UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

LUCAS JORJAN ALVES

DESENVOLVIMENTO DE TESTES PARA PROGRAMAS EM CLP VIA OPC-UA

Joinville
2022

LUCAS JORJAN ALVES

DESENVOLVIMENTO DE TESTES PARA PROGRAMAS EM CLP VIA OPC-UA

Trabalho de Conclusão de Curso apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica no curso de Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville.

Orientador: Dr. Gian Ricardo Berkenbrock

Joinville
2022

AGRADECIMENTOS

Agradeço à todos que estiveram presentes durante a graduação, minha família, professores e amigos que com seu incentivo me fizeram chegar à conclusão do meu curso, em especial meus pais desejo sinceros agradecimentos. Vocês desempenharam um papel significativo no meu crescimento. Também gostaria de agradecer meu professor orientador, que me auxiliou durante a construção desse projeto e sua paciência com os desafios encontrados.

RESUMO

No desenvolvimento de programas para CLP, a garantia de qualidade é fundamental, visto que os testes de software visam evitar falhas que podem ocasionar problemas de desempenho e gerar altos custos no projeto. As verificações antecipam a descoberta de falhas a partir de métricas e técnicas que auxiliam na execução. O presente trabalho tem o propósito de simular casos de testes para programas em linguagem ladder, em função do que, por meio do protocolo OPC UA é feita a comunicação entre softwares de simulação de CLP, supervisório e de ferramentas para testes de software. A partir de técnicas, métodos e processos de testes de software, é realizado o desenvolvimento das análises de cada exemplo executado. Os resultados obtidos buscam a validação de cada caso de teste, da forma mais adequada para seus respectivos processos.

Palavras-chave: Ladder. CLP. Testes de software. OPC UA.

ABSTRACT

In the development of programs for CLP, quality assurance is fundamental, given that the software tests are made to avoid failures that can cause performance issues and excessive project costs. The tests anticipate the recognition of failures from metrics and techniques that help in execution. The purpose of this project is to simulate test cases for programs in ladder language, using the OPC UA protocol, in wich the communication is done by PLC simulation software, supervisory and software testing tools. This article analyzes each executed exemple through techniques, methods and software testing processes. The results obtained seek validation of each test case, with the appropriate method for their respective operations.

Keywords: Ladder. CLP. software testing. OPC UA.

LISTA DE FIGURAS

Figura 1 – Pirâmide de automação 14
Figura 2 – Terminais gráficos PanelView 5000 16
Figura 3 – Interface do Elipse E3 17
Figura 4 – Controlador Lógico Programável PLC300 18
Figura 5 – Controlador CompactLogix 5480 19
Figura 6 – Exemplo de um diagrama ladder 21
Figura 7 – Padrões de interface OPC clássicos 22
Figura 8 – Diagrama de conexão com o OPC-UA 23
Figura 9 – Ambiente OPC UA 25
Figura 10 – Entidades Publisher e Subscriber 26
Figura 11 – symbol configuration no Codesys 32
Figura 12 – Testes de comunicação via UaExpert 32
Figura 13 – Uso da biblioteca opcua 33
Figura 14 – Funções de escrita e leitura dos dados 34
Figura 15 – Programa em ladder do caso 1 37
Figura 16 – Programa em ladder do caso 2 38
Figura 17 – Programa em ladder do caso 2 com os blocos de adição e subtração 38
Figura 18 – Fluxograma geral do sistema 39
Figura 19 – Exemplo de teste criado em XML 40
Figura 20 – Exemplo elaborado no figma 41
Figura 21 – Tela de diagnóstico dos testes no figma 41
Figura 22 – Exemplo elaborado no QtDesginer 42
Figura 23 – Tela de configurações da aplicação 43
Figura 24 – Execução dos testes a partir da interface do PyQt 44
Figura 25 – Adicionando variáveis aos testes 45
Figura 26 – Função de criar testes 46
Figura 27 – Tela inicial da aplicação 47
Figura 28 – Popup ao clicar em detalhes no teste selecionado 48
Figura 29 – função teste linha 48
Figura 30 – método parametrize 49
Figura 31 – método teste XML 49
Figura 32 – funções de acionamento para as entradas 50
Figura 33 – Comparação se a saída é menor ou maior com o resultado esperado 50
Figura 34 – Comparação entre a saída com o resultado esperado 51
Figura 35 – tanque criogênico no Codesys 52

Figura 36 – Resultado do teste de reset 53
Figura 37 – Arquivo XML do teste de falha 54
Figura 38 – Verificação do teste de falha 54
Figura 39 – Resultado via terminal para o teste de falha 54
Figura 40 – Variáveis para o teste do motor fechando 55
Figura 41 – Variáveis para o teste do motor abrindo 55
Figura 42 – Testes para o motor atuar 56

LISTA DE TABELAS

Tabela 1 – Instruções para diagrama ladder 20
Tabela 2 – Características dos trabalhos relacionados 30
Tabela 3 – Requisitos Funcionais 35
Tabela 4 – Requisitos não Funcionais 36
Tabela 5 – Partição de equivalência para o caso 1 36
Tabela 6 – Partição de equivalência para o caso 2 37

LISTA DE SIGLAS E ABREVIATURAS

CLP	*Controlador lógico programável*
IHM	*Interface homem máquina*
SCADA	*Supervisory control and data acquisition*
LD	*Ladder*
FBD	*Diagrama de blocos funcionais*
OPC	*Open Platform Communications*
DA	*Data Access*
SDK	*Software Development Kit*
ERP	*Enterprise Resource Planning*
RFN	*Requisitos não-funcionais*
CTT	*Combinatorial Test Tool*
CNC	*Comando Numérico Computadorizado*
XML	*Extensible Markup Language*

SUMÁRIO

1	**INTRODUÇÃO**	**11**
1.1	Objetivo Geral	11
1.2	Objetivos Específicos	11
2	**FUNDAMENTAÇÃO TEÓRICA**	**13**
2.1	Automação industrial	13
2.1.1	**Arquitetura da automação industrial**	**13**
2.1.2	**Sistemas supervisórios**	**15**
2.1.2.1	Interface Homem Máquina	15
2.1.2.2	SCADA	16
2.1.3	**Controladores lógicos programáveis**	**17**
2.1.3.1	SoftPLC	19
2.1.3.2	Linguagem ladder	19
2.2	OPC	21
2.2.1	**DA**	**22**
2.2.2	**UA**	**23**
2.2.3	**Segurança**	**24**
2.2.4	**Publisher/Subscriber**	**26**
2.2.5	**Aplicações**	**26**
2.3	Teste de software	27
2.3.1	**Testes caixa-preta**	**27**
2.4	Trabalhos Relacionados	29
3	**MATERIAIS E MÉTODOS**	**31**
3.1	Comunicação OPC UA	31
3.2	Elaboração dos testes	34
3.3	Gerenciamento dos casos de testes	38
3.4	Criação da interface	40
4	**RESULTADOS E DISCUSSÕES**	**43**
4.1	Configurações iniciais	43
4.2	Desenvolvimento dos testes	44
4.3	Execução dos testes	46
4.4	Validação dos requisitos	55
4.5	Limitações	56
4.6	Considerações finais	57

5 CONCLUSÕES 58

REFERÊNCIAS 60

1 INTRODUÇÃO

A Engenharia de Software evoluiu significativamente nas últimas décadas, procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de software (DELAMARO et al., 2000). Como resultado da crescente utilização de sistemas baseados em computação, a área de testes de software tornou-se cada vez mais relevante, representando uma etapa do processo de desenvolvimento, a qual tem objetivo validar se uma aplicação está funcionando corretamente.

Para Javaroni (2014), a validação de um programa de CLP pode se dar por verificação formal ou por testes. A verificação formal consiste em transformar o programa em um modelo formal equivalente e, por meio de fórmulas, ter suas propriedades verificadas. Já os testes se baseiam na definição de conjuntos de valores de entrada e de saída esperada, analisando o correto funcionamento do programa.

O desempenho é uma das características de qualidade mais importantes em um produto de software. Avaliar o desempenho de um software não é uma tarefa trivial, pois requer não apenas conhecimento prático de uma ferramenta de automação de testes, mas também aspectos teóricos relacionados (SOUZA, 2018).

Considerando tal problemática, propõe-se a pesquisa aplicada a testes de software. O trabalho contribui para a comunidade acadêmica, tendo em vista que apresenta técnicas e ferramentas para acelerar o processo de testes para programas de controladores lógicos programáveis, realizando modelos de verificações que objetivam garantir a qualidade dos programas, além de sua integração da comunicação via protocolo de plataforma aberta arquitetura unificada (OPC UA).

O conjunto que compreende os testes gerados, a partir do modelo criado, será executado usando a biblioteca de testes do Pytest. O modelo comportamental é projetado com base nas especificações de requisitos, a qual descreve a saída esperada para cada combinação de entradas. Essa abordagem segue o esperado para um teste do tipo caixa-preta.

1.1 OBJETIVO GERAL

Realizar testes de unidade remotos em programas para controlador lógico programável usando do protocolo OPC UA.

1.2 OBJETIVOS ESPECÍFICOS

- Revisar a literatura sobre testes;

- Realizar a comunicação com OPC UA com CLPs (virtuais ou reais), supervisórios e de ferramentas de teste por modelos de cliente-servidor;
- Desenvolver casos de testes de caixa-preta;
- Elaborar uma interface para criação de testes;
- Testar a abordagem desenvolvida.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão abordados conceitos da automação industrial, explicando componentes da sua organização, modos de comunicação e aplicações.

2.1 AUTOMAÇÃO INDUSTRIAL

A automação industrial visa reduzir o esforço ou a interferência humana sobre um processo e, de acordo com Castrucci e Moraes (2006), entende-se por automação qualquer sistema, apoiado em computadores, que substitua o trabalho humano em favor da segurança das pessoas, da qualidade dos produtos, da rapidez da produção ou da redução de custos, aperfeiçoando os complexos objetivos das indústrias e dos serviços.

O principal motivador da automação é a busca pela maior qualidade e flexibilidade dos processos, tornando os equipamentos mais eficientes e consequentemente reduzindo os custos operacionais, resultando em empresas cada vez mais competitivas (MARCAL; GUIMARAES; RESENDE, 2013).

Segundo Castrucci e Moraes (2006) a automação envolve a implantação de sistemas interligados e assistidos por redes de comunicação, compreendendo sistemas supervisórios e interfaces homem máquina que possam auxiliar os operadores no exercício da supervisão e da análise dos problemas que porventura venham a ocorrer.

