Samara Lima Cardoso

Desenvolvimento de um Gêmeo Digital para um Manipulador Robótico Utilizando Padrões Abertos de Automação

Campina Grande, PB

2024

Samara Lima Cardoso

Desenvolvimento de um Gêmeo Digital para um Manipulador Robótico Utilizando Padrões Abertos de Automação

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Universidade Federal de Campina Grande – UFCG

Centro de Engenharia Elétrica e Informática

Departamento de Engenharia Elétrica

Orientador: George Acioli Júnior, D.Sc

Campina Grande, PB

2024

Samara Lima Cardoso

Desenvolvimento de um Gêmeo Digital para um Manipulador Robótico Utilizando Padrões Abertos de Automação

Trabalho de Conclusão de Curso (TCC) submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Trabalho aprovado. Campina Grande, PB, _____/_____/_____.

George Acioli Júnior, D.Sc
Orientador

Pericles Rezende Barros, Ph.D.
Convidado

Campina Grande, PB
2024

À minha família, com todo o meu amor e gratidão.

Agradecimentos

Primeiramente, gostaria de expressar minha profunda gratidão aos meus pais, Marcos e Maria Cláudia, e à minha avó, Maria Izabel, pelo apoio incondicional, pelos sacrifícios, pelo suporte constante e por sempre acreditarem em mim. Agradeço também aos meus irmãos, Sabrina e Marcus Alexandre, por estarem ao meu lado e me encorajarem nos momentos difíceis. Vocês são minha fonte de inspiração e motivação. Esta conquista não é apenas minha, mas também de vocês.

Um agradecimento especial à Leticia Paz, pelo amor, paciência e encorajamento ao longo desta jornada. Sou grata por ter você ao meu lado e por tornar esta etapa da vida muito mais leve e superável.

Gostaria de expressar minha gratidão à minha dupla de universidade, Julia Ramalho, por toda a parceria e apoio durante a graduação. Desde o segundo período, sua colaboração nos trabalhos e seu suporte foram fundamentais para enfrentar os desafios que o curso e a vida apresentaram. Não consigo imaginar como teria sido sem a sua companhia.

Aos meus amigos da graduação, Gabriel Henrique, Clarice, Marcos Paulo, Maria Gabrilela, Sarah, Biancca e Airon, agradeço profundamente pelo apoio nos grupos de estudos, pelos trabalhos em grupo, pela ajuda nas disciplinas e pela companhia constante. Sem vocês, muitos desafios teriam sido muito mais difíceis de enfrentar. Agradeço por cada momento de colaboração. Agradeço aos meus colegas de universidade pelas discussões, ideias e apoio durante nossa trajetória acadêmica.

Agradeço aos colegas do LIEC pela colaboração e amizade, que tornaram este período de pesquisa mais produtivo e agradável. Em especial, gostaria de expressar minha gratidão aos doutorandos Egydio Tadeu e Matheus pelos valiosos conselhos e por compartilharem seus conhecimentos comigo.

Ao meu orientador, George Acioli, agradeço pela orientação e sabedoria, que foram importantes para a realização deste trabalho. Agradeço também ao professor Péricles Barros, pelos os ensinamentos, atenção, compreensão e apoio me dado durante minha trajetória no LIEC.

"Não tenho medo de errar, só medo de desistir;
Mas tenho vinte e poucos anos e não vou parar aqui"
Lagum

Lista de ilustrações

Figura 1 – Esquemático conceitual do GD. 20
Figura 2 – Fluxo de dados em um DM, DS e DT. 20
Figura 3 – AX-12A Smart Robotic Arm. 22
Figura 4 – Construção do Motor AX-12A. 22
Figura 5 – Dispositivo USB2DYNAMIXEL. 23
Figura 6 – Conexão USB2DYNAMIXEL com os servosmotores. 24
Figura 7 – Conexão do Motor AX-12A. 24
Figura 8 – Conexão em de múltiplos atuadores. 25
Figura 9 – Tela do Unity para a configuração do *GameObject* Pai. 25
Figura 10 – Tela do Unity para a configuração do *GameObject* Filho. 26
Figura 11 – Tela do Unity para a configuração das propriedades da Âncora de Junta. 26
Figura 12 – Aplicação do OPC UA dentro da pirâmide de automação. 29
Figura 13 – Sistema com funcionalidade distribuída. 30
Figura 14 – Dados e fluxos de eventos do Bloco de Função. 31
Figura 15 – Características do Bloco de Função 32
Figura 16 – Modelo de execução para Bloco de Função. 33
Figura 17 – Exemplo de aplicação IEC 61499. 34
Figura 18 – Exemplo de aplicação IEC 61499 distribuída. 35
Figura 19 – Modelo de visão 4+1. 36
Figura 20 – Diagrama de contexto do sistema. 38
Figura 21 – Diagrama de sequência do sistema. 39
Figura 22 – Diagrama de atividade do sistema. 40
Figura 23 – Diagrama de componente do sistema. 41
Figura 24 – Diagrama de implementação. 42
Figura 25 – Caso de uso do sistema. 43
Figura 26 – Interface gráfica desenvolvida no AppDesigner. 45
Figura 27 – *Callbacks* da interface. 45
Figura 28 – Função *JuntaBaseEditFieldValueChanged*. 46
Figura 29 – Função *ConectarButtonPushed*. 46
Figura 30 – Função *DesconectarButtonPushed*. 46
Figura 31 – Função *TimerCallback()*. 47
Figura 32 – Programa para o controle das juntas. 48
Figura 33 – Função de acionamento do servomotor da base. 49
Figura 34 – Função de acionamento dos servomotores do ombro. 49
Figura 35 – *Links* do braço robótico 51
Figura 36 – Hierarquia dos *GameObjetcs* do braço robótico 51

Figura 37 – Juntas do braço robótico. 52
Figura 38 – Configuração do componente *ArticulationBody base_link*. 53
Figura 39 – Configuração do componente *ArticulationBody ombro_link*. 53
Figura 40 – Braço robótico desenvolvido no Unity. 54
Figura 41 – Declaração das instâncias. 54
Figura 42 – Código para criação do método *OpcUa_Client_Configuration*. 55
Figura 43 – Código para criação da sessão *OpcUa_Create_Session*. 55
Figura 44 – Código para criação do método *opcua_Inicia*. 56
Figura 45 – Componentes de interação do Unity. 56
Figura 46 – Interface gráfica no Unity. 57
Figura 47 – Obtenção dos *TMP_InputField* para cada articulação do braço robótico. 57
Figura 48 – código para manipulação dos valores obtidos a partir do componente *TMP_InputField*. 58
Figura 49 – código *Controller* incluído no *GameObject* Pai. 58
Figura 50 – Trecho do código *Controller* que adiciona o componente *JointControll* nos *GameObjects* filhos. 59
Figura 51 – Método *Update()* do código *Controller*. 60
Figura 52 – Método *UpdateDirection()* do código *Controller*. 61
Figura 53 – Criação de *BaseObjectTypes*. 61
Figura 54 – Tela do UaModeler para criação do objeto base tipo *JuntaTipo*. 62
Figura 55 – Tela do UaModeler para criação do objeto base tipo *BracoRoboticoTipo*. 62
Figura 56 – Tela do UaModeler para criação do objeto *BracoRobotico1*. 63
Figura 57 – Diagrama do modelo de informação do objeto *BracoRobotico1*. 63
Figura 58 – Tela do UaModeler para exportação do modelo. 64
Figura 59 – Comando do UA-ModelCompiler. 64
Figura 60 – Arquivos gerados pelo UA-ModelCompiler. 65
Figura 61 – Criação e preparação da aplicação para o desenvolvimento do servidor. 65
Figura 62 – Interface gráfica do servidor. 66
Figura 63 – Aplicação IEC 61499 no 4diac para leitura e escrita em um servidor OPC UA. 67
Figura 64 – Interface do bloco de função *ReadWrite*. 68
Figura 65 – Rede dos blocos de funções para a contrução do bloco de função *ReadWrite*. 68
Figura 66 – Aplicação IEC 61499 no 4diac para leitura e para escrita dos ângulos das juntas. 69
Figura 67 – Manipulação do braço virtual no Unity, com os valores de rotação inseridos pelo UaExpert (1) e lidos pelo script Controller (2). 71
Figura 68 – Manipulação do braço virtual no Unity, com os valores de rotação inseridos pela própria interface do Unity (1) e lidos pelo cliente UaExpert (2). 71

Figura 69 – Manipulação dos ângulos das articulações na aplicação do MATLAB, com os valores de rotação inseridos pela própria interface do AppDesigner (1) e lidos pelo cliente UaExpert (2). 72
Figura 70 – Manipulação dos ângulos das articulações na aplicação IEC 61499. Com os valores de rotação inseridos no bloco de função (1), escritos no servidor a partir do acionamento de um evento (2) e lidos pelo cliente UaExpert (3). 73
Figura 71 – Manipulação dos ângulos das articulações na aplicação IEC 61499. Com os valores de rotação inseridos pelo UaExpert (1) e lidos pela aplicação IEC 61499 (2). 74

Lista de tabelas

Tabela 1 – Propriedades da Âncora de Junta 27

Lista de abreviaturas e siglas

CLP	*Controlador Lógico Programável*
DT	*Digital Twin*
DM	*Digital Model*
DS	*Digital Shadow*
ECC	*Execution Control Chart*
FBs	*Function Blocks*
IEC	*International Electrotechnical Commission*
IHM	*Interface Humano-Máquina*
IoT	*Internet of Things*
JSON	*JavaScript Object Notation*
LIEC	*Laboratório de Instrumentação Eletrônica e Controle*
OPC UA	*Open Platform Communications Unified Architecture*
SDK	*Software Development Kit*
XML	*Extensible Markup Language*

Resumo

Nas últimas décadas, os avanços industriais na manufatura têm sido impulsionados pela integração de robôs, principalmente os manipuladores robóticos, que se destacam pela capacidade de realizar tarefas repetitivas com muita precisão e velocidade. Entretanto, na Indústria 4.0, há uma busca por sistemas mais avançados e integrados, e os Gêmeos Digitais surgiram como ferramenta para alcançar esse objetivo. A implementação de Gêmeos Digitais é importante para melhorar a competitividade e eficiência das indústrias, simplificando desde o desenvolvimento até o descarte dos ativos industriais. No entanto, ainda há tecnologias para melhorar a eficiência dos Gêmeos Digitais, como a sua aplicação em sistemas de controle distribuídos, que ainda é limitada e amplamente inexplorada. Essa integração de Gêmeos Digitais em sistemas de controle distribuídos possibilita a criação de sistemas de controle inteligentes distribuídos, automatizados e capazes de autogerenciamento. Neste trabalho tem-se como objetivo construir um gêmeo digital para o braço robótico disponível no Laboratório de Instrumentação Eletrônica e Controle (LIEC), implementando um canal de cominucação utilizando OPC UA e o controle distribuído baseado em bloco de funções definidos na IEC 61499.

Palavras chave: Indústria 4.0, Gêmeos Digitais, OPC UA, Controle Distribuído.

Abstract

In recent decades, industrial advancements in manufacturing have been driven by the integration of robots, particularly robotic manipulators, which excel at performing repetitive tasks with high precision and speed. However, with Industry 4.0, there arose a need for more advanced and integrated systems, such as digital twins. The implementation of digital twins is important for improving the competitiveness and efficiency of industries, simplifying everything from development to the disposal of industrial assets. Nevertheless, there are still technologies to enhance the efficiency of digital twins, such as their application in distributed control systems, which is still limited and largely unexplored. This integration of digital twins in distributed control systems enables the creation of intelligent, automated, and self-managing distributed control systems. This work aims to construct a digital twin for the robotic arm available at the Laboratory of Electronic Instrumentation and Control (LIEC), implementing OPC UA in its communication channel and utilizing IEC 61499 to enable the implementation of distributed control.

Keywords: Industry 4.0, Digital Twins, OPC UA, Distributed Control.

Sumário

1 INTRODUÇÃO 15
1.1 Justificativa 15
1.2 Objetivo geral 17
1.3 Objetivos específicos 17
1.4 Metodologia 17
1.5 Organização do Documento 18

2 REVISÃO BIBLIOGRÁFICA E TECNOLÓGICA 19
2.1 Gêmeos Digitais 19
2.2 Braço Robótico AX-12A 21
2.3 Unity 24
2.4 MATLAB 27
2.5 Visual Studio 28
2.6 OPC UA 28
2.6.1 UaModeler 29
2.6.2 UaExpert 29
2.7 Controle Distribuido com IEC 61499 30
2.8 *Eclipse* 4diac 34
2.8.1 4diac IDE 34
2.8.2 4diac FORTE 35

3 ARQUITETURA DO SISTEMA 36
3.1 Modelo de Visão 4+1 36
3.2 Arquitetura Proposta 38
3.2.1 Visão Lógica 39
3.2.2 Visão de Processo 40
3.2.3 Visão de Desenvolvimento 41
3.2.4 Visão Física 41
3.2.5 Cenários 42

4 DESENVOLVIMENTO 44
4.1 *Software* do Braço Robótico 44
4.1.1 Interface de Comunicação e Manipulação de Dados 44
4.1.2 Programa de Controle das Juntas 47
4.2 Modelo Digital 50
4.2.1 Construção do Objeto Digital 50

4.2.2 Cliente OPC UA . . . 54
4.2.3 Interface Gráfica . . . 56
4.2.4 Código de controle do braço robótico . . . 57
4.2.5 código de controle das articulações . . . 59
4.3 Canal de Comunicação . . . 59
4.3.1 Modelo de Informação . . . 59
4.3.2 Servidor OPC UA . . . 63
4.4 Controle com IEC 61499 . . . 66

5 RESULTADOS . . . 70
5.1 Aplicação do Modelo Digital . . . 70
5.2 Aplicação de comunicação com o braço robótico . . . 72
5.3 Aplicação de controle distribuído . . . 72
5.4 Integração das aplicações . . . 73

6 CONCLUSÃO . . . 75
6.1 Trabalhos Futuros . . . 75

REFERÊNCIAS . . . 76

1 Introdução

1.1 Justificativa

Nas últimas décadas, o uso de robôs na indústria de manufatura vem proporcionando um aumento significativo na produtividade e na qualidade dos produtos. Os braços robóticos são uma classe de robôs aplicados na indústria de manufatura para manuseio de cargas e execução precisa de tarefas (MORAN, 2007). O principal papel deles é executar um conjunto de tarefas repetitivas com alta velocidade e precisão ao longo do tempo (WANG et al., 2018).

Com o crescente desenvolvimento digital dos processos de manufatura, as empresas têm cada vez mais buscado agregar valor com o uso de soluções digitais. Nesse cenário, os Gêmeos Digitais vêm ganhando bastante popularidade, devido à sua característica principal de prover um equivalente digital de um sistema físico, o que facilita diversas operações, como detecção de falhas ou previsões mais precisas de resultados, implicando em uma otimização da produção. Nesse contexto, na indústria, o Gêmeo Digital representa um conceito fundamental nessa revolução industrial (FEI et al., 2022).

Um Gêmeo Digital (do inglês *Digital Twin* - DT) , conforme proposto originalmente por Grieves (2015), é uma representação virtual de um ativo físico no espaço digital com o propósito de caracterizar de perto as operações do processo ou sistema físico original. A representação precisa do sistema físico no ciberespaço é possibilitada pela sincronização contínua de dados e pela troca de informações entre o Gêmeo Digital e a contraparte física.

Contar com uma representação digital da entidade física de interesse traz uma série de vantagens significativas que permeiam todo o seu ciclo de vida. Isso engloba desde a fase de desenvolvimento, passando pela fase de fabricação, até a fase de serviço e a fase de descarte (ADAM et al., 2022). O Gêmeo Digital é uma ferramenta utilizada por empresas para aumentar sua competitividade, produtividade e eficiência (KRITZINGER et al., 2018), e devido à sua característica de conectar sistemas físicos e virtuais em tempo real, pode ser utilizado para prover informações mais realistas em cenários imprevisíveis (PARROT; WARSHAW, 2017).

O Gêmeo Digital possui dois elementos fundamentais para o seu desenvolvimento (STARK; KIND; NEUMEYER, 2017). O primeiro é uma representação virtual do produto que se baseia nas ferramentas e metodologias de *software* que permitem projetar, simular e otimizar produtos e seus sistemas de produção. O segundo é a coleção de dados operacionais gerados e coletados em tempo real por sensores e outras tecnologias da Indústria 4.0 (CHRYSSOLOURIS et al., 2009).

No quesito de troca de dados em tempo real, o padrão OPC UA (do inglês *Open Platform Communications Unified Architecture*), um protocolo de comunicação industrial aberto, está crescendo rapidamente. O OPC UA oferece mecanismos de segurança embutidos que protegem a autenticidade, integridade e confidencialidade dos dados, o que é essencial para garantir a confiabilidade dos Gêmeos Digitais (KOHNHäUSER et al., 2021). A integração de modelos de informação do OPC UA em Gêmeos Digitais permite a monitorização de eventos e a disponibilização de informações importantes com base em critérios específicos, contribuindo para uma representação precisa e em tempo real dos sistemas físicos (VACLAVOVA et al., 2022).

Apesar dos grandes avanços dos Gêmeos Digitais na indústria, ainda há áreas que necessitam de desenvolvimento na tecnologia dos Gêmeos Digitais, como sua aplicação em sistemas de controle distribuído (LESAGE; BRENNAN, 2022). Desenvolver Gêmeos Digitais que possam ser implementados em sistemas de controle distribuído torna esses sistemas mais inteligentes, totalmente automatizados e autogerenciáveis, transformando o Gêmeo Digital em um produto valioso.

Nesse contexto, o padrão definido na norma IEC 61499 é aplicado no desenvolvimento dos sistemas de controle, devido ao seu suporte à computação distribuída (LYU et al., 2023). O padrão IEC 61499 apresenta uma arquitetura de automação de referência para componentes industriais, ampliando a arquitetura tradicional de programação de CLP (Controlador Lógico Programável) da IEC 61131-3 como suporte à computação distribuída e maior compatibilidade com tecnologias da informação (VYATKIN, 2009).

Este trabalho tem como objetivo construir um Gêmeo Digital de um braço robótico, implementando OPC UA em seu canal de comunicação e utilizando a IEC 61499 para permitir a implementação de controle distribuído.

1.2 Objetivo geral

Neste trabalho, tem-se como objetivo o desenvolvimento de um Gêmeo Digital, utilizando padrões abertos de automação, para um braço robótico inteligente modelo AX-12A, fabricado pela empresa CrustCrawler (NORRIS, 2008), que está disponível no Laboratório de Instrumentação Eletrônica e Controle (LIEC).

1.3 Objetivos específicos

A fim de alcançar o objetivo geral, pontua-se os seguintes objetivos específicos:

- Treinar o aluno para uso das ferramentas de desenvolvimento e simulação disponíveis no laboratório;
- Estudar as características construtivas e de operação do braço robótico AX-12A;
- Implementar o braço robótico AX-12A;
- Estudar os conceitos básicos acerca de Gêmeos Digitais e sua implementação;
- Desenvolver o modelo virtual do braço robótico;
- Estudar e compreender o protocolo de comunicação OPC UA;
- Desenvolver o canal de comunição em OPC UA;
- Estudar e compreender o *software* Eclipse 4diac e FORTE;
- Desenvolver um cliente utilizando o eclipse 4diac;
- Realizar testes para validação do Gêmeo Digital.

1.4 Metodologia

Nesta seção, serão descritos os métodos utilizados para atingir os objetivos propostos. Primeiramente, foi realizada uma revisão bibliográfica, com buscas em bases científicas usando os termos "Gêmeos Digitais", "OPC UA", "Unity"e "controle distribuído". Após a busca, leitura e compreensão do problema, foram selecionados artigos para serem utilizados como referência.

Posteriormente, foi realizado o estudo do funcionamento do braço robótico, utilizando documentos do fabricante (NORRIS, 2008), e em seguida, o mesmo foi colocado em operação. Para o software de simulação, foi escolhido o Unity, devido à sua versatilidade em sistemas operacionais e aplicabilidade em robótica (AUDONNET; HAMILTON;

ARAGON-CAMARASA, 2022). Ainda com base nas pesquisas, foi escolhido o protocolo OPC UA para a construção do canal de comunicação.

Inicialmente foi realizado o estudo aprofundado do funcionamento da tecnologia. E após compreender completamente o conceito por trás dessa tecnologia, iniciou à sua aplicação prática.

O desenvolvimento do projeto foi realizado de maneira sequencial, onde após a operacionalização do ativo físico, a construção da simulação e o desenvolvimento do canal de comunicação, os elementos foram então integrados entre si. Por fim, novas funcionalidades foram incorporadas com base nos comentários recebidos e nas recentes descobertas.

1.5 Organização do Documento

Este trabalho está estruturado em 6 capítulos que têm como objetivo detalhar todos os passos realizados para o desenvolvimento do projeto e apresentar os resultados obtidos. O **Capítulo 1** apresenta uma visão geral do trabalho, bem como sua justificativa, os objetivos e os métodos aplicados para alcançar os objetivos. No **Capítulo 2**, é explicada as tecnologias que foram utilizadas.

No **Capítulo 3** é apresentada a arquitetura do sistema. No **Capítulo 4**, é apresentado o desenvolvimento do projeto. No **Capítulo 5** é apresentado os resultados obtidos e as suas validações. Por fim, o **Capítulo 6** resume brevemente o que foi alcançado no trabalho.

2 Revisão Bibliográfica e Tecnológica

Neste capítulo são apresentadas as principais ferramentas e conceitos necessários para o desenvolvimento do projeto. Inicialmente, discutiremos os Gêmeos Digitais, uma tecnologia onde réplicas virtuais de sistemas físicos são criadas, conectadas a eles e se comportam de maneira similar. Em seguida, é apresentado o Braço Robótico AX-12A, o ativo físico que será replicado no mundo digital. Para a criação e simulação dos modelos 3D necessários, é utilizado o Unity, uma plataforma de desenvolvimento de jogos que também é usada na indústria para crialçao de Gêmeos Digitais.

Outro tópico abordado é o MATLAB, amplamente utilizado para cálculos matemáticos avançados, análise de dados, controle de sistemas e que no trablaho foi usado para controlar as juntas do ativo físico. Também é explorado o Visual Studio, que é o principal ambiente de desenvolvimento para codificação e desenvolvimento de software usado no trabalho.

Em seguida, é apresentado o padrão de comunicação para automação industrial, o OPC UA, que permite a integração e a interoperabilidade entre diferente sistemas e dispositivos e ele é usado na construção do canal de comunicação do trabalho. Para criar o modelo de informação do sistemas e simular a comunicação OPC UA, é utilizado o UaModeler e o UaExpert.

Além disso, é discutido o Controle Distribuído com IEC 61499, um padrão internacional para a automação industrial, que permite a distribuição de funções de controle em redes industriais. Nesse sentido, é utilizado o Eclipse 4diac como framework de desenvolvimento para aplicações baseadas em IEC 61499, aproveitando suas capacidades de modelagem e execução de sistemas distribuídos.

Nas seções desse capítulo serão apresentados em detalhes esses tópicos, fornecendo uma base para a compreensão e implementação do trabalho.

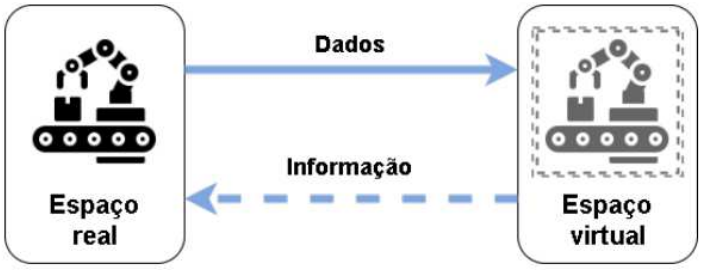
2.1 Gêmeos Digitais

O conceito de Gêmeo Digital foi inicialmente apresentado por Grieves durante uma de suas aulas no curso de Gerenciamento do Ciclo de Vida do Produto, em 2003. Embora o termo Gêmeo Digital ainda não estivesse associado a essa ideia na época, Grieves definiu-o como uma representação virtual do produto final, permitindo uma compreensão mais clara das disparidades entre o que foi planejado e o que foi efetivamente produzido (GRIEVES, 2015).

