste trabalho. Todos os pacotes necessários para a comunicação e operação do Turtlebot2i estavam previamente instalados, permitindo o acesso direto aos tópicos e serviços oferecidos pelo robô, sem a necessidade de implementações adicionais. O Turtlebot2i utiliza o ROS para compartilhar dados dos sensores e receber comandos de movimento. Sua estrutura modular é composta por sensores e uma base móvel equipada com rodas, permitindo deslocamentos em diversas direções.

3.1.1.5 Esteira Industrial

O ambiente de desenvolvimento conta com uma esteira industrial, conforme ilustrado na Figura 31. Sua função é simular uma linha de produção básica,permitindo a movimentação de peças para interação com outros elementos, como o robô UR10 e o Turtlebot2i. Esse fluxo contínuo permite atividades como identificação, manipulação e transferência de peças de um ponto a outro. Diferente dos outros dispositivos presentes no ambiente, a esteira não possui controle integrado ao sistema ROS. Dessa forma, sua operação é contínua e independente, funcionando de maneira autônoma, sem receber comandos ou enviar informações para o sistema de controle principal.

Embora a esteira não seja controlada, sua integração ao ambiente experimental é fundamental para replicar cenários de produção realistas, onde diferentes dispositivos robóticos precisam interagir com sistemas de transporte passivos. A ausência de controle impõe desafios adicionais, exigindo que os demais dispositivos ajustem seus comportamen-

Figura 31 – Esteira industrial.



Fonte: Autoria própria

tos de acordo com a posição e o movimento dos objetos, que são detectados e monitorados através de sensores externos. Essa configuração permite que o sistema execute tarefas em sincronia com o movimento constante da esteira, garantindo que o ambiente experimental seja representativo de cenários industriais reais, onde sistemas de transporte automatizado nem sempre são integrados ao controle central.

3.2 Integração dos dispositivos físicos com o ROS

Para configurar e gerenciar os dispositivos físicos no laboratório *Smart Factory*, foi desenvolvido o pacote `smartfactory_bringup`. Esse pacote organiza os arquivos de configuração e inicialização necessários para a integração dos dispositivos com o ambiente ROS, permitindo uma configuração centralizada e simplificada. A estrutura de diretórios do pacote é apresentada na Figura 32.

O pacote `smartfactory_bringup` possui uma estrutura organizada em três diretórios principais, descritos a seguir:

- `config/`: armazena os arquivos de configuração dos dispositivos, incluindo parâmetros de calibração das câmeras e tópicos específicos para comunicação no ROS. Esses arquivos facilitam a personalização de parâmetros de cada dispositivo, tornando o ambiente experimental mais flexível e adaptável a diferentes configurações.
- `launch/`: contém os arquivos de inicialização, que são responsáveis por carregar as configurações e ativar os dispositivos físicos no ambiente ROS.
- `smartfactory_bringup/`: armazena os nós que gerenciam a comunicação e controle dos dispositivos no sistema distribuído.

Figura 32 – Estrutura de diretórios do pacote `smartfactory_bringup`

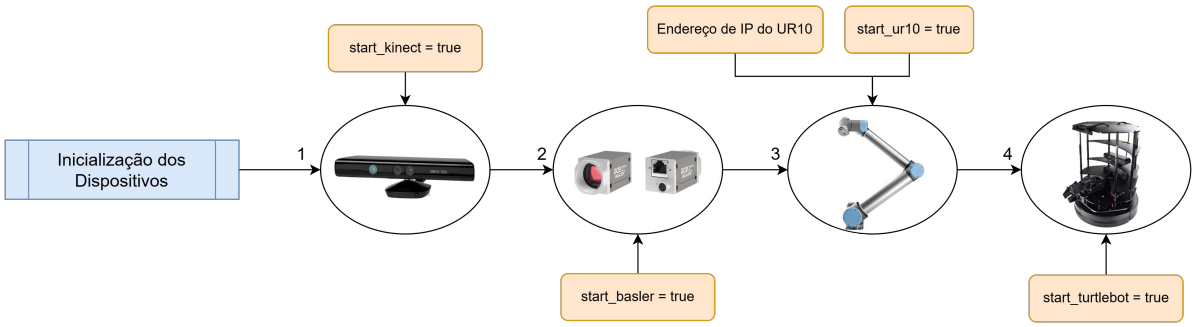
```
smartfactory_ws/
|-- src/
    |-- smartfactory_bringup/
    |   |-- config/
    |   |-- launch/
    |   |-- smartfactory_bringup/

```

Fonte: Autoria própria

Para a inicialização dos dispositivos, o pacote `smartfactory_bringup` utiliza um fluxo de lançamento que organiza a ativação dos dispositivos na ordem desejada. Cada dispositivo — incluindo o robô UR10, o TurtleBot2i, e as câmeras Kinect e Basler — possui um nó dedicado no ROS que é acionado por meio deste arquivo `launch`. O diagrama na Figura 33 ilustra o processo, onde cada dispositivo é ativado conforme o argumento correspondente esteja configurado como `true`.

Figura 33 – Fluxo de inicialização dos dispositivos.



Fonte: Autoria própria

1. Inicialização das Câmeras Kinect e Basler:

   a) Kinect: O nó d do Kinect, é lançado se o argumento `start_kinect` estiver ativo. Esse nó `kinect_ros2_node` publica as imagens RGB e de profundidade para o ambiente ROS, facilitando a detecção de marcadores e a integração com o controle visual.

   b) Basler: Similar ao Kinect, o nó `pylon_ros2_camera_wrapper` captura e publica imagens RGB para operações de controle e monitoramento.

2. Inicialização do UR10: Quando o argumento `start_ur10` está ativo, o sistema inclui o arquivo de lançamento do UR10. Esse arquivo configura a conexão do manipulador ao ambiente ROS, definindo o endereço IP do controlador.

3. Inicialização do TurtleBot2i: Similarmente, com o argumento `start_turtlebot` ativo, o pacote `turtlebot2i_bringup` lança o nó integrando o robô ao ROS para permitir controle autônomo ou teleoperado.

Esse fluxo de inicialização flexível permite que o usuário configure quais dispositivos devem ser ativados na cena experimental, garantindo que todos estejam prontos para realizar as operações de controle e monitoramento conforme necessário.

3.3 Cinemática do UR10

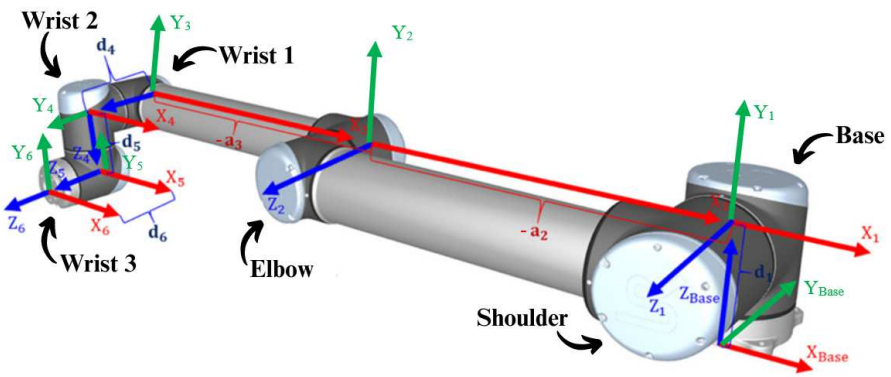
A cinemática inversa do manipulador UR10 foi desenvolvida com base na teoria apresentada na seção 2.4. Para esse cálculo, foram utilizados os parâmetros de Denavit-Hartenberg (DH) específicos do UR10, descritos na Tabela 4. Esses parâmetros fornecem uma descrição das relações geométricas entre as juntas do robô, facilitando o desenvolvimento de uma solução analítica.

Tabela 4 – Parâmetros DH do UR10

Kinematics	θ [rad]	a [m]	d [m]	α [rad]
Junta 1 - *Base*	0	0	0.1273	$\pi/2$
Junta 2 - *Shoulder*	0	-0.612	0	0
Junta 3 - *Elbow*	0	-0.5723	0	0
Junta 4 - *Wrist* 1	0	0	0.163941	$\pi/2$
Junta 5 - *Wrist* 2	0	0	0.1157	$-\pi/2$
Junta 6 - *Wrist* 3	0	0	0.0922	0

Fonte: adaptado de (Universal Robots, 2023a)

Figura 34 – Definição dos parâmetros DH para o UR10



Fonte: (NASCIMENTO et al., 2023)

Para a implementação, foi desenvolvido um código em Python com base nos trabalhos de (HAWKINS, 2013) e (YANG et al., 2023). Esse código calcula 6 soluções possíveis, permitindo que o manipulador alcance a posição desejada com diferentes configurações das juntas, adaptando-se a possíveis restrições do ambiente e preferências de operação.