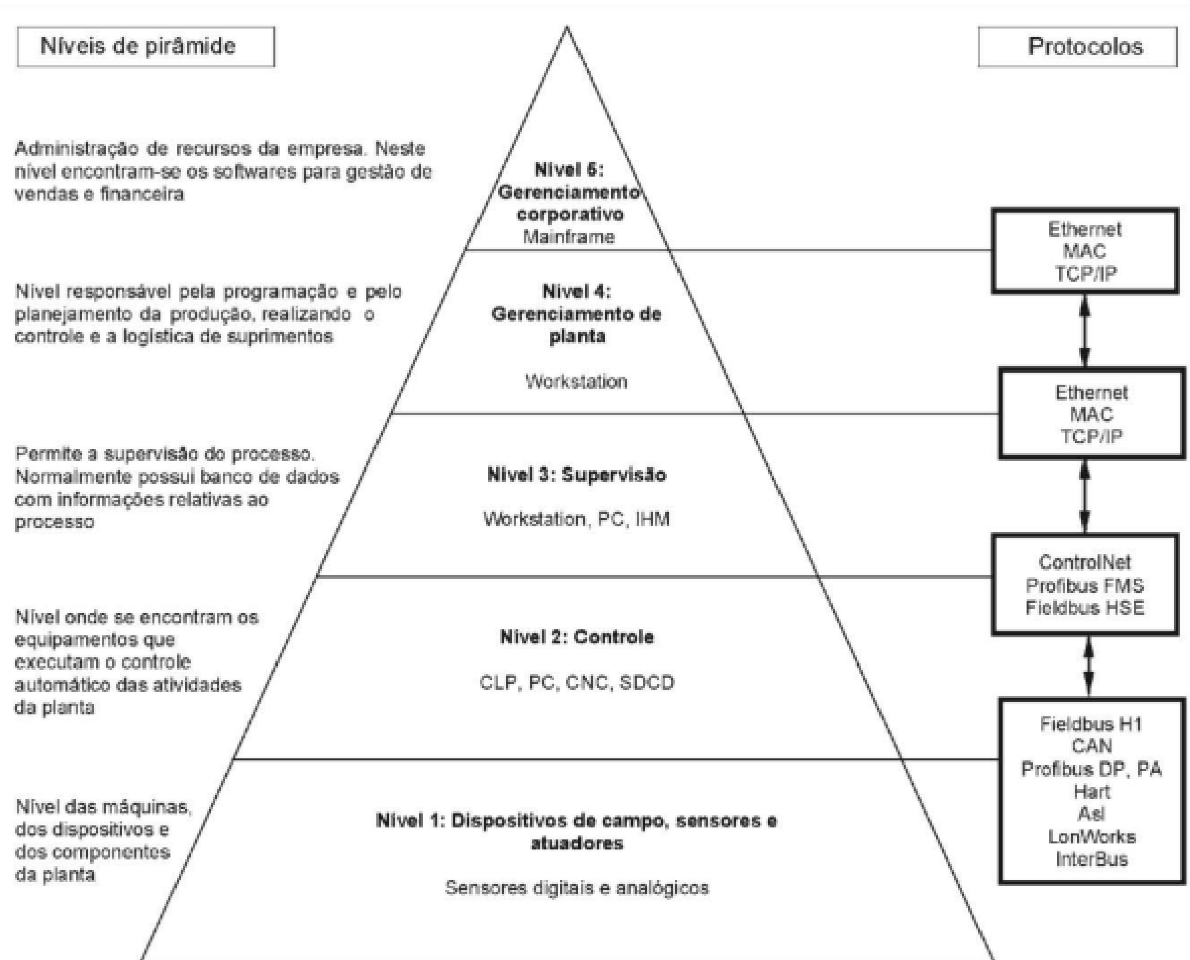
O primeiro termo utilizado para definir a automação foi "controle automático de processo", e foram usados instrumentos capazes de medir, transmitir, comparar e atuar no processo para conseguir um produto desejado com o mínimo de intervenção humana (RIBEIRO, 2001). O conceito de automação está ligado ao uso de técnicas, softwares e equipamentos eletrônicos que, aplicados em máquinas inclusas em um processo, tem como objetivo aumentar a eficiência, maximizar a produção e a sua qualidade.

Dentre os assuntos que fazem parte da automação industrial, as próximas seções descrevem a estrutura e conceitos necessários para entender o funcionamento para o controle, supervisão e visualização de um sistema de automação industrial.

2.1.1 Arquitetura da automação industrial

A automação industrial requer o gerenciamento de diversas tarefas e pode ser dividida em cinco níveis dentro de uma planta industrial, a Figura 1 representa a chamada Pirâmide de Automação, que divide os diferentes níveis de automação encontrados em estruturas industriais.

Figura 1 – Pirâmide de automação



Fonte: Castrucci e Moraes (2006, p. 13)

Os diferentes níveis da piramide de automação são descritos de acordo com Castrucci e Moraes (2006) como:

- Nível 1: é o nível das máquinas, dispositivos e componentes (chão de fábrica);
- Nível 2: é o nível dos controladores digitais, dinâmicos e lógicos, e de algum tipo de supervisão associada ao processo;
- Nível 3: permite o controle do processo produtivo da planta; normalmente é constituído por bancos de dados com informações dos índices de qualidade da produção, relatórios e estatísticas de processo, índices de produtividade, algoritmos de otimização da operação produtiva;
- Nível 4: é responsável pela programação e pelo planejamento da produção, realizando o controle e a logística dos suprimentos;
- Nível 5: é responsável pela administração dos recursos da empresa, em que se encontram os softwares para gestão de vendas e gestão financeira.

A automação industrial está em constante evolução, trazendo cada vez

mais vantagens para as plantas industriais em que é aplicada. Há em todos os níveis da piramide de automação novas tecnologias sendo criadas e aperfeiçoadas, principalmente com a evolução de diversas áreas como a internet das coisas, indústria 4.0, a inteligência artificial e Machine Learning.

2.1.2 Sistemas supervisórios

Sistemas supervisórios são sistemas digitais comumente implementados em plantas industriais que possuem como finalidade a monitoração, operação e gerenciamento das variáveis do processo. Estas variáveis são armazenadas em bancos de dados locais ou remotos e são utilizadas como registro histórico, podem servir de base para estudos no auxílio de tomada de decisões estratégicas de uma empresa (CASTRUCCI; MORAES, 2006).

Sistemas de Supervisão como Interface Homem Máquina (IHM), SCADA e Workstation pertencem ao nível 3 da pirâmide de automação, de acordo com Castrucci e Moraes (2006). Os sistemas supervisórios podem ser classificados quanto a complexidade, robustez e número de entradas e saídas monitoradas. Os dois grandes grupos atualmente conhecidos são:

- IHM / HMI (Interface homem máquina/Human machine interface)
- SCADA (Supervisory control and data acquisition - Aquisição de dados e controle do supervisório)

2.1.2.1 Interface Homem Máquina

Segundo Castrucci e Moraes (2001) IHM são sistemas supervisórios que surgiram da necessidade de uma interface amigável, eficiente e ergonômica entre sistemas de automação complexos e a equipe encarregada da sua operação. No CLP as IHMs podem ser configuradas para enviarem sinais de atuação ou simplesmente monitorá-lo.

Na Figura 2 é ilustrado um exemplo de IHM, essa interface gráfica é uma integração do Logix, a qual usa o software Studio 5000 View Designer da Rockwell Automation.

Figura 2 – Terminais gráficos PanelView 5000

Fonte: Rockwellautomation (2021)

Uma IHM é um hardware industrial composto normalmente por uma tela de cristal líquido e um conjunto de teclas para navegação ou inserção de dados que utiliza um software proprietário para sua programação (CASTRUCCI; MORAES, 2006).

2.1.2.2 SCADA

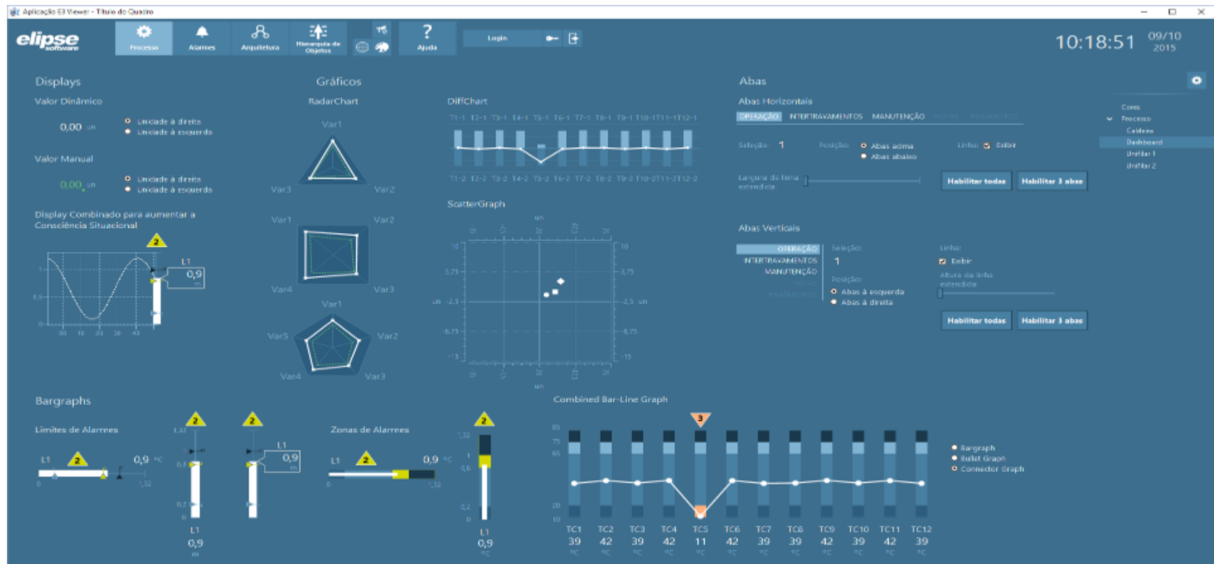
Os sistemas de supervisão e controle comumente chamados de sistemas SCADA são sistemas configuráveis, destinados à supervisão, ao controle e à aquisição de dados de plantas industriais. A interação do operador com o processo é garantida através de interfaces gráficas (CASTRUCCI; MORAES, 2006).

O equipamento usado para coletar dados é chamado de sistema de aquisição de dados, a aquisição de dados segundo Ribeiro (2001) é a coleta de informação para fins de armazenamento e uso posterior, como a análise dos dados e o consequente controle e monitoração do processo. Nas aplicações industriais, a aquisição de dados deve ser em tempo real, ou seja, o sistema deve ter a habilidade de coletar os dados ou fazer uma tarefa de controle dentro de uma janela aceitável de tempo.

Na Figura 3 é apresentado a interface do Elipse E3, que é um exemplo de

ferramenta SCADA utilizada para monitoramento e controle de processos.

Figura 3 – Interface do Elipse E3



Fonte: Elipse (2021)

Para haver o monitoramento das variáveis de um processo entre o CLP e o Sistema SCADA é necessário existir uma comunicação, segundo Castrucci e Moraes (2006) essa comunicação é feita através de TAGs (Etiquetas) que o CLP envia junto aos sinais para o sistema de supervisão. Esses TAGs carregam informações como o endereço dentro do CLP e o tipo de TAG.

2.1.3 Controladores lógicos programáveis

Dentre o grande número de dispositivos utilizados na automação industrial, um grupo que possui grande destaque são os controladores lógicos programáveis. Castrucci e Moraes (2006) explicam a vantagem de utilizar sistemas que envolvam diretamente a informatização e a possibilidade da expansão utilizando recursos de fácil acesso. Nesse contexto, são de extraordinária importância os controladores lógicos programáveis (CLP), que tornam a automação industrial uma realidade onipresente.

Atualmente, na maioria das aplicações industriais, a aquisição de dados é feita por controladores lógico programáveis, que possuem as interfaces de entrada e saída já padronizadas e com preço mais conveniente que as interfaces Entrada/Saída do sistema digital de controle distribuído. Outra vantagem de se usar um CLP como sistema de coleta de dados é a facilidade de driver de comunicação entre ele e o microcomputador onde será rodado o aplicativo para realizar o controle supervisório do processo (RIBEIRO, 2001). As principais características que justificam seu uso segundo Castrucci e Moraes (2006) são as seguintes:

- Linguagens de programação de alto nível, caracterizando um sistema bastante amigável com relação ao operador. Depois de concluído e depurado, o programa

pode ser transferido para outros CLP, garantindo confiabilidade na sua utilização;

- Simplificação nos quadros e painéis elétricos. Toda a fiação do comando fica resumida a um conjunto de entradas e saídas. Como consequência, qualquer alteração necessária torna-se mais rápida e barata;
- Confiabilidade operacional. Uma vez que as alterações podem ser realizadas através de um aplicativo, necessitando de muito pouca ou de nenhuma alteração da fiação elétrica, a possibilidade de haver erro é minimizada, garantindo sucesso nos desenvolvimentos ou melhorias a serem implementadas.