O modelo proposto para este conceito consistia principalmente de três partes: o

produto físico no espaço real, o produto virtual no espaço digital e os fluxos de dados e informações que conectam ambos os sistemas, conforme ilustrado no esquema da Figura 1.

Figura 1 – Esquemático conceitual do GD.

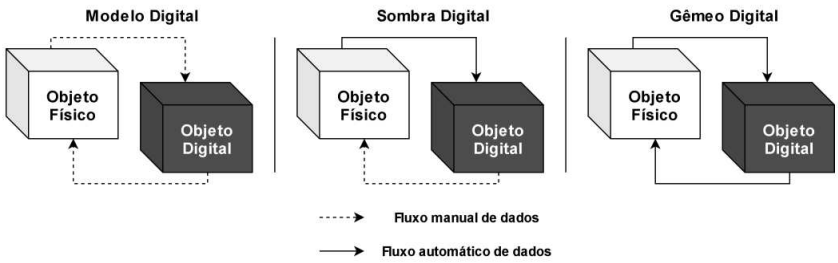


Fonte: Grieves e Vickers (2017).

Inicialmente focado no gerenciamento de produtos de manufatura, esse conceito foi posteriormente expandido para outros contextos, e diversas definições foram propostas ao longo dos anos seguintes, com o objetivo de ampliar o escopo do conceito quanto aplicá-lo de forma mais específica a diferentes contextos.

Desde a introdução do conceito de Gêmeos Digitais, várias implementações têm sido desenvolvidas ao longo dos anos, com base em diferentes definições propostas. No entanto, alguns autores observam que a existência de múltiplas soluções e conceitos de Gêmeos Digitais em diversos setores reflete uma compreensão diversa e, por vezes, incompleta desse conceito. Essa percepção surgiu da constatação de que muitas dessas implementações não se distinguem o suficiente dos simuladores já disponíveis, a ponto de poderem ser denominadas de Gêmeos Digitais (KRITZINGER et al., 2018). Diante disso, foi sugerida a separação dessas implementações em três categorias distintas, com base no nível de integração, como representado na Figura 2.

Figura 2 – Fluxo de dados em um DM, DS e DT.



Fonte: Kritzinger et al. (2018).

Dentro dessa categorização, a primeira categoria, Modelo Digital (DM do inglês *Digital Model*), refere-se a uma representação digital onde não há trocar dados de forma automática com o ativo físico. Isso significa que uma mudança de estado no objeto físico não resulta em mudanças no objeto digital, e vice-versa. Na segunda categoria, Sombra

Digital (DS do inglês *Digital Shadow*), existe alteração automática do objeto digital a partir da mudança de estado do objeto físico, mas não o inverso. Por fim, a terceira categoria, Gêmeo Digital (DT), envolve uma representação digital que permite a alteração automática do objeto digital a partir da mudança de estado do objeto físico, assim como o inverso. Esse fluxo bidirecional de informações possibilita até mesmo que o objeto digital atue como controlador do objeto físico.Nesse contexto, é possível distinguir os Gêmeos Digitais das outras implementações, que se enquadram nas duas primeiras categorias propostas, com base na presença da funcionalidade de comunicação bidirecional em tempo real com o ativo físico.

Do ponto de vista de modelagem, o modelo de Gêmeo Digital pentadimensional proposto por Adam et al. (2022) pode ser considerado uma versão estendida do modelo tridimensional proposto por Grieves (2015), e é definido pela seguinte função:

$$\mathrm{DT} = F(\mathrm{PS},\ \mathrm{VS},\ \mathrm{P2V},\ \mathrm{V2P},\ \mathrm{OPT}), \tag{2.1}$$

onde o Gêmeo Digital (DT) abrange um sistema físico (PS do inglês *Physical System*), um sistema virtual (VS do inglês *Virtual System*), um mecanismo de atualização (P2V do inglês *Physical to Virtual*), um mecanismo de predição (V2P do inglês *Virtual to Physical*) e um mecanismo de otimização (OPT do inglês *Optimization*).

O sistema físico (PS) possui capacidades de detecção, coletando dados de diversas fontes por meio de várias técnicas de detecção e aquisição. O mecanismo de atualização (P2V) é utilizado para ajustar o estado do modelo digital com base nessas informações sensoriais. Em seguida, os modelos digitais atualizados (VS) são utilizados para prever o estado futuro do sistema físico, por meio do mecanismo de predição (V2P), permitindo que as decisões preditivas sejam aplicadas ao sistema físico por meio de atuadores (controle, manutenção, planejamento de caminho, etc.). Por fim, a dimensão de otimização (OPT), complementa as funcionalidades das outras quatro dimensões no Gêmeo Digital, aprimorando a coleta de dados, a modelagem, a estimativa de estado, a tomada de decisões, etc., tanto offline quanto online. Por meio da integração coesa dessas cinco dimensões (PS, VS, P2V, V2P e OPT) os Gêmeos Digitais atendam à necessidade de um reflexo em tempo real do ciclo de vida de um sistema físico, facilitando assim a tomada de decisões.

2.2 Braço Robótico AX-12A

O braço robótico inteligente AX-12A, apresentado na Figura 3, possui uma estrutura em alumínio galvanizado, possui 7 servomotores do tipo AX-12A de forma a proporcionar 4 graus de liberdade além da garra.

Quatro dos AX-12A estão emparelhados de forma a constituírem as articulações do punho e ombro. As demais articulações possuem apenas um único AX-12A. Na base existem

Figura 3 – AX-12A Smart Robotic Arm.

Fonte: Norris (2008).

quatro esferas de aço para a facilitar o trato com cargas mais pesadas e uma articulação ajustável manualmente. A garra possui um prolongamento de maneira a permitir a conexão de câmeras, sensores de pressão, toque ou outros sensores. Há outras garras opcionais disponíveis no mercado fabricadas pela CrustCrawler.

No que tange o comprimento do braço, possui 53,78 cm de extensão máxima e garra com abertura máxima de 6,35 cm. Para fixação de sua base utilizou-se uma estrutura feita em madeira garantindo assim uma boa estabilidade.

Desenvolvidos pela empresa ROBOTIS, os servomotores AX-12A fazem parte de uma família de motores inteligentes, os quais se diferenciam entre si pela capacidade de torque e de velocidade. Na Figura 4 são apresentadas imagens do servomotor fechado, do circuito de controle e da sua caixa de engrenagens.

Figura 4 – Construção do Motor AX-12A.

(a) Motor AX-12A.

(b) Circuito de Controle e motor DC.

(c) Caixa de Engrenagens.

Fonte: Robotis (2017).

O servomotor AX-12A é um atuador de alto desempenho, projetado para ser modular e encadeado em projetos robóticos com a finalidade de produzir movimentos mais

precisos e flexíveis. A sua construção consiste em um motor DC integrado, um redutor de engrenagens, um controlador interno e um circuito de drive do motor. Por ser programável e em rede, o status do atuador pode ser lido e monitorado por meio de um fluxo de pacotes de dados.

Apesar do seu tamanho compacto, o AX-12A tem resistência estrutural necessária para suportar grandes forças externas. Ademais, seu circuito interno consegue detectar e agir a condições internas, como mudanças na temperatura interna, tensão de alimentação e extrapolamento de posição.

O atuador tem um controle de precisão de 1024 níveis com faixa de posicionamento de 300°, e pode trabalhar com velocidades de comunicação de até 1 Mpbs. Inclui um conjunto de memórias, EEPROM e RAM, nas quais são armazenados os parâmetros de controle e as instruções de leitura. Modificando esse valores torna possível o controle do motor com pacotes de instrução.

Os pacotes de instrução podem ser tanto de leitura, *read*, como de escrita, *write*. O usuário pode verificar o status atual do dispositivo lendo dados específicos da tabela de controle com pacotes de instruções de leitura. Utilizando os pacotes de instruções de escrita o usuário pode controlar o dispositivo alterando dados específicos na tabela de controle. Para ler ou guardar os dados, o usuário deve informar um endereço específico no pacote de instruções de acordo com a tabela de controle daquela memória. O Endereço é um valor exclusivo ao acessar dados específicos na tabela de controle com pacotes de instruções.

A comunicação entre os motores e o computador é realizada pelo dispositivo USB2DYNAMIXEL, um conversor USB para RS485, apresentado na Figura 5, também produzido pela ROBOTIS. Nele é estabelecida uma conexão serial TTL do tipo Half-Duplex, em que o AX-12A é identificado por um ID único, e o dispositivo atua como um servidor que recebe comandos e retorna respostas ao controlador externo, que nesse caso é o computador.

Figura 5 – Dispositivo USB2DYNAMIXEL.

Fonte: Robotis (2017).

A comunicação entre o controlador e os servos é realizada pela troca de pacotes de informações. Existem duas categorias distintas de pacotes: pacotes de instrução, que o controlador principal despacha para os servos; e pacotes de status, que o servo envia de

volta para o controlador principal. A forma de conexão entre o controlador e o servos é representada na Figura 6.

Figura 6 – Conexão USB2DYNAMIXEL com os servosmotores.

Fonte: Robotis (2017).

O USB2DYNAMIXEL é conectado à porta USB do PC, e conectores 3P e 4P são oferecidos para que vários atuadores possam ser conectados. Para a comunicação com o AX-12A é utilizado o conector 3P que é a configuração que estabelece a comunicação TTL. A conexão é ilustrada na Figura 7.

Figura 7 – Conexão do Motor AX-12A.

(a) Conexão USB2DYNAMIXEL.

3 Pin Cable		
Pin No.	Signal	Pin Figure
1	GND	
2	NOT Connected	
3	DATA (TTL)	

(b) Pinagem para Configuração 3P.

Fonte: Robotis (2017).

A comunicação com mais de um atuador é feita de maneira serial, onde, cada motor tem um ID, e um servo recebe a conexão com o USB2DYNAMIXEL e os outros servos são conectados a ele. Como o USB2DYNAMIXEL não fornece energia para o servo, para a alimentação é necessário conectar uma fonte externa a um dos motores. A conexão é esquematizada na Figura 8.

2.3 Unity

O Unity é uma plataforma de desenvolvimento usado principalmente na criação de jogos 3D de alta qualidade e de simulações interativas. Ele oferece um ambiente imersivo que permite o desenvolvimento de experiências únicas e personalizadas de Gêmeos Digitais, que envolvem e engajam o usuário em um nível mais profundo do que no caso dos métodos tradicionais (DOSOFTEI, 2023).

Figura 8 – Conexão em de múltiplos atuadores.

Fonte: Robotis (2017).

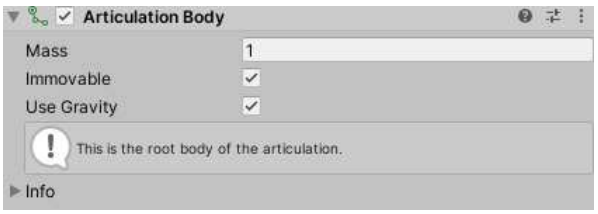
Gêmeos digitais são réplicas virtuais de sistemas físicos e o Unity é uma ferramenta usada para criar e explorar essas réplicas de uma forma altamente imersiva e interativa, onde o ambiente externo pode influenciar o comportamento dos Gêmeos Digitais. Ao criar um Gêmeo Digital influenciado por fatores externos, é possível compreender melhor como o sistema funciona sob diferentes condições e tomar decisões mais informadas sobre o seu projeto e operação.

No Unity são disponilizadas funções para criação de robôs, utlizando a modelagem 3D. Elementos que representam articulações, como braços robóticos ou cadeias cinemáticas, podem ser criados utilizando *GameObject* hierarquicamente organizados do tipo *Articulation Body*. Esses componentes ajudam a obter comportamentos físicos realistas no contexto de simulação para aplicações industriais.

O uso do componente *Articulation Body* pelo usuário permite a definição, em um único componente, das propriedades de um corpo rígido e uma junta comum. Dito isso, essas propriedades dependem da posição do *GameObject* na hierarquia:

- Conforme apresentado na Figura 9, para o *GameObject* pai da articulação, é feita a configuração apenas das propriedades do corpo físico:

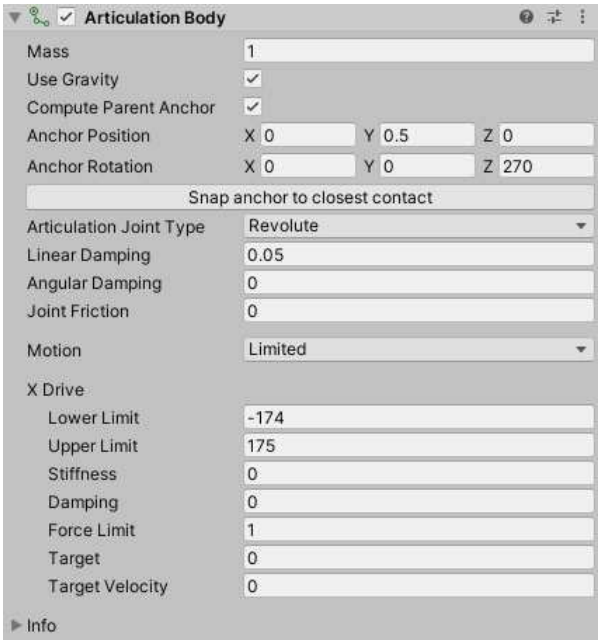
Figura 9 – Tela do Unity para a configuração do *GameObject* Pai.



Fonte: Autoria própria.

- Conforme apresentado na Figura 10, para qualquer *GameObject* filho dentro da articulação, é feita a configuração das propriedades do corpo físico, bem como o tipo e as propriedades da junta que liga esse *GameObject* ao seu *GameObject* pai:

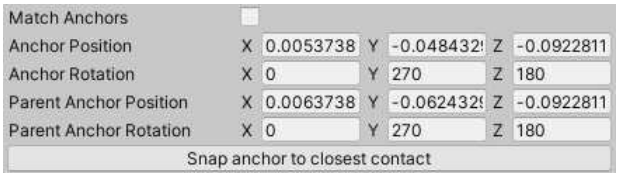
Figura 10 – Tela do Unity para a configuração do *GameObject* Filho.



Fonte: Autoria própria.

A propriedade do *Articulation Body* usada para este trabalho, foi a propriedade da Âncora de Junta. Dentro desta categoria, é preciso definir as coordenadas das âncoras de junta tanto para o *Articulation Body* quanto para seu *GameObject* pai. As principais propriedades da âncora de junta são apresentadas na Figura 11 e na Tabela 1.

Figura 11 – Tela do Unity para a configuração das propriedades da Âncora de Junta.



Fonte: Autoria própria.

Ademais, é necessário selecionar o tipo de junta que conecta o *Articulation Body* atual ao seu *Articulation Body* pai e definir suas propriedades comuns e específicas. Sendo possível a escolha de 4 tipos de juntas: a fixa, a prismática, a de revolução e a esférica. Também é necessário estabelecer o amortecimento linear, o amortecimento angular e o atrito articular.

Nesse contexto, Unity tornou-se uma plataforma proeminente para simular e controlar robôs em diversas aplicações. Pesquisadores têm utilizado a interface do Unity, suas capacidades de renderização em tempo real e suporte para ambientes 3D para criar Gêmeos Digitais e simulações para sistemas robóticos diversos.

Além disso, o Unity tem sido mostrado como facilitador de instruções em linguagem

Tabela 1 – Propriedades da Âncora de Junta

Propriedade	**Função**
Match Anchor	Esta propriedade faz a âncora relativa ao pai corresponder à âncora do *Articulation Body* atual. Se desabilitada, esta propriedade, poderá definir separadamente valores para *Parent Anchor Position* e *Parent Anchor Rotation*.
Anchor Position	As coordenadas de posição da âncora, em relação ao corpo de articulação atual.
Anchor Rotation	As coordenadas de rotação da âncora, em relação ao *Articulation Body* atual.
Parent Anchor Position	As coordenadas de posição da âncora pai, em relação ao *Articulation Body* pai. Esta propriedade só aparece se *Match Anchor* estiver desabilitada.
Parent Anchor Rotation	As coordenadas de rotação da âncora pai, em relação ao *Articulation Body* pai. Esta propriedade só aparece se *Match Anchor* estiver desabilitada.
Snap Anchor to closest contact	Calcula o ponto na superfície deste *Articulation Body* que está mais próximo do centro de massa do *Articulation Body* pai e define a âncora para ele. Se *Match Anchor* estiver habilitado, o Unity também atualiza a âncora pai adequadamente.

natural para robôs, correção de erros de robôs por meio de estruturas de realidade virtual e permitir métodos de navegação adaptativa para ações estáveis de robôs em diversos estudos (MIZUCHI; INAMURA, 2017). Ademais, a fusão do Unity com tecnologias de realidade virtual e aumentada tem sido investigada para aprimorar interações humano-robô e mecanismos de controle (JI et al., 2022).

2.4 MATLAB

O MATLAB é amplamente utilizado em diversas aplicações, incluindo a robótica. Em particular, ele é utilizado para controlar os servomotores da ROBOTICS, que atuam como juntas do braço robótico (ROBOTIS, 2017). Além disso, o MATLAB é empregado no desenvolvimento de algoritmos de controle e modelagem dinâmica de sistemas robóticos (KRISHNAN; VIJAYAN; ASHOK, 2020).

Ademais, no MATLAB está disponível o "App Designer", que é um ambiente integrado que permite ao usuário aplicações que possuam Interface Gráfica do Usuário (GUI- do inglês *Graphics User Interface*). Nesse sentido, o MATLAB foi utilizado no trabalho para realizar o controle de cada junta do braço robótico físico.

2.5 Visual Studio

O Visual Studio é um Ambiente de Desenvolvimento Integrado (IDE) usado para desenvolver aplicativos em várias linguagens de programação. Nele, são disponibilizados diversos compiladores, ferramentas de conclusão de código, controle de origem, extensões e muitos outros recursos.

O Visual Studio oferece suporte completo para o desenvolvimento de aplicações .NET, tornando possível desenvolver aplicativos para área de trabalho, web, celulares, jogos e IoT. Os aplicativos .NET podem ser desenvolvidos nas linguagens C#, F# ou Visual Basic.

No trabalho, o Visual Studio foi usado no desenvolvimento de uma aplicação .NET para servidor OPC UA e na criação de *scripts* para o Unity na linguagem C#.

2.6 OPC UA

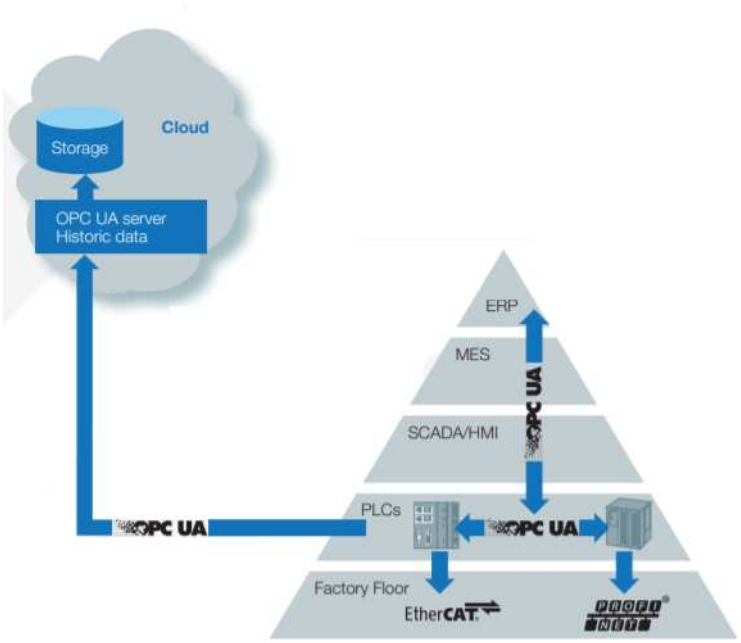
O OPC UA (do inglês *Open Platform Communication Unified Architecture*) é um padrão de comunicação industrial que permite a troca de dados entre diversos tipos de sistemas e dispositivos. Ele pode ser aplicado em vários níveis da pirâmide de automação industrial, abrangendo desde sensores e atuadores até sistemas de controle e planejamento de recursos empresariais (RINALDI, 2016). A principal finalidade com a utilização do OPC UA é facilitar a interoperabilidade entre dispositivos e sistemas de diferentes fabricantes, promovendo a integração e a eficiência dos processos industriais.

No padrão, é especificado um modelo de informação de infraestrutura comum para facilitar a troca de informações entre os dispositivos e camadas industriais. Esse modelo de informações é usado para representar estrutura, comportamento e semântica. Também é determinado pelo padrão o modelo de mensagens para interação entre aplicativos, o modelo de comunicação para transferir dados entre *endpoints* e o modelo de conformidade para garantir a interoperabilidade entre sistemas.

O OPC UA pode ser mapeado para uma variedade de protocolos de comunicação e os dados podem ser codificados de várias maneiras para equilibrar portabilidade e eficiência. Nele é permitido que os dados sejam expostos em muitos formatos diferentes, incluindo estruturas binárias e documentos XML ou JSON (OPC Foundation, 2022).

No mais, a utilização do OPC UA dentro do modelo de conformidade assegura que os sistemas possam interoperar, independentemente do fabricante. Como mostrado na Figura 12, o OPC UA não é direcionado apenas às interfaces de chão de fábrica, mas também é usado como uma maneira de fornecer maior interoperabilidade entre funções de nível superior.

Figura 12 – Aplicação do OPC UA dentro da pirâmide de automação.



Fonte: Envisia (2020).

2.6.1 UaModeler

O UaModeler é uma das ferramentas disponibilizadas pelo SDK da *Unified Automation*. Sua funcionalidade é construir modelos de informação a partir de design gráfico de espaço de endereço, adicionando nós e referências. Ele não apenas acelera a implementação, mas também aumenta a qualidade do software ao produzir código bem estruturado e sem erros. Os modelos gerados por ele são salvos no formato XML e podem ser usados para construção do servidor OPC UA.

No UaModeler é mostrada a representação hierárquica e gráfica do modelo projetado. A representação gráfica segue a notação e sintaxe OPC UA. Nele é gerado código para a SDK do servidor OPC UA baseado em C++, SDK do servidor OPC UA baseado em ANSI C, SDK de cliente e servidor OPC UA baseado em .NET e SDK de servidor OPC UA de alto desempenho.

2.6.2 UaExpert

O UaExpert é um cliente de teste OPC UA multiplataforma programado em C++ e desenvolvido pela *Unified Automation*. Nele é suportado recursos como *DataAccess*, alarmes e condições, acesso histórico e chamada de métodos UA. Sua estrutura básica inclui funcionalidades gerais como manipulação de certificados, descoberta de servidores UA, conexão com servidores UA, navegação no modelo de informações, exibição de atributos e referências de nós UA específicos. O UaExpert está disponível para Windows e Linux.

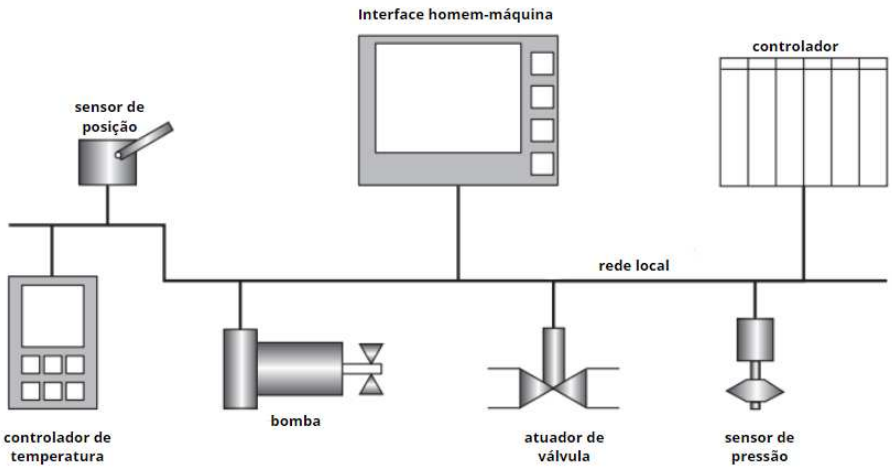
2.7 Controle Distribuido com IEC 61499

Atualmente, há um crescente interesse em novas tecnologias e arquiteturas para desenvolver sistemas distribuídos de automação industrial, onde o *software* é organizado como conjuntos de componentes cooperantes, em vez de grandes unidades de software personalizadas (AGUIAR; MURGATROYD; EDWARDS, 1996).