1. Cálculo de θ_1: ângulo da base

   O primeiro ângulo, é determinado usando a projeção do ponto posição da quinta junta (p_{05}) no plano xy:

   $$p_{05} = \mathcal{K}(\mathbf{q}) \cdot \begin{bmatrix} 0 \\ 0 \\ -d_6 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_{05x} \\ p_{05y} \\ p_{05z} \\ 1 \end{bmatrix}$$

   onde $\mathcal{K}(\mathbf{q})$ é a matriz de transformação desejada para a pose do efetuador final, seu desenvolvimento pode ser visto na Equação 2.9. E d_6 é o parâmetro DH que representa a distância entre a quinta e o efetuador final ao longo do eixo z_5, isso pode ser visto na Figura 34. Com isso,

   $$\theta_1 = \frac{\pi}{2} + \psi \pm \phi \tag{3.1}$$

   onde $\psi = \arctan\left(\frac{p_{05y}}{p_{05x}}\right)$, $\phi = \arccos\left(\frac{d_4}{\sqrt{p_{05x}^2 + p_{05y}^2}}\right)$, d_4 é o parâmetro DH que representa a distância entre a quarta e a quinta junta ao longo do eixo z_4, como mostrado na Figura 34. Essa fórmula resulta em duas possíveis soluções para θ_1, indicando posições diferentes do "ombro", sendo uma para a direita e outra para a esquerda.

2. Cálculo de θ_5: ângulo do *wrist 2*

   Esse ângulo é calculado usando a coordenada z do ponto p_{16} no sistema de referência da primeira junta. Aqui, p_{16} representa a posição do efetuador final em relação ao sistema de coordenadas da junta 1, o que facilita o cálculo da orientação do pulso. Esse ponto é obtido pela transformação entre a primeira junta e o efetuador final, utilizando:

   $$T_{10} = {}^{0}\xi_1{}^{-1}$$

   $$T_{16} = T_{10} \cdot \mathcal{K}(\mathbf{q}) = \begin{pmatrix} r_{11} & r_{12} & r_{13} & p_{16x} \\ r_{21} & r_{22} & r_{23} & p_{16y} \\ r_{31} & r_{32} & r_{33} & p_{16z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   onde T_{10} é a transformação da base para a primeira junta. O desenvolvimento da matriz de transformação é descrito na Equação 2.12. Com p_{16z} determinado, θ_5 é calculado pela expressão:

   $$\theta_5 = \arccos\left(\frac{p_{16z} - d_4}{d_6}\right) \tag{3.2}$$

Essa expressão fornece duas soluções para θ_5, indicando orientações diferentes do pulso.

3. Cálculo de θ_6: ângulo do *wrist 3*

   Para o cálculo de θ_6, utilizamos o sistema de coordenadas da junta 1

$$T_{61} = (\mathcal{K}(\mathbf{q}))^{-1} \cdot T_{10} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & p_{61x} \\ r_{21} & r_{22} & r_{23} & p_{61y} \\ r_{31} & r_{32} & r_{33} & p_{61z} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

   Com T_{61} obtido, θ_6 é calculado pela expressão:

$$\theta_6 = \arctan\left(-\frac{r_{23}}{\sin(\theta_5)}, \frac{r_{13}}{\sin(\theta_5)}\right) \tag{3.3}$$

   Esse ângulo orienta o efetuador final na direção desejada em relação ao eixo z do robô, definindo a rotação

4. Cálculo de θ_3: ângulo de *elbow*

   Para encontrar θ_3, foi utilizado o comprimento efetivo dos elos e a posição projetada de p_{13} entre as juntas 2 e 3. Esse ponto p_{13} representa a posição da terceira junta em relação ao sistema de coordenadas da primeira junta. O cálculo é feito da seguinte forma:

$$T_{14} = T_{10} \cdot (\mathcal{K}(\mathbf{q})) \cdot T_{65} \cdot T_{54}$$

   na qual T_{14} é a transformação composta entre as juntas 1 e 4, $T_{65} = {}^{5}\xi_{6}{}^{-1}$ é a transformação da junta 5 para a junta 6, $T_{54} = {}^{4}\xi_{5}{}^{-1}$ é a transformação da junta 4 para a junta 5. Com isso,

$$p_{13} = T_{14} \cdot \begin{bmatrix} 0 \\ -d_4 \\ 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} p_{13x} \\ p_{13y} \\ p_{13z} \\ 1 \end{bmatrix}$$

$$\|p_{13}\| = \sqrt{p_{13x}^2 + p_{13y}^2 + p_{13z}^2}$$

   Com p_{13} definido, o ângulo θ_3 é calculado como:

$$\theta_3 = \arccos\left(\frac{\|p_{13}\|^2 - a_2^2 - a_3^2}{2a_2a_3}\right) \tag{3.4}$$

   onde a_2 e a_3 são os comprimentos dos elos entre as juntas 2 e 3 e entre as juntas 3 e 4, respectivamente (ilustrado na Figura 34). Esse cálculo fornece duas soluções para θ_3, correspondendo a diferentes configurações do cotovelo: uma com o cotovelo apontando para cima e outra com o cotovelo apontando para baixo.

5. Cálculo de θ_2: ângulo de *shoulder*

$$\theta_2 = -\arctan(p_{13y}, -p_{13x}) + \arcsin\left(\frac{a_3 \sin(\theta_3)}{\|p_{13}\|}\right) \tag{3.5}$$

6. Cálculo de θ_4: ângulo de *wrist 1*

$$T_{34} = T_{32} \cdot T_{21} \cdot T_{14} = \begin{pmatrix} T_{34_{11}} & T_{34_{12}} & T_{34_{13}} & T_{34_{14}} \\ T_{34_{21}} & T_{34_{22}} & T_{34_{23}} & T_{34_{24}} \\ T_{34_{31}} & T_{34_{32}} & T_{34_{33}} & T_{34_{34}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

na qual $T_{32} = {}^{2}\xi_3{}^{-1}$ é a transformação da junta 2 para a junta 3, $T_{21} = {}^{1}\xi_2{}^{-1}$ é a transformação da junta 2 para a junta 1. Agora, $T_{34_{11}}$ e $T_{34_{21}}$ representam as projeções utilizadas no cálculo de θ_4 como segue:

$$\theta_4 = \arctan(T_{34_{21}}, T_{34_{11}}) \tag{3.6}$$

Neste trabalho, foi necessário considerar restrições nos ângulos das juntas devido à limitação do espaço de trabalho do manipulador no laboratório *Smart Factoring*. Devido ao posicionamento da esteira, uma das restrições é que o robô deve adotar uma configuração com o cotovelo para cima. Portanto, a solução ideal para a aplicação desenvolvida é a segunda configuração de juntas.

3.4 Simulação

O objetivo principal de uma simulação em robótica é verificar o funcionamento e a integração de sistemas antes de aplicá-los no mundo real. Assim, ela permite que dispositivos e algoritmos sejam testados e ajustados em condições semelhantes às do ambiente físico, possibilitando a identificação e correção de problemas sem os riscos e os custos associados a falhas em sistemas reais.

Neste trabalho, foi desenvolvida uma simulação com o objetivo de criar uma representação virtual do laboratório *Smart Factoring*, replicando o layout e os dispositivos presentes no ambiente físico. O simulador Gazebo foi escolhido por sua integração nativa com o ROS, facilitando o desenvolvimento e a comunicação entre dispositivos simulados e sistemas de controle. Essa integração permite o uso dos mesmos pacotes e configurações que serão aplicados posteriormente no ambiente físico, reduzindo o esforço de adaptação entre a simulação e o sistema físico.

3.4.1 Estrutura dos pacotes de simulação

Para desenvolver e controlar a simulação no Gazebo, foram criados dois pacotes principais em Python: `smartfactory_description` e `smartfactory_simulation`. Cada pacote possui uma estrutura de diretórios organizada para armazenar os arquivos e configurações necessários, facilitando o desenvolvimento modular e a organização dos recursos. A estrutura desses diretórios pode ser vista na Figura 35.