Um exemplo de CLP é ilustrado na Figura 4, desenvolvido pela WEG, foi projetado para atender às necessidades no controle de máquinas e pequenos processos industriais.

Figura 4 – Controlador Lógico Programável PLC300

Fonte: Weg (2021)

Na Figura 5 é mostrado outro exemplo de controlador programável, fabricado pela Rockwell Automation, o CompactLogix 5480 é um CLP com mecanismo de controle voltado para internet das coisas, assim como outros CLP, é utilizado para coleta de dados de modo a permitir análises em tempo real.

Figura 5 – Controlador CompactLogix 5480

Fonte: Rockwellautomation (2021)

A forma com que o CLP se comunica com o computador e os equipamentos de campo é feita através de diversos protocolos via rede, alguns exemplos desses protocolos são Ethernet, Profibus, Modbus, DeviceNet e OPC UA, o qual foi utilizado como meio de comunicação no presente trabalho.

2.1.3.1 SoftPLC

A tecnologia SoftPLC é um novo tipo de tecnologia de controle baseada no PC, em comparação com o tradicional CLP, que tem uma arquitetura aberta, capacidade de comunicação de rede poderosa e dados aumentados, pode atender aos requisitos da automação industrial moderna (LIANG; LI, 2011).

O SoftPLC é um recurso que tem as funcionalidades de um CLP, agregando flexibilidade e permitindo que o usuário desenvolva seus próprios aplicativos, como Liang e Li (2011) explica, o objetivo do soft PLC é usar o PC como plataforma de suporte de hardware, usar o software para alcançar a função básica de um CLP tradicional. As funções de controle do CLP são encapsuladas no software, rodando no ambiente do PC. Este sistema de controle fornece um sistema de controle CLP com os mesmos recursos.

2.1.3.2 Linguagem ladder

Durante muito tempo, foram desenvolvidas diferentes técnicas de programação para aplicações industriais de controle e para CLPs. Isso exigiu que os profissionais

envolvidos no trabalho em plantas tivessem que buscar conhecimento de várias marcas de fabricantes de CLPs (SILVA, 2016). A fim de padronizar as linguagens de programação para a automação industrial foi elaborada a norma IEC 61131-3, que apresenta e define cinco linguagens de programação que podem cobrir uma ampla gama de aplicações. De acordo com a norma IEC 61131-3, destacam-se as linguagens de programação para CLPs: lista de instrução (IL), ladder (LD), texto estruturado (ST), diagrama de blocos funcionais (FBD) e sequencial gráfico de função (SFC).

Entre as várias linguagens de programação utilizadas em controladores programáveis, o diagrama Ladder é uma das mais utilizadas para programar CLP por ser uma linguagem de fácil compreensão, pois é muito semelhante com as lógicas de contatos de comandos elétricos, como Castrucci e Moraes (2006) explicam, a linguagem ladder, a qual permite programar desde funções binárias até funções matemáticas complexas. A sua representação originou-se dos diagramas elétricos em ladder (escada), cujo princípio provém da lógica de relés e contatos.

Os elementos constituintes de um diagrama ladder podem ser divididos em componentes de entrada e de saída. O principal componente de entrada é o contato. Quanto à operação o contato pode ser retentivo ou não retentivo. Quanto à lógica, o contato pode ser normalmente aberto (NA) ou normalmente fechado (NF) (RIBEIRO, 2001). Na Tabela 1 são ilustradas representações para cada instrução em ladder.

Tabela 1 – Instruções para diagrama ladder

Instrução	Representação
Contato normalmente aberto (NA)	-| |-
Contato normalmente fechado (NF)	-|/|-
Bobina	-( )-
Bobina inversa (acionada, desenergizada)	-(I)-
Bobina set	-(S)-
Bobina reset	-(R)-

A Tabela 1 ilustra as instruções básicas que aparecem em qualquer CLP, onde os contatos representam as entradas do sistema e as bobinas suas saídas. A imagem 6 representa um exemplo de diagrama ladder, o qual mostra a ligação em paralelo entre as entradas S e Q1, componentes normalmente abertos e um Contato normalmente fechado.

Figura 6 – Exemplo de um diagrama ladder

Fonte: Zoubek, Roussel e Kwiatkowska (2003, p. 7)

O procedimento básico para analisar o diagrama ladder envolve a consideração de cada componente do circuito por vez, para decidir o que acontece com este componente quando um contato é fechado, a determinação da função de cada componente em suas posições normal e energizada (RIBEIRO, 2001). Ribeiro (2001) resume como analisar um diagrama ladder, o qual é importante para entender a lógica de controle empregada.

2.2 OPC

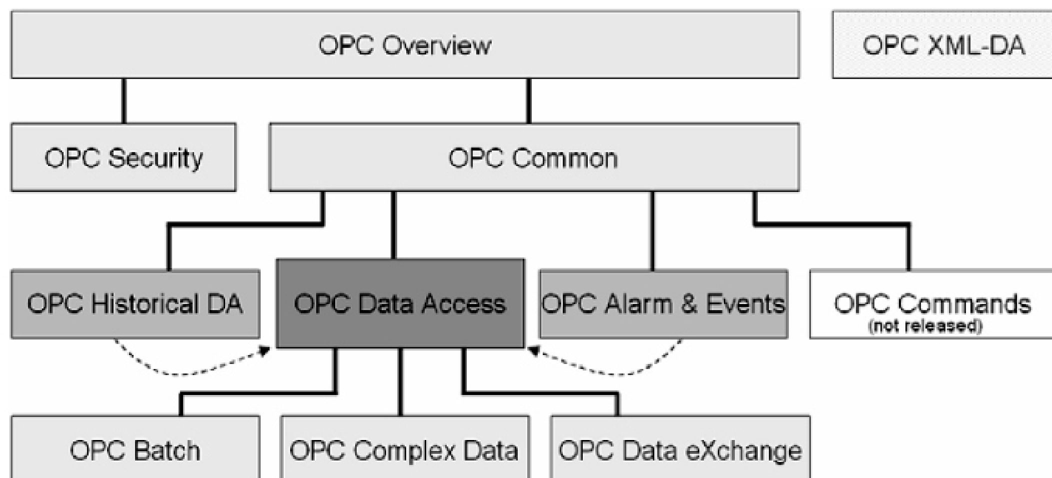
Com o aumento do número de dispositivos instalados numa indústria, torna-se essencial a interligação entre os diversos sistemas, de forma a permitir a troca de informação e a coordenação de operações. Assim, surgiram as redes de comunicação industriais para possibilitar a partilha e a troca de informação entre os diferentes níveis hierárquicos, bem como o controle e monitorização dessa informação (ROCHA, 2013).

OPC (Open Platform Communications - Plataforma de Comunicação Aberta) é o padrão de interoperabilidade para a troca segura e confiável de dados no espaço da automação industrial e em outras indústrias. Esse protocolo é independente de plataforma e suporta o fluxo contínuo de informações entre dispositivos de vários fornecedores. A fundação OPC é responsável pelo desenvolvimento e manutenção deste padrão (FOUNDATION, 2021).

Conforme Franco et al. (2011) descreve, o protocolo OPC é um padrão de comunicação de dados em que várias informações localizadas em diversos pontos (controladores, dispositivos de E/S, banco de dados, etc.) podem ser conectadas diretamente a um “cliente” (aplicação) e “Servidor” (fonte de dados), fazendo com que o uso de drivers e conversores sejam minimamente utilizados.

O padrão OPC consiste em uma série de especificações desenvolvidas por fornecedores da indústria, usuários finais e desenvolvedores de software. Estas especificações definem a interface entre clientes e servidores, incluindo acesso a dados em tempo real, monitoramento de alarmes e eventos, acesso a dados históricos e outras aplicações (FOUNDATION, 2021). A Figura 7 oferece uma visão geral de todas as especificações clássicas do OPC.

Figura 7 – Padrões de interface OPC clássicos



Fonte: Mahnke, Leitner e Damm (2009, p. 7)

Cada versão do OPC era designada para determinadas funções como é descrito por Lehnhoff et al. (2011) o OPC UA é o sucessor de ex-padrões OPC clássicos estabelecidos OPC DA (acesso a dados), OPC A&E (Alarmes e Eventos) e OPC HDA (Histórico de acesso a dados físicos). A seguir será abordado de forma mais profunda as especificações do OPC DA e o OPC UA.

2.2.1 DA

O OPC DA (Data Access - acesso de dados) foi lançado em agosto de 1996, onde quase todos os fornecedores de sistemas para automação industrial tornaram-se membros da OPC Foundation. A OPC Foundation foi capaz de definir e adotar padrões relevantes mais rápido do que outras organizações. Uma das razões para este sucesso foi a simplificação das principais características e a restrição para a definição de APIs usando componentes de tecnologias da Microsoft (MAHNKE et al., 2009).

Ainda segundo Mahnke et al.(2009) a interface OPC Data Access permite a leitura, gravação e monitoramento de variáveis contendo dados de processo atuais. O principal caso de uso é mover dados em tempo real de CLP, DCS e outros dispositivos de controle para IHM e outros clientes de exibição. O OPC DA foi projetado como interface para drivers de comunicação, permitindo uma leitura padronizada e acesso de gravação aos dados atuais em dispositivos de automação.

Segundo a Foundation (2021) OPC DA cumpre as seguintes funções:

- Os conceitos de tecnologia cliente / servidor OPC e definição de dados;
- Atividades gerais que incluem definir um espaço de endereço e navegar nele, ler, escrever e assinar notificações de atualização de dados;
- Descrições detalhadas de tipos e estruturas de dados;

- Descrições detalhadas de interfaces, métodos, parâmetros e comportamentos esperados.

2.2.2 UA

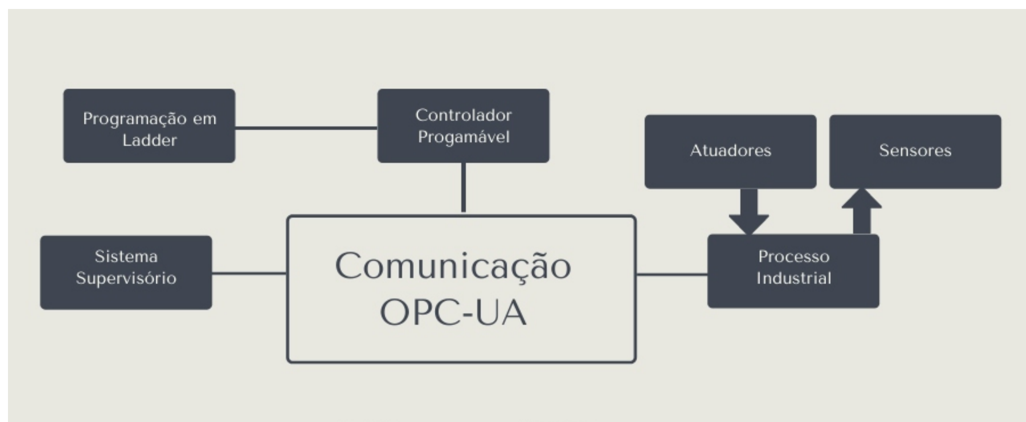
OPC UA (FOUNDATION, 2021) é um protocolo de comunicação industrial desenvolvido pela OPC foundation, representa uma evolução do OPC clássico. Teve seu surgimento em 2008, sendo atualmente um dos principais padrões de troca de dados entre diversos dispositivos de chão de fábrica. Sua arquitetura é baseada no sistema de serviços, onde há troca de mensagens entre servidor e cliente.

Segundo Pogacean, Broschei e Süss (2018) o OPC clássico é baseado no modelo COM/DCOM (Distributed) Component Object Model da Microsoft, usado como a tecnologia base para o padrão OPC. Este laço estreito impôs uma série de restrições ao uso da tecnologia OPC. Por exemplo, a tecnologia OPC baseada em DCOM não é adequada para comunicação pela Internet, para o uso de firewalls ou para operação em plataformas não-Windows.