Nesse cenário, os sistemas para controle de processos industriais, de manufatura e de negócios vão se integrar. E para alcançar altos níveis de integração e permitir a criação de sistemas flexíveis que possam ser redefinidos conforme as necessidades industriais e empresariais mudam, é necessária uma nova abordagem para o design de software, baseada na interação de objetos distribuídos (ORFALI; HARLEY; EDWARDS, 1996).

Na Figura 13 é apresentada uma parte de um sistema com funcionalidade distribuída. Nesse tipo de sistema, a funcionalidade de controle pode ser distribuída entre dispositivos diferentes. Dispositivos inteligentes, como bombas, válvulas ou sensores, possuem funcionalidade de controle incorporada que pode ser vinculada por software com dispositivos mais inteligentes, como painéis interface Humano-Máquina (IHM), controladores de temperatura e controladores de software, para formar a funcionalidade total do sistema de controle (ZOITL; LEWIS, 2014).

Figura 13 – Sistema com funcionalidade distribuída.



Fonte: Adaptado de Zoitl e Lewis (2014).

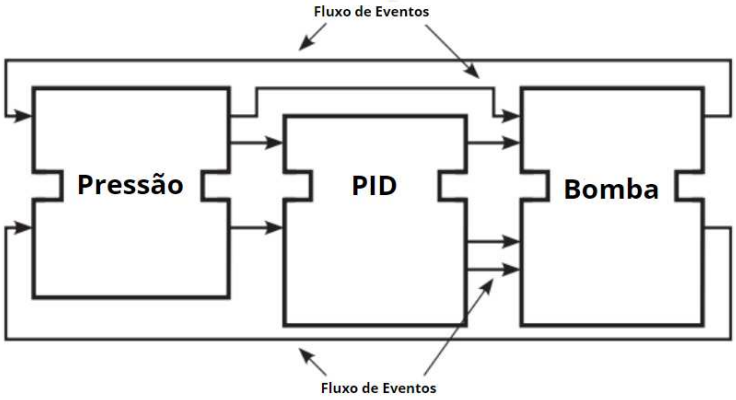
Com o desenvolvimento de novos padrões de comunicação, como o OPC UA, foi possível toda a integração técnica de componentes distribuídos, mas o grande desafio é a integração semântica, ou seja, definir um significado por trás dos dados. Nesse contexto, a Comissão Eletrotécnica Internacional (IEC) desenvolveu um padrão, o IEC 61499 Commission (2012), que define como blocos de função podem ser usados em sistemas distribuídos de processos industriais, medição e controle. O uso do padrão ajuda a resolver

parte do problema de integração semântica. No padrão é definido modelos de sistema que ajudam não apenas no design da funcionalidade em sistemas distribuídos, mas também na integração das ferramentas do sistema através da definição de modelos de dados e informações.

No padrão IEC 61499, foi desenvolvido um modelo geral e uma metodologia para descrever blocos de função em um formato que é independente da implementação. A metodologia pode ser usada por projetistas de sistemas para construir sistemas de controle distribuídos. O uso dessa metodologia permite que um sistema seja definido em termos de blocos de função logicamente conectados que são executados em diferentes dispositivo de processamento.

No IEC 61499, é usado blocos de função para encapsular funcionalidades e algoritmos de software em um formato padrão. O uso do padrão permite que ferramentas e outros padrões que lidam com blocos de função usem os mesmos conceitos e metodologia (ZOITL; LEWIS, 2014). Na Figura 14 é apresentado três blocos de função interconectados, representando as conexões entre um transmissor de pressão, um bloco de função de controle PID e uma bomba usando conceitos do IEC 61499.

Figura 14 – Dados e fluxos de eventos do Bloco de Função.



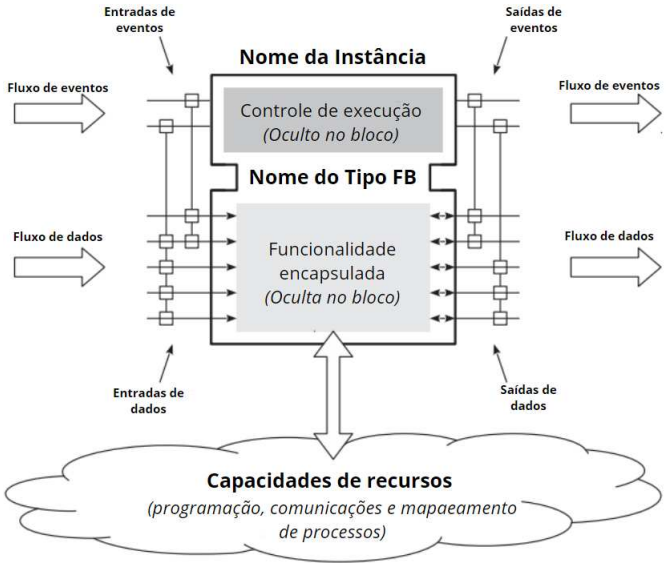
Fonte: Adaptado de Zoitl e Lewis (2014).

No centro do padrão está o modelo de bloco de função que sustenta toda a arquitetura IEC 61499. Um bloco de função é descrito como uma "unidade funcional de software"que possui sua própria estrutura de dados, a qual pode ser manipulada por um ou mais algoritmos (ZOITL; LEWIS, 2014).

Os blocos de função possuem um nome de tipo e um nome de instância, exibidos graficamente, e podem ter entradas e saídas de eventos e dados para comunicação entre blocos. Na Figura 15, são ilustradas as principais características de um bloco de função IEC 61499. A parte superior do bloco, denominada 'Controle de execução', fornece uma definição para associar eventos à funcionalidade encapsulada. Isso significa que ela determina qual 'Funcionalidade encapsulada', localizada na parte inferior do bloco, será ativada quando

eventos chegarem ao 'Controle de execução', bem como quando os eventos de saída são gerados. No padrão é descrito como estabelecer as relações entre os eventos que chegam nas entradas de eventos, a execução da funcionalidade encapsulada e a emissão de eventos de saída, conforme o tipo de bloco de função (COMMISSION, 2012).

Figura 15 – Características do Bloco de Função



Fonte: Adaptado de Zoitl e Lewis (2014)

No modelo de execução geralmente blocos de função são definidos como elementos passivos. Eles requerem um gatilho por um evento de entrada para invocar sua funcionalidade encapsulada. O ambiente de execução no qual um bloco de função existe deve fornece um escalonador que garante que cada fase da execução do bloco de função ocorra na ordem correta e na prioridade correta. Há várias fases necessárias para que o bloco de função seja executado; cada fase depende de interações definidas entre o bloco de função e o escalonador (ZOITL; LEWIS, 2014).

Na Figura 16 é apresentado os oito passos que devem ocorrer sequencialmente para o bloco de função operar; a conclusão de cada fase é definida por um passo numerado específico.

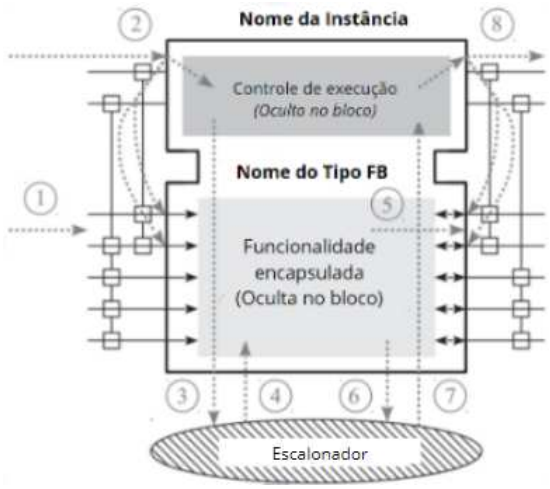
Cada passo é descrito a seguir:

1. Os valores de dados provenientes de outros blocos de função são disponibilizados nas entradas de dados do bloco de função.

2. Quando um evento de entrada chega, as variáveis de entrada associadas são amostradas, e o controle de execução é notificado.

3. Com base no estado atual do bloco de função, o controle de execução sinaliza ao escalonador que uma certa funcionalidade encapsulada está pronta para ser executada.

Figura 16 – Modelo de execução para Bloco de Função.



Fonte: Adaptado de Zoitl e Lewis (2014).

4. Após um período determinado, o escalonador começa a executar a funcionalidade encapsulada solicitada.

5. A funcionalidade que foi solicitada processa os valores de entrada e, em alguns casos, valores internos, criando novos valores de saída que são escritos nas saídas do bloco de função.

6. Ao completar sua tarefa, a funcionalidade sinaliza ao escalonador que os valores de saída estão estáveis.

7. O escalonador notifica o controle de execução sobre a conclusão, permitindo que ele gere um evento de saída apropriado

8. Este evento de saída aciona a execução de blocos de função a jusante, sinalizando que eles agora podem usar os valores de saída gerados.

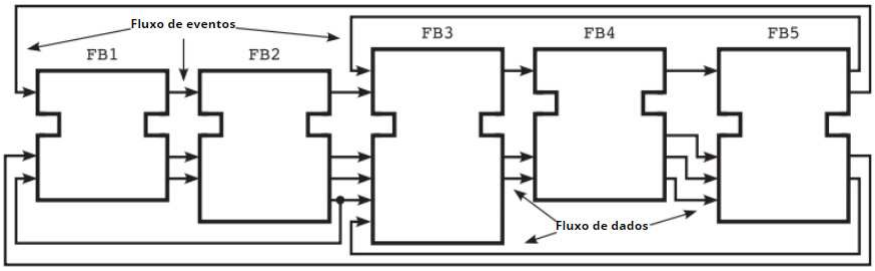
Um conceito importante no IEC 61499 é a capacidade de definir um tipo de bloco de função que estabelece o comportamento e as interfaces das instâncias de bloco de função que podem ser criadas a partir do tipo (ZOITL; LEWIS, 2014).

Atualmente existem três tipos de bloco de funções, sendo eles os blocos de função básicos, os blocos de função compostos e os blocos de função de interface de serviço (COMMISSION, 2012). Os blocos de função básicos são definidos por algoritmos que respondem a eventos de entrada e acionam eventos de saída para sinalizar mudanças de estado, utilizando diagramas de controle de execução (ECC do inglês *Execution Control Chart*) para mapear eventos a algoritmos. Blocos de função compostos, por outro lado, são definidos por uma rede de instâncias de blocos de função, com conexões de dados e eventos entre essas instâncias. Já os blocos de função de interface de serviço atuam

como intermediários entre o domínio do bloco de função e serviços externos, como a comunicação com dispositivos remotos ou a leitura de um relógio em tempo real de hardware, sendo definidos por diagramas de sequência de serviço e desenvolvidos em linguagens de programação de TI(Tecnologia da Informação), como C, C++ e Python.

Um conjunto de blocos de função interconectados, ligados por fluxos de eventos e dados é definida como uma aplicação IEC 61499. As características do modelo de aplicação IEC 61499 são apresentadas na Figura 17.

Figura 17 – Exemplo de aplicação IEC 61499.



Fonte: Adaptado de Zoitl e Lewis (2014).

Por fim, execução de uma aplicação IEC 61499 pode ser distribuida em diversos dispositivos, como apresentado na Figura 18.

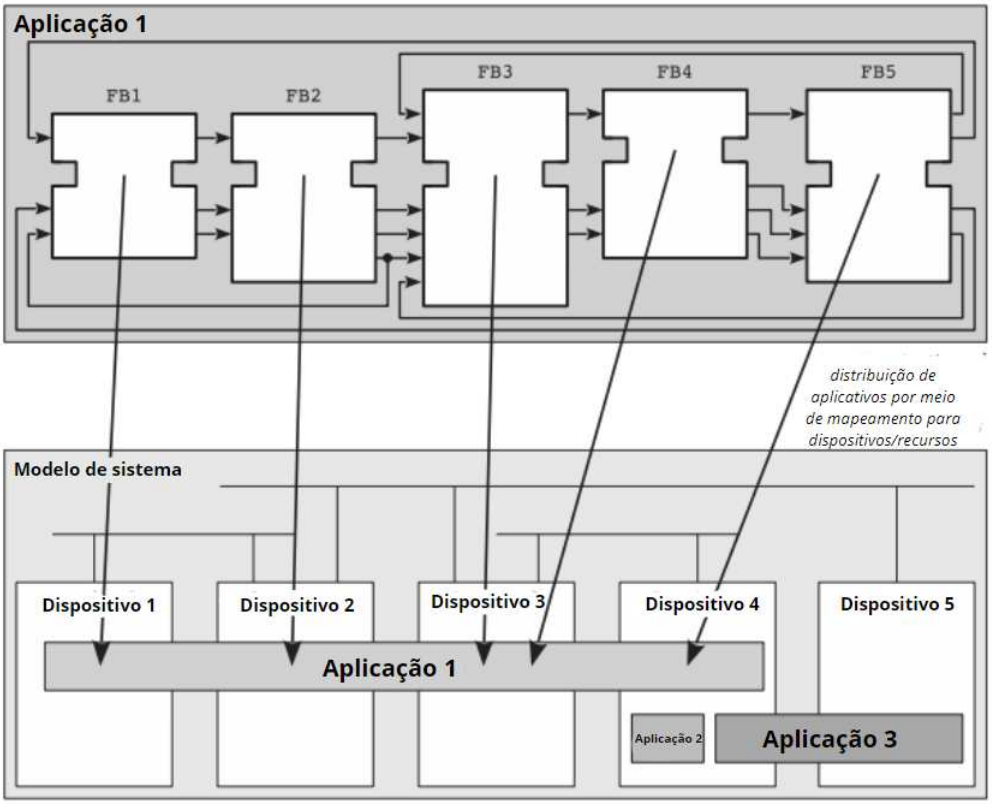
2.8 *Eclipse* 4diac

O Eclipse 4diac é um *framework* que fornece uma infraestrutura de código aberto para sistemas de medição e controle de processos industriais distribuídos com base no padrão IEC 61499. No *framework* 4diac é disponibilizado um ambiente de desenvolvimento, um ambiente de execução, biblioteca de blocos de função e projetos de exemplo (Eclipse Foundation, 2007).

2.8.1 4diac IDE

O 4diac IDE é o ambiente de desenvolvimento integrado, desenvolvido utilizando a linguagem de programção java e baseado no *framework* Eclipse. Nele é oferecido uma plataforma extensível para modelar aplicações de controle distribuído conforme o padrão IEC 61499. Com o 4diac IDE, é possível criar blocos de função (FBs), desenvolver aplicativos, configurar dispositivos e realizar outras tarefas relacionadas ao IEC 61499. Além disso, também é possível implantar esses resultados em dispositivos que executam o 4diac FORTE ou outros ambientes de execução compatíveis.

Figura 18 – Exemplo de aplicação IEC 61499 distribuída.



Fonte: Adaptado de Zoitl e Lewis (2014).

2.8.2 4diac FORTE

O 4diac FORTE é o ambiente de execução IEC 61499 implementado em C++, que suporta a execuções de FBs em dispositivos embarcados. Este ambiente de execução é multithreaded e otimizado para baixo consumo de memória. Normalmente, o 4diac FORTE opera sobre um sistema operacional em tempo real. Ele é compatível com os seguintes sistemas operacionais: Windows, Linux (i386, amd64, ppc, xScale, arm), NetOS, eCos, rcX da Hilscher, vxWorks e freeRTOS.

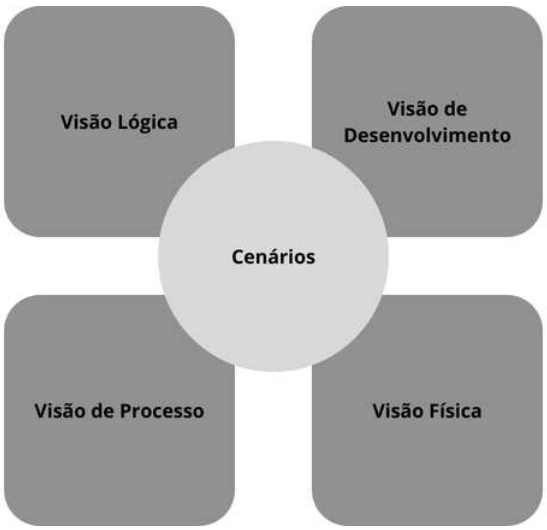
3 Arquitetura do Sistema

3.1 Modelo de Visão 4+1

Para muitos sistemas, o projeto de *software* é uma parte complexa de ser desenvolvida especialmente em aplicações de controle distribuído onde há *software* rodando em diferentes recursos de processamento (ZOITL; LEWIS, 2014). Isso ocorre pois algumas visões expressam aspectos abstratos do sistema, outras visões expressam aspectos físicos e outras mostram como o *software* está organizado. No contexto de engenharia de *software*, o termo visão se refere a uma representação do sistema em desenvolvimeto, que foca em um aspecto particluar para facilitar o entedimento e gerenciamento do sistema. Logo, para uma boa estruturação do sistema, é necessário a construção de várias visões para permitir que diferentes aspectos de construção do sistema sejam expressos e analisados.

Nesse contexto, a maioria dos projetos de *software* podem exigir pelo menos quatro visões de projetos diferentes, além da construção de cenários. Esse conjunto é conhecido como uma arquitetura chamada Modelo de Visão 4+1, proposta por Kruchten (1995). Conforme apresentado na Figura 19, a arquitetura é composta pelas seguintes visões: Visão Lógica, Visão de Processo, Visão de Desenvolvimento, Visão Física e a apresentação de cenários.

Figura 19 – Modelo de visão 4+1.



Fonte: Autoria própria.

Visão Lógica

A Visão Lógica é utilizada para representar os requisitos funcionais do sistema. Nela é representada as funcionalidades do *software* exigida pelo usuário. Um dos diagramas utizados para representar a visão lógica é o diagrama de sequência. No diagrama de sequência, é detalhado de que forma e em que ordem um conjunto de elementos trabalha em conjunto. Nele, são apresentados os objetos do sistema, que são representados por retângulos e possuem linhas de vida, e a troca de mensagens durante a interação entre eles.

Visão de Processo

A Visão de Processo é utilizada para representar os requisitos não funcionais do sistema. Nela, é possível visualizar as partes dinâmicas, onde se pode explicar os processos e como eles se comunicam, focando no comportamento do sistema. Segundo Kruchten (1995), a visão de processo representa redes lógicas de programas comunicantes que são distribuídos por um conjunto de recursos de hardware. A visão de processo trata da concorrência, distribuição, integração, desepenho e escalabilidade. O diagrama usado nessa visão é o diagrama de atividades.

No diagrama de atividades é descrito os acontecimentos necessários para o sistema modelado. Nele, é fornecida a visualização do comportamento do sistema, descrevendo as ações do processo. Consiste em um fluxograma com as ações de uma atividade. No entanto, o diagrama de atividade também pode representar fluxos paralelos ou simultâneos, bem como fluxos alternativos.

Visão de Desenvolvimento

A Visão de Desenvolvimento é usada no gerenciamento de projeto e apresenta o sistema do ponto de vista do desenvolvedor. Nela, são mostrados os componentes do projeto e os relacionamentos entre eles. O diagrama de componentes é o mais utilizado para exemplificar essa visão. Neste, os componentes representam sistemas ou subsistemas independentes que possuem capacidade de interagir entre si. O diagrama de componentes apresenta como o sistema de *software* é estruturado, descrevendo seus componentes, suas interfaces e suas dependências.

Visão Física

Na visão física o sistema é representado do ponto de vista da engenharia. Uma visão física abrange a configuração física do sistema, mostrando a localização dos dispositivos e fornecendo detalhes sobre a comunicação e barramentos. Nela o diagrama de implementação é usado.

No diagrama de implementação é descrita quais elementos de *hardware* são embarcados os elementos de *software*. Nele é apresentada a visão da topologia do sistema do *hardware* e comunicação entre os componentes.

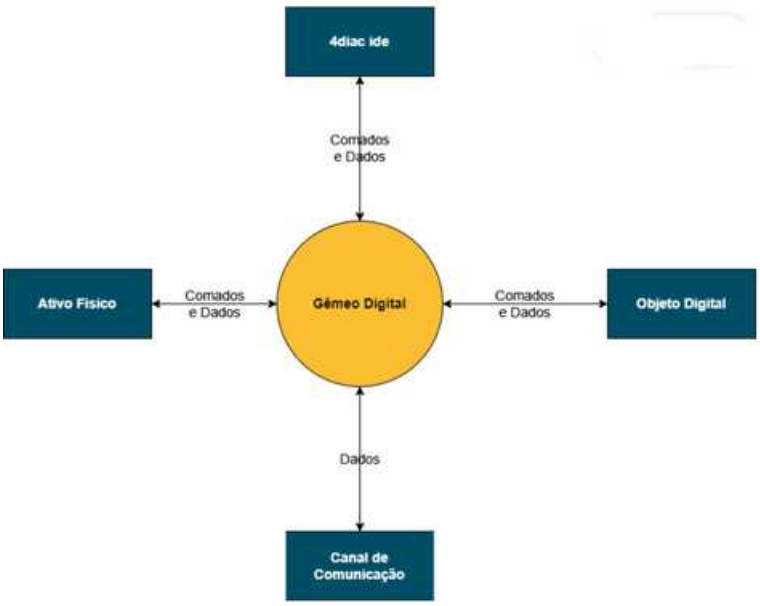
Cenário

O cenário é uma visão usada para complementar as outras, representando as principais interações entre as unidades do software a fim de fornecer a principal funcionalidade do sistema. Um cenário pode apresentar aspectos das visões lógicas e das visões de processo. Para a construção de cenários, é utilizado o diagrama de casos de uso, que descreve as sequências de interações entre os objetos e processos. Esses diagramas são usados para identificar elementos de arquitetura, ilustrar e validar o design da arquitetura. Nos diagramas de caso de uso é descrito as funções de alto nível e o escopo do sistema.

3.2 Arquitetura Proposta

Para a contextualização do sistema, é necessária a construção do diagrama de contexto. Na Figura 20, é apresentado esse diagrama para o Gêmeo Digital. Nele, é possível visualizar as principais partes que compõem o sistema, sendo elas: uma aplicação no 4diac IDE, o ativo físico que é o braço robótico AX-12A presente no LIEC, o objeto digital construído no Unity e o canal de comunicação desenvolvido utilizando OPC UA. Nesse contexto, a arquitetura Modelo 4+1 será utilizada para o desenvolvimento do trabalho.

Figura 20 – Diagrama de contexto do sistema.