Figura 35 – Estrutura de diretórios dos pacotes

```
smartfactory_ws/
|-- src/
    |-- smartfactory_description/
    |   |-- meshes/
    |   |-- rviz/
    |   |-- urdf/
    |
    |-- smartfactory_simulation/
        |-- launch/
        |-- models/
        |-- world/

```

Fonte: Autoria própria

O pacote `smartfactory_description` define e organiza os modelos dos dispositivos, com três diretórios principais::

- *meshes*: Armazena arquivos *STL* (*Standard Tessellation Language*) que representam a geometria 3D dos dispositivos, como o UR10, Kinect, Basler, Turtlebot2i e a esteira. Esses arquivos são usados para exibir a aparência dos objetos na simulação.
- *rviz*: Inclui arquivos de configuração para o *Rviz*, permitindo visualizar robôs, sensores e imagens das câmeras em 3D.
- *urdf*: Contém arquivos *Xacro* e *URDF* (*Unified Robot Description Format*), que descrevem a estrutura de cada dispositivo, incluindo links, juntas, posição e orientação dos objetos.

O pacote `smartfactory_simulation` gerencia a inicialização e controle de todos os elementos simulados no Gazebo, centralizando o layout e os parâmetros operacionais da simulação. Suas principais pastas incluem:

- *launch*: Armazena arquivos `.launch.py` que inicializam todos os dispositivos na simulação, como robôs, câmeras e elementos físicos do laboratório. Esses arquivos de inicialização organizam todos os nós necessários para configurar a cena completa,

incluindo as funcionalidades dos robôs UR10 e Turtlebot2i, e das câmeras Kinect e Basler.

- *models*: Armazena configurações específicas para cada modelo, como parâmetros de posição, orientação e outras características operacionais. Cada elemento na cena, incluindo móveis e equipamentos, possui um modelo configurado para refletir sua localização dentro do laboratório.

- *world*: Armazena o arquivo `.world`, que define o layout completo do laboratório no Gazebo. Esse arquivo replica a disposição física dos móveis, robôs e câmeras, proporcionando uma representação virtual precisa do ambiente de trabalho real.

3.4.2 Configuração da cena e integração com o ROS

O processo de criação da cena envolveu a modelagem do ambiente físico, a inserção dos robôs e sensores, e a configuração dos arquivos de lançamento para inicializar a simulação completa.

Para representar a estrutura física do laboratório no ambiente virtual, foram coletadas medidas da sala com o auxílio de uma trena, garantindo que cada elemento estivesse posicionado conforme o ambiente real. A estrutura da sala foi criada utilizando modelos do repositório *Gazebo Models and Worlds Collection*[9], que oferece uma variedade de objetos e materiais prontos para compor ambientes no Gazebo. Com base nessas medidas, foi construído o arquivo `lab_smart_factory.world`, onde foram inseridos móveis, computadores e outros elementos do laboratório, posicionados de acordo com a configuração real.

Os robôs UR10 e Turtlebot2i foram inseridos na cena utilizando pacotes ROS prontos, ajustados para garantir que suas posições coincidissem com as do laboratório físico:

- UR10: O manipulador UR10 foi adicionado à simulação através do pacote *Universal Robots ROS2 Gazebo Simulation*[10], que fornece uma representação do manipulador. Após carregar o modelo, a posição do UR10 foi ajustada no ambiente virtual para coincidir com a posição real no laboratório.

- Turtlebot2i: Para o Turtlebot2i, foi utilizado o pacote *Turtlebot2i Simulation*[11], que fornece uma simulação do robô móvel.

[9] Repositório Gazebo Models and Worlds Collection: <https://github.com/leonhartyao/gazebo_models_worlds_collection>

[10] Repositório Universal Robots ROS2 Gazebo Simulation: <https://github.com/UniversalRobots/Universal_Robots_ROS2_Gazebo_Simulation>

[11] Repositório Turtlebot2i Simulation: <https://github.com/SmartFactoryLab-UFCG/smartfactory_pkg/tree/main/smartfactory_ws/src/smartfactory/smartfactory_simulation>

As câmeras Kinect e Basler foram adicionadas à simulação com o uso de arquivos *xacro* desenvolvidos especificamente para esse trabalho. Os arquivos `basler.urdf.xacro` e `kinect.urdf.xacro` permitiram incluir os modelos *STL* das câmeras, definir suas posições no ambiente e configurar o plugin `libgazebo_ros_camera`, que simula as funcionalidades das câmeras no ROS, incluindo parâmetros de resolução, campo de visão e frequência de captura. As câmeras foram posicionadas de acordo com suas localizações reais no laboratório.

Para inicializar a simulação, foi desenvolvido um arquivo de lançamento que organiza a inicialização completa da simulação no Gazebo, configurando todos os dispositivos da cena e integrando-os ao ROS. Esse arquivo de lançamento chama os seguintes nós:

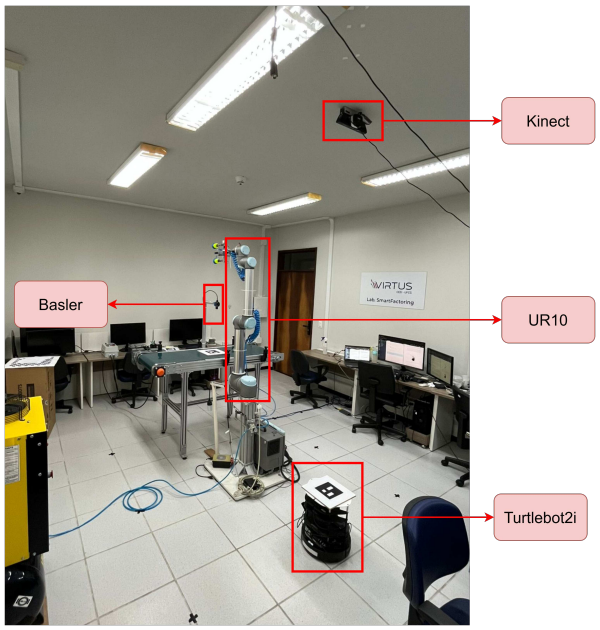
1. Gazebo: inicializa o ambiente de simulação, carregando o arquivo *world* que define a estrutura completa do laboratório.
2. TurtleBot2i: lança o robô móvel, configurado para operar de acordo com a posição e orientações definidas na cena, replicando seu papel no laboratório real.
3. UR10: inicializa o manipulador, utilizando o pacote oficial do ROS, com sua posição ajustada para coincidir com a disposição no laboratório físico.
4. Câmeras Kinect e Basler: configura as câmeras, incluindo parâmetros de captura de imagem e profundidade, por meio do *plugin*.
5. Rviz: Lança o Rviz, permitindo a visualização em tempo real de todos os elementos da simulação, facilitando o monitoramento e ajuste dos dispositivos na cena.
6. Transformações estáticas: Configura nós de transformações estáticas que sincronizam os sistemas de coordenadas para garantir que todos os dispositivos na simulação estejam no mesmo referencial.

Na Figura 36, é possível ver a configuração do laboratório físico com os dispositivos reais, como o UR10, o TurtleBot2i, a câmera Kinect, a câmera Basler, a esteira e os móveis do laboratório.

A Figura 37 apresenta a simulação no Gazebo, onde o ambiente virtual foi configurado para replicar a estrutura física do laboratório, incluindo a disposição dos móveis, os computadores, a esteira e os robôs.

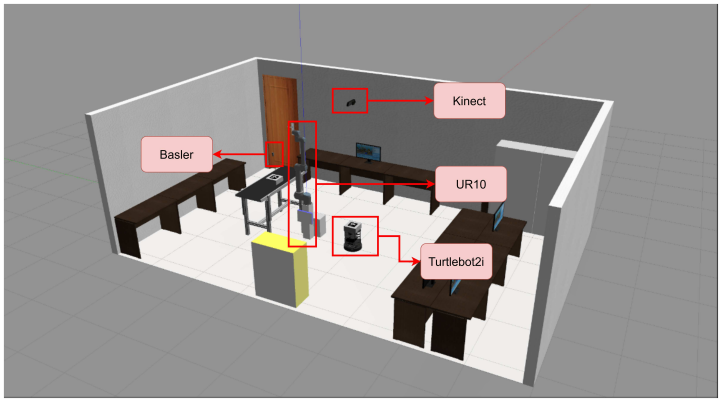
Por fim, a Figura 38 mostra a visualização no *Rviz*.

Figura 36 – Laboratório físico *Smart Factoring*.