Devido as restrições do OPC clássico, foi criado um novo padrão capaz de oferecer uma solução para as necessidades e exigências específicas da indústria. Como o OPC-UA é um padrão aberto e gratuito, pode ser usada por outros sistemas operacionais como afirma Lehnhoff et al. (2011) dentre os diversos avanços fornecidos pelo OPC-UA tem-se a possibilidade de desenvolvimento empregando múltiplas plataformas, como Windows, Linux e sistemas embarcados. Este foi viabilizado devido à disponibilização de SDK (Software Development Kit - Kits de Desenvolvimento de Software) estruturados em diversas linguagens de programação, como C/C++, C# e Java.

Na Imagem 8 é apresentado um exemplo de conexão para um processo industrial, utilizando o OPC-UA como protocolo de comunicação.

Figura 8 – Diagrama de conexão com o OPC-UA



Fonte: Autor (2021)

OPC UA (Open Platform Communications Unified Architecture – Plataforma

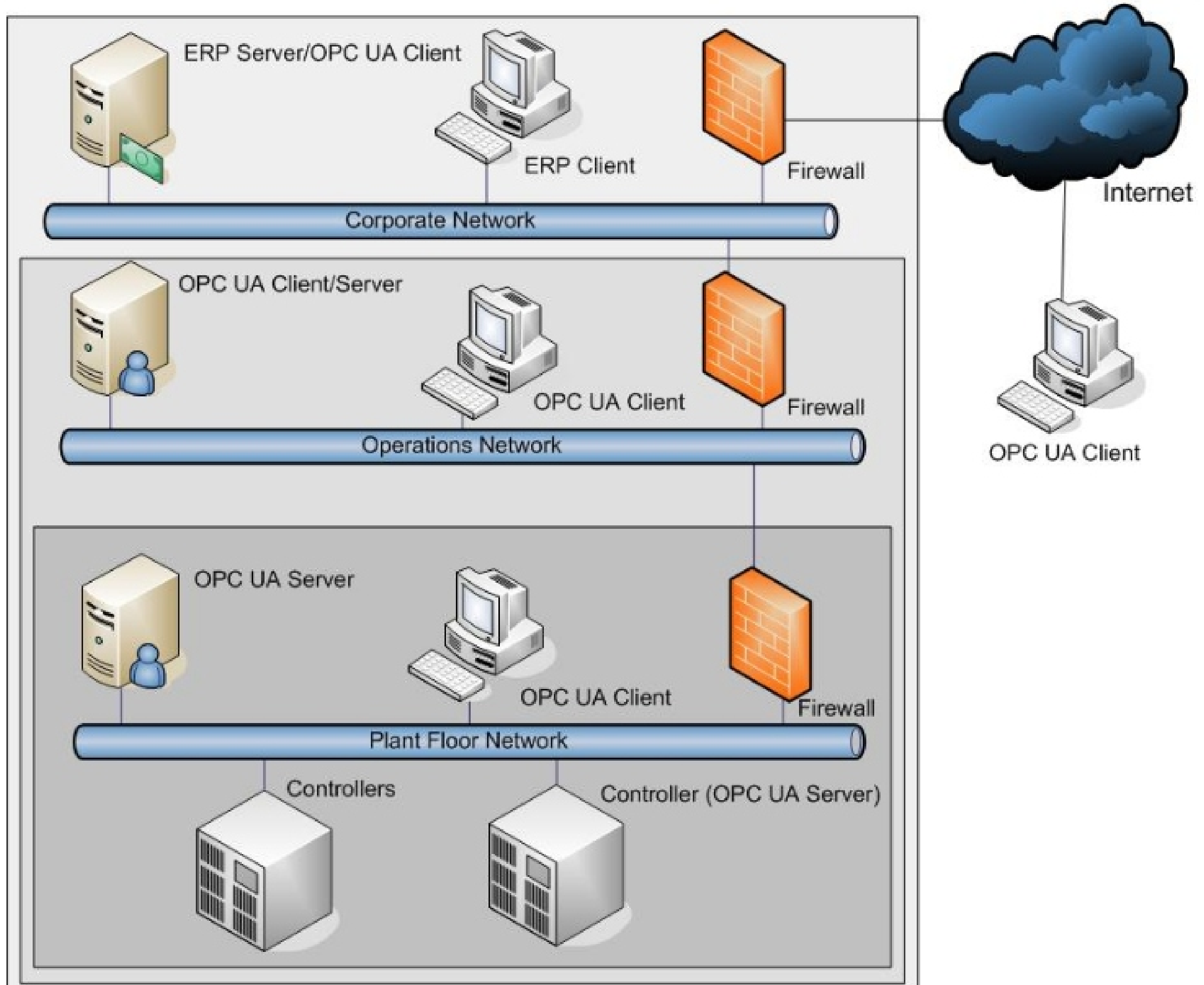
de Comunicação Aberta de Arquitetura Unificada) denomina-se dessa maneira por unificar a funcionalidade das diferentes especificações clássicas do OPC em aplicações exibindo os dados atuais, notificações de eventos e o histórico de ambos em um *Address Space* (MAHNKE et al., 2009).

2.2.3 Segurança

A Segurança do protocolo OPC segundo Mahnke et al.(2009) especifica como controlar o acesso do cliente aos servidores para proteger informações confidenciais e proteger contra modificação não autorizada dos parâmetros do processo. Os aplicativos OPC UA serão executados em ambientes variados, com diferentes requisitos de segurança, ameaças e políticas de segurança.

A Figura 9 mostra um exemplo de como os aplicativos OPC UA podem ser implantados. Nesse caso o OPC UA é aplicado em diferentes níveis da pirâmide de automação, no nível do chão de fábrica, um servidor OPC UA pode ser executado pelos controladores, fornecendo dados do dispositivos de campo para clientes OPC UA. Enquanto no nível mais alto um cliente OPC UA integrado em um sistema ERP (Enterprise Resource Planning - Planejamento dos Recursos da Empresa) poderia obter informações sobre as horas de trabalho dos dispositivos usados no chão de fábrica.

Figura 9 – Ambiente OPC UA



Fonte: Mahnke, Leitner e Damm (2009, p. 210)

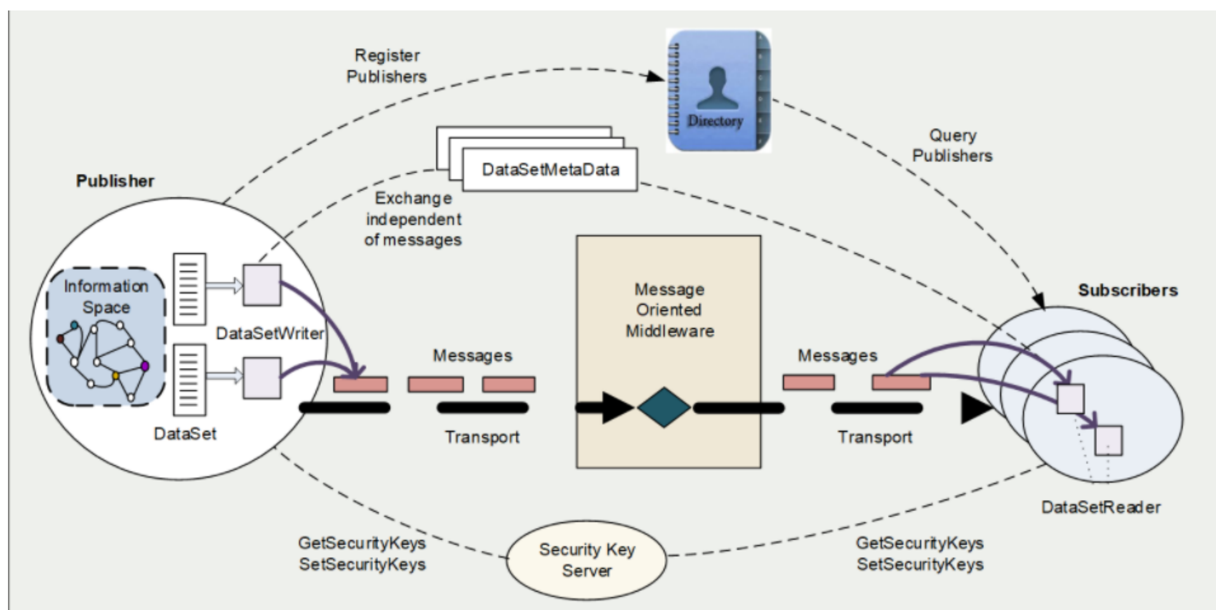
Ainda segundo Mahnke et al.(2009) o cenário do exemplo da Figura 9 mostra que OPC UA pode ser usado em vários lugares e para finalidades diferentes dentro do mesmo ambiente. A segurança para esses aplicativos também podem diferir de várias maneiras. A troca entre segurança e desempenho são um bom exemplo. No nível mais alto, a segurança pode ser mais importante do que o desempenho, uma vez que esta rede está conectada à Internet. No nível mais baixo os requisitos podem ser completamente diferentes, onde o desempenho pode ser mais importante do que a segurança, quando os dados devem ser adquiridos de forma rápida e eficiente para controlar um processo produtivo.

Conforme é afirmado pela OPC Foundation (2021) o OPC UA é um protocolo usado entre componentes na operação de uma instalação industrial em múltiplos níveis: de gerenciamento empresarial de alto nível ao controle de processo direto de baixo nível de um dispositivo. Assim ratificando seu uso para toda piramide industrial, a partir do nível do chão de fábrica até o nível mais alto, que é representado pela rede corporativa.

2.2.4 Publisher/Subscriber

A fim de preparar o OPC UA para o futuro, uma extensão para o Publisher/Subscriber foi desenvolvida. A extensão foi lançada pela Fundação OPC na parte 14 das especificações do OPC UA em fevereiro de 2018. A Figura 10 fornece uma visão geral das entidades Publisher e Subscriber, ilustrando o fluxo de mensagens de um Publisher para um ou mais Subscribers.

Figura 10 – Entidades Publisher e Subscriber



Fonte: Foundation (2021)

Com o Publisher/Subscriber, os aplicativos OPC UA não trocam solicitações e respostas diretamente. Em vez disso, publishers enviam mensagens para um Middleware, sem saber se pode haver subscribers. Da mesma forma, os subscribers requisitam tipos específicos de dados, e processam mensagens que contenham esses dados, sem conhecimento de quais publishers existem (FOUNDATION, 2021).

De acordo com Pfrommer et al. (2018) o OPC UA segue estritamente ao padrão de comunicação de Solicitação/Resposta. Contudo, uma vez que OPC UA é um protocolo assíncrono, um mecanismo de assinatura pode ser usado para transmissão push de notificações apenas quando eles ocorrem.

2.2.5 Aplicações

A automação industrial está presente em diversos níveis de complexidade no cotidiano da maioria das pessoas, seus níveis podem ser divididos entre *menor* e *maior* complexidade. Alguns exemplos segundo Castrucci e Moraes (2006) são aparelhos de TV, de vídeo, telefones celulares, eletrodomésticos, automóveis para *menor* complexidade.

Ainda de acordo com Castrucci e Moraes (2006) entre as de *maior* complexidade estão os grandes sistemas de automação, onde envolvem muitos computadores de vários tipos e capacidades. São, por exemplo, os sistemas de controle de voo nos aeroportos, os de controle metroviário, os de defesa militar. Sua programação envolve, além de programas de aplicação comercial e científica, a plena engenharia de software de tempo real.