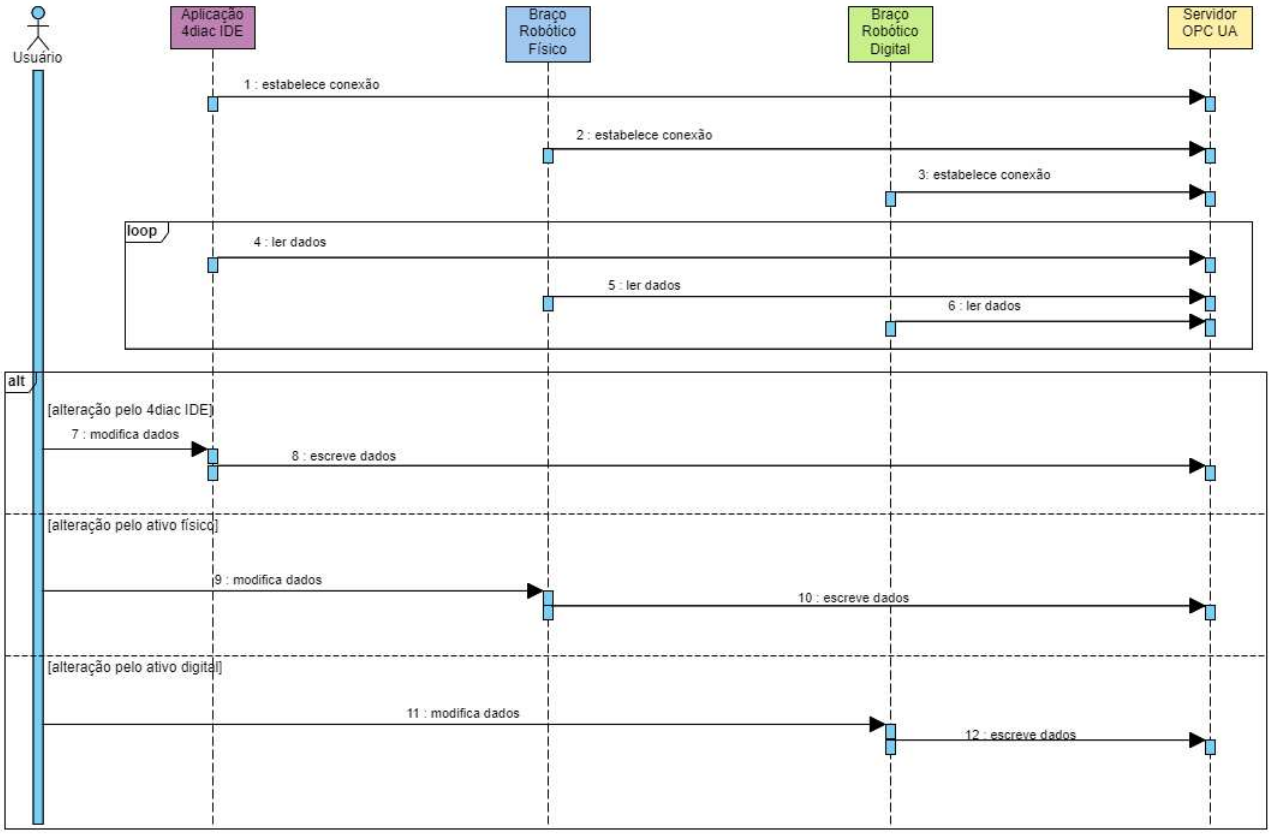
Fonte: Autoria própria.

3.2.1 Visão Lógica

Na Figura 21, é apresentado o diagrama de sequência construído para o sistema, onde os objetos são: o usuário, a aplicação 4diac IDE, o braço robótico AX-12A físico, o braço robótico no Unity e o servidor OPC UA. Nele, é possível perceber que, antes que o usuário troque informações com os objetos do sistema, é necessário que a aplicação 4diac IDE, o braço robótico AX-12A físico e o braço robótico no Unity estabeleçam conexão com o servidor OPC UA. Ainda no diagrama, é apresentado o bloco *loop*, que representa um bloco de repetição referente a leitura dos dados disponibilizados no servidor.

Por fim, é apresentado o bloco *alt*, que é usado para representar uma escolha entre duas ou mais sequências de mensagens. Quando o usuário deseja modificar a angulação de uma determinada junta por meio de um determinado objeto, seja a aplicação 4diac IDE, o braço robótico AX-12A físico ou o braço robótico no Unity, o sistema seguirá a seguinte ordem de troca de mensagens: alteração do dado no objeto e a escrita do dado pelo objeto no servidor OPC UA. É importante perceber que, como os dados estão sendo lidos no servidor de maneira contínua, os dados dos demais objetos serão atualizados.

Figura 21 – Diagrama de sequência do sistema.

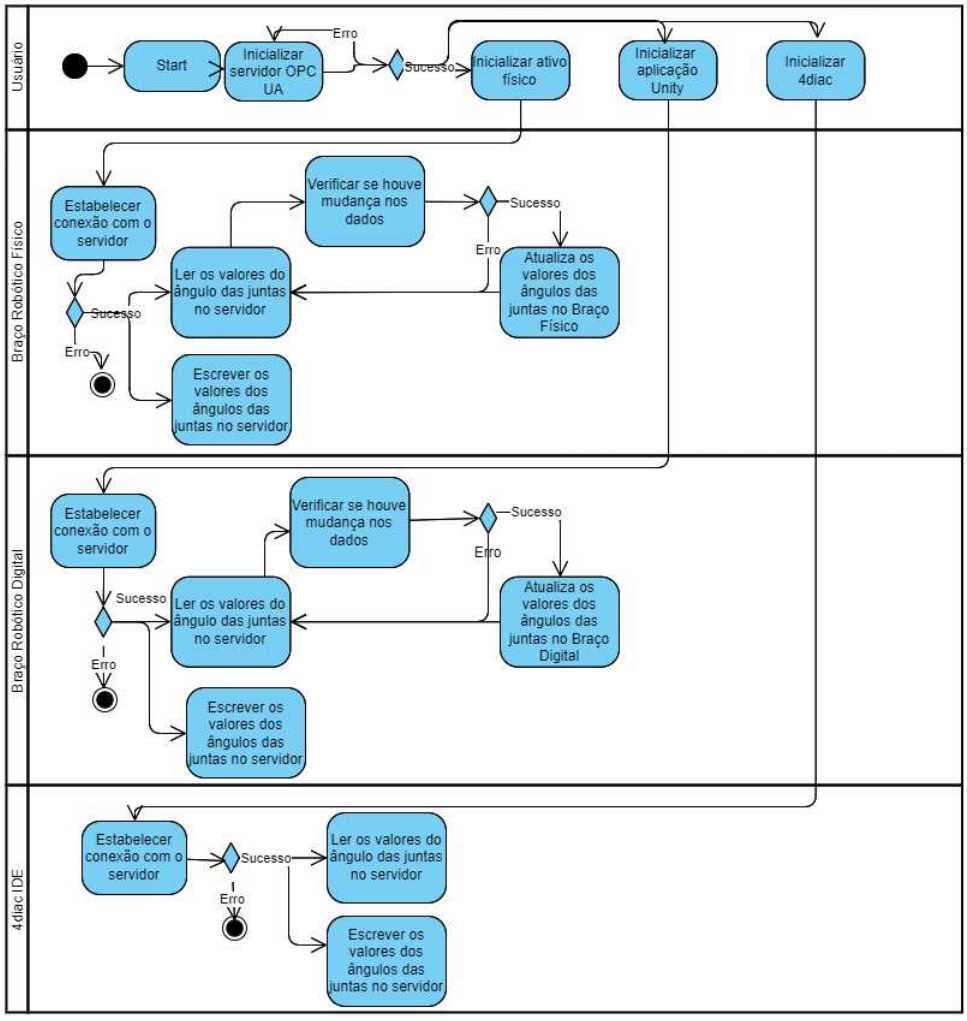


Fonte: Autoria própria.

3.2.2 Visão de Processo

No diagrama de atividades apresentado na Figura 22, é possível visualizar como o sistema se comportará a partir dos fluxos de ações de cada um de seus componentes. Nele, é apresentado um usuário e três componentes: uma aplicação 4diac IDE, o braço robótico físico e o braço robótico digital. A atividade do usuário consiste em inicializar o sistema, inicializar o servidor OPC UA e, caso obtenha sucesso na inicialização do servidor, o usuário poderá inicializar o ativo físico, o ativo digital e a aplicação 4diac IDE. Com todos os componentes inicializados, cada um deles deverá estabelecer a conexão com o servidor que já foi inicializado pelo usuário. A partir disso, será feita a leitura em periódica dos dados, onde a cada ciclo de leitura será verificado se os dados estão sendo alterados. Caso estejam, os valores serão atualizados no sistema de cada componente e o processo retornará ao *loop* de leitura. Se não houver alterações dos dados, o processo também retornará ao *loop*. O usuário também pode escrever os valores desejados no servidor utilizando as interfaces de cada componente.

Figura 22 – Diagrama de atividade do sistema.



Fonte: Autoria própria.

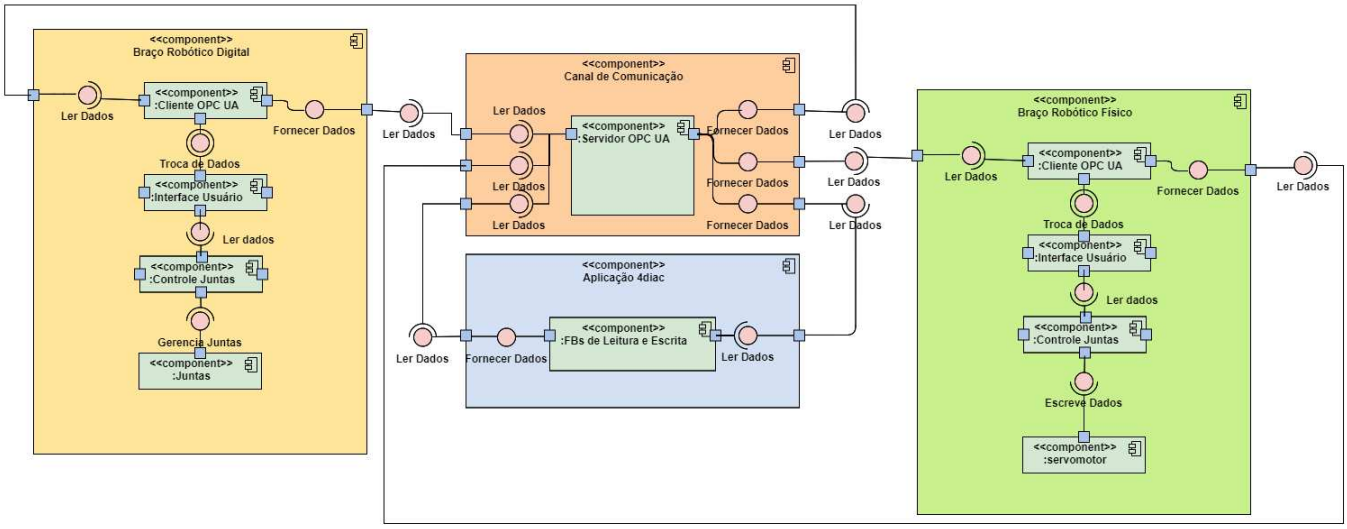
3.2.3 Visão de Desenvolvimento

Na Figura 23, é apresentado o diagrama de componentes do Gêmeo Digital. Nele, são exibídos os quatro componentes principais: o canal de comunicação, o braço robótico digital, a aplicação 4diac e o braço robótico físico. As interfaces entre os componentes principais são de leitura e escrita de dados, sendo a interface consumidora a de leitura de dados e a interface fornecedora a de fornecimento de dados. O componente braço robótico digital é composto por outros quatro componentes, sendo eles:

- o cliente OPC UA, que faz a comunicação com o servidor;
- a interface de usuário, que adquire dados do ambiente usuário;
- o controle de juntas, que, a partir dos dados adquiridos, determina qual junta irá funcionar;
- e as juntas, que rotaciona a articulação específica para o ângulo determinado.

O componente braço robótico físico funciona da mesma maneira do digital, com a única diferença que o componente *juntas* agora é o servomotor do braço físico. Já a aplicação 4diac possui componentes de blocos de função de leitura e escrita. Ainda há o componente do canal de comunicação, composto pelo servidor OPC UA, onde é possível realizar a escrita e a leitura dos dados.

Figura 23 – Diagrama de componente do sistema.



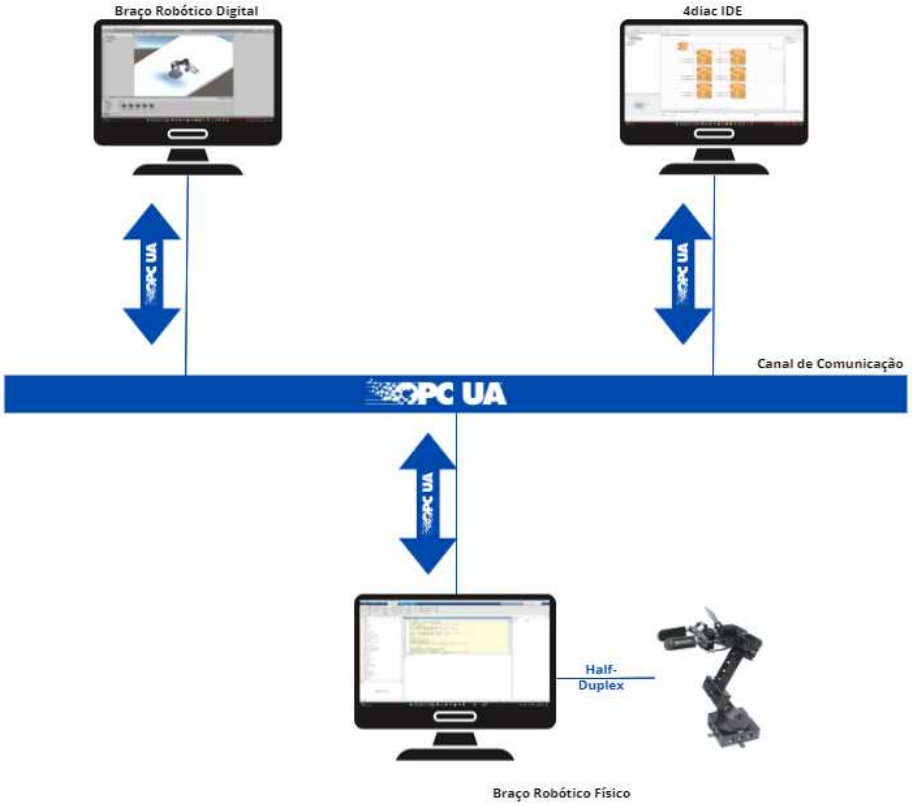
Fonte: Autoria própria.

3.2.4 Visão Física

Na Figura 24, é apresentado o diagrama de implementação do sistema. Nele, é mostrado o braço robótico físico conectado a um computador via comunicação Half-Duplex.

O computador no qual o braço físico esta conectado, se conecta ao canal de comunicação.O braço robótico digital, executado em outro computador, também se conecta ao canal de comunicação. Além disso, há a aplicação 4diac, que é executada em um computador e também se conecta no canal de comunicação. Todos os dispositivos so conectam utilizando o protocolo OPC UA via Ethernet.

Figura 24 – Diagrama de implementação.

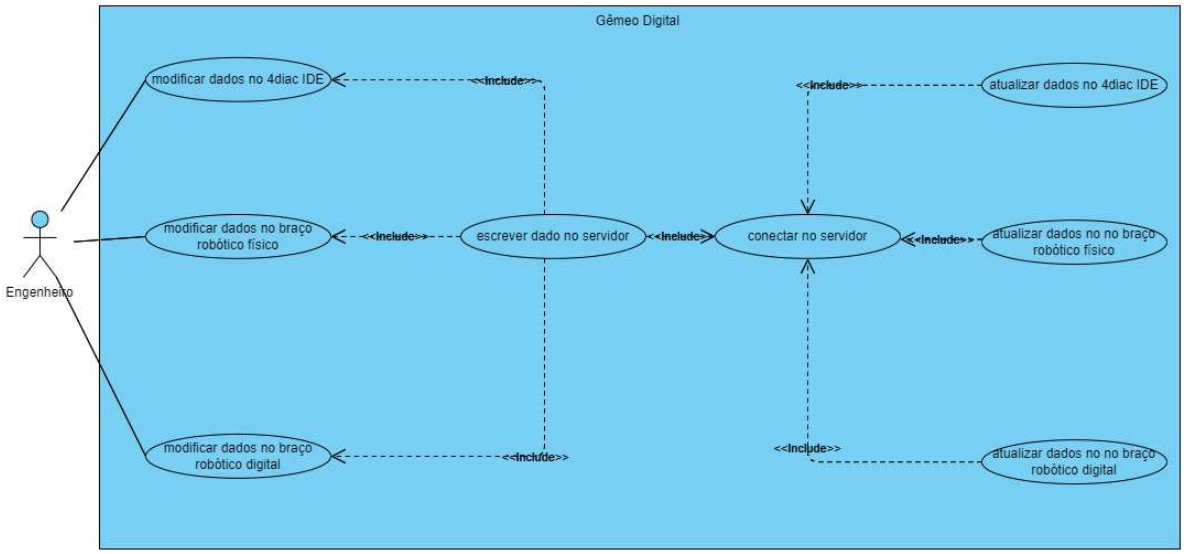


Fonte: Autoria própria.

3.2.5 Cenários

O caso de uso apresentado na Figura 25 descreve as sequências de interações que o usuário tem com os componentes do sistema. O ator do sistema, o engenheiro, pode ter três tipos de interações: realizar a mudança da angulação das juntas do braço tanto pela aplicação 4diac, como pela interface de usuário do ativo físico, como pela interface de usuário do ativo digital. Essas ações incluem escrever dados no servidor e conectar-se ao servidor, o que, por sua vez, inclui a atualização dos dados na aplicação 4diac IDE, no braço robótico físico e no braço robótico digital.

Figura 25 – Caso de uso do sistema.



Fonte: Autoria própria.

4 Desenvolvimento

Neste capítulo são apresentados os detalhes da implementação do trabalho descrito nos capítulos anteriores. O fluxo de desenvolvimento consistiu na programação do *software* de controle do braço robótico utilizando o MATLAB, o objeto digital no Unity, um canal de comunicação utilizando OPC UA e uma aplicação de controle distribuído utilizando blocos de funções.

4.1 *Software* do Braço Robótico

Para a construção do *software* que controla o braço robótico físico, foram utilizados o MATLAB e o AppDesigner. A aplicação é dividida em duas partes, sendo a primeira um código desenvolvido que controla as articulações do braço, e a segunda uma interface criada, onde ocorre a conexão com o servidor e a manipulação dos dados. A seguir, é apresentado como cada parte foi desenvolvida.

4.1.1 Interface de Comunicação e Manipulação de Dados

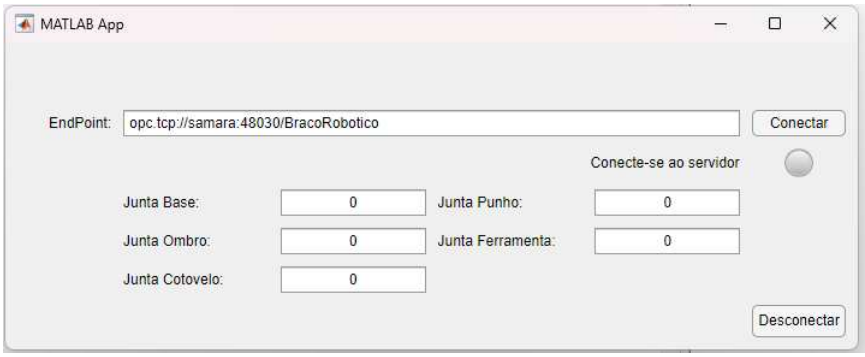
A interface gráfica no AppDesigner do MATLAB é utilizada para acionar as articulações do braço robótico físico e para a conexão com o servidor OPC UA, a fim de manipular as variáveis das articulações. No AppDesigner, são disponibilizados componentes gráficos de interação com o usuário, onde o desenvolvedor pode arrastar e soltar para adicionar componentes da GUI, como botões, caixas de texto, tabelas, gráficos, sliders, etc. Para a criação de aplicativos usando essa ferramenta, é necessário abrir o MATLAB, ir na guia de APPS e selecionar no menu Design App. Por fim, será aberta uma janela para a criação de aplicativos.

Como apresentado na Figura 26, a interface criada para a manipulação do braço robótico permite ao usuário conectar-se a partir da inclusão do *endpoint* do servidor OPC UA e do acionamento do botão *Conectar*, e desconectar-se pelo botão *Desconectar*. O usuário também pode modificar a angulação de uma articulação específica inserindo o valor desejado nos campos apresentados que ficam em frente às *labels*, *Junta Base*, *Junta Ombro*, *Junta Cotovelo*, *Junta Punho* e *Junta Ferramenta*.

Cada componente de interface pode ser associado a um *callback*, que é uma função usada para realizar alguma funcionalidade. Para essa interface foram usados 7 *callbacks*, sendo eles apresentados na Figura 27.

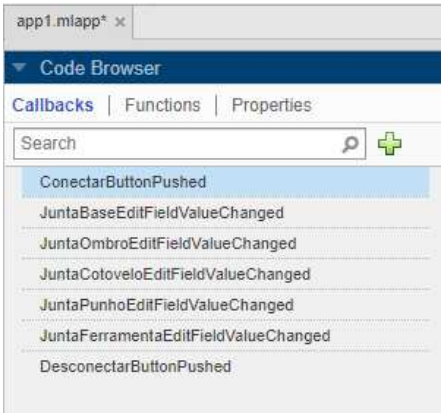
Os 5 *callbacks*, do tipo *EditFieldValueChanged*, seguem o mesmo comportamento: eles são acionados caso o valor do componente a que estão vinculados mude. A lógica

Figura 26 – Interface gráfica desenvolvida no AppDesigner.



Fonte: Autoria própria.

Figura 27 – *Callbacks* da interface.



Fonte: Autoria própria.

dentro da função é que, caso a caixa de entrada seja editada, será realizada a conversão desse valor, ele será escrito no servidor, e será chamada a função que aciona o movimento de determinada junta. Um exemplo desse tipo é apresentado na Figura 28. As outras funções do mesmo tipo seguem o mesmo padrão, mudando a chamada da função de acionamento do servomotor e o endereço da variável do servidor.

No *callback* *ConectarButtonPushed*, caso o botão Conectar seja pressionado, será executada a função apresentada na Figura 29, na qual é feita a conexão com o servidor OPC UA, a inicialização e configuração de um *timer* com execução indefinida que acionará a função *TimerCallback()* a cada meio segundo. Nela também são definidos o endereço de cada variável do servidor e armazenados em uma variável. Por fim, a função que realiza a inicialização do braço robótico é acionada.

A função *DesconectarButtonPushed* é o *callback* do botão Desconectar. Nela o servidor é desconectado, o *timer* é pausado e a função que desabilita o braço robótico é acionada. O código dessa função é apresentado na Figura 30.

A função chamada pelo *timer*, *TimerCallback()*, é apresentada na Figura 31. Nela

Figura 28 – Função *JuntaBaseEditFieldValueChanged*.

```
function JuntaBaseEditFieldValueChanged(app, event)
    % leitura e armazenamento do campo de entrada
    baseAngulo = app.JuntaBaseEditField.Value;
    % conversão da variável para double
    app.DoublebaseAngulo = str2double(baseAngulo);
    if isnan(app.DoublebaseAngulo)
       uialert(app.UIFigure, 'Insira um valor numérico válido.', 'Erro de Entrada');
       return;
    end
    %chamada da função para acionamento do servomotor
    base(app.DoublebaseAngulo);
    %escrita da variavel no servidor
    try
       writeValue(findNodeByName(app.nodoBase, 'angulo', '-once'), app.DoublebaseAngulo);
    catch ME
       uialert(app.UIFigure, ['Falha na escrita do valor: ' ME.message], 'Erro de Escrita');
    end
end
```

Fonte: Autoria própria.

Figura 29 – Função *ConectarButtonPushed*.

```
function ConectarButtonPushed(app, event)
    % Colocar o codigo da conexão aq
    %definição do endpoint
    endpointurl = app.EndPointEditField.Value;
    %cria o cliente
    app.uaCliente = opcua(endpointurl);
    %tenta conectar ao servidor
    try
        %estabelecer conexão
        connect(app.uaCliente);
        %mensagem de sucesso
        app.ConecteseaoservidorLamp.Color = 'green';
        app.ConecteseaoservidorLampLabel.Text = 'Conexão bem-sucedida';
        % Inicia o temporizador
        app.myTimer = timer('Period', 0.5, ...
            'ExecutionMode', 'fixedSpacing', ...
            'TasksToExecute', Inf, ...   % Quantas vezes o timer deve executar a função (Inf para infinito)
            'TimerFcn',  @(~,~) app.TimerCallback());  % Função de callback do timer

        start(app.myTimer);
        %define os nos
        app.nodoBase = findNodeByName(app.uaCliente.Namespace(4), 'Base', '-once');
        app.nodoOmbro = findNodeByName(app.uaCliente.Namespace(4), 'Ombro', '-once');
        app.nodoCotovelo = findNodeByName(app.uaCliente.Namespace(4), 'Cotovelo', '-once');
        app.nodoPunho = findNodeByName(app.uaCliente.Namespace(4), 'Punho', '-once');
        app.nodoFerramenta = findNodeByName(app.uaCliente.Namespace(4), 'Ferramenta', '-once');
        %Chama a função que realiza a inicialização do braço
        %robotico
        Inicializacao();
    catch ME
        %mensagem de erro
        app.ConecteseaoservidorLamp.Color = 'red';
        app.ConecteseaoservidorLampLabel.Text = 'Erro de Conexão';
        uialert(app.UIFigure, ['Falha na conexão: ' ME.message], 'Erro de Conexão');
    end
end
```

Fonte: Autoria própria.