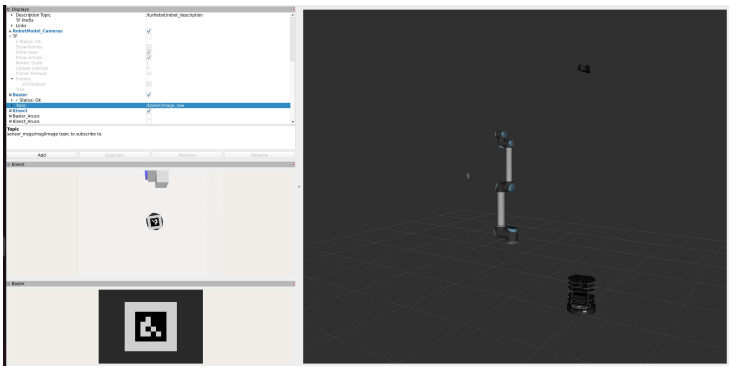
Fonte: Autoria própria

Figura 37 – Simulação do laboratório no Gazebo.



Fonte: Autoria própria

Figura 38 – Visualização no *Rviz*.



Fonte: Autoria própria

4 Controle Orientado a Eventos em uma Fábrica Inteligente

Neste capítulo é apresentada a contribuição deste trabalho para o controle orientado a eventos de uma fábrica inteligente, explorando a aplicação de árvores de comportamento em um contexto de produção automatizada. A abordagem proposta visa modelar e gerenciar a interação entre diferentes dispositivos e sistemas da fábrica — incluindo robôs, câmeras e esteiras — de forma adaptativa e flexível. Esse controle é baseado em uma estrutura de programação distribuída, que permite a adaptação do comportamento dos dispositivos em resposta a eventos detectados em tempo real, garantindo uma produção contínua e eficiente.

Para validar a flexibilidade e adaptabilidade do modelo de controle proposto, foi definida uma cena de fábrica que simulam cenários reais de produção automatizada. Cada cena representa uma tarefa específica, configurando uma sequência de ações que devem ser executadas pelo sistema de controle. Essa cena, descrita a seguir, envolve a interação entre o TurtleBot2i, o manipulador UR10 e uma esteira transportadora, utilizando visão computacional para monitoramento e detecção dos objetos a serem transportados.

4.1 Definição das Tarefas

Para validar o modelo de controle orientado a eventos em uma fábrica inteligente, foi desenvolvida uma cena experimental em que o TurtleBot2i transporta um objeto até a área de trabalho do UR10. Essa cena representa uma operação de *pick-and-place* (pegar e deixar) automatizada, onde o objetivo é que o UR10 mova o objeto do TurtleBot2i para a esteira transportadora, simulando um processo de produção típico em ambientes industriais.

Na cena, o TurtleBot2i é responsável por transportar um objeto ao longo da fábrica até uma posição próxima ao UR10. A movimentação do TurtleBot2i é feita por teleoperação. O objeto transportado por ele possui um marcador ArUco na superfície superior, que permite a sua identificação e rastreamento. O Kinect, posicionado estrategicamente para monitorar a área de trabalho do UR10, detecta o marcador ArUco e calcula a pose do objeto em relação ao sistema de coordenadas de referência central da fábrica.

A sequência de tarefas no processo segue um fluxo orientado a eventos, conforme descrito a seguir:

1. Detecção do objeto pelo Kinect: ao chegar na área de trabalho do UR10, o Turtle-

Bot2i para e o Kinect identifica o marcador ArUco no objeto. A câmera "calcula" a pose do objeto (posição e orientação) em relação ao centro da fábrica e publica essa informação em um tópico do ROS.

2. Posicionamento do UR10 para captura do objeto: O UR10, localizado na posição fixa $(0, 0, 0.75)$ em relação ao sistema de coordenadas do mundo, recebe a pose do objeto pelo tópico publicado pelo Kinect. Com isso, é realizando o cálculo de cinemática inversa para definir os ângulos de junta que o UR10 precisa para atingir a posição do objeto. Esses ângulos são então enviados para o UR10 via ROS, permitindo que ele ajuste suas juntas para "pegar" o objeto usando sua ventosa.

3. Transporte do objeto para a esteira: Após capturar o objeto, o UR10 o move até a posição inicial da esteira transportadora. Como a esteira é estática e suas dimensões e localização são conhecidas, a posição de entrega do objeto já foi previamente calculada e salva no sistema para otimizar o processo, evitando novos cálculos de cinemática inversa a cada operação. O UR10 então posiciona o objeto na esteira, permitindo que ele siga para o próximo estágio do processo produtivo.

Essa cena exemplifica a integração entre visão computacional e robótica.

4.2 Modelo SED do UR10

O modelo baseado teoria de Sistemas a Eventos Discretos (SED), abordada na seção 2.2, desenvolvido para o manipulador robótico UR10 descreve seu comportamento em um ambiente de trabalho, onde o robô alterna entre estados de espera, movimento e execução de tarefas. Além disso, o modelo inclui eventos de segurança, como paradas de emergência, que interrompem as operações em casos de risco. Essa representação permite desenvolver uma estrutura de controle responsiva em tempo real, ajustando o comportamento do robô conforme os estados e transições definidos no autômato.

O comportamento básico do manipulador pode ser descrito da seguinte forma:

1. Inicialmente, o robô encontra-se desligado e precisa ser ligado manualmente.

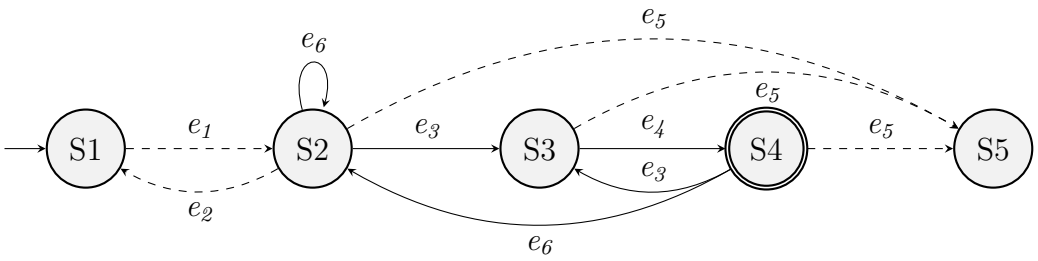
2. Após ligado, o robô permanece em estado de espera até receber uma tarefa.

3. Ao receber um comando de tarefa, o robô se move até a posição necessária para realizar a ação.

4. Durante o movimento ou ao alcançar a posição final, o robô pode entrar em um estado de emergência caso ocorra uma situação inesperada.

A sequência de estados e transições desse ciclo operacional é formalizada no autômato a seguir.

Figura 39 – Autômato do modelo SED para um manipulador UR10



Fonte: Autoria própria

O sistema de controle do manipulador é representado pela estrutura formal do autômato G, descrita Equação 2.6, com os seguintes elementos: $X = \{S_1, S_2, S_3, S_4, S_5\}$ é o conjunto de estados do sistema, representando cada etapa de operação do robô; $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ é o conjunto de eventos que geram transições entre os estados; $E_{uc} = \{e_1, e_2, e_5\}$ representa os eventos não controláveis, que ocorrem sem intervenção do sistema de controle; $f : X \times E \to X$ é a função de transição, que define a mudança de estados em resposta a eventos; $x_0 = S1$ é o estado inicial, onde o robô se encontra desligado; $X_m = \{S4\}$ é o conjunto de estados marcados.

A Tabela 5 detalha os estados do manipulador no modelo SED.

Tabela 5 – Estados do manipulador UR10

Estado	**Descrição**
S_1	Robô desligado
S_2	Robô ligado e em espera (Idle)
S_3	Movendo para a pose da tarefa
S_4	Executando tarefa
S_5	Parada de emergência

Fonte: Autoria própria

A Tabela 6 lista os eventos que ocorrem, indicando se são controláveis ou não.