2.3 TESTE DE SOFTWARE

A Engenharia de Software evoluiu significativamente nas últimas décadas procurando estabelecer técnicas, critérios, métodos e ferramentas para a produção de software. Apesar das técnicas, métodos e ferramentas empregados, erros no produto ainda podem ocorrer (DELAMARO et al., 2000). Para verificação dos possíveis erros os testes de software são necessários, onde determinam se um produto atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado.

O teste de software é um processo relacionado ao desenvolvimento de software, e tem como principal objetivo revelar falhas por meio da análise dinâmica de programas [ISO/IEC/IEEE 2013]. Trata-se de uma análise dinâmica porque há a necessidade de colocar o software em execução para avaliar se determinados dados de entrada geram determinados dados de saída, sob determinadas condições de contexto (SOUZA, 2018).

Segundo Delamaro et al. (2000) os critérios de software são estabelecidos, basicamente, a partir de três técnicas:

- Funcional: os critérios e requisitos de teste são estabelecidos a partir da função de especificação do software;
- Estrutural: os critérios e requisitos são derivados essencialmente a partir das características de uma implementação particular em teste;
- Baseada em erros: os critérios e requisitos de teste são oriundos do conhecimento sobre erros típicos cometidos no processo de desenvolvimento de software.

Já Souza (2018) afirma que os testes de software podem ser divididos em dois tipos, os testes podem ser classificados em testes funcionais ou testes não-funcionais. A definição do tipo de teste depende do tipo de requisito que se deseja avaliar. Requisitos funcionais demandam testes funcionais, enquanto que requisitos não-funcionais (RNF) exigem a realização de testes não-funcionais.

2.3.1 Testes caixa-preta

O teste funcional conhecido como caixa-preta segundo Delamaro et al. (2000) denomina-se dessa maneira pelo fato de tratar o software como uma caixa cujo conteúdo é desconhecido e da qual só é possível visualizar o lado externo, ou seja, os

dados de entrada fornecidos e as respostas produzidas como saída. Nos testes do tipo caixa preta são verificadas as funções do sistema sem se preocupar com os detalhes de implementação.

Dados de entrada de são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. Haverá sucesso no teste se o resultado obtido for igual ao resultado esperado. O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade (PRESSMAN, 1997).

De acordo com Pressman (1997) O teste de caixa-preta envolve dois passos principais:

- Identificar as funções que o software deve realizar;
- Criar casos de teste capazes de checar se essas funções estão sendo realizadas pelo software.

As funções que o software deve possuir são identificadas a partir de sua especificação. Assim, uma especificação bem elaborada e de acordo com os requisitos do usuário é essencial para esse tipo de teste (DELAMARO et al., 2000).

No teste baseado em especificação (ou teste caixa-preta) o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados. O problema é que, em geral, a especificação existente é informal e, desse modo, a determinação da cobertura total da especificação que foi obtida por um dado conjunto de casos de teste também é informal (PERRY; KAISER, 1990).

Entretanto como Delamaro et al. (2000) afirma os critérios de teste de caixa-preta podem ser utilizados em qualquer contexto (procedimental ou orientado a objetos) e em qualquer fase de teste sem a necessidade de modificação. Exemplos desses critérios segundo Pressman (1997) são:

- Particionamento em Classes de Equivalência: a partir das condições de entrada de dados identificados na especificação, divide-se o domínio de entrada de um programa em classes de equivalência válidas e inválidas. Em seguida seleciona-se o menor número possível de casos de teste, baseando-se na hipótese que um elemento de uma dada classe seria representativo da classe toda, sendo que para cada uma das classes inválidas deve ser gerado um caso de teste distinto;
- Análise do Valor Limite: é um complemento ao critério Particionamento em Classes de Equivalência, sendo que os limites associados às condições de entrada são exercitados de forma mais rigorosa; ao invés de selecionar-se qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes, pois nesses pontos se concentra um grande número de erros. O espaço de saída do programa também é particionado e são exigidos casos de teste que produzam resultados nos limites dessas classes de saída;

- Grafo de Causa-Efeito: os critérios anteriores não exploram combinações das condições de entrada. Este critério estabelece requisitos de teste baseados nas possíveis combinações das condições de entrada. Primeiramente, são levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. A seguir é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão a partir da qual são derivados os casos de teste.

O teste de caixa-preta desempenha um papel significativo nos testes de softwares, auxiliando na validação geral das funcionalidades do sistema. O teste de caixa-preta é feito com base nos requisitos do usuário final, para que quaisquer requisitos incompletos ou imprevisíveis possam ser facilmente identificados e resolvidos posteriormente. A principal importância do teste de caixa-preta é lidar com entradas válidas e inválidas na perspectiva do usuário final (MURNANE; REED, 2001).

2.4 TRABALHOS RELACIONADOS

Esta seção descreve brevemente trabalhos relevantes feitos na área de validação de programas de testes para CLP, foi analisado trabalhos como de Zoubek, Roussel e Kwiatkowska (2003) em que propõem testes de um programa de controle para uma unidade de linha de bombeamento, focado na linguagem ladder é feito a verificação automática em relação aos requisitos que incluem tempo.

No caso de Sudheerbabu (2021) é proposto um modelo de teste, onde a execução é realizada para sistemas CLP de segurança crítica. Um sistema de controle de máquinas é implementado usando a linguagem de programação de diagramas de blocos. A estrutura de teste Pytest usa o protocolo de comunicação OPC UA para se conectar ao CLP em um dispositivo simulado no ambiente de desenvolvimento CODESYS.

Kormann, Tikhonov e Heuser (2012) apresentam uma arquitetura de sistema que incorpora diagramas UML de sequencia para testes em CLP. Usando o CoDeSys V3 para simulação, a aplicação teste é escrita em diagrama de blocos. O trabalho tem foco em preocupações arquitetônicas e semânticas de diagramas de sequência UML para especificações de casos de teste.

Já Charbachi e Eklund (2016) utilizaram a técnica de Pairwise para gerar comparações entre testes gerados automaticamente em CLPs com testes criados manualmente por engenheiros industriais. Os testes foram avaliados em termos de detecção de falhas, cobertura do código e custo. Uma ferramenta criada anteriormente chamada Combinatorial Test Tool (CTT) foi estendida para oferecer suporte aos testes propostos.

Na Tabela 2 são resumidas as principais características dos trabalhos

relacionados.

Tabela 2 – Características dos trabalhos relacionados

Autor(es):	Ferramenta utilizada:	Entrada:	Saída:
Zoubek, Roussel e Kwiatkowska (2003)	Uppaal.	Programas em ladder.	Verificação de controle para especificações não triviais.
Sudheerbabu (2021)	UPPAAL e Pytest.	Programas em diagrama de blocos.	Geração automática de casos de teste são significativamente mais rápidos do que o processo de teste manual.
Kormann, Tikhonov e Heuser (2012)	sequência de diagramas UML usados para descrever o comportamento esperado da máquina / planta.	Programas em diagrama de blocos.	Diagramas de sequência UML como um sistema integrado de solução para especificação e execução de casos de teste.
Charbachi e Eklund (2016)	Combinatorial Test Tool (CTT).	Programas em ladder.	Testes manuais tiveram um desempenho ligeiramente melhor em relação a técnica de Pairwise.

Fonte: Elaborado pelo autor (2021).

A partir das pesquisas feitas, é visto que a utilização de modelo baseada em UPPAAL para verificar e testar é a mais recorrente. Assim como o uso da linguagem em ladder e diagrama de blocos, que são as linguagens mais populares para programação em CLP.

3 MATERIAIS E MÉTODOS

Este capítulo detalha a metodologia de desenvolvimento para realizar os testes do sistema proposto. Abordando como é feita a comunicação cliente e servidor via OPC UA, os critérios para elaboração dos casos de testes, as ferramentas utilizadas que conseguem cobrir os testes estudados e sobre a interface da aplicação.

3.1 COMUNICAÇÃO OPC UA

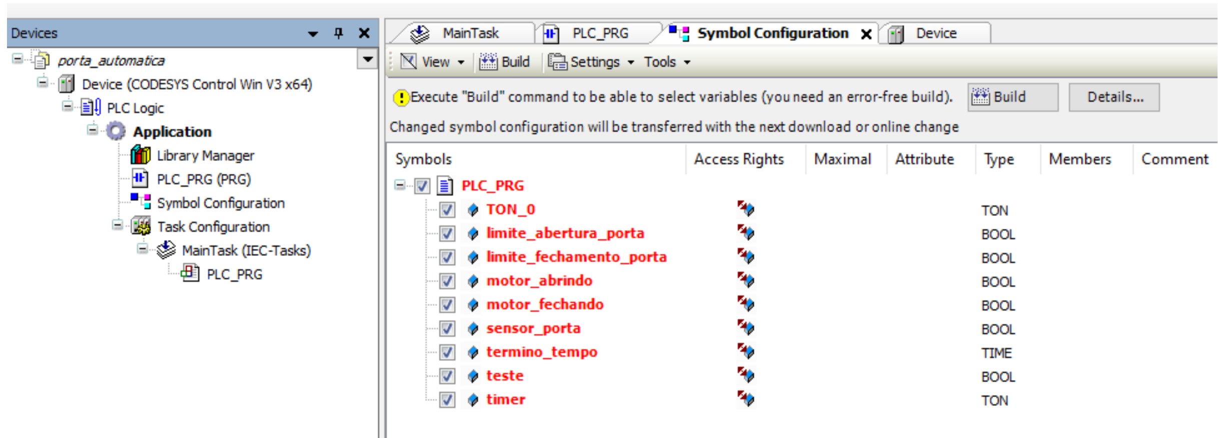
Para alcançar os objetivos, foi necessário investir tempo para compreender o funcionamento do protocolo OPC UA e como utilizá-lo nos softwares de simulação. Foi utilizado o Codesys, o qual já tem por padrão o protocolo OPC UA para sua comunicação. O Codesys tem função de modelar automações industrias simulando um CLP virtual.

Codesys é um ambiente de desenvolvimento para controladores programáveis de acordo com a norma IEC 61131-3, é desenvolvido e comercializado pela 3S-Smart Software Solutions GmbH. Sua interface de desenvolvimento é gratuita e não é orientado a um tipo específico de hardware de controlador programável ou sistema embarcado, inclusive não dependendo do fabricante do hardware. É amplamente utilizado na indústria para diversos tipos de automações, variando de lógicas simples à robótica, controle de movimento e CNC (Comando Numérico Computadorizado) (GROUP, 2022).

O Codesys foi utilizado no projeto por possuir um simulador de CLP integrado e oferecer configurações de comunicações com protocolos industriais em forma de biblioteca. Permitindo simular um programa de CLP e realizar sua comunicação por protocolo OPC UA via software.

Para tornar a estrutura acessível via OPA UA no Codesys, ela deve ser definida em uma configuração de símbolos como um objeto, o objeto deve ser adicionado a aplicação de forma que seja possível chamá-lo. Também deve ser alterada a política de comunicação do device para permitir login anônimo em versões do Codesys V3 SP17 ou maiores, possibilitando o sistema fazer a comunicação OPC UA sem fornecer credenciais. Na Figura 11 é ilustrada a configuração de símbolos no Codesys, onde é adicionado na aplicação as variáveis desejadas, assim sendo possível fazer a leitura e escrita destes dados a partir da comunicação pelo protocolo OPC UA.