Figura 30 – Função *DesconectarButtonPushed*.

```
function DesconectarButtonPushed(app, event)
    disconnect(app.uaCliente);
    stop(app.myTimer);
    app.ConecteseaoservidorLamp.Color = '#CCCCCC';
    app.ConecteseaoservidorLampLabel.Text = 'Conecte-se ao servidor';
    ENCERRAR();
end
```

Fonte: Autoria própria.

é realizada a leitura das variáveis do servidor a cada meio segundo e elas são comparadas com o último valor de entrada do usuário. Caso sejam diferentes, o valor lido no servidor é apresentado na interface para o usuário e é chamada a função que aciona o servomotor.

Figura 31 – Função *TimerCallback()*.

```
function TimerCallback(app, ~, ~)
    %Leitura no servidor
    NewBaseAngulo = readValue(app.uaCliente, opcuanode(2,6002,app.uaCliente));
    NewOmbroAngulo = readValue(app.uaCliente, opcuanode(2,6019,app.uaCliente));
    NewCotoveloAngulo = readValue(app.uaCliente, opcuanode(2,6024,app.uaCliente));
    NewPunhoAngulo = readValue(app.uaCliente, opcuanode(2,6029,app.uaCliente));
    NewFerramentaAngulo = readValue(app.uaCliente, opcuanode(2,6034,app.uaCliente));
    %Comparação para Base - Modificação
    if app.DoublebaseAngulo ~= NewBaseAngulo
        srtBaseAngulo = num2str(NewBaseAngulo);
        app.JuntaBaseEditField.Value = srtBaseAngulo;
        app.DoublebaseAngulo = NewBaseAngulo;
        base(app.DoublebaseAngulo);
    end
    %Comparação para Ombro - Modificação
    if app.DoubleOmbroAngulo ~= NewOmbroAngulo
        srtOmbroAngulo = num2str(NewOmbroAngulo);
        app.JuntaOmbroEditField.Value = srtOmbroAngulo;
        app.DoubleOmbroAngulo = NewOmbroAngulo;
        ombro(app.DoubleOmbroAngulo);
    end
    %Comparação para Cotovelo - Modificação
    if app.DoubleCotoveloAngulo ~= NewCotoveloAngulo
        srtCotoveloAngulo = num2str(NewCotoveloAngulo);
        app.JuntaCotoveloEditField.Value = srtCotoveloAngulo;
        app.DoubleCotoveloAngulo = NewCotoveloAngulo;
        cotovelo(app.DoubleOmbroAngulo);
    end
    %Comparação para Punho - Modificação
    if app.DoublePunhoAngulo ~= NewPunhoAngulo
        srtPunhoAngulo = num2str(NewPunhoAngulo);
        app.JuntaPunhoEditField.Value = srtPunhoAngulo;
        app.DoublePunhoAngulo = NewPunhoAngulo;
        punho(app.DoublePunhoAngulo);
    end
    %Comparação para Ferramenta - Modificação
    if app.DoubleFerramentaAngulo ~= NewFerramentaAngulo
        srtFerramentaAngulo = num2str(NewFerramentaAngulo);
        app.JuntaFerramentaEditField.Value = srtFerramentaAngulo;
        app.DoubleFerramentaAngulo = NewFerramentaAngulo;
        ferramenta( app.DoubleFerramentaAngulo);
    end
end
```

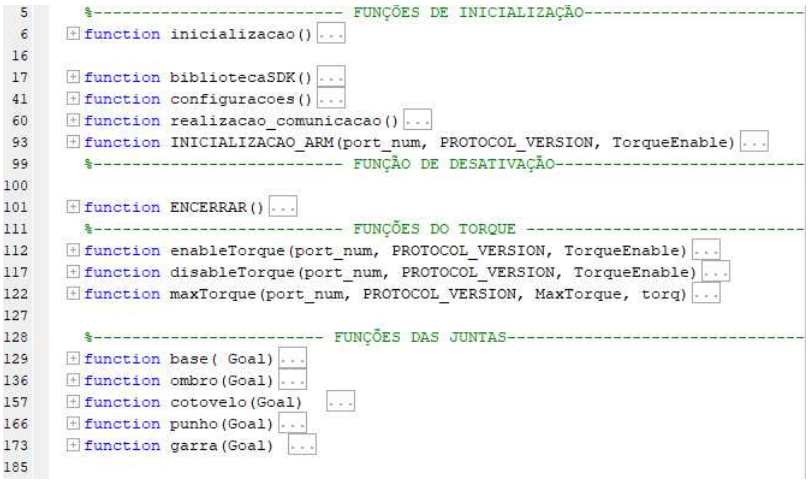
Fonte: Autoria própria.

4.1.2 Programa de Controle das Juntas

Para ocorrer a movimentação do braço robótico físico de acordo com as entradas do usuário e as variações do servidor foi desenvolvido um código que se realiza a comunicação com os servomotores e os aciona. A fabricante dos servomotores, contendo uma SDK que possui suporte para o MATLAB, e disponibiliza um conjunto de funções para realizar o controle dos servomotores da familia DYNAMIXEL. Logo para o desenvolvimento do código foi realizada a instalação da SDK, e adicionado ao caminho de diretórios do MATLAB os arquivos disponibilizados por ela.

Com a configuração do ambiente de desenvolvimento concluída, foi realizada a construção do script, que pode ser dividido em quatro partes: inicialização, desativação, funções de torque e funções das juntas. Na Figura 32 é apresentado o código.

Figura 32 – Programa para o controle das juntas.



Fonte: Autoria própria.

A primeira parte consiste em uma função chamada *inicializacao()*, que chama as funções *bibliotecaSDK()*, *configuracoes()*, *realizacao_comunicacao()* e *INICIALIZACAO_ARM()*. Na função *bibliotecaSDK()*, são realizadas todas as configurações da SDK, por meio da importação de arquivos de bibliotecas desenvolvidas na linguagem C, que serão utilizados pelo MATLAB na construção da aplicação. Na função *configuracoes()*, são definidos os IDs dos servomotores, a porta de comunicação onde o dispositivo USB2DYNAMIXEL está conectado, e o *upload* da Tabela de Controle, que é uma estrutura que armazena o status ou controla o dispositivo. Os métodos usados para escrita e leitura da tabela de controle são *write2ByteTxRx(port_num, PROTOCOL_VERSION, id, TorqueEnable, 1)*, que escreve um *byte* com parâmetros como a porta de comunicação, a versão do protocolo (pode ser 1 ou 2), o ID do Dynamixel, o nome da variável de controle na tabela e o valor a ser escrito.

No método *read2ByteTxRx(port_num, PROTOCOL_VERSION, ID_DXL, PresentPosition)* são lidos dois *bytes* com os mesmos parâmetros, sendo o último o dado na tabela de controle a ser lido. Na função *realizacao_comunicacao()* é estabelecida a comunicação com o dispositivo USB2DYNAMIXEL. Na função *INICIALIZACAO_ARM()*, o torque é habilitado, chamando a função da terceira parte *enableTorque()*, e o valor máximo de torque é definido chamando a função *maxTorque*.

Na parte de desativação, o torque é desabilitado com a função *disableTorque()*, a porta de conexão com o dispositivo USB2Dynamixel é fechada e as bibliotecas da SDK são descarregadas. A terceira parte envolve a ativação e desativação do torque, com três funções: a função para habilitar o torque de todos os servomotores, *enableTorque()*, a

função para desabilitar o torque de todos os servomotores, *disableTorque()*, e a função para definir o torque máximo para todos os servomotores, *maxTorque()*.

Na última parte, são criadas funções para cada articulação. Por exemplo, para a base, foi criada uma função que escreve na tabela de controle tanto a velocidade quanto o ângulo, sendo a velocidade definida com um valor constante e o ângulo é passado pela interface contruída. Como os valores do ângulo são passados em graus e na tabela de controle os valores estão em bytes, é necessário realizar a modulação de 0°a 300°. As funções para acionamento do servomotor utilizam a mesma função de escrita, mas com lógicas diferentes. Por exemplo, a junta do ombro e do cotovelo são compostas por dois servomotores, então a escrita na tabela de controle é feita para ambos os servomotores. Na Figura 33 é apresentada a função de acionamento da base, para o acionamento de um servomotor. Já na Figura 34 é apresentada a função de acionamento do ombro, que possui dois servomotores.

Figura 33 – Função de acionamento do servomotor da base.

```
131     %------------------------ FUNÇÕES DAS JUNTAS-------------------------------
132     function base( Goal)
133 -       global port_num PROTOCOL_VERSION MovingSpeed GoalPosition;
134 -       GoalByte = Goal*1024/300;
135 -       speedByte = 15*1024/114;
136 -       write2ByteTxRx(port_num, PROTOCOL_VERSION, 1, MovingSpeed, speedByte);
137 -       write2ByteTxRx(port_num, PROTOCOL_VERSION, 1, GoalPosition, GoalByte);
138 -   end
139     function ombro(Goal) ...
160     function cotovelo(Goal)  ...
169     function punho(Goal) ...
176     function garra(Goal)  ...
```

Fonte: Autoria própria.

Figura 34 – Função de acionamento dos servomotores do ombro.

```
function ombro(Goal)
    global port_num PROTOCOL_VERSION MovingSpeed GoalPosition PresentPosition;
    % calculo dos angulos de acordo com o dado, sendo o dado o id2
    GoalByte2 = Goal*1024/300;
    GoalByte3 = (300-Goal)*1024/300;
    speedByte = 15*1024/114;

    %Verificar se chegou na posição certa
    while true
        dxl_present_position2 = read2ByteTxRx(port_num, PROTOCOL_VERSION, 2, PresentPosition);
        erro = abs(dxl_present_position2-GoalByte2);
        if(erro< 15)
            break;
        else
            write2ByteTxRx(port_num, PROTOCOL_VERSION, 2, MovingSpeed, speedByte);
            write2ByteTxRx(port_num, PROTOCOL_VERSION, 2, GoalPosition, GoalByte2);
            write2ByteTxRx(port_num, PROTOCOL_VERSION, 3, MovingSpeed, speedByte);
            write2ByteTxRx(port_num, PROTOCOL_VERSION, 3, GoalPosition, GoalByte3);
        end
    end
end
```

Fonte: Autoria própria.

4.2 Modelo Digital

Para a virtualização do braço robótico, primeiramente foi desenvolvido o modelo 3D no unity e a aplicação da física, depois foi contruído o seu cliente OPC UA e *scripts* para controlar a movimentação das articulações. A seguir, é apresentado como cada parte foi desenvolvida.

4.2.1 Construção do Objeto Digital

O design auxiliado por computador (CAD do inglês *Computer-Aided Design*) é uma técnica de modelagem de sólidos virtuais onde objetos 3D são criados a partir de uma série de desenhos 2D usando software de computador. Muitos componentes individuais podem ser combinados em uma montagem para criar um modelo sólido 3D de um sistema maior. Os modelos sólidos ajudam os designers e engenheiros a ver a aparência e o funcionamento de um produto no mundo real.

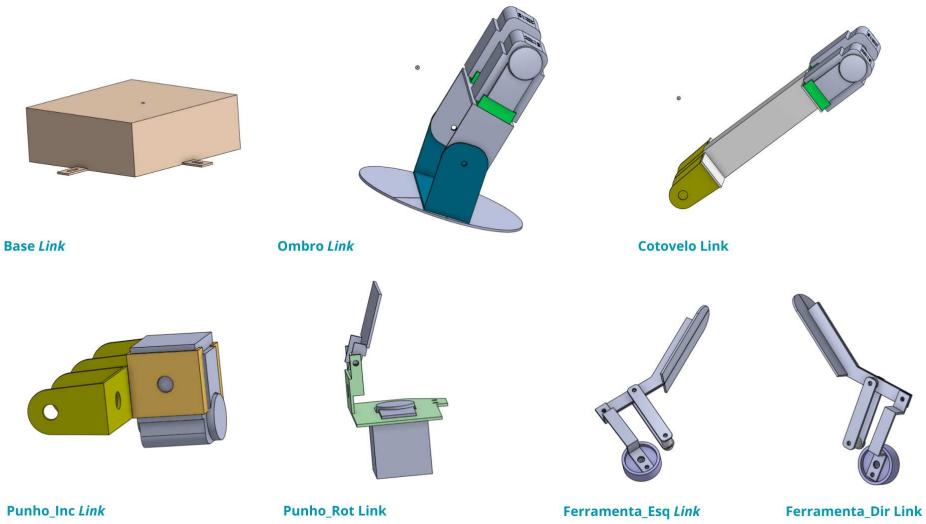
O modelo tridimensional (3D) do braço robótico está disponível no site da fabricante, CrustCrawler (NORRIS, 2008). Considerando a utilização do ambiente de simulação Unity, foi necessário realizar a importação do modelo 3D juntamente com sua física definida. Este processo demandou etapas específicas, uma vez que o Unity só aceita determinados tipos de arquivos, e o formato .iges do arquivo que continha o modelo 3D não é compatível.

Diante disso, foi realizado o *upload* do modelo 3D no OneShape, uma aplicação gratuita e online para manipulação de arquivos 3D. Nele foi realizada a fragmentação do braço robótico. O modelo do braço robótico foi dividido em *links*. Cada *link* foi exportado em formato Collada (do inglês *COLLAborative Design Activity*), que é um formato de arquivo usado para a troca de dados de gráficos 3D entre diferentes aplicações e plataformas. Na Figura 35 é apresentado a divisão e nomenclatura dos *links*.

No contexto de robótica e cinemática de robôs, um *link* refere-se a uma parte ou segmento físico de um robô. Em termos simples, é uma peça ou componente que compõe a estrutura mecânica do robô. Cada *link* pode ter características únicas, como comprimento, massa, inércia e conectividade com outros *link*s através de juntas ou articulações. Na montagem de um braço robótico, por exemplo, cada segmento que conecta uma junta a outra é considerado um *link*. Esses *link*s são fundamentais para determinar a cinemática e a dinâmica do robô, bem como para calcular trajetórias e controlar seu movimento.

Dentro do Unity, um *link* pode ser representado por um *GameObject* que contém informações sobre sua posição, orientação e escala no espaço tridimensional, além de outros componentes e propriedades necessários para o seu comportamento e interação com o ambiente virtual. Organizar os *links* como *GameObjects* dentro da hierarquia do Unity permite uma representação visual e funcional do robô e facilita a implementação de lógica de controle e simulação.

Figura 35 – *Links* do braço robótico

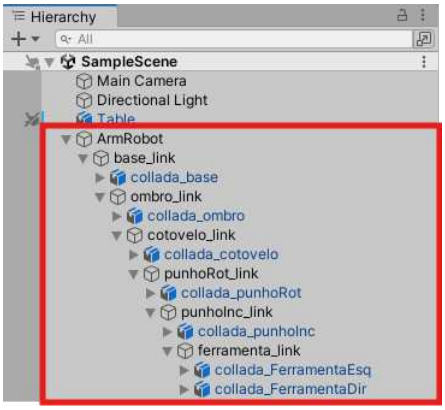


Fonte: Autoria própria.

Com a representação geométrica dos *links* no formato aceito pelo Unity, foi realizada a importação dos tais para a plataforma, onde foram utilizados na criação de *GameObjetc* para representar o braço robótico e suas articulações. Para a criação do *GameObjetc* braço robótico, a partir dos arquivos COLLADA é necessario o uso da propriedade do *Articulation Body* nos *GameObjetcs*, e também uma organização hierarquica entre eles. Logo foi criado um *GameObjetcs* pai, *ArmRobot*, que possui *GameObjetcs* filhos com os *links* exportados.

Na Figura 36 é apresentada como está a relação de hierarquia dos *GameObjetcs*, e nela é possível perceber que a hierarquia do Unity segue um sistema pai-filho.

Figura 36 – Hierarquia dos *GameObjetcs* do braço robótico



Fonte: Autoria própria.

Essa organização não apenas facilita a compreensão da estrutura do robô, mas também simplifica o processo de interação e manipulação durante o desenvolvimento e implementação de comportamentos e funcionalidades no projeto. Isso significa que qualquer

movimento ou rotação aplicado a um objeto pai será herdado por todos os seus filhos. Por exemplo, se o *base_link* for rotacionado, todos os links e objetos abaixo dele (*ombro_link*, *cotovelo_link*, etc.) também serão rotacionados.

Com a estrutura do braço montada no Unity, foi adicionado nos *GameObjetcs* o componente *ArticulationBody*. Cada *ArticulationBody* pode ter propriedades físicas e cinemáticas, como massa, colisores, propriedades de ancoragem da junta, seleção e propriedades do tipo de junta, propriedades de acionamento da junta, restrições de movimento e comportamento físico. Eles são responsáveis por definir como as diferentes partes do braço robótico interagem entre si e com o ambiente ao seu redor. Para a estruturação dos *ArticulationBody*, foi necessário a definição das juntas e qual o sentido de rotação que elas rotacionam. Na Figura 37 é apresentada as configurações que o objeto digital têm que corresponder, tanto na quantidade de juntas rotacionais quanto no sentido de rotação.

Figura 37 – Juntas do braço robótico.

(a) Definição das juntas do braço robótico.

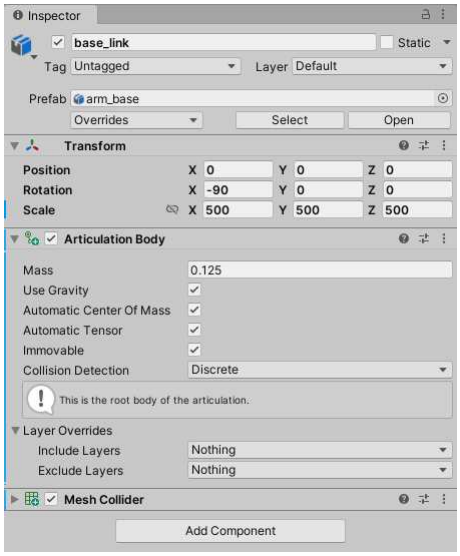
(b) Sentido de rotação das juntas do braço robótico.

Fonte: Adaptado de Norris (2008).

Após determinada as caracteristicas das juntas, é necessário configurar o componente *ArticulationBody* em cada *GameObject*. Para o *GameObject* raiz da articulação, o *base_link*, a configuração realizada é apresentada na Figura 38. Nele, apenas propriedades do corpo físico foram configuradas, enquanto para os demais *GameObjects* filhos dentro da hierarquia, além das propriedades do corpo físico, também foi configurado o tipo e as propriedades da junta que conecta esse *GameObject* ao seu pai, comforme apresentado na Figura 39.

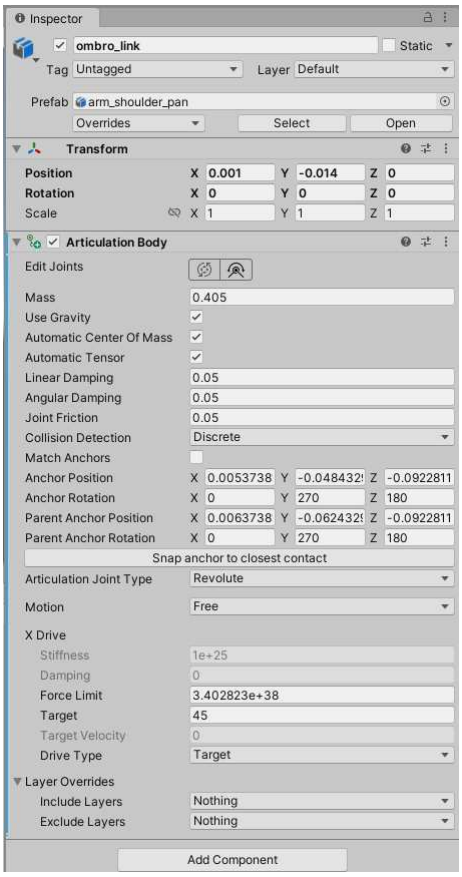
Com todos os *GameObjects* configurados, foi finalizada a parte da construção no Unity do objeto digital, apresentado na Figura 40.

Figura 38 – Configuração do componente *ArticulationBody base_link*.



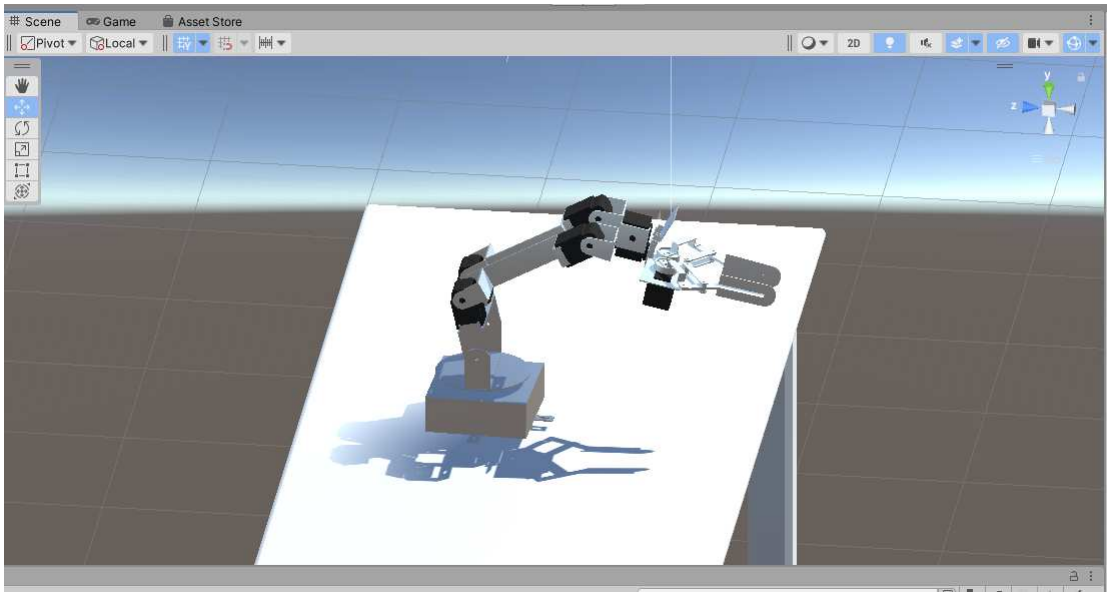
Fonte: Autoria própria.

Figura 39 – Configuração do componente *ArticulationBody ombro_link*.



Fonte: Autoria própria.

Figura 40 – Braço robótico desenvolvido no Unity.



Fonte: Autoria própria.

4.2.2 Cliente OPC UA

De forma análoga ao braço robótico físico, para a realização da troca de informções com o servidor, foi desenvolvido um cliente OPC UA, utilizando a SDK da OPC *Fundation* disponibilizados no GITHUB da OPC Foundation (2016).

A primeira etapa foi a declaração e a inicialização de uma instância da classe *ApplicationConfiguration* que realiza as configurações do cliente, e a criação de uma sessão a qual é usada para gerenciar a comunicação com o servidor e definido o endereço do servidor OPC UA, essa etapa é apresentada na Figura 41.

Figura 41 – Declaração das instâncias.

```
private ApplicationConfiguration app_configuration = new ApplicationConfiguration();
private Session m_session;
public string server_address = @"opc.tcp://localhost:48030/BracoRobotico";
```

Fonte: Autoria própria.