4.3 Modelagem do Supervisor em Árvore de Comportamento

A escolha por uma árvore de comportamento possibilita uma estrutura modular e flexível, favorecendo a reatividade e recuperação de falhas — aspectos essenciais para sistemas de controle em tempo real (GUGLIERMO et al., 2024). Para a cena desenvolvida, o supervisor foi implementado usando uma árvore de comportamento, o que permite que

Tabela 6 – Eventos do manipulador UR10

Evento	**Descrição**	**Controlável**
e_1	Ligar o robô	×
e_2	Desligar o robô	×
e_3	Recebeu uma tarefa	✓
e_4	Executar tarefa	✓
e_5	Parada de emergência	×
e_6	Sem tarefa	✓

Fonte: Autoria própria

o sistema ajuste dinamicamente as ações do UR10 em resposta ao evento de chegada do TurtleBot2i e à detecção do objeto, permitindo uma operação autônoma e eficiente. Essa configuração torna-se vantajosa em relação ao uso de Sistemas a Eventos Discretos (SED) convencionais, pois o SED não suporta a mesma flexibilidade na execução e recuperação de tarefas em tempo real com a mesma granularidade de controle dos estados.

Na árvore de comportamento projetada para a fábrica inteligente, cada nó desempenha uma função específica, garantindo que as tarefas sejam executadas de forma ordenada. A estrutura da árvore inicia-se com o nó "Raiz", que é um nó paralelo. Esse nó permite que todos os seus filhos sejam executados simultaneamente, retornando sucesso apenas quando todos os nós filhos completam suas respectivas tarefas. Esse tipo de nó é utilizado para coordenar de forma sincronizada a execução de todas as ações principais.

Logo abaixo do nó raiz, temos a "Sequência Principal", que organiza os passos necessários para que o UR10 realize a tarefa de pegar o objeto e transportá-lo até a esteira. Esse nó é sequencial, executando cada comportamento da esquerda para a direita e prosseguindo apenas se cada nó filho retornar sucesso, o que garante que a ordem das operações seja mantida. Abaixo estão os detalhes dos principais nós dessa sequência:

1. Verificação da pose do ArUco: este nó verifica se a pose do marcador ArUco foi informada pelo Kinect. É um nó de condição que retorna sucesso se a pose estiver disponível, permitindo que a sequência prossiga. Caso contrário, ele retorna em execução, aguardando que a pose seja detectada. Esse nó funciona como um pré-requisito, assegurando que o UR10 só inicie seu movimento após a detecção do objeto.

2. Envio dos ângulos de junta para o UR10: esse é um nó de ação responsável por enviar os ângulos de junta calculados ao UR10, que o posicionarão até o objeto. Ele permanece em execução enquanto o processo de envio está ativo e retorna sucesso quando a tarefa é concluída. Esse comportamento representa o movimento inicial do UR10 para posicioná-lo corretamente antes de pegar o objeto.

3. Ativação da ventosa: após o posicionamento, o próximo nó ativa a ventosa do UR10 para segurar o objeto. Esse nó é configurado como um nó de sequência com timeout. Ele envia o comando de ativação da ventosa e espera um curto período, após o qual assume que a ventosa está ativa e o objeto foi segurado, retornando sucesso. Isso permite ao sistema avançar para a próxima tarefa sem esperar uma confirmação.

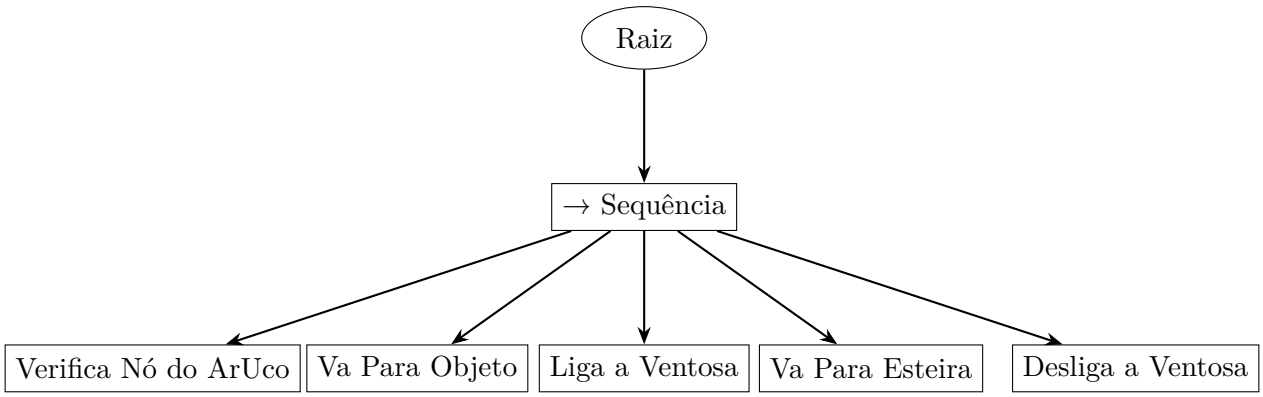
4. Movimento até a esteira: esse nó de ação comanda o UR10 a transportar o objeto até a posição da esteira. Ele utiliza os ângulos de junta pré-calculados para alcançar a posição desejada de maneira precisa. Esse nó é executado até que a posição seja alcançada e retorna sucesso ao concluir a tarefa.

5. Desativação da ventosa: o último nó da sequência desativa a ventosa, soltando o objeto na esteira para que ele continue no processo de produção. Esse nó é implementado como um nó de sequência com timeout. Após enviar o comando para desativar a ventosa, ele espera um tempo predefinido antes de retornar sucesso, assumindo que o objeto foi liberado com segurança na esteira. Esse design permite que o UR10 conclua o processo de liberação sem precisar de confirmação sensorial, garantindo uma transição suave para a etapa final do transporte do objeto.

A estrutura da árvore de comportamento permite que o sistema responda de forma autônoma aos eventos, ajustando as ações do UR10 em tempo real e garantindo uma operação coordenada. Essa configuração lógica e modular facilita também a adaptação a novas tarefas e a recuperação em caso de falhas. O diagrama dessa estrutura pode ser visto na Figura 40.

Cabe ressaltar que o cálculo da cinemática inversa é realizado fora da árvore de comportamento, sendo processado por um sistema externo que envia os ângulos diretamente ao UR10, garantindo a eficiência e simplificando a implementação dentro da árvore de comportamento.

Figura 40 – Árvore de comportamento do surpevisor do sistema.



Fonte: Autoria própria.

5 Resultados

É importante esclarecer que o escopo deste trabalho não inclui o desenvolvimento dos algoritmos que controlam a navegação do TurtleBot2i até a área de trabalho do UR10 ou o cálculo da pose do marcador ArUco pelo Kinect. Ambos os dispositivos fazem parte do ambiente pré-configurado no laboratório *Smart Factoring*, e suas informações são enviadas para o supervisor (a árvore de comportamento) através do ROS. Esse trabalho, portanto, concentra-se na criação e gerenciamento do fluxo de controle do robô manipulador UR10, utilizando as informações recebidas do TurtleBot2i e do Kinect para guiar suas operações.

5.1 Implementação da Árvore de Comportamentos no ROS

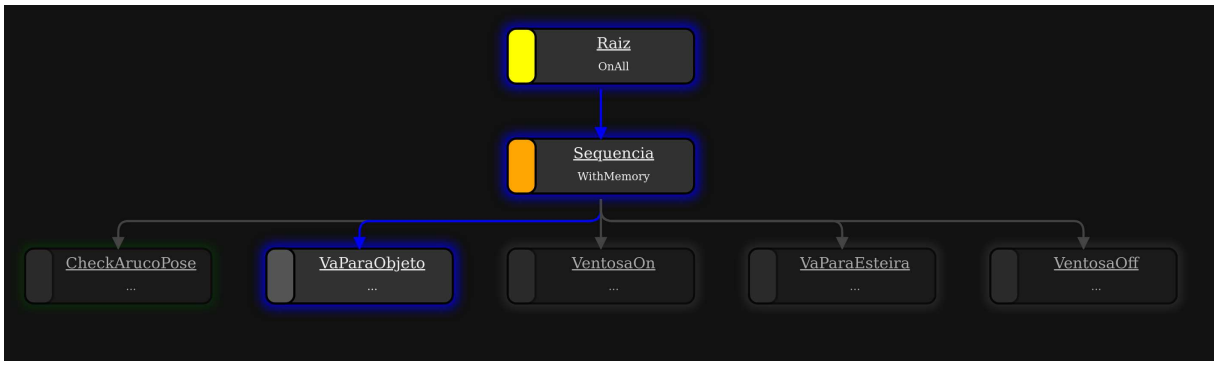
A implementação da árvore de comportamento para o sistema de controle da fábrica inteligente foi realizada utilizando o framework ROS, que permite modularizar e coordenar as diferentes ações e condições envolvidas no processo. Essa abordagem modular facilita o controle de cada dispositivo do sistema, como o TurtleBot2i, a câmera Kinect e o UR10, garantindo a comunicação em tempo real e a reatividade necessária para a execução das tarefas. A árvore de comportamento desenvolvida foi projetada para integrar e coordenar diversos dispositivos, incluindo sensores e atuadores, assegurando a execução eficiente e ordenada das tarefas de manipulação e transporte de objetos.