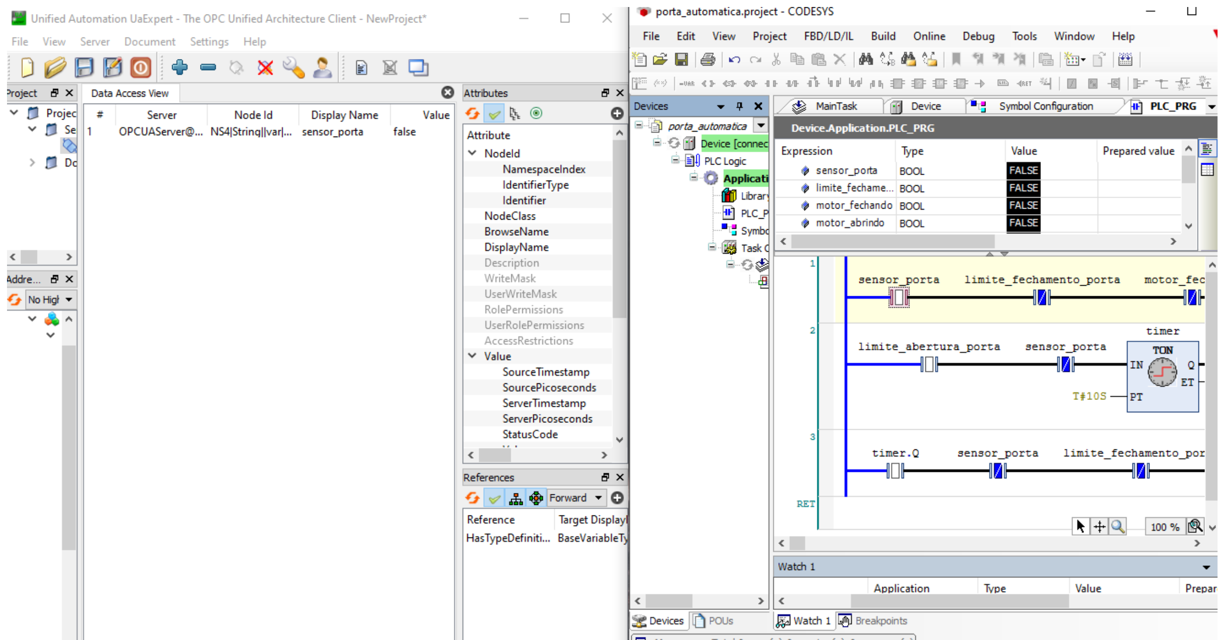
Figura 11 – symbol configuration no Codesys



Fonte: Autor (2022)

Para fins de teste foi utilizado o software UaExpert (UAEXPERT, 2022), que tem função de acessar dados e métodos via OPC UA. A Figura 12 demonstra a comunicação pelo software. Pelo UaExpert é possível identificar o caminho de cada variável e acompanhar o estado atual dela no programa feito no Codesys. A configuração para comunicar nos programas de testes em python depende dos parâmetros do endereço IP, porta de entrada e tags, onde deve ser descrito o caminho até as variáveis do programa.

Figura 12 – Testes de comunicação via UaExpert



Fonte: Autor (2022)

O Codesys possui um servidor próprio, que pode ser utilizado para fazer a comunicação OPC UA entre o servidor e cliente. No projeto foi utilizado a biblioteca opcua para python para criar o cliente que comunica com o servidor do codesys. A sua instalação é realizada pelo terminal com o seguinte comando do pip3:

```
$ pip3 install opcua
```

A declaração da biblioteca opcua e suas funções que inicializam e interrompem a conexão entre cliente e servidor são ilustradas na Figura 13. Client é a classe que faz a conexão e pega os dados do servidor e ua são os módulos que contém ferramentas como tipos de conversão. Client nas funções de conexão tem como parâmetro os dados da rede a ser utilizadas, contendo o número do IP e porta da rede.

Figura 13 – Uso da biblioteca opcua

```
from opcua import Client
from opcua import ua
import py
import time
import pytest


def OPC_Disconnect(client):
    client.disconnect()


def OPC_Connect(client) :

    client.connect()
```

Fonte: Autor (2022)

É utilizado um arquivo em python somente para declaração das dependências e métodos atribuídos a comunicação OPC UA. Na Figura 14 é apresentado outras funções que são utilizadas para os testes unitários, onde os parâmetros para as funções de escrita e leitura dos dados são baseados no objects, que é o nó da raiz do servidor, que contém a base para todas variáveis. Outros parâmetros incluem o nome, o valor e o Vpath, que é o caminho da variável,

Figura 14 – Funções de escrita e leitura dos dados

```
def read_opc_var (objects,vpath,var_name) :

    vpath[len(vpath)-1]=str("4:")+ str(var_name)
    var_path=  objects.get_child(vpath)
    var_type= get_opc_var_type(var_path)

    return var_path.get_value()

def write_opc_var (objects,vpath, var_name, value):

    vpath[len(vpath)-1]=str("4:")+ str(var_name)
    var_path=  objects.get_child(vpath)
    var_type= get_opc_var_type(var_path)

    return var_path.set_value(value, var_type)
```

Fonte: Autor (2022)

3.2 ELABORAÇÃO DOS TESTES

Os programas de CLP em ladder a serem testados são sistemas do tipo caixa-preta, no qual somente se tem acesso às entradas e saídas do sistema. No teste baseado em especificação (ou teste caixa-preta) o objetivo é determinar se o programa satisfaz aos requisitos funcionais e não-funcionais que foram especificados (LABICHE, 1997).

Conforme Labiche (1997) explica por definição, um programa é considerado adequadamente testado se tiver sido coberto de acordo com os critérios selecionados. Desta forma foram elaborados dois casos de testes no Codesys com seus respectivos critérios.

- Caso 1) Porta Automatizada:
  Modelo de porta automatizada tem seu funcionamento baseado nos sensores, que quando estiverem ligados e os interruptores de limite da porta não estiverem ativados, então o motor estará atuando para abertura ou fechamento da porta;
- Caso 2) Tanque Industrial:
  Modelo de um tanque de ressonância magnética para o controle de nível e fluxo. Tem como objetivo controlar o líquido para variar entre seu limite máximo e mínimo, verificando o nível de hélio para manter no estado líquido de forma que faça seu papel de resfriamento. Uma máquina de ressonância magnética deve manter seu hélio líquido até um limite, no qual o hélio é usado como agente criogênico para resfriar bobinas magnéticas supercondutoras, que por sua vez mantém o campo magnético funcionando.

Os casos de teste são descritos nos requisitos funcionais e não funcionais, cobrindo todos os cenários descritos. Cada caso compõe uma série de entradas ou pré condições, sendo o cenário atual, e saídas ou pós condições, apresentando o comportamento esperado do sistema. As entradas e saídas devem ser descritas objetivamente de forma formal, que possam ser posteriormente comprovadas através de testes. As Tabelas 3 e 4 apresentam cada caso com seus respectivos requisitos funcionais e não funcionais.

Tabela 3 – Requisitos Funcionais

Casos de uso	Requisito	Descrição
Porta Automatizada	1.1	A porta deve abrir quando o sensor for ativado.
	1.2	A porta deve permanecer aberta enquanto o sensor estiver ativado.
	1.3	Se o sensor estiver desligado e o interruptor de limite de fechamento não estiver ativado o motor deverá fechar a porta.
	1.4	Deve haver o uso de sensores fim de curso para indicar o máximo deslocamento da porta.
Tanque Industrial	1.1	Deve haver regulagem do nível do tanque.
	1.2	Caso o botão de reset seja apertado, deve haver o processo de quench. O quench ocorre quando há uma emergência no equipamento, onde o hélio é esquentado rapidamente se transformando em gás e sendo dissipado pela abertura de válvulas, dessa forma interrompendo a atuação do campo magnético.
	1.3	Após ultrapassar o nível do tanque em 100 porcento, como exemplo é utilizado o valor da pressão de 5 PSI para 100 porcento, a válvula de abertura deve ser acionada para diminuir seu volume.
	1.4	Deve haver um controle das válvulas para o sistema variar entre 75 porcento até 100 porcento continuamente.

Tabela 4 – Requisitos não Funcionais

Casos de uso	Requisito	Descrição
Porta Automatizada	1.1	Tempo para porta permanecer aberta depois do sensor desativar ser de 10 segundos ou mais.
	1.2	Abertura da porta variar entre 15 a 25 segundos.
	1.3	Velocidade de abertura da porta ser maior em relação a seu fechamento.
Tanque	1.1	Indicadores informando quando os sensores são ativado
	1.2	Alarmes para indicar o nível máximo e mínimo do tanque.
	1.3	Permitir a visualização do líquido e mostrar via display o volume atual.

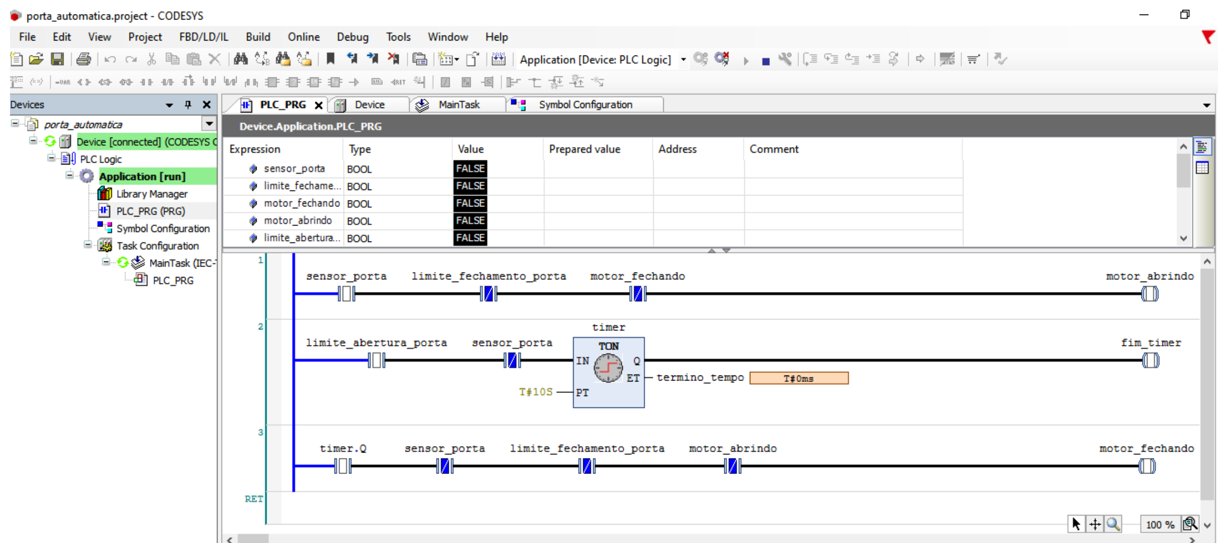
Parar criar critérios para cada caso de teste foi utilizando a técnica de particionamento em classes de equivalência, onde é preciso identificar as condições de entrada, suas classes válidas e invalidas. Usando o exemplo da porta automatizada, onde se espera que ela permaneça por pelo menos 10 segundos aberta. Na Tabela 5 é apresentado os critérios para o caso da porta começar a fechar.

Tabela 5 – Partição de equivalência para o caso 1

Condições de entrada	Classes válidas	Classes inválidas
Tempo para porta permanecer aberta	$0s<t<10s$	$t>10s$
Sensor ativo	Não	Sim.
Limite da chave de fim de curso	Sim	Não

Na Figura 15 é apresentado o programa da porta automatizada executado no codesys, o qual possibilita entender a lógica em ladder para a variável ser verdadeira, onde deve ter suas condições de entradas verdadeiras e seu timer estabelecido acima de 10 segundos.