Com as instanciações feitas, foi desenvolvido o método *OpcUa_Client_Configuration* do tipo *ApplicationConfiguration*, que configura a aplicação cliente OPC UA, incluindo a inicialização de certificados de segurança, configuração de transporte e tempo limite, além da configuração do validador de certificados para aceitar certificados não confiáveis automaticamente. O código da classe *OpcUa_Client_Configuration* é apresentado na Figura 42.

A próxima etapa foi a criação da sessão *OpcUa_Create_Session*, utilizada para realizar as configurações do cliente e o *endpoint* do servidor, e para criar a sessão de forma

Figura 42 – Código para criação do método *OpcUa_Client_Configuration*.

```
ApplicationConfiguration OpcUa_Client_Configuration()
{
    // Configuração do Cliente OPCUa  {W/R }
    var config = new ApplicationConfiguration()
    {
        // Inicialização (Name, Uri, etc.)
        ApplicationName = "OPCUa_AS", //
        ApplicationUri = @"opc.tcp://localhost:48030",
        // Configurando o tipo da aplicação como Cliente
        ApplicationType = ApplicationType.Client,
        // Configuração de Segurança
        SecurityConfiguration = new SecurityConfiguration...,
        // Configurações de Transporte
        TransportConfigurations = new TransportConfigurationCollection(),
        // Definição de Quotas de Transporte
        TransportQuotas = new TransportQuotas { OperationTimeout = 10000 },
        // Configuração do tempo de sessão
        ClientConfiguration = new ClientConfiguration { DefaultSessionTimeout = 50000 },
        TraceConfiguration = new TraceConfiguration()
    };
    config.Validate(ApplicationType.Client).GetAwaiter().GetResult();
    // Configuração do validador de certificado para aceitar automaticamente certificados não confiáveis
    if (config.SecurityConfiguration.AutoAcceptUntrustedCertificates)
    {
        config.CertificateValidator.CertificateValidation += (s, e) => { e.Accept = (e.Error.StatusCode == StatusCodes.BadCertificateUntrusted); };
    }
    // Instância da Aplicação
    var application = new ApplicationInstance
    {
        ApplicationName = "OPCUa_AS",
        ApplicationType = ApplicationType.Client,
        ApplicationConfiguration = config
    };
    return config;// Retorna a configuração do cliente
}
```

Fonte: Autoria própria.

assincrona foi utilizado o método *GetAwaiter().GetResult()*. O código para a criação da sessão *OpcUa_Create_Session* é apresentado na Figura 43.

Figura 43 – Código para criação da sessão *OpcUa_Create_Session*.

```
1 referência
Session OpcUa_Create_Session(ApplicationConfiguration client_configuration, EndpointDescription client_end_point)
{
    return Session.Create(
        client_configuration,
        new ConfiguredEndpoint(
            null,
            client_end_point,
            EndpointConfiguration.Create(client_configuration)
        ),
        false,
        "",
        10000,
        null,
        null
    ).GetAwaiter().GetResult();
}
```

Fonte: Autoria própria.

Por fim, foi criado o método *opcua_Inicia()* o qual coordena a configuração do cliente e a criação da sessão, encapsulando todo o processo de inicialização. Nele, a configuração do cliente é obtida através do método *OpcUa_Client_Configuration()*, e o endpoint do servidor é selecionado utilizando *CoreClientUtils.SelectEndpoint*. Em seguida, a sessão é criada chamando *OpcUa_Create_Session()*. O método *opcua_Inicia()* é inicializado assim que a aplicação é executada. O código para a criação do método *opcua_Inicia()* é apresentado na Figura 44.

Figura 44 – Código para criação do método *opcua_Inicia*.

```
1 referência
void opcua_Inicia()
{
    try
    {
        app_configuration = OpcUa_Client_Configuration();
        EndpointDescription end_point = CoreClientUtils.SelectEndpoint(server_address, useSecurity: false);
        m_session = OpcUa_Create_Session(app_configuration, end_point);

    }
    catch (Exception e)
    {
        Console.WriteLine("Problema de Comunicação: {0}", e);
    }
}
```

Fonte: Autoria própria.

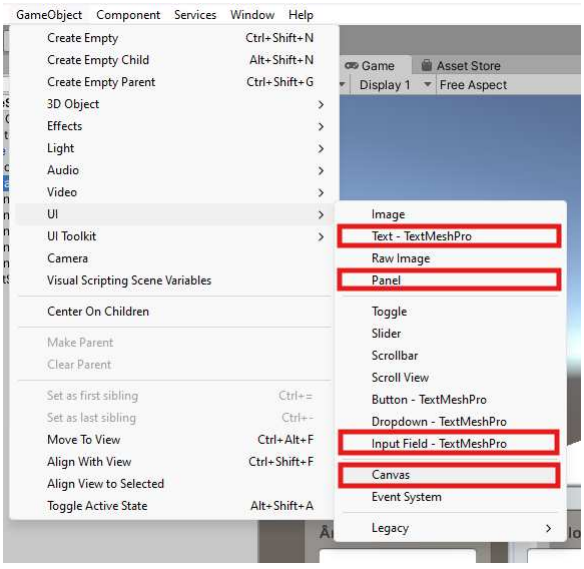
4.2.3 Interface Gráfica

Para a manipulação do braço robótico por meio do Unity, foi desenvolvida a interface gráfica. No Unity, é possível criar uma interface usando elementos pré-configurados, acessando o menu *GameObject* e selecionando a opção *UI*. Os elementos utilizados na criação do trabalho são os destacados na Figura 45.

O *Canvas* é uma área onde todos os outros elementos *UI* devem, obrigatoriamente, ser inseridos e organizados. O *Panel* é um contêiner para outros elementos *UI*, contendo uma área de fundo personalizável, utilizada para agrupar e organizar outros elementos, como botões, textos e imagens.

O *Input Field* é um campo de entrada de texto onde os usuários inserem informações, e o *Text* é um componente de exibição de texto. Ambos, *Input Field* e *Text*, utilizam a tecnologia *TextMeshPro* para renderizar texto na tela.

Figura 45 – Componentes de interação do Unity.

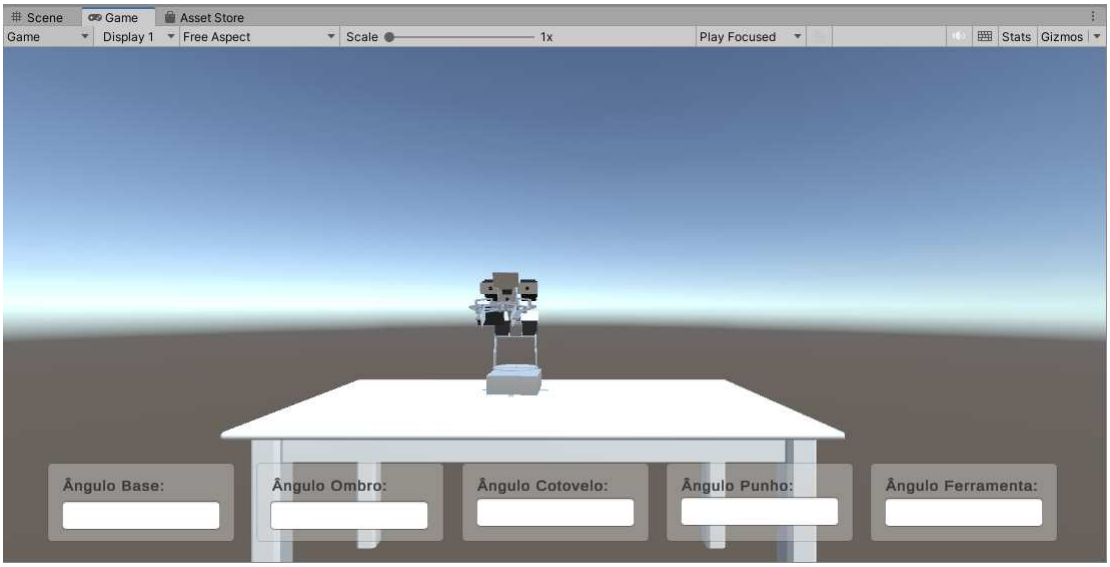


Fonte: Autoria própria.

Os elementos da interface de usário foram organizados conforme apresentado na

Figura 46. Foram utilizados cinco *Panel* para agrupar os componentes *Input Field* e *Text* de cada junta.

Figura 46 – Interface gráfica no Unity.



Fonte: Autoria própria.

Para realizar a manipulação das entradas do usuário é necessário obter os componentes *TMP_InputField* da interface, conforme apresentado na Figura 47 e implementar 2 métodos no código principal da aplicação, o *Controller*.

Figura 47 – Obtenção dos *TMP_InputField* para cada articulação do braço robótico.

```
// Obtém os componentes TMP_InputField do próprio GameObject
base_InputField = GameObject.Find("baseInp").GetComponent<TMP_InputField>();
ombro_InputField = GameObject.Find("ombroInp").GetComponent<TMP_InputField>();
cotovelo_InputField = GameObject.Find("cotoveloInp").GetComponent<TMP_InputField>();
punho_InputField = GameObject.Find("punhoInp").GetComponent<TMP_InputField>();
ferramenta_InputField = GameObject.Find("ferramentaInp").GetComponent<TMP_InputField>();
```

Fonte: Autoria própria.

Os métodos implementados são apresentados na Figura 48. Em *ReadInput()* é realizada a conversão da *string* enviada pelo usuario para um *double*, que é o tipo de dado do servidor, e caso a conversão seja bem-sucedida, o valor é escrito no servidor utilizando o método *writeValue*. Já o método *attModifacaoInputField()* adiciona *listeners* aos campos de entrada específicos das juntas, e quando o usuário finaliza a edição do *TMP_InputField* na interface, é acionado o método *ReadInput()* com o valor e o identificador do nó correspondente.

4.2.4 Código de controle do braço robótico

O código *Controller* é o principal da aplicação do objeto digital, devendo ser adicionado ao objeto pai do braço robótico, como apresentado na Figura 49. Assim,

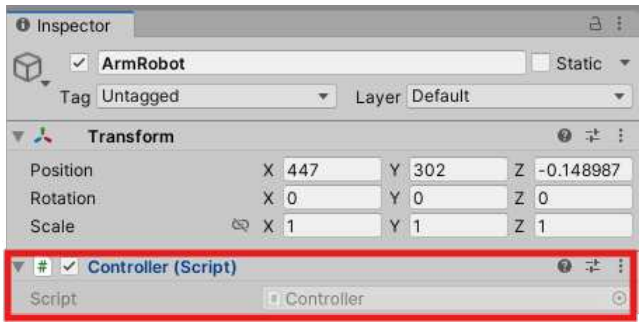
Figura 48 – código para manipulação dos valores obtidos a partir do componente *TMP_InputField*.

```
public void ReadInput(string s, string nodeId)
{

    double value;
    if (double.TryParse(s, out value))
    {
        Debug.Log($"valor convertido: {value}");
        WriteValue(nodeId, value);
    }
    else
    {
        Debug.LogError("ENTRADA INVALIDA, DIGITE UM NUMERO!!");
    }
}
1 referência
public void attModifacaoInputField()
{
    base_InputField.onEndEdit.AddListener(delegate { ReadInput(base_InputField.text, "ns=2;i=6002"); });
    ombro_InputField.onEndEdit.AddListener(delegate { ReadInput(ombro_InputField.text, "ns=2;i=6019"); });
    cotovelo_InputField.onEndEdit.AddListener(delegate { ReadInput(cotovelo_InputField.text, "ns=2;i=6024"); });
    punho_InputField.onEndEdit.AddListener(delegate { ReadInput(punho_InputField.text, "ns=2;i=6029"); });
    ferramenta_InputField.onEndEdit.AddListener(delegate { ReadInput(ferramenta_InputField.text, "ns=2;i=6034"); });
}
```

Fonte: Autoria própria.

quando a aplicação for inicializada, ele também será inicializado, adicionando o código *JointControll* em cada componente de junta do braço, ou seja, em cada *GameObject* filho, conforme mostrado na Figura 50. Nele, também é realizada toda a construção do cliente OPC UA e a manipulação dos componentes de entrada, *TMP_InputField*, cujo funcionamento já foi explicado anteriormente.

Figura 49 – código *Controller* incluído no *GameObject* Pai.



Fonte: Autoria própria.

Utilizando o método padrão do Unity, o *Update()*, é realizada a leitura periódica das variáveis do servidor. O método *Update()* é chamado a cada quadro da aplicação, o que permite verificar continuamente o estado das variáveis do servidor. Essas variáveis são comparadas com valores auxiliares para detectar qualquer mudança no ângulo de rotação do braço. Caso uma mudança seja detectada, o método *UpdateDirection()* é acionado com o novo valor de rotação. O método *Update()* é apresentado na Figura 51.

O método *UpdateDirection()* verifica a direção e o ângulo de rotação e aciona a articulação correspondente, utilizando seu índice pré-definido e atualizando o sentido da

Figura 50 – Trecho do código *Controller* que adiciona o componente *JointControll* nos *GameObjects* filhos.

```
// Um array articulationChain é atribuído aos componentes ArticulationBody do GameObject atual e de seus filhos.
articulationChain = this.GetComponentsInChildren<ArticulationBody>();
foreach (ArticulationBody joint in articulationChain) // Para cada junta (ArticulationBody) em articulationChain:
{
    joint.gameObject.AddComponent<JointControl>(); // Um componente JointControl é adicionado ao GameObject da junta.
    //As propriedades jointFriction e angularDamping da junta são definidas como defDynamicVal (que é 10 neste caso).
    joint.jointFriction = defDyanmicVal;
    joint.angularDamping = defDyanmicVal;
    // O xDrive da junta é recuperado, sua propriedade forceLimit é definida com o valor de forceLimit
    ArticulationDrive currentDrive = joint.xDrive;
    currentDrive.forceLimit = 20;
    //  o xDrive modificado é atribuído de volta à junta.
    joint.xDrive = currentDrive;
}
```

Fonte: Autoria própria.

rotação e o valor da angulação. O método *UpdateDirection()* é apresentado na Figura 52.

4.2.5 código de controle das articulações

No código *JointControl*, é realizado o controle de uma articulação específica do braço robótico. Ele recebe os parâmetros de controle definidos no *Controller* e os aplica à articulação, ajustando dinamicamente sua posição com base no ângulo e na direção de rotação especificados. Essa implementação assegura o movimento preciso e restrito das articulações, viabilizando a simulação de comportamentos realistas do braço robótico no ambiente virtual do Unity.

4.3 Canal de Comunicação

Para a criação do canal de comunicação, foi utilizado o protocolo OPC UA. Para o desenvolvimento do servidor OPC UA, é necessária a criação de um modelo de informações do sistema. A seguir, será apresentado como foram desenvolvidos tanto o modelo de informações quanto o servidor OPC UA.

4.3.1 Modelo de Informação

O modelo de informação foi desenvolvido usando a aplicação UaModeler, disponibilizada pela *Unified Automation*. Após concluir a construção do modelo, pode ser realizada a sua exportação para um arquivo XML contendo o *nodeset*. A escolha do UaModeler se deve à sua simplicidade na criação do modelo de informação.

Para a construção do modelo de informação do braço robótico, primeiramente foram levantadas as variáveis de interesse para manipulação, que afetariam o movimento por completo. As variáveis que podem ser manipuladas em cada junta, tanto no braço físico quanto no digital, são: aceleração, velocidade, torque, direção e ângulo de rotação.

Figura 51 – Método *Update()* do código *Controller*.

```
void Update() // metodo chamado a cada quadro de atualização do unity
{
    NovasLeituraVariaveis();
    //--------------------------base---------------------------------
    if (Angle_base != newAngle_base)
    {
        Angle_base = m_session.ReadValue(NodeId.Parse("ns=2;i=6002")).ToString();
        Angle_basef = float.Parse(Angle_base);
        UpdateDirection(1, Angle_basef);
    }
    //-----------------------------Cotovelo--------------------------
    if (Angulo_Cotovelo != newAngulo_Cotovelo)
    {
        Angulo_Cotovelo = m_session.ReadValue(NodeId.Parse("ns=2;i=6024")).ToString();
        Angulo_Cotovelof = float.Parse(Angulo_Cotovelo);
        UpdateDirection(3, Angulo_Cotovelof);
    }
    //-----------------------------Ferramenta-----------------------------
    if (Angulo_Ferramenta != newAngulo_Ferramenta)
    {
        Angulo_Ferramenta = m_session.ReadValue(NodeId.Parse("ns=2;i=6034")).ToString();
        Angulo_Ferramentaf = float.Parse(Angulo_Ferramenta);
        UpdateDirection(5, Angulo_Ferramentaf);
        if (Angulo_Ferramentaf == 0)
        {
            float open1 = 0.0f;
            // Angulo_Ferramentaf2 = open1;
            UpdateDirection(6, open1);
        }
        else
        {
            float open1 = 45.0f;
            //Angulo_Ferramentaf2 = open1;
            UpdateDirection(6, open1);
        }
    }
    //-----------------------------Ombros----------------------
    if (Angulo_Ombros != newAngulo_Ombros)
    {
        Angulo_Ombros = m_session.ReadValue(NodeId.Parse("ns=2;i=6019")).ToString();
        Angulo_Ombrosf = float.Parse(Angulo_Ombros);
        UpdateDirection(2, Angulo_Ombrosf);
    }
    //----------------------Punho-----------------------------
    if (Angulo_Punho != newAngulo_Punho)
    {
        Angulo_Punho = m_session.ReadValue(NodeId.Parse("ns=2;i=6029")).ToString();
        Angulo_Punhof = float.Parse(Angulo_Punho);
        UpdateDirection(4, Angulo_Punhof);
    }
}
```

Fonte: Autoria própria.

Além disso, conforme apresentado na Figura 37, o braço robótico possui cinco juntas, logo, para a construção do modelo de informação foi preciso definir dois tipo base de objetos, *BaseObjectTypes*, no UaModeler.

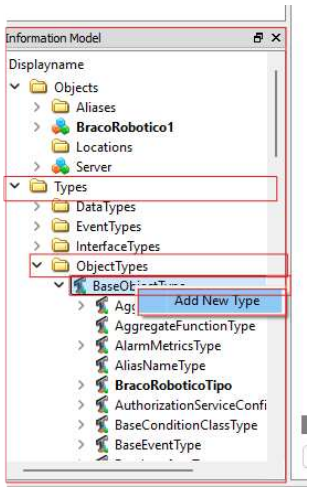
Os *BaseObjectTypes* são criados, conforme apresentado na Figura 53, indo na aba *Information Model*, pressionando o botão direito do *mouse* sobre o item *BaseObjectTypes*, que está dentro da pasta *ObjectTypes*, a qual está inserida na pasta *Types*.

Figura 52 – Método *UpdateDirection()* do código *Controller*.

```
private void UpdateDirection(int jointIndex, float Target)
{

    if (jointIndex < 0 || jointIndex >= articulationChain.Length)
    {
        return;
    }

    float moveDirection = Target;
    JointControl current = articulationChain[jointIndex].GetComponent<JointControl>();

    if (previousIndex != jointIndex)
    {
        JointControl previous = articulationChain[previousIndex].GetComponent<JointControl>();
        previous.direction = RotationDirection.None;
        previousIndex = jointIndex;
    }

    if (current.controltype != control)
    {
        UpdateControlType(current);
    }

    if (moveDirection != 0)
    {
        current.direction = moveDirection > 0 ? RotationDirection.Positive : RotationDirection.Negative;
        current.angle = Math.Abs(moveDirection);
    }
    else
    {
        current.direction = RotationDirection.None;
        current.angle = 0;
    }

}
```

Fonte: Autoria própria.

Figura 53 – Criação de *BaseObjectTypes*.

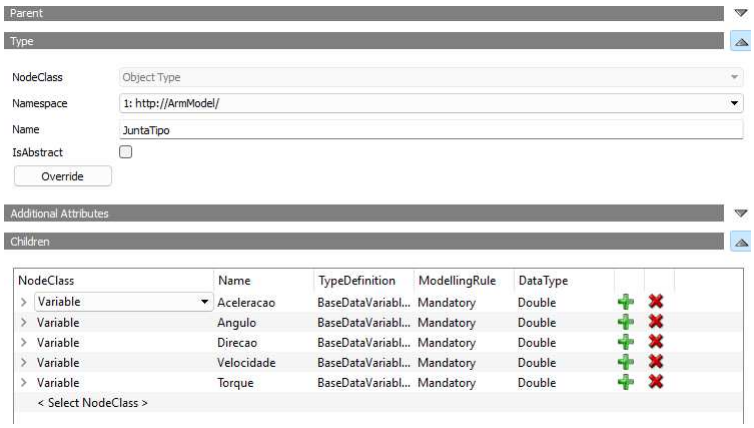


Fonte: Autoria própria.

Assim, com o levantamento das variáveis, foi definido o *BaseObjectTypes* no UaModeler, *JuntaTipo*, com cinco variáveis de interesse, que são do tipo *double* e componentes

do objeto definido. Na Figura 54, é apresentada a tela do UaModeler onde foi construído o tipo de objeto *JuntaTipo*.

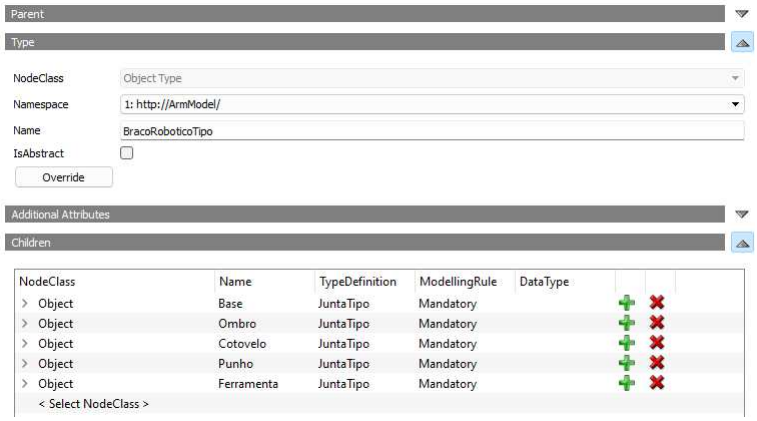
Figura 54 – Tela do UaModeler para criação do objeto base tipo *JuntaTipo*.



Fonte: Autoria própria.

Após a definição do tipo *JuntaTipo*, foi construído o tipo *BracoRoboticoTipo*, e para isso, foram instanciados cinco objetos do tipo *JuntaTipo*. Na Figura 55, é apresentada a tela do UaModeler onde foi construído o tipo base de objeto *BracoRoboticoTipo*. Nela, são mostrados os cinco objetos, sendo eles: base, ombro, cotovelo, punho e ferramenta, que são do tipo *JuntaTipo*.

Figura 55 – Tela do UaModeler para criação do objeto base tipo *BracoRoboticoTipo*.

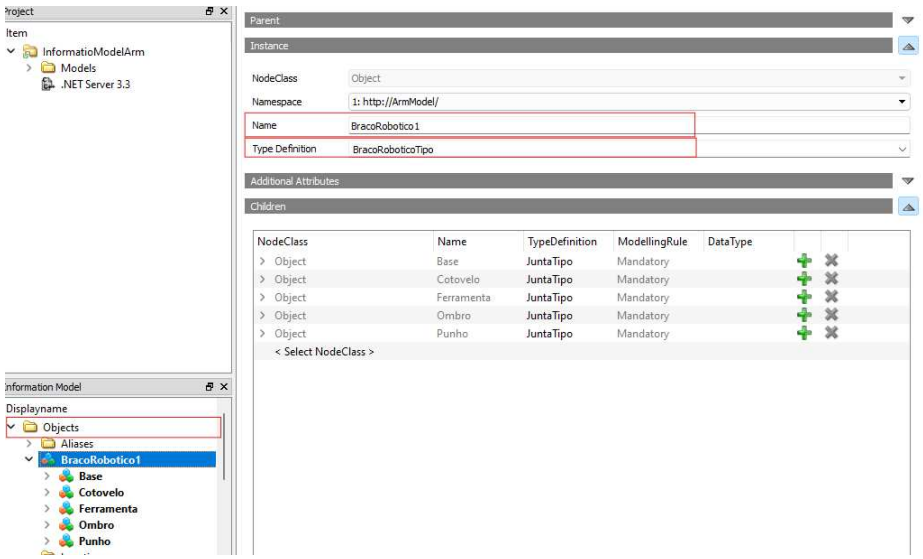


Fonte: Autoria própria.