A árvore de comportamento foi implementada em Python utilizando a biblioteca `py_tree_ros`[1], que permite a criação de nós representando comportamentos específicos e condições que definem a sequência de execução. Cada nó da árvore representa uma ação ou condição, como a detecção do objeto, o movimento do UR10 e a ativação e desativação da ventosa. Essa estrutura hierárquica foi organizada para possibilitar a execução sequencial ou paralela dos nós, conforme a necessidade de cada tarefa. A Figura 41 ilustra a árvore de comportamento desenvolvida para o controle do sistema, visualizada na interface gráfica utilizando a biblioteca `py_tree_ros_viewer`[2].

A árvore de comportamento desenvolvida é composta por um nó raiz configurado como nó paralelo (*Parallel*), com a política *SuccessOnAll*, o que significa que ele só retorna sucesso após a conclusão bem-sucedida de todos os nós filhos. Esse tipo de nó foi escolhido para garantir que todas as ações necessárias na sequência sejam executadas de forma sincronizada, proporcionando uma estrutura robusta e permitindo que cada dispositivo realize sua tarefa no momento adequado. Abaixo do nó raiz, a árvore segue com uma

[1] Repositório py_tree_ros no GitHub: <https://github.com/splintered-reality/py_trees_ros>
[2] Repositório py_tree_ros_viewer no GitHub: <https://github.com/splintered-reality/py_trees_ros_viewer>

Figura 41 – Visualização da árvore de comportamento no ROS para o controle de manipulação do UR10



Fonte: Autoria própria

sequência principal, detalhada a seguir.

O primeiro nó da sequência é um nó de condição (`CheckArucoPose`) que verifica se a pose do marcador ArUco foi detectada e está disponível para o UR10. Esse nó monitora o tópico que fornece a pose do marcador e retorna sucesso ao detectar uma pose válida, permitindo que a sequência continue. Caso contrário, ele permanece em execução (*RUNNING*), indicando que o sistema deve aguardar até que o objeto seja identificado. Esse comportamento assegura que o UR10 só inicie sua movimentação ao objeto após a confirmação da presença do marcador.

Após a confirmação da pose do objeto, (`SendJointAnglesAction`) é o próximo nó da sequência e ele envia os ângulos de junta calculados para o UR10, movendo-o até a posição do objeto. Esse nó é configurado como um nó de ação e utiliza um subprocesso para comunicar os ângulos ao controlador do UR10. Ele permanece em execução enquanto o robô está em movimento, retornando sucesso apenas quando o posicionamento do UR10 é concluído. Para verificar se o robô chegou corretamente à posição desejada, foi implementada uma comparação entre os valores das juntas atuais e os ângulos de junta recebidos do cálculo da cinemática inversa. Esse procedimento assegura que o UR10 se posicione corretamente antes de tentar pegar o objeto.

Com o UR10 posicionado, o próximo nó (`VentosaOn`) ativa a ventosa para segurar o objeto. Esse nó de ação foi configurado com um *timeout* para garantir que a ativação ocorra dentro de um intervalo de tempo específico. Ele retorna sucesso ao concluir a ativação da ventosa e falha caso o processo exceda o tempo estipulado. É importante mencionar que, como o sistema não dispõe de sensores na ventosa para confirmar a captura do objeto, o único parâmetro monitorado é a ativação ou não da ventosa. Esse nó assegura que o objeto está seguro antes de ser transportado até a esteira. Infelizmente no ambiente experimental, não havia outra câmera que pudesse monitorar a ventosa.

O nó subsequente (`SendConveyor`) instrui o UR10 a mover o objeto até a posição

da esteira transportadora, utilizando uma pose previamente calculada e armazenada no sistema para otimizar o processo. Esse nó de ação envia o comando de movimento ao UR10 e fica em execução até que o robô atinja a posição de destino. Ele retorna sucesso ao alcançar a posição correta na esteira, permitindo que o processo avance para a próxima etapa, onde o objeto será liberado.

O último nó da sequência (`VentosaOff`) desativa a ventosa, soltando o objeto na esteira para que ele continue no processo de produção. Esse nó também é configurado com um *timeout* para assegurar que a desativação ocorra dentro de um período seguro. Ele retorna sucesso ao concluir a soltura do objeto, finalizando a sequência e garantindo que o UR10 libere o objeto de forma controlada.

É importante mencionar que, devido aos erros da estimativa da altura do ArUco detectado pelo Kinect, ter uma margem de erro considerável. Foi adotado uma altura fixa para o eixo z. Com isso, os valores recebidos pelo Kinect são apenas com relação aos valores de x e y da pose do objeto.

A utilização de uma árvore de comportamento para modelar o supervisor do sistema proporciona uma estrutura modular, que é fundamental para a flexibilidade e reatividade do sistema de controle. Cada nó representa uma unidade funcional independente, permitindo que o sistema se adapte a diferentes cenários com facilidade. Além disso, o uso de nós de sequência e de ação com *timeout* garante que o sistema responda a falhas e eventos inesperados de forma eficaz, retomando a execução correta sempre que possível.

Em resumo, a implementação da árvore de comportamento no ROS organiza o fluxo das operações de maneira sequencial e controlada, aproveitando as informações recebidas dos diferentes dispositivos, como o TurtleBot2i e o Kinect, para guiar as ações do UR10. Essa abordagem permite que o sistema opere de forma coordenada e eficiente em tempo real, ajustando-se automaticamente às condições do ambiente e garantindo a execução confiável das tarefas de manipulação e transporte de objetos.

5.2 Validação da Árvore de Comportamentos no ROS

A validação da árvore de comportamentos foi realizada em um ambiente simulado e controlado no laboratório *Smart Factoring*. O objetivo foi verificar se o sistema conseguia executar as tarefas programadas de forma sincronizada e autônoma, coordenando a movimentação do UR10 com a chegada do objeto transportado pelo TurtleBot2i e detectado pelo Kinect. As Figuras 42 e 43 ilustram momentos chave do processo, com o UR10 interagindo com o TurtleBot2i para pegar e soltar o objeto na esteira.

É importante notar que a simulação focou principalmente na validação do envio dos ângulos de junta para o UR10 e sua movimentação, enquanto o controle da ventosa

não pôde ser totalmente validado na simulação. Em testes reais, o processo de pegar e soltar o objeto pela ventosa teria uma importância crucial, mas, devido às limitações de sensores na simulação, a verificação do sucesso da captura pelo UR10 foi feita apenas pela ativação do comando de sucção.

A sequência da árvore de comportamento foi monitorada para garantir que cada nó de ação, como o envio de ângulos de junta e a ativação da ventosa, fosse acionado na ordem correta e dentro dos parâmetros de *timeout* definidos. Esse controle de tempo foi especialmente importante para assegurar que a ventosa fosse ativada e desativada no momento certo, ainda que a simulação não permitisse a validação completa de sua funcionalidade física.

Nos testes feitos, foi possível verificar que o UR10 recebia os ângulos de junta calculados e movia-se para a posição desejada, demonstrando que a integração do nó `SendJointAnglesAction` com o controlador do UR10 funcionou conforme esperado. Além disso, a árvore de comportamento coordenou corretamente os movimentos do UR10, com cada nó executando a ação esperada antes de passar para a próxima etapa.

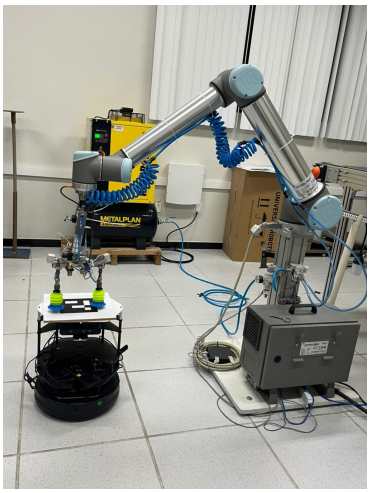
A detecção do objeto pelo Kinect e o transporte até a área de trabalho pelo TurtleBot2i foram bem integrados no fluxo de controle, apesar de a navegação do TurtleBot2i e o cálculo da pose pelo Kinect não estarem no escopo deste trabalho. Esses dispositivos enviaram suas informações diretamente ao sistema através do ROS, permitindo que a árvore de comportamento tomasse decisões baseadas em eventos em tempo real.