Figura 15 – Programa em ladder do caso 1



Fonte: Autor (2022)

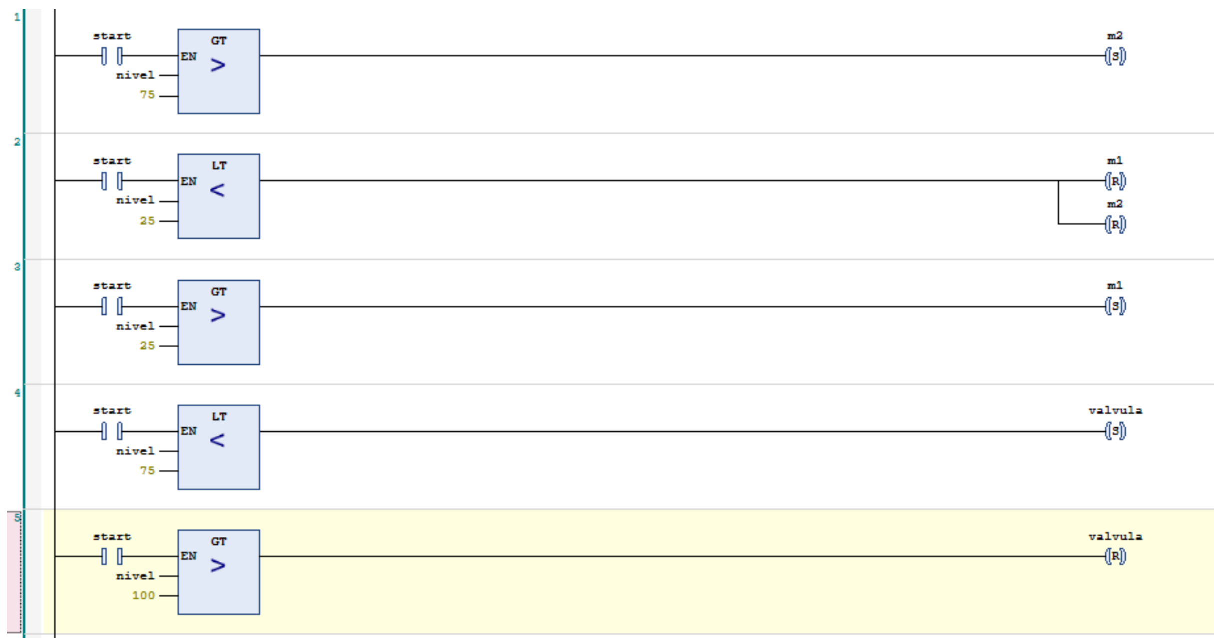
Utilizando o segundo caso do tanque de criogenia, o sistema deve variar o o líquido entre 75 e 100 porcento do seu volume, chegando ao máximo de 5 PSI para que o sistema de refrigeração continue eficiente. A Tabela 6 apresenta a partição de equivalência deste caso.

Tabela 6 – Partição de equivalência para o caso 2

Condições de entrada	Classes válidas	Classes inválidas
Válvula aberta	75% <nível< 100%	75% >nível> 100%
Reset	Não	Sim
bomba m1 atuando	sim	não
bomba m2 atuando	sim	não
Start	Sim	Não
Pressão do hélio no seu limite máximo	pressão < 5 PSI	pressão > 5 PSI

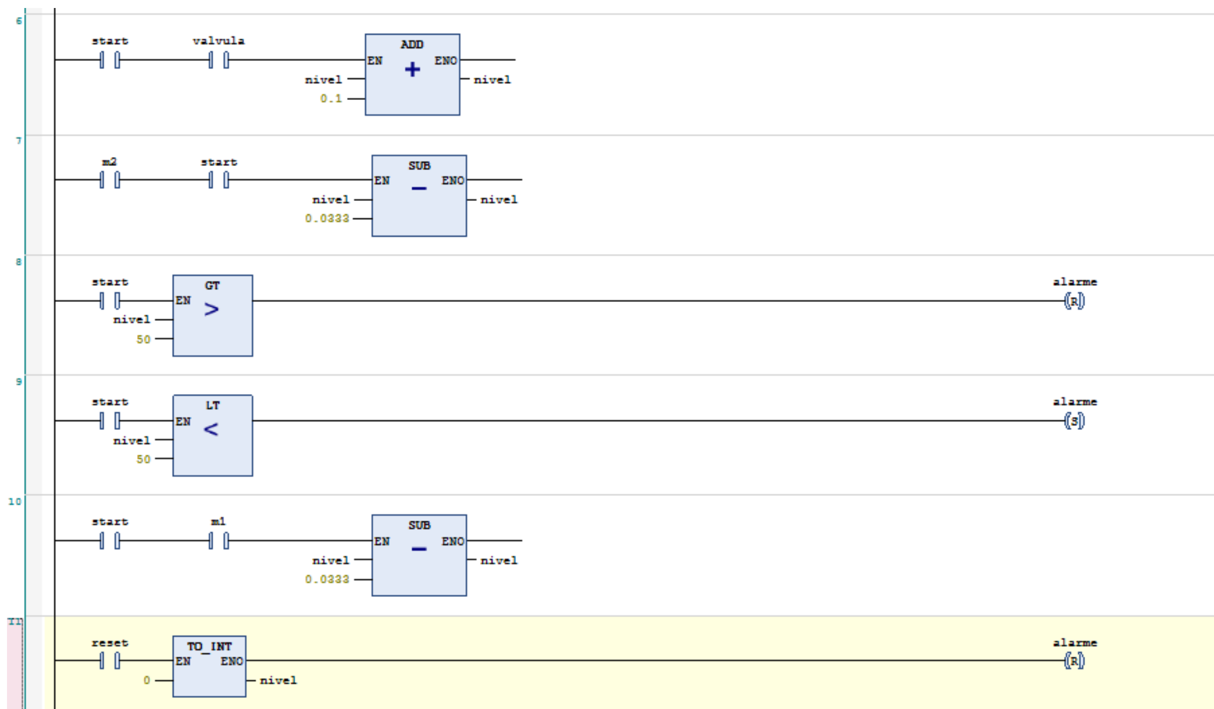
O sistema do tanque criogênico elaborado no Codesys é ilustrado nas Figuras 16 e 17. O diagrama em ladder proposto é composto por onze linhas, onde há uma série de blocos de inequações, adição e subtração que controlam o nível do tanque. Através das bombas denominadas m1 e m2 é aumentado o volume do líquido no tanque e com o acionamento ou não da válvula é mantido o equilíbrio do sistema.

Figura 16 – Programa em ladder do caso 2



Fonte: Autor (2022)

Figura 17 – Programa em ladder do caso 2 com os blocos de adição e subtração



Fonte: Autor (2022)

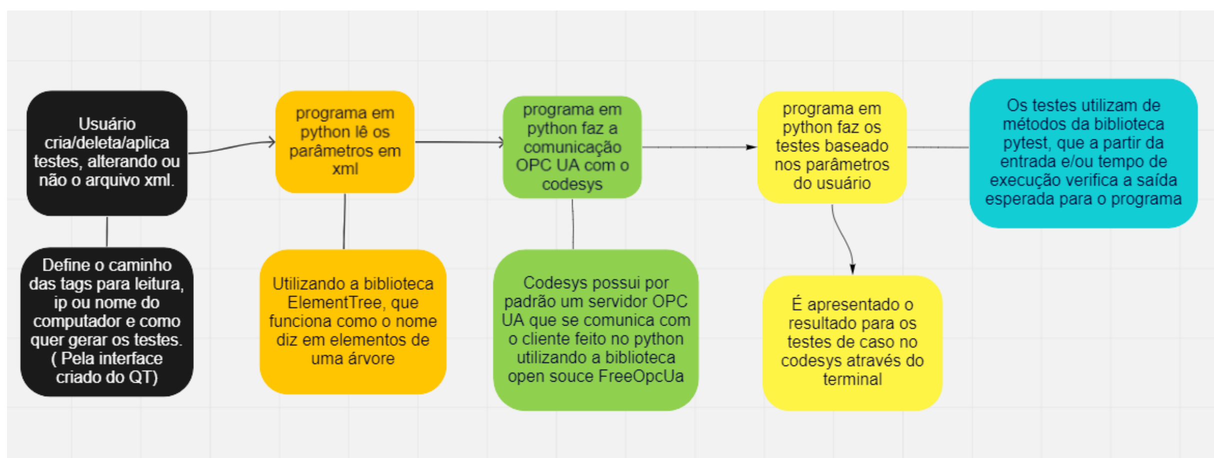
3.3 GERENCIAMENTO DOS CASOS DE TESTES

O gerenciamentos de cada caso de teste foi feito após um levantamento para escolha das ferramentas, foram escolhidas as que conseguem cobrir os critérios de testes estudados e possibilitam sua utilização livre (Open Source). As ferramentas de programação com suas respectivas dependências que foram utilizadas:

- A biblioteca Pytest na versão pytest-6.2.5 para testes unitários, onde é verificado cada entrada e saída esperada com seu tempo de execução;
- O Python na versão 3.9.13, que foi utilizado em todo projeto simplificando a comunicação e execução de todos códigos;
- Codesys na versão 3.5 SP17, software usado para criação de programas em ladder e com protocolo de comunicação OPC UA disponível;
- QTdesigner e a biblioteca PyQt5 para criação da interface do usuário. QTdesigner é a IDE para criação do designer e o PyQt5 é a dependência que permite usar o frameork Qt GUI do Python;
- A biblioteca ElementTree, responsável pela escrita e leitura de cada teste nos arquivos em XML (Extensible Markup Language), com suas respectivas entradas e saídas esperadas;
- A dependência OS no python é uma biblioteca de comandos do sistema operacional, que é utilizada no projeto para executar comandos no terminal;
- A biblioteca time no python, responsável pelo gerenciamento do tempo no código de execução dos testes;
- O software Anaconda, que é uma distribuição das linguagens de programação Python, que visa simplificar o gerenciamento e implantação de pacotes. Usado para instalar o PyQt, onde foi convertido arquivos .ui do QTdesigner para python;
- O editor VS Code, utilizado para edição e compilação de todos arquivos.

O sistema de criação de testes unitários foi elaborado de forma que o usuário possa criar e executar testes diretamente do software criado pelo QT. Na Figura 18 é apresentado um fluxograma do funcionamento para execução dos testes.

Figura 18 – Fluxograma geral do sistema



Fonte: Autor (2022)

Os programas em python tem como objetivo obter os dados de rede da comunicação OPC UA, verificar se os dados são providos pelo sistema, compilar a interface para o usuário, manipular os arquivos em XML e se as informações retornam

da forma esperada de acordo com cada teste.

Para manipulação dos arquivos em XML foram criadas hierarquias com o uso da biblioteca ElementTree, no exemplo da Figura 19 é apresentado um dos casos de teste e como foi organizado suas informações. Cada caso de teste é separado com uma id e seu nome. Suas sub-hierarquias são divididas entre entradas, saídas, seu histórico de resultados, seu último resultado e a última data que o teste ocorreu. As variáveis tem as informações dos nomes, valores em forma de string, seu tempo e caso seja atribuído ”True” em ”menor” como no exemplo, o código interpreta para que o valor esperado seja menor que o estabelecido.