Finalizadas as definições dos *BaseObjectTypes*, foi criado um objeto do tipo *BracoRoboticoTipo*, chamado de *BracoRobotico1*. Na Figura 56, é apresentada a tela do UaModeler onde foi construído o objeto *BracoRobotico1*.

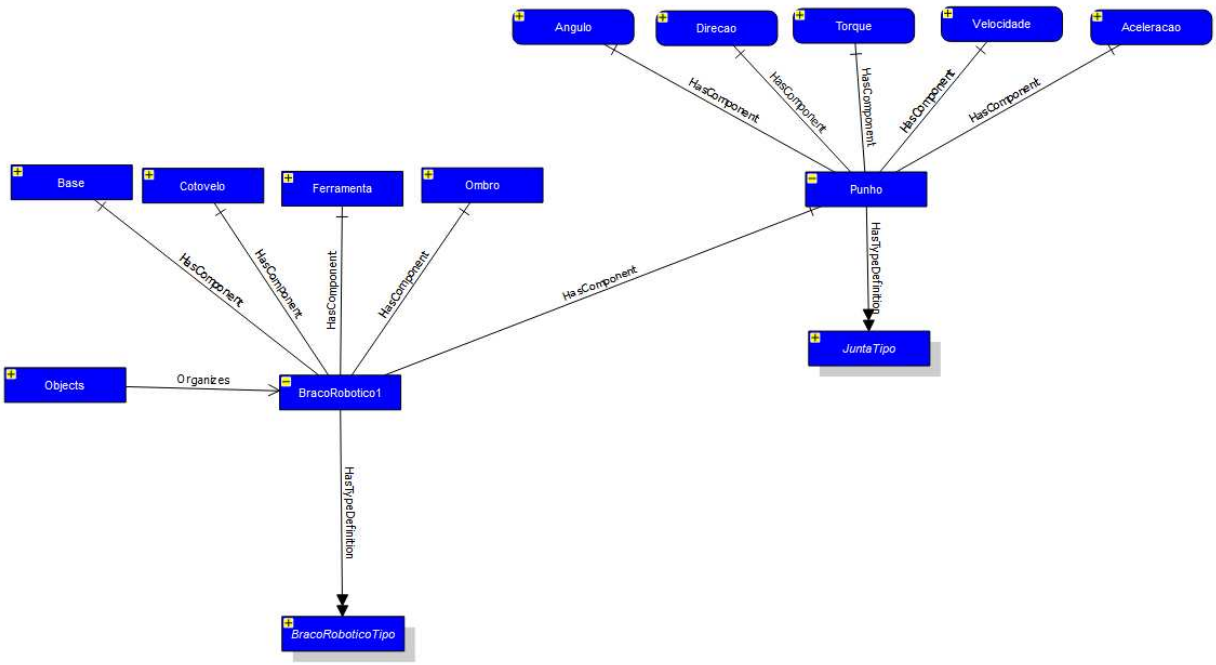
No diagrama do modelo de informação do braço robótico, apresentado na Figura 57, o objeto *BracoRobotico1* está organizado dentro da pasta *Objects*. Este objeto é definido pelo tipo *BracoRoboticoTipo* e, por esse motivo, herda seus componentes do tipo *JuntaTipo*. Cada uma dessas juntas herda as cinco variáveis definidas no seu tipo.

Figura 56 – Tela do UaModeler para criação do objeto *BracoRobotico1*.



Fonte: Autoria própria.

Figura 57 – Diagrama do modelo de informação do objeto *BracoRobotico1*.



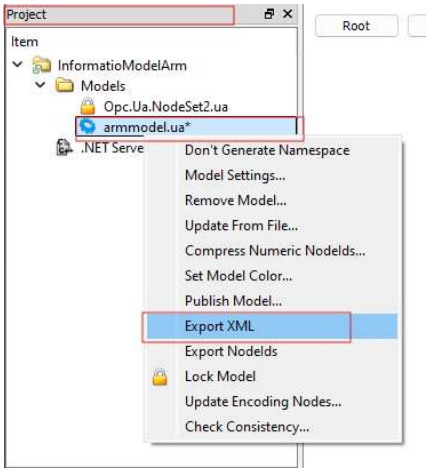
Fonte: Autoria própria.

Por fim, na aba *Project* do UaModeler, o modelo de informação foi exportado para um arquivo XML, para ser usado na criação do servidor.

4.3.2 Servidor OPC UA

O servidor OPC UA do sistema foi desenvolvido na linguagem C#, utilizando a IDE do Visual Studio, como um aplicativo .NET. Entretanto, antes da criação do servidor,

Figura 58 – Tela do UaModeler para exportação do modelo.



Fonte: Autoria própria.

é preciso, a partir do modelo de informação exportado do UaModeler, gerar os arquivos que irão configurar os nós do servidor. Para esse objetivo, foi utilizada a aplicação UA-ModelCompiler, disponibilizada no GitHub da OPCFoundation (2019). Com o repositório clonado no computador, é feita a comilação da aplicação pelo Visual Studio. Após uma compilação bem-sucedida, o arquivo gerado é executado na linha de comando, conforme apresentado na Figura 59. A aplicação compila o modelo de informação, *armmodel.xml*, e gera os arquivos para contrução do servidor e armazena na pasta *Models*. Os arquivos gerados são apresentados na Figura 60.

Figura 59 – Comando do UA-ModelCompiler.

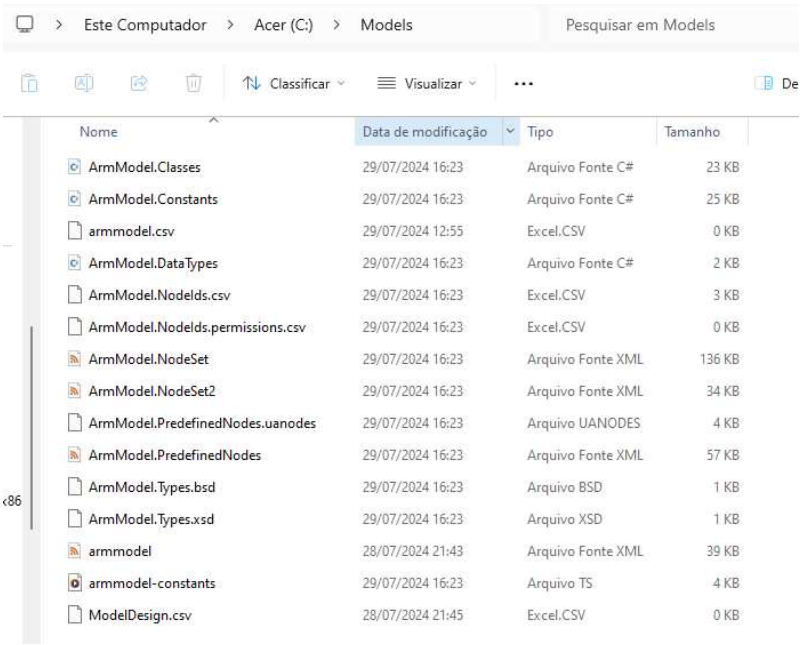
```
C:\UA-ModelCompiler\build\bin\Debug\net8.0>.\Opc.Ua.ModelCompiler.exe compile -d2 C:\Models\armmodel.x
ml -cg C:\Models\armmodel.csv -o2 C:\Models -version v104
```

Fonte: Autoria própria.

Após a geração dos arquivos de configuração do modelo de informação para a criação do servidor OPC UA, um novo projeto com o nome *ArmModel*, do tipo de aplicativo Windows Forms (.NET Framework), foi criado no Visual Studio. A tela do Visual Studio para a criação do projeto é apresentada na Figura 61. Ademais, foi realizada, por meio do gerenciador de pacotes NuGet, a inclusão do pacote OPC UA desenvolvido pela OPC UA *Fundation*.

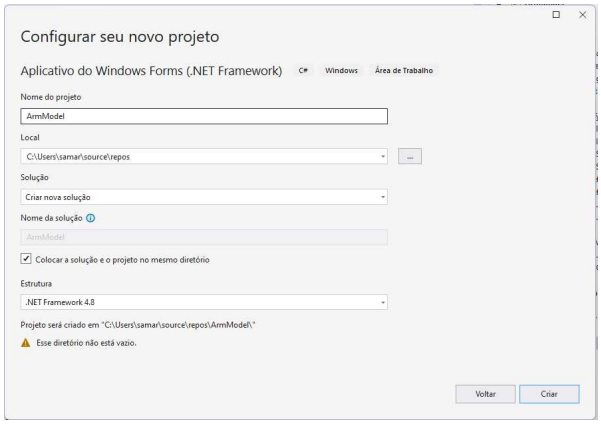
Finalizada a etapa de criação e configuração da aplicação, os arquivos gerados a partir do modelo de informção foram copiados para a raiz do projeto. Também foi construído um modelo XML para configurar o servidor, especificando a porta, o tipo de acesso, o *endpoint* e o nome do servidor. Também é definido nele os detalhes do certificado, como o tipo e o local de armazenamento, assim como as políticas de comunicação, taxas de transmissão e outros parâmetros relevantes. Esse arquivo também foi copiado para a raiz do projeto.

Figura 60 – Arquivos gerados pelo UA-ModelCompiler.

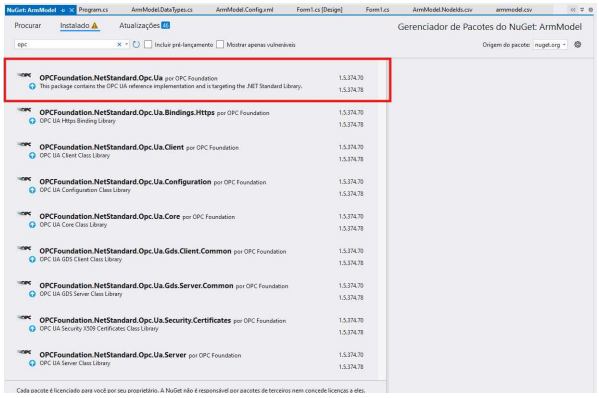


Fonte: Autoria própria.

Figura 61 – Criação e preparação da aplicação para o desenvolvimento do servidor.



(a) Criação de projeto no Visual Studio.



(b) Inclusão do pacote OPC UA no Visual Studio.

Fonte: Autoria própria.

Por fim, foram desenvolvidos no projeto os seguintes arquivos:

- *Program.cs*, o ponto de entrada da aplicação, onde é realizada a inicialização da aplicação, a inicialização do ambiente gráfico e a configuração da aplicação como um servidor, a partir do arquivo XML montado.
- *ArmModelServer.cs*, é responsável pela configuração e a inicialização do servidor OPC UA, é criado e gerenciado os gerenciadores de nós(objetos), *ArmModelNodeManager.cs*, do servidor. Além disso, nele é definido as propriedades do servidor, como

o nome do fabricante, o nome do produto, a URI do produto, a versão do *software*, o número da *build* e a data da *build*.

- *ArmModelNodeManager.cs*, onde é realizado o gerenciamento dos nós do servidor, definindo os *NodeId*, os *namespaces* e carregamento das configurações específicas do servidor.
- *ArmModelServerConfiguration.cs*, onde é armazenada a configuração do servidor, e é permitido que a configuração seja convertida para e a partir de um formato que pode ser salvo e carregado, ou seja, a configuração pode ser serializada e desserializada.

Por fim, foi desenvolvida a interface gráfica, apresentada Na Figura 62, no Windows Forms, com o intuito de facilitar a visualização do *endpoint* do servidor e ela é configurada pelo código *Forms1.cs*.

Figura 62 – Interface gráfica do servidor.

Fonte: Autoria própria.

4.4 Controle com IEC 61499

Para a construção da aplicação de controle distribuído com IEC 61499, foi utilizado o 4diac IDE, nele foi desenvolvido um sistema que lê e escreve no servidor o valor de rotação que determinada junta terá que realizar. Para isso, inicialmente foi desenvolvida uma aplicação utilizando blocos de função. A cada 1 segundo, a aplicação lê os dados do servidor e atualiza a saída correspondente. Se o usuário desejar modificar a angulação de alguma junta para um valor específico, ele pode fornecer o ângulo de rotação desejado em um bloco de função, e a aplicação atualizará o servidor com o novo valor.

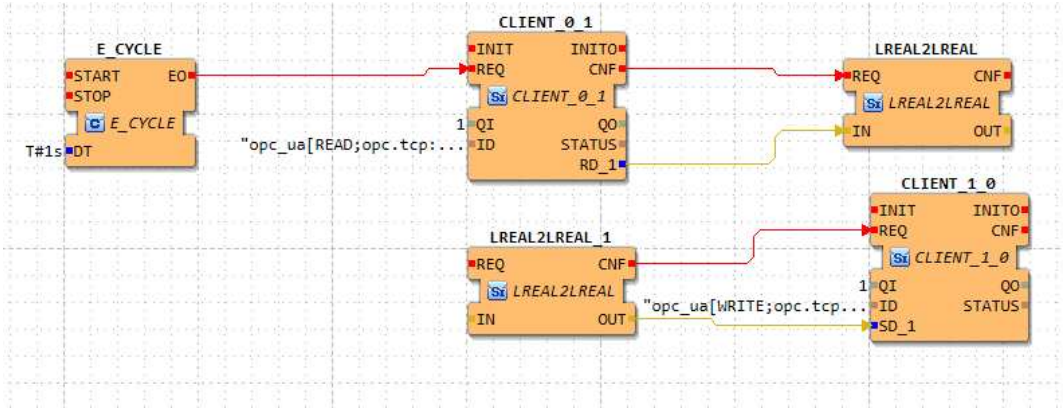
Na Figura 63 é apresentada a primeira versão da aplicação contruida no 4diac IDE, para a leitura e escrita de apenas uma variavel de uma junta. Nela, são apresentados cinco blocos de funções, sendo eles:

- *E_CYCLE*, é um temporizador e a cada segundo, é disparado um evento chamado *EO*.
- *CLIENT_0_1*, é um cliente que se conecta ao servidor *OPC UA* por meio de um *ID* fornecido, onde o *ID* contém o endereço da variável de interesse. No *ID* é feita a

determinação se a operação será de leitura (quando "*READ*") ou escrita (quando "*WRITE*"). Portanto, nesse bloco será realizada a leitura do servidor a cada evento que é disparado do *E_CYCLE*.

- *LREAL2LREAL*, é um bloco de função de conversão, que converte o número lido para um tipo *Long Real*.
- *CLIENT_1_0*, também é um cliente que se conecta ao servidor *OPC UA* por meio de um *ID*. No entanto ele é um cliente que realiza a escrita da variável especificada no servidor *OPC UA*.

Figura 63 – Aplicação IEC 61499 no 4diac para leitura e escrita em um servidor OPC UA.



Fonte: Autoria própria.

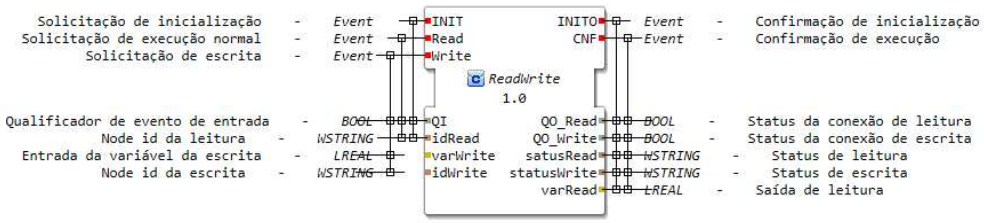
Logo, o funcionamento da aplicação consiste em que, a cada segundo, o bloco de função *CLIENT_0_1* realiza a leitura dos dados disponíveis no servidor e os disponibiliza. Caso o usuário deseje escrever um determinado valor para uma junta específica, ele deve inserir o número desejado em *IN* do bloco de função *LREAL2LREAL_1* e disparar o evento *REQ*.

Entretanto, utilizar quatro blocos de funções para realizar a escrita e leitura de apenas uma variável de interesse torna a visualização dessa aplicação muito complicada para o usuário. Por esse motivo, se fez necessária a construção de um bloco de função composto.

Tanto a Norma IEC 61499 quanto o 4diac fornecem suporte para a construção de blocos de funções a partir de outros blocos de funções, os chamados blocos de funções compostos. Para essa aplicação, foi construído um bloco de função com o nome *ReadWrite*, apresentado na Figura 64, que é composto pelos quatro blocos de funções apresentados na Figura 63.

Na Figura 65, é apresentada a construção da rede dos blocos de funções para a contrução do bloco de função composto. A lógica explicada anteriormente foi repetida

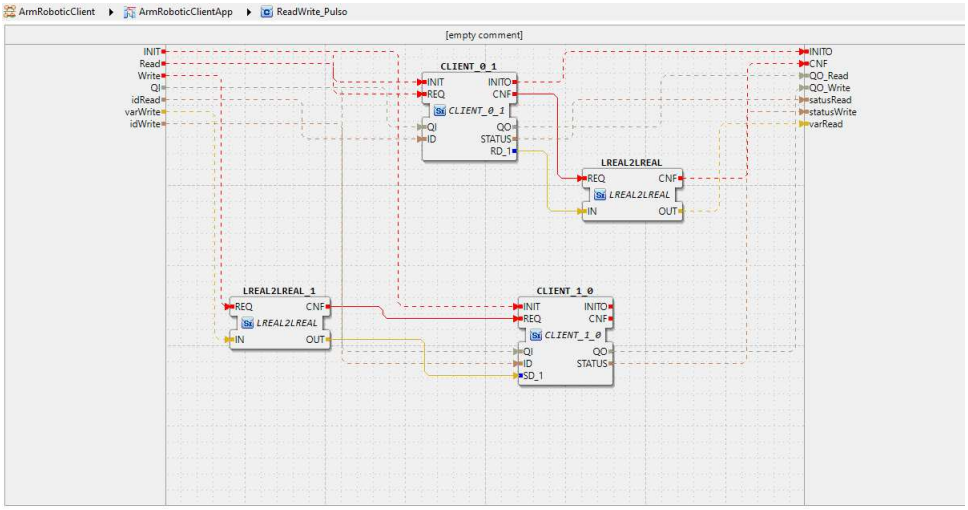
Figura 64 – Interface do bloco de função *ReadWrite*.



Fonte: Autoria própria.

para cada junta, ou seja, o bloco de função composto terá a mesma funcionalidade que o conjunto de blocos de funções descrito anteriormente.

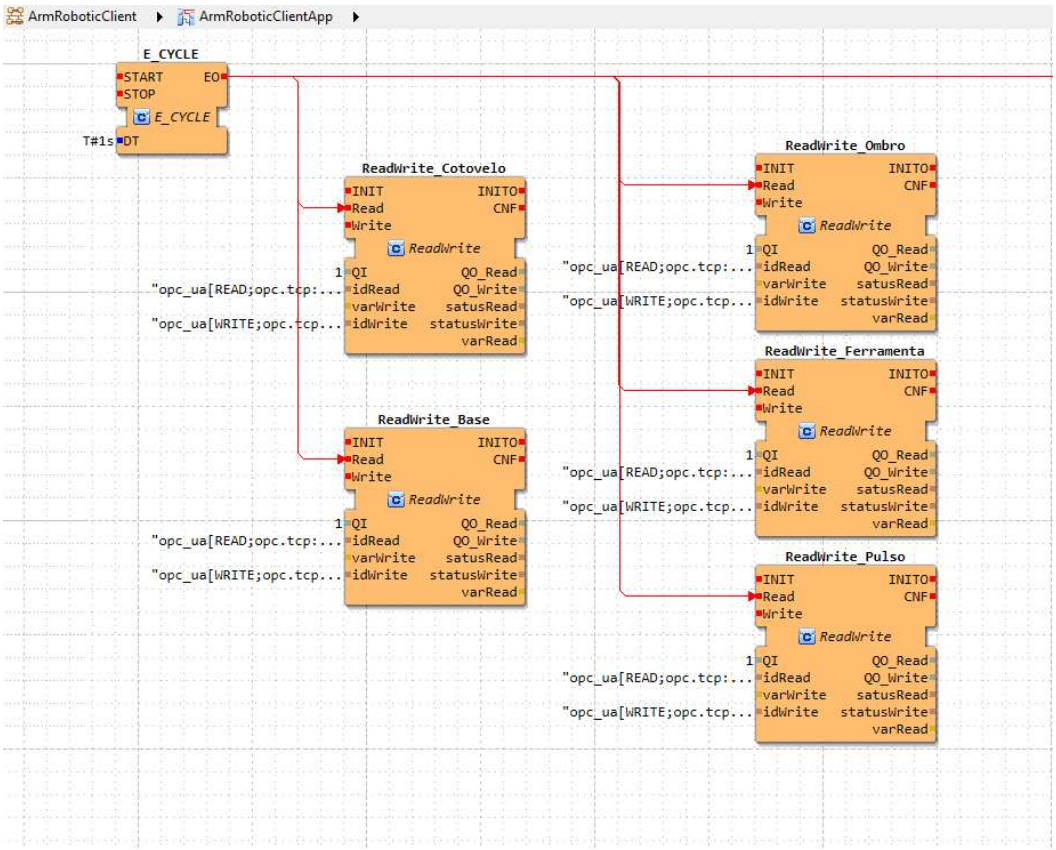
Figura 65 – Rede dos blocos de funções para a contrução do bloco de função *ReadWrite*.



Fonte: Autoria própria.

Com o desenvolvimento do bloco finalizado, aplicação geral foi construída para as cinco juntas no 4diac. Nesse sentido, foi utilizado um FB *ReadWrite* para realizar a leitura e escrita do valores de rotação para as juntas da base, do ombro, do punho, do cotovelo e da ferramenta. A aplicação desenvolvida é apresentada na Figura 66.

Figura 66 – Aplicação IEC 61499 no 4diac para leitura e para escrita dos ângulos das juntas.



Fonte: Autoria própria.

5 Resultados

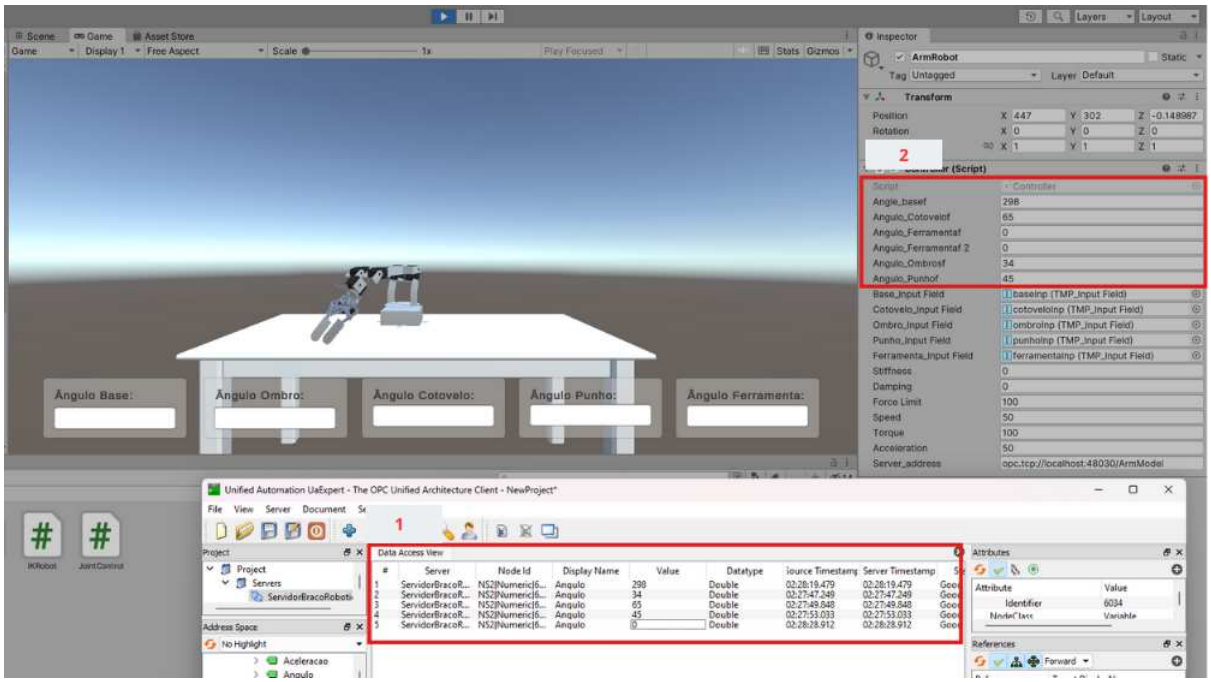
Como resultado do desenvolvimento apresentado no capítulo anterior, foi possível alcançar os objetivos específicos definidos no início deste trabalho. Foi realizada a construção de um gêmeo digital utilizando o padrão de comunicação OPC UA, permitindo a implementação de projetos de automação e controle distribuído. A apresentação dos resultados é dividida em quatro partes, sendo elas a validação de cada uma das aplicações e sua integração. Primeiramente, é realizada a validação da aplicação do Gêmeo Digital no Unity, depois a validação da aplicação de comunicação com o braço robítco no MATLAB, depois a validação da aplicação de controle distribuído no 4diac e, por fim, a integração das aplicações. Para as validações das aplicações, foi usado o cliente UaExpert disponibilizado pela OPC *Foundation*.