As Figuras 42 e 43 mostram o UR10 em momentos diferentes da execução, validando a coordenação entre o robô e o sistema de transporte e detecção de objetos.

Figura 42 – UR10 posicionando o objeto na esteira transportadora.

A construção da cena no *RViz* foi essencial para validar a integração dos dispositivos e monitorar em tempo real o posicionamento e a interação entre eles no ambiente simulado. A cena montada no *RViz* inclui a visualização dos modelos dos principais dispositivos físicos utilizados no projeto: o robô manipulador UR10, o TurtleBot2i, e a câmera

Figura 43 – UR10 interagindo com o TurtleBot2i para pegar o objeto com auxílio da ventosa.



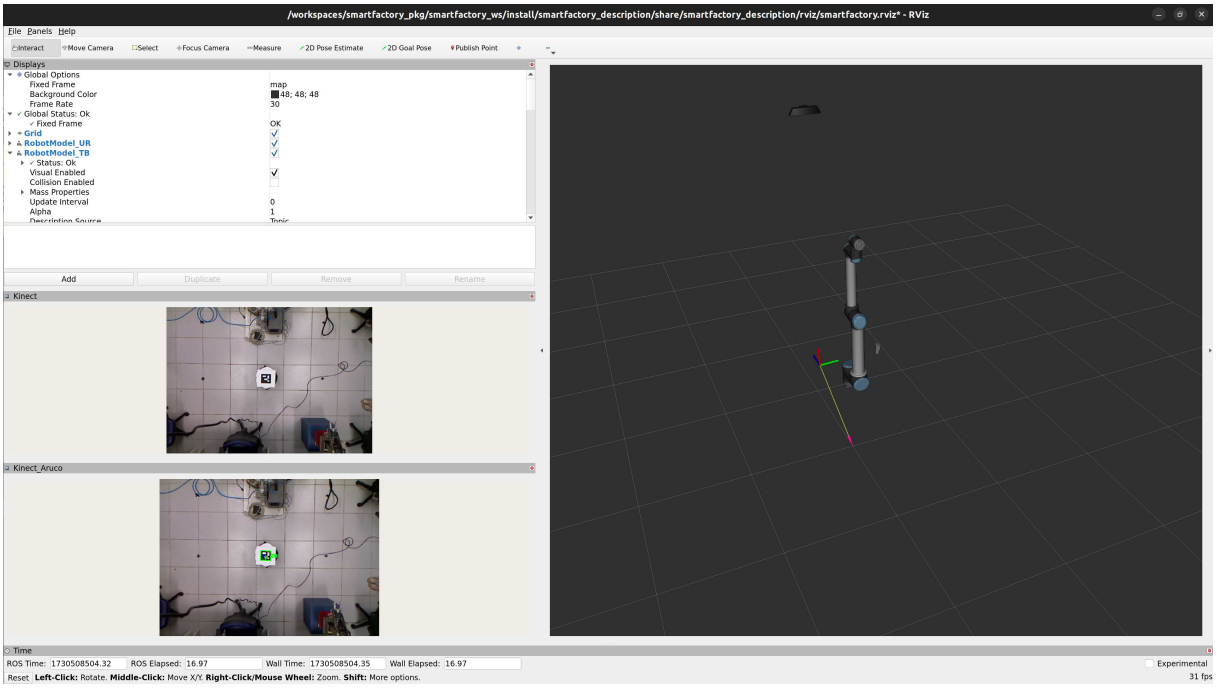
Fonte: Autoria própria.

Kinect. Conforme ilustrado na Figura 44, o *RViz* é configurado para exibir as seguintes informações relevantes para o sistema de fábrica inteligente:

- **Modelo do UR10**: O modelo tridimensional do UR10 é exibido na cena, permitindo visualizar o estado e o posicionamento das suas juntas. Isso facilita a validação do movimento do robô em resposta às instruções de controle enviadas pela árvore de comportamento.
- **Modelo do TurtleBot2i**: A posição e orientação do TurtleBot2i são atualizadas em tempo real na cena do *RViz*, possibilitando o monitoramento do transporte de objetos até a área de trabalho do UR10.
- **Dados da Câmera Kinect**: As imagens capturadas pela câmera Kinect são transmitidas para o *RViz*, onde são exibidas em uma janela dedicada. Essas imagens incluem tanto a visualização RGB quanto a detecção dos marcadores ArUco, o que é fundamental para a identificação da pose dos objetos manipulados.
- **Referência Global e Eixos de Referência**: A cena também inclui a grade e os eixos de referência, que ajudam a verificar a posição relativa dos dispositivos em relação ao sistema de coordenadas da fábrica inteligente.

A configuração dos elementos no *RViz* foi realizada a partir dos parâmetros especificados no pacote `smartfactory_bringup`, que organiza e padroniza a visualização dos dispositivos. Para cada dispositivo, um modelo tridimensional correspondente foi inserido na cena, junto com os tópicos relevantes para receber dados de posição e orientação. Isso proporciona uma visualização completa e integrada do ambiente de fábrica, permitindo monitorar as operações e validar a interação entre os dispositivos em tempo real.

Figura 44 – Visualização da cena do *RViz* com os dispositivos integrados



Fonte: Autoria própria.

Com essa estrutura, o *RViz* desempenha um papel importante na verificação da comunicação entre os dispositivos. Ele permite observar, por exemplo, o momento em que o TurtleBot2i entra na área de trabalho do UR10, ativando a detecção de pose pelo Kinect e acionando o supervisor (árvore de comportamento) para coordenar a ação do manipulador. Essa visualização em tempo real facilita o desenvolvimento, a depuração e a validação das operações da fábrica inteligente, garantindo que os dispositivos atuem de forma coordenada e precisa.

Essa validação inicial demonstrou que o sistema é capaz de coordenar ações entre dispositivos heterogêneos de forma autônoma, e a árvore de comportamento provou ser uma estrutura eficaz para gerenciar o fluxo de controle em uma fábrica inteligente. Todos os pacotes e códigos desenvolvidos ao longo deste trabalho foram disponibilizados de forma aberta no repositório <https://github.com/SmartFactoryLab-UFCG/smartfactory_pkg>. Essa disponibilização visa facilitar a replicação dos experimentos, bem como promover a continuidade e evolução das implementações por outros pesquisadores e desenvolvedores interessados em sistemas de automação industrial e fábricas inteligentes.

6 Conclusão

Neste trabalho foi apresentada uma abordagem para controle orientado a eventos em uma fábrica inteligente, utilizando árvores de comportamento implementadas no ROS (Robot Operating System). A estrutura desenvolvida visa promover uma automação modular, flexível e escalável, integrando diferentes dispositivos como o manipulador robótico UR10, o TurtleBot2i e as câmeras Kinect e Basler. Através da implementação de uma árvore de comportamento, foi possível coordenar as tarefas de detecção, movimentação e manipulação de objetos, proporcionando uma resposta eficiente e organizada em tempo real às demandas do ambiente industrial.

A arquitetura baseada em eventos se mostrou adequada para o controle de dispositivos em ambientes dinâmicos, onde a sincronização e a comunicação entre sensores e atuadores são essenciais. A árvore de comportamento permitiu modularizar as operações em nós específicos, garantindo que cada dispositivo desempenhasse sua função de forma independente, mas coordenada. Esse modelo facilitou o desenvolvimento e a implementação do sistema, possibilitando a adição ou modificação de comportamentos sem comprometer a estabilidade do sistema como um todo.

No entanto, algumas limitações foram identificadas durante o processo de implementação e validação. Uma delas foi a ausência de sensores que permitissem a validação completa do funcionamento da ventosa, sendo possível apenas verificar o envio de ângulos de junta para o UR10. Isso restringiu a avaliação do sistema em termos de confiabilidade de captura dos objetos. Além disso, a configuração atual ainda depende de intervenções manuais para determinar alguns estados iniciais e condições operacionais, o que representa uma oportunidade de aprimoramento em futuros trabalhos.