5.1 Aplicação do Modelo Digital

Para a aplicação no Unity, foram realizados dois testes. O primeiro consistiu em escrever pelo cliente UaExpert, simulando uma entrada no servidor, e observar o que acontece com o objeto digital no Unity. Na Figura 67 é apresentado o resultado dessa manipulação, onde, ao inserir valores de rotação pelo UaExpert, os valores são lidos pelo código *Controller* e é feita a rotação das articulações.

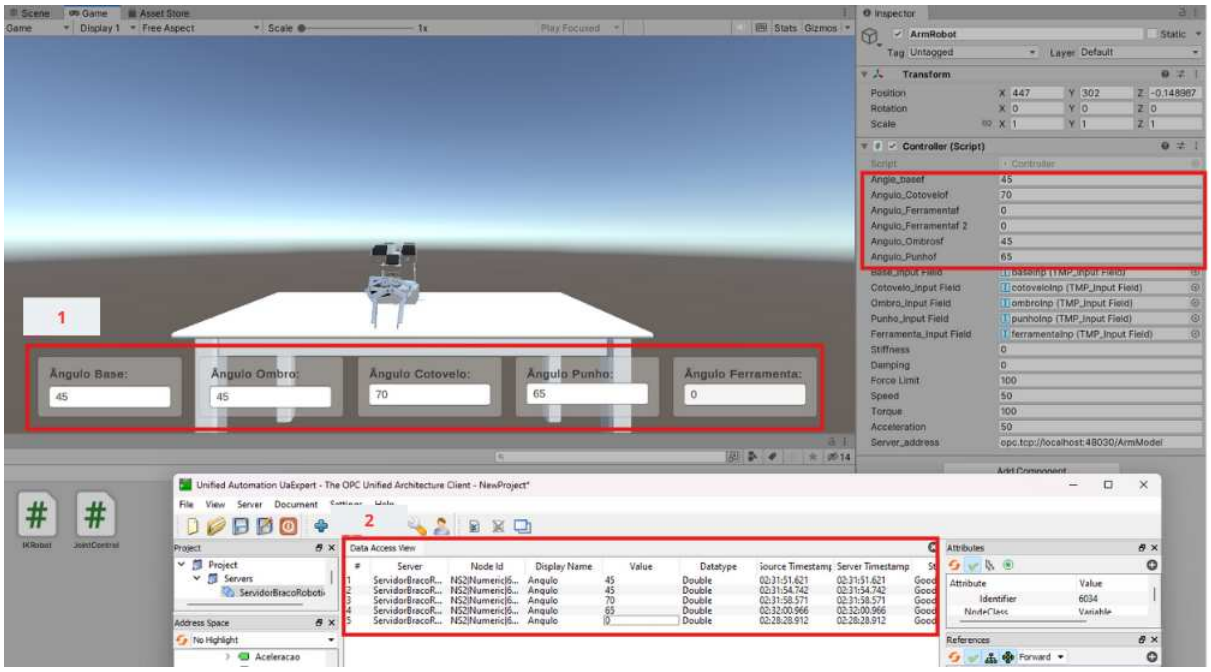
No segundo teste realizado, a manipulação do braço foi feita pela interface do Unity. Na Figura 68, é apresentado o resultado dessa operação, onde os valores escritos na interface também foram lidos pelo cliente UaExpert, que resulta na rotação das articulações.

Figura 67 – Manipulação do braço virtual no Unity, com os valores de rotação inseridos pelo UaExpert (1) e lidos pelo script Controller (2).



Fonte: Autoria própria.

Figura 68 – Manipulação do braço virtual no Unity, com os valores de rotação inseridos pela própria interface do Unity (1) e lidos pelo cliente UaExpert (2).

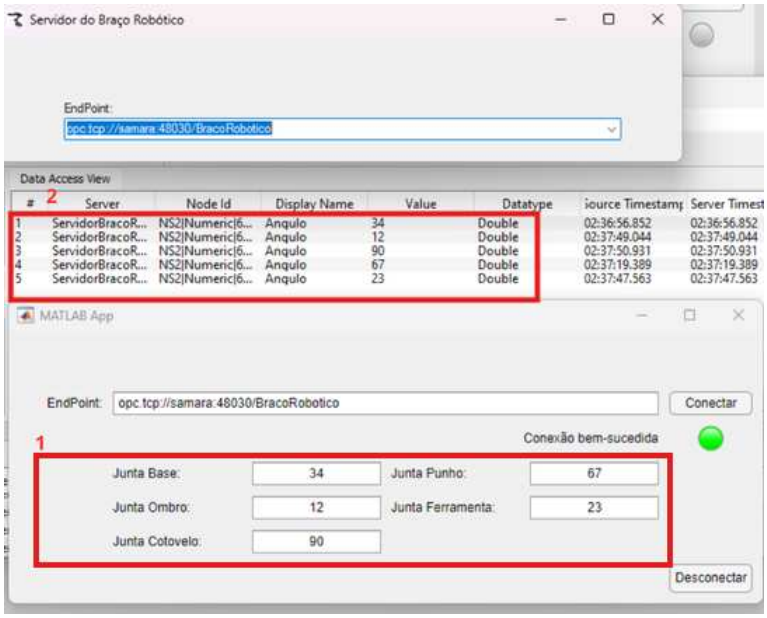


Fonte: Autoria própria.

5.2 Aplicação de comunicação com o braço robótico

Para a aplicação no MATLAB, foi realizado o teste de manipulação pela interface gráfica. Onde, um valor é inserido utilizando a interface do MATLAB e, depois é realizada a verificação no cliente UaExpert se o valor foi escrito do servidor. Foi realizado também o teste de maneira inversa, escrevendo pelo cliente do UaExpert e verificando na interface se o valor foi lido corretamente. O resultado dos testes é apresentado na Figura 69. Com esses dois teste foi feita a validação da aplicação no MATLAB.

Figura 69 – Manipulação dos ângulos das articulações na aplicação do MATLAB, com os valores de rotação inseridos pela própria interface do AppDesigner (1) e lidos pelo cliente UaExpert (2).



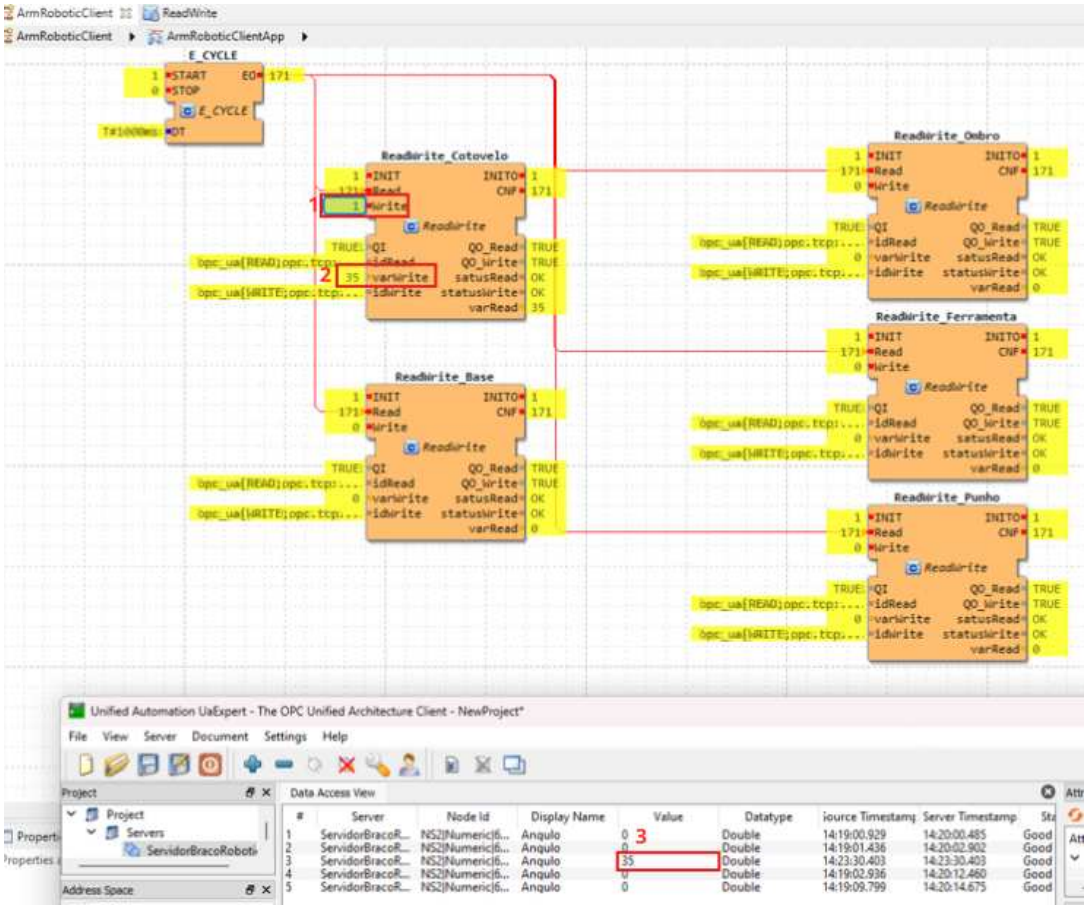
Fonte: Autoria própria.

5.3 Aplicação de controle distribuído

A aplicação de controle distribuído IEC 61499, desenvolvida no 4diac IDE, foi carregada no 4diac FORTE, que foi previamente colocado em execução no computador. Verificou-se se os valores do servidor estavam sendo lidos e se, quando acionado o evento de escrita, o dado em *varWrite* estava sendo escrito na variável. O passo a passo seguido para a realização do teste foi:

- Foi realizada a escrita do ângulo da articulação do cotovelo por meio do bloco de função *ReadWrite_Cotovelo*. Na Figura 70 é apresentado o resultado dessa operação. Onde foi inserido o valor no dado de entrada *varWrite*, e após foi acionado o evento *write* para realizar a escrita do valor no servidor. Por fim, foi verificado se o valor foi lido pelo cliente UaExpert.

Figura 70 – Manipulação dos ângulos das articulações na aplicação IEC 61499. Com os valores de rotação inseridos no bloco de função (1), escritos no servidor a partir do acionamento de um evento (2) e lidos pelo cliente UaExpert (3).



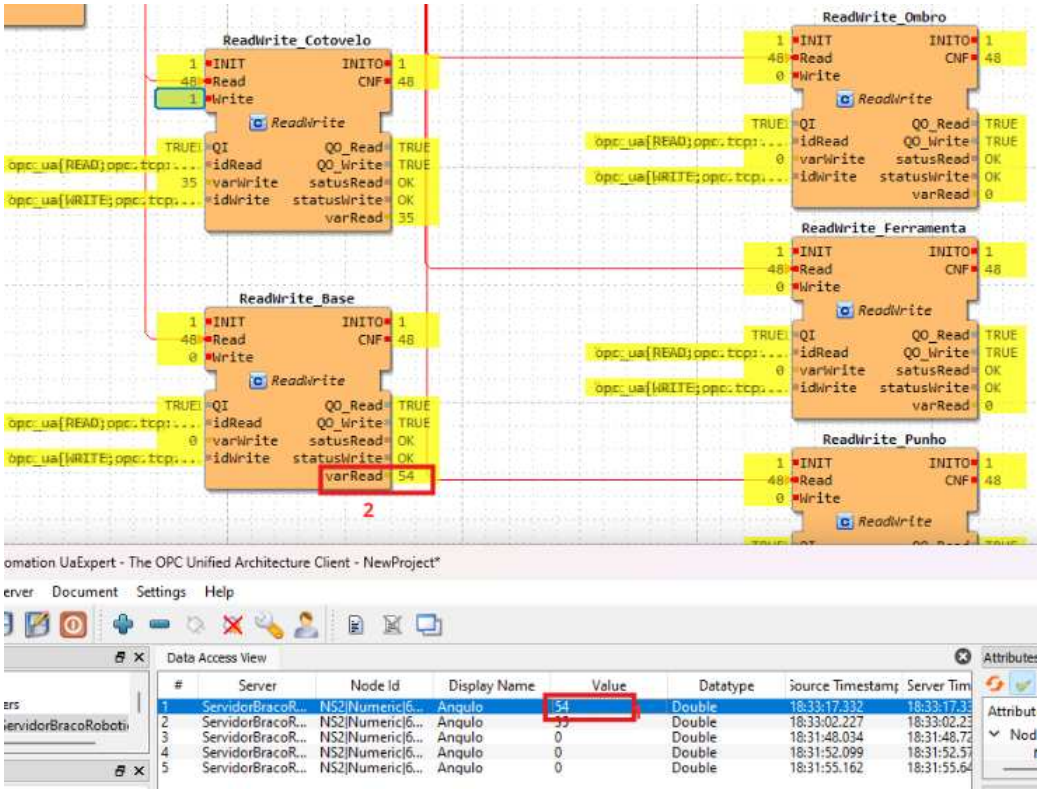
Fonte: Autoria própria.

- Para verificar se a aplicação esta realizando a leitura em tempo real do servidor OPC UA, foi escrito um valor na variável da angulação da articulação da base pelo cliente UaExpert. Essa operação é apresentada na Figura 71, em que o valor escrito (indicado pelo número 1) é apresentado no dado de saída *varRead* (indicado pelo número 2) no bloco de função *ReadWrite_Base*.

5.4 Integração das aplicações

Após testar cada aplicação de forma individual, foi realizada a validação da integração entre elas e do gêmeo digital. Para isso, os dados foram escritos pela aplicação IEC 61499, e analisou-se o comportamento nas demais aplicações e no gêmeo digital. Observou-se que os valores escritos pela aplicação 4diac foram corretamente lidos nas outras aplicações e refletidos no gêmeo digital.

Figura 71 – Manipulação dos ângulos das articulações na aplicação IEC 61499. Com os valores de rotação inseridos pelo UaExpert (1) e lidos pela aplicação IEC 61499 (2).



Fonte: Autoria própria.

6 Conclusão

Neste trabalho, foi possível alcançar os objetivos específicos definidos no início e construir um Gêmeo Digital de um braço robótico utilizando padrões abertos de automação. Para isso, foi desenvolvido um modelo virtual do braço robótico AX-12A no Unity, que espelha suas operações físicas. Também foi implementado um canal de comunicação utilizando o protocolo OPC UA e criada uma interface para o braço robótico físico, garantindo a troca de dados em tempo real e o controle de suas articulações. Além disso, foram utilizadas ferramentas de controle distribuído, como o Eclipse 4diac, para desenvolver um cliente que gerencia e coordena a operação do gêmeo digital e do braço robótico. Este desenvolvimento permitiu a implementação de um projeto de automação e controle distribuído, demonstrando a viabilidade do uso dessa tecnologia em ambientes de simulação e controle.

A aplicação desenvolvida permite que o usuário implemente o controle das articulações do braço robótico a partir de plataformas que interoperam por meio do canal de comunicação OPC UA. Além da implementação de controle, o usuário também pode utilizar o sistema como um ambiente de simulação, desde que não inicialize a aplicação do MATLAB.

Em resumo, os objetivos foram alcançados, e mostrou o potencial de uso de Gêmeos Digitais e padrões de comunicação abertos em ambientes de automação. Espera-se que este trabalho contribua para futuras pesquisas e desenvolvimentos na área, promovendo a adoção de tecnologias interoperáveis e eficientes.

6.1 Trabalhos Futuros

Levando em consideração sua aplicabilidade, para trabalhos futuros sugere-se a implementação de um sistema de controle para cada articulação do manipulador robótico, bem como o desenvolvimento de um sistema de controle de trajetória para o objeto físico e o objeto digital. O controle poderia ser implementado no 4diac, enquanto o controle de trajetória poderia ser desenvolvido no MATLAB e no ROS.

Referências

ADAM, T. et al. A comprehensive review of digital twin — part 1: modeling and twinning enabling technologies. *Structural and Multidisciplinary Optimization*, Springer Nature, v. 65, n. 354, 2022. Citado 2 vezes nas páginas 15 e 21.

AGUIAR, M.; MURGATROYD, I.; EDWARDS, J. Object-oriented resource models: their role in specifying components of integrated manufacturing systems. *Computer Integrated Manufacturing Systems*, v. 9, n. 1, p. 33–48, 1996. Citado na página 30.

AUDONNET, F. P.; HAMILTON, A.; ARAGON-CAMARASA, G. Digital manufacturing: history, perspectives, and outlooka systematic comparison of simulation software for robotic arm manipulation using ros2. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, Nov 2022. Citado na página 18.

CHRYSSOLOURIS, G. et al. Digital manufacturing: history, perspectives, and outlook. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, v. 223, n. 5, p. 451–462, May 2009. Citado na página 15.

COMMISSION, I. E. *Function blocks - Part 1: Architecture*. 2nd. ed. Suíça: IEC Geneva, 2012. Also British Standard BS EN61499-1: 2012. Citado 3 vezes nas páginas 30, 32 e 33.

DOSOFTEI, C.-C. Simulation power vs. immersive capabilities: Enhanced understanding and interaction with digital twin of a mechatronic system. *Applied Sciences*, v. 13, n. 11, 2023. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/13/11/6463>. Citado na página 24.

Eclipse Foundation. *O que é Eclipse 4diac™?* 2007. Disponível em: <https://eclipse.dev/4diac/>. Citado na página 34.

Envisia. *O que é OPC UA?* 2020. Acesso em: 27 jul. 2024. Disponível em: <https://www.envisia.com.br/2020/06/27/o-que-e-opc-ua/>. Citado na página 29.

FEI, T. et al. Digital twin modeling. *Journal of Manufacturing Systems*, v. 64, p. 372–389, 2022. Citado na página 15.

GRIEVES, M. Digital twin: Manufacturing excellence through virtual factory replication. 03 2015. Citado 3 vezes nas páginas 15, 19 e 21.

GRIEVES, M.; VICKERS, J. Digital twin manufacturing through virtual factory replication. *Journal of Engineering Science and Researches*, v. 18, p. 6–15, 03 2017. Citado na página 20.

JI, L. et al. Adaptiveon: adaptive outdoor local navigation method for stable and reliable actions. 2022. Citado na página 27.

KOHNHäUSER, F. et al. On the security of iiot deployments: an investigation of secure provisioning solutions for opc ua. *Ieee Access*, v. 9, p. 99299–99311, 2021. Citado na página 16.

KRISHNAN, M. G.; VIJAYAN, A.; ASHOK, S. Interfacing an industrial robot and matlab for predictive visual servoing. *Industrial Robot: The International Journal of Robotics Research and Application*, v. 48, p. 110–120, 2020. Citado na página 27.

KRITZINGER, W. et al. Digital twin in manufacturing: A categorical literature review and classification. *IFAC-PapersOnLine*, v. 51, n. 11, p. 1016–1022, 2018. 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018. Citado 2 vezes nas páginas 15 e 20.

KRUCHTEN, P. Architectural blueprints—the "4+1" view model of software architecture. *IEEE Software*, Rational Software Corp, v. 12, n. 6, p. 42–50, November 1995. Citado 2 vezes nas páginas 36 e 37.

LESAGE, J.; BRENNAN, R. Digital twins for distributed intelligent sensing and control systems. In: BORANGIU, T. et al. (Ed.). *Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future*. Cham: Springer International Publishing, 2022. p. 119–130. ISBN 978-3-030-99108-1. Citado na página 16.

LYU, T. et al. Mechatronic swarm and its virtual commissioning. In: *2023 IEEE International Conference on Mechatronics (ICM)*. Loughborough, Reino Unido: IEEE, 2023. ISBN 978-1-6654-6661-5. Citado na página 16.

MIZUCHI, Y.; INAMURA, T. Cloud-based multimodal human-robot interaction simulator utilizing ros and unity frameworks. *2017 IEEE/SICE International Symposium on System Integration (SII)*, 2017. Citado na página 27.

MORAN, M. E. Evolution of robotic arms. *Journal of robotic surgery*, Springer, v. 1, n. 2, p. 103–111, 2007. Citado na página 15.

NORRIS, S. *CRUSTCRAWLER AX-12+ Smart Robotic Arm*. 2008. CrustCrawler. Disponível em: <https://www.crustcrawler.com/products/AX12A%20Smart%20Robotic%20Arm/>. Citado 4 vezes nas páginas 17, 22, 50 e 52.

OPC Foundation. *UA-.NETStandard*. 2016. Acesso em: 6 agosto 2024. Disponível em: <https://github.com/OPCFoundation/UA-.NETStandard/tree/master>. Citado na página 54.

OPC Foundation. *OPC Unified Architecture Part 1: Overview and Concepts*. 2022. OPC 10000-1 - UA Specification Part 1 - Overview and Concepts 1.05.02.docx. Document Number OPC 10000-1, Software: MS-Word, Status: Release. Citado na página 28.

OPCFOUNDATION. *UA-ModelCompiler*. 2019. Disponível em: <https://github.com/OPCFoundation/UA-ModelCompiler>. Citado na página 64.

ORFALI, R.; HARLEY, D.; EDWARDS, J. *The Essential Distributed Objects Survival Guide*. USA: Wiley, 1996. Citado na página 30.

PARROT, A.; WARSHAW, L. Industry 4.0 and the digital twin: Manufacturing meets its match. *Deloitte University Press*, v. 1, n. 1, p. 1–17, 2017. Citado na página 15.

RINALDI, J. S. *OPC UA: The Everyman's Guide to OPC UA*. Estados Unidos: CreateSpace Independent Publishing Platform, 2016. ISBN 978-1530505111. Citado na página 28.

ROBOTIS. *Dynamixel SDK*. 2017. Disponível em: <https://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_sdk/overview/>. Citado 5 vezes nas páginas 22, 23, 24, 25 e 27.

STARK, R.; KIND, S.; NEUMEYER, S. Innovations in digital modelling for next generation manufacturing system design. *CIRP Annals*, v. 66, n. 1, p. 169–172, 2017. Citado na página 15.

VACLAVOVA, A. et al. Proposal for an iiot device solution according to industry 4.0 concept. *Sensors*, v. 22, p. 325, 2022. Citado na página 16.

VYATKIN, V. The iec 61499 standard and its semantics. *IEEE Industrial Electronics Magazine*, v. 3, n. 4, p. 40–48, 2009. Citado na página 16.

WANG, Y. et al. Perception of demonstration for automatic programing of robotic assembly: Framework, algorithm, and validation. *IEEE/ASME Transactions on Mechatronics*, v. 23, n. 3, p. 1059–1070, 2018. Citado na página 15.

ZOITL, A.; LEWIS, R. *Modelling Control Systems Using IEC 61499*. 2nd. ed. Londres: The Institution of Engineering and Technology, 2014. ISBN 978-1-84919-761-8. Citado 7 vezes nas páginas 30, 31, 32, 33, 34, 35 e 36.