Como perspectivas de continuidade, são propostas algumas extensões para o sistema desenvolvido. Primeiramente, a implementação do TurtleBot2i com navegação autônoma utilizando o pacote de navigation do ROS permitiria ao robô móvel navegar de forma independente até a área de trabalho do UR10, aumentando a automação do processo. Adicionalmente, a criação de cenas com setorização, onde o UR10 possa direcionar os objetos para diferentes setores definidos por identificadores, como QRCodes, ampliaria a capacidade do sistema de lidar com diferentes fluxos de trabalho e tomaria decisões baseadas nas características dos objetos, para inserir a câmera Basler na cena.

Outro desenvolvimento interessante seria a criação de um gêmeo digital do sistema, que permitiria simulações mais precisas e testes de estratégias de controle antes da aplicação no ambiente real. Além disso, a implementação de um controle cinemático direto no UR10, ao invés de apenas enviar ângulos de junta, contribuiria para um controle

mais eficiente e preciso, especialmente em tarefas que exijam movimentos complexos e uma maior precisão no posicionamento do robô.

Assim, o trabalho realizado serve como uma base promissora para o desenvolvimento de sistemas de automação industrial flexíveis e inteligentes, destacando-se pela aplicação de árvores de comportamento para o controle orientado a eventos. A continuidade e aprimoramento desse sistema podem trazer contribuições significativas para o avanço da automação e da integração de dispositivos em ambientes de manufatura inteligente.

Referências

ALVARES, A. et al. Robotic additive manufacturing by laser metal deposition in the context of industry 4. 0: Manufatura aditiva robotizada por deposição de metal a laser no contexto da indústria 4. 0. *Concilium*, v. 23, n. 23, p. 79–103, 2023.

Basler AG. *Basler ace 2 A2A1920-51gcPRO Documentation*. [S.l.], 2023. Accessed: 2023-10-31. Disponível em: <https://docs.baslerweb.com/a2a1920-51gcpro#installation>.

BENOTSMANE, R.; KOVÁCS, G.; DUDÁS, L. Economic, social impacts and operation of smart factories in industry 4.0 focusing on simulation and artificial intelligence of collaborating robots. *Social Sciences*, MDPI, v. 8, n. 5, p. 143, 2019.

CAGGIANO, A.; TETI, R. Digital factory technologies for robotic automation and enhanced manufacturing cell design. *Cogent Engineering*, Taylor & Francis, v. 5, n. 1, p. 1426676, 2018.

CASSANDRAS, C. G.; LAFORTUNE, S. *Introduction to discrete event systems*. [S.l.]: Springer, 2008.

COLLEDANCHISE, M.; ÖGREN, P. *Behavior trees in robotics and AI: An introduction*. [S.l.]: CRC Press, 2018.

CORKE, P. *Robotics, Vision and Control: fundamental algorithms in Python*. [S.l.]: Springer Nature, 2023. v. 146.

CSALÓDI, R. et al. Industry 4.0-driven development of optimization algorithms: A systematic overview. *Complexity*, Wiley Online Library, v. 2021, n. 1, p. 6621235, 2021.

DELOITTE. *Digital twin applications bridging the physical and digital*. 2020. Disponível em: <https://www2.deloitte.com/us/en/insights/focus/tech-trends/2020/digital-twin-applications-bridging-the-physical-and-digital.html>. Acesso em: 20 Abr. 2024.

EL-IAITHY, R. A.; HUANG, J.; YEH, M. Study on the use of microsoft kinect for robotics applications. *Proceedings of the IEEE Conference*, IEEE, p. 1280–1288, 2012.

GORI, R. S. L.; GORI, D. D. L. Smart factory e a indústria 4.0: uma revisão sistemática de literatura. *Revista Sítio Novo*, v. 6, n. 2, p. 141–155, 2022.

GUGLIERMO, S. et al. Evaluating behavior trees. *Robotics and Autonomous Systems*, v. 178, p. 104714, 2024.

HAWKINS, K. P. Analytic inverse kinematics for the universal robots ur-5/ur-10 arms. Georgia Institute of Technology, 2013.

HEPPNER, G. et al. Distributed behavior trees for heterogeneous robot teams. In: IEEE. *2023 IEEE 19th International Conference on Automation Science and Engineering (CASE)*. [S.l.], 2023. p. 1–8.

IBM. *What is Industry 4.0?* 2020. Disponível em: <https://www.ibm.com/topics/industry-4-0>. Acesso em: 22 Abr. 2024.

NASCIMENTO, F. H. et al. Synchronizing a collaborative arm's digital twin in real-time. In: IEEE. *2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE)*. [S.l.], 2023. p. 230–235.

ONU, P.; PRADHAN, A.; MBOHWA, C. Industry 4.0 and beyond: Enabling digital transformation and sustainable growth in industry x.0. *IEEE International Conference on Industrial Engineering and Engineering Management*, 2023.

Open Robotics. *ROS 2 Documentation: Humble Hawksbill*. [S.l.], 2024. Accessed: 2024-10-30. Disponível em: <https://docs.ros.org/en/humble/index.html>.

Quasi AI. *Behavior Trees for Autonomous Mobile Robots (AMRs)*. 2023. Acesso em: 2 nov. 2024. Disponível em: <https://www.quasi.ai/behavior-trees-for-autonomous-mobile-robots-amrs/>.

RAHMAN, A.; ARTHUR, S. D. Smart factory for future industry development.

ROBOTICS, O. *Beginner: GUI*. 2014. Disponível em: <https://classic.gazebosim.org/tutorials?cat=guided_b&tut=guided_b2>. Acesso em: 29 Abr. 2024.

ROBOTICS, O. *About Gazebo*. 2024. Disponível em: <https://gazebosim.org/about>. Acesso em: 29 Abr. 2024.

ROBOTICS, O. *Gazebo Simulator*. 2024. Disponível em: <https://gazebosim.org/home>. Acesso em: 29 Abr. 2024.

SEARS-COLLINS, A. *What is the Difference Between RViz and Gazebo?* 2020. Disponível em: <https://automaticaddison.com/what-is-the-difference-between-rviz-and-gazebo/>. Acesso em: 07 Mai. 2024.

TurtleBot. *TurtleBot2i - Official Website*. 2023. Accessed: 2023-10-31. Disponível em: <https://turtlebot2i.weebly.com/>.

TurtleBot. *TurtleBot2i GitHub Repository*. 2023. Accessed: 2023-10-31. Disponível em: <https://github.com/turtlebot/turtlebot2i>.

TurtleBot. *TurtleBot2i Wiki - ROS Documentation*. 2023. Accessed: 2023-10-31. Disponível em: <https://wiki.ros.org/turtlebot2i>.

Universal Robots. *DH Parameters for Calculations of Kinematics and Dynamics*. 2023. Accessed: 2024-11-01. Disponível em: <https://www.universal-robots.com/articles/ur/application-installation/dh-parameters-for-calculations-of-kinematics-and-dynamics/>.

Universal Robots. *UR Robots - CB3 Series*. 2023. Accessed: 2023-10-31. Disponível em: <https://www.universal-robots.com/products/cb3/>.

Universal Robots. *UR10 Technical Specifications*. 2023. Accessed: 2023-10-31. Disponível em: <https://www.universal-robots.com/media/1828035/ur10_tech_spec_web_en.pdf>.

Universal Robots. *UR10 Working Area Diagram*. [S.l.], 2023. Accessed: 2023-10-31. Disponível em: <https://www.siemens-pro.ru/docs/ur/ur10/100400.PDF>.

Universal Robots. *User Manual: CB3 Series Robots*. [S.l.], 2023. Accessed: 2023-10-31. Disponível em: <https://s3-eu-west-1.amazonaws.com/ur-support-site/18369/manual_en_1.3.pdf>.

Wikimedia Commons. *Xbox 360 Kinect Standalone*. 2024. <https://commons.wikimedia.org/wiki/File:Xbox-360-Kinect-Standalone.png>. Accessed: 2024-10-31.

WIKIPEDIA. Gazebo simulator. 2024. Disponível em: <https://en.wikipedia.org/wiki/Gazebo_simulator>.

YANG, T. et al. Kinematics analysis and trajectory planning of ur5 robot. In: IEEE. *2023 2nd International Conference on Automation, Robotics and Computer Engineering (ICARCE)*. [S.l.], 2023. p. 1–4.

ŽEMLA, F. et al. Smart platform for monitoring and control of discrete event system in industry 4.0 concept. *Applied Sciences*, MDPI, v. 13, n. 19, p. 10697, 2023.

ÖGREN, P.; SPRAGUE, C. I. Behavior trees in robot control systems. *Annual Review of Control, Robotics, and Autonomous Systems*, v. 5, p. 1–25, 2022.