Figura 19 – Exemplo de teste criado em XML

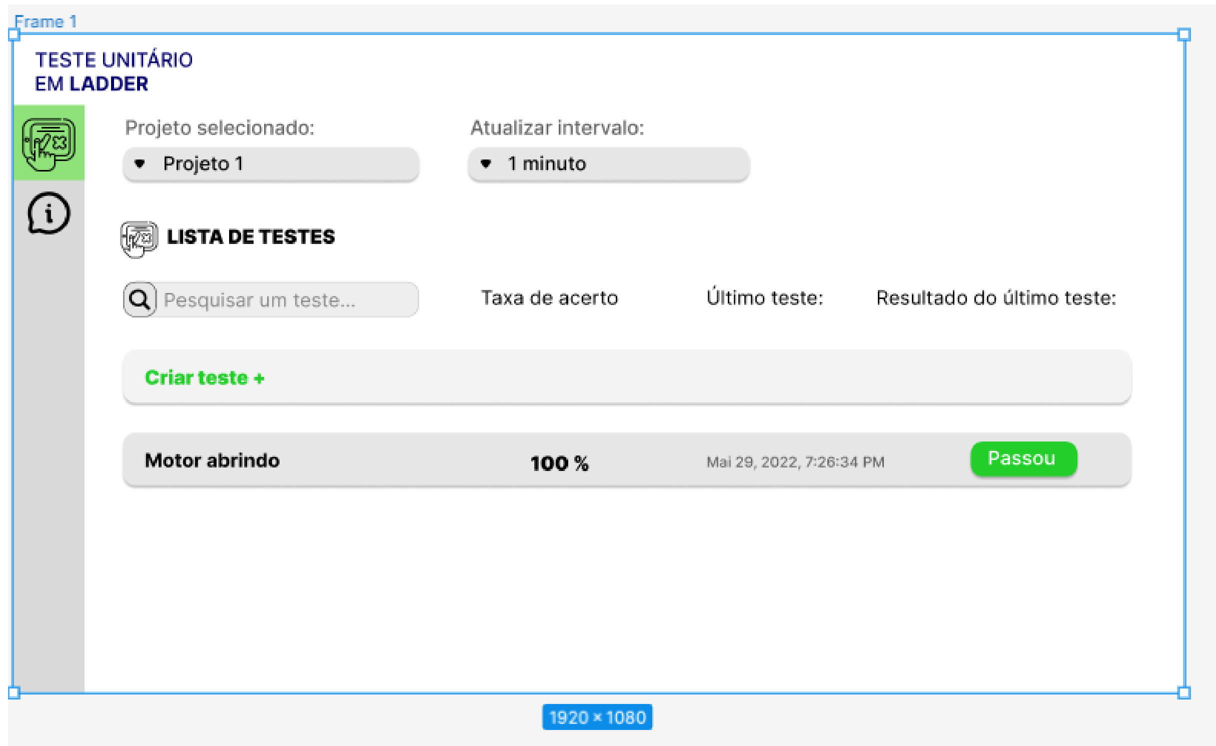
```
<test-case name="teste minimo por 60s" id="6">
    <input>
        <var name="start" value="True" />
        <var name="nivel" value="76" />
    </input>
    <expected>
        <var name="nivel" value="75" menor="true" time="60" />
    </expected>
    <result>
        <var name="True" value="3" />
        <var name="False" value="15" />
    </result>
    <last_result>
        <var name="False" value="1" />
    </last_result>
    <last_time>
        <var name="tempo" value="2022-11-27 22:34:39.459612" />
    </last_time>
</test-case>
```

Fonte: Autor (2022)

3.4 CRIAÇÃO DA INTERFACE

Inicialmente foram elaboradas telas de protótipo para aplicação no figma (FIGMA, 2022), que é um software pra edição gráfica utilizado para aplicativos desktop e mobile. As telas do figma serviram de modelo para a versão final, que foi modificada de acordo com as funcionalidades e limitações do projeto. Nas Figuras 20 e 21 são apresentadas as telas criadas no figma. O design das telas foi inspirado numa interface limpa e moderna, utilizando de cores claras e com bordas arredondas para os botões, campos de texto e as caixa de combinação.

Figura 20 – Exemplo elaborado no figma



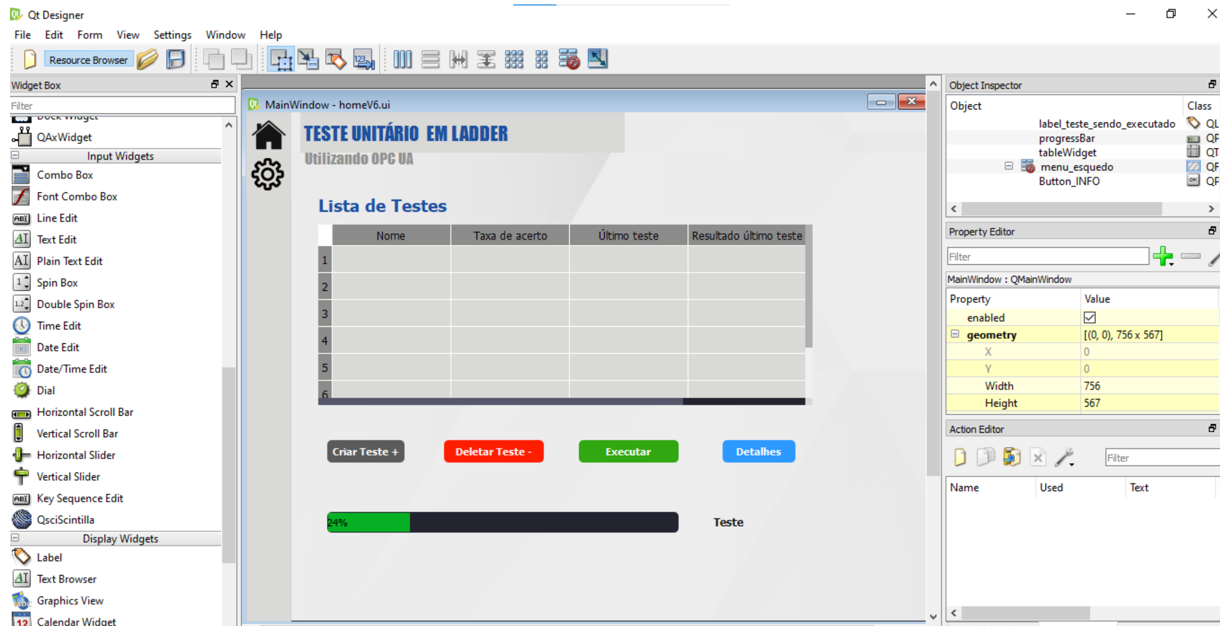
Fonte: Autor (2022)

Figura 21 – Tela de diagnóstico dos testes no figma

Fonte: Autor (2022)

Para desenvolver a interface do figma para python foi utilizado a IDE QtDesigner, onde é criado o layout inicial da aplicação na forma de um arquivo .ui que pode ser convertido para python ou C++. Na Figura 22 é exibida a tela inicial da aplicação criada no QtDesigner.

Figura 22 – Exemplo elaborado no QtDesginer



Fonte: Autor (2022)

O QtDesigner já possui a organização de classes do PyQt, de maneira que ao ter arquivo compilado em python é possível ter o layout do aplicativo pronto para uso. Para sua conversão foi utilizado o software Anaconda, no terminal do anaconda foi instalado através do pip3, que é um sistema de gerenciamento de pacotes para python 3, o PyQt5 com o seguinte comando:

```
$ pip3 install PyQt5
```

E para sua conversão dos arquivos .ui a linha de comando utilizada:

```
$ pyuic5 -x homeV6.ui -o homeV6.py
```

A biblioteca do PyQt é composta por vários widgets, incluindo botões, checkboxes, labels, caixas de texto e barras de navegação. Todos widgets são organizados por meio de classes que possuem seus respectivos métodos e atributos. Outra função do Qtdesginer é ao ter o arquivo convertido já formar uma classe chamada UiMainWindow, a qual possui todos os widgets utilizados, com seus parâmetros e seu StyleSheet determinado no software.

4 RESULTADOS E DISCUSSÕES

As características gerais do software estão baseadas na aplicação onde seja possível que o usuário crie os testes, os execute e tenha seus resultados obtidos no software. Todos testes são executados com base no programa do Codesys que o usuário escolher.

4.1 CONFIGURAÇÕES INICIAIS

Para criação dos testes no software deve-se primeiramente configurar os dados de rede, a versão do Codesys e o nome atribuído ao programa do Codesys. Na interface da tela inicial presente na figura 23, ao clicar no ícone de ferramentas é aberto a tela de configurações, que grava cada uma das informações num arquivo XML. Este arquivo XML tem sua leitura realizada no programa em python, que por sua vez tem função de configurar o caminho das variáveis do Codesys a serem utilizadas e ajustar os padrões de rede do usuário.

Figura 23 – Tela de configurações da aplicação

Fonte: Autor (2022)

4.2 DESENVOLVIMENTO DOS TESTES

Para criação dos testes por meio da aplicação foi elaborado a tela apresentada na Figura 24, onde tem cada parâmetro que vai ser salvo no arquivo XML dos casos de testes. Na tela é pedido ao usuário o nome do teste ou selecionar na tabela um teste já criado. Escolher entre saída ou entrada, valor da variável, tempo de execução e caso seja saída como vai ser a saída esperada (igual/maior que/menor que).

Figura 24 – Execução dos testes a partir da interface do PyQt

Fonte: Autor (2022)

Todas variáveis serão strings que ao passar pelas condições de cada teste utilizam o método eval(), a qual é uma função integrada em Python que transforma uma cadeia de caracteres em código utilizável. Dessa maneira possibilitando entradas e saídas em formato de string, booleano, números inteiros e reais.

Na Figura 25 é ilustrado o código que adiciona variáveis aos testes presentes na tabela. Caso alguma célula da tabela esteja selecionada é adicionado a entrada ou saída que o usuário inseriu, que corresponde a linha da célula na tabela de testes. Ao percorrer o caminho do arquivo XML de testes é acrescentado como um sub elemento do nó do loop a variável desejada ao teste.

Figura 25 – Adicionando variáveis aos testes

```
#---------------Leitura dos arquivos xml, declarando o nó principal de cada um-----------------#
xmlfile_ = 'linha_cria_teste.xml'
tree_ = ET.parse(xmlfile_)
root_ = tree_.getroot()
xmlfile = 'output.xml'
tree = ET.parse(xmlfile)
root = tree.getroot()
#---------------Caso o usuário escolha um teste da tabela-----------------#
for elm in root_:
        numero_linha_ = elm.text
        numero_linha = int(numero_linha_)
if numero_linha != None:
  if (entrada_saida == "Entrada"): # Verifica se é uma entrada ou saída
        for elm in root.findall("./test-case[@id='"+str(numero_linha + 1)+"']/input"):
                elm.attrib= {'name' : nome_variavel , 'value' : valor_variavel, 'time' : tempo}
  else:
   #---------------Condições de inequações-----------------#
   if (saida_esperada == 'Maior que (>)'):
     for elm in root.findall("./test-case[@id='"+str(numero_linha + 1)+"']/expected"):
         SubElement(elm,'var',name=nome_variavel,value=valor_variavel,time =tempo,menor='',maior='True')
   if (saida_esperada == 'Igual (=)'):
     for elm in root.findall("./test-case[@id='"+str(numero_linha + 1)+"']/expected"):
         SubElement(elm,'var',name=nome_variavel,value=valor_variavel,time =tempo,menor='',maior='')
   if (saida_esperada == 'Menor que (<)'):
     for elm in root.findall("./test-case[@id='"+str(numero_linha + 1)+"']/expected"):
         SubElement(elm,'var',name=nome_variavel,value=valor_variavel,time =tempo,menor='true',maior='')
  #---------------gravação dos testes e identação no arquivo xml-----------------#
  tree = ET.ElementTree(root)
  ET.indent(tree, space="\t", level=0)
  tree.write(xmlfile, encoding="utf-8")
  self.prencher_tabela() #Atualiza a tabela de testes
```

Fonte: Autor (2022)

O restante da função que adiciona testes no arquivo XML é mostrada na Figura 26, onde o usuário pode escolher por criar um novo teste. Para criação é adicionado uma série de elementos a partir do id do teste para organizar os valores de entrada, saída, último resultado, histórico dos resultados e o tempo. Por fim é chamado o método de preencher tabela, que faz a leitura do arquivo XML atualizado e insere os dados em cada linha.

Figura 26 – Função de criar testes

```
#---------------Caso o usuário escolha criar um novo teste----------------#
else:
        numero_testes=0
        for value in root.findall("./test-case"):
                numero_testes=numero_testes+1
        head = SubElement(root, 'test-case', name= nome_teste, id = str(numero_testes + 1))
        expected = SubElement(head, 'expected')
        input = SubElement(head, 'input')
        if (entrada_saida == "Entrada"):
            var=SubElement(input, 'var', name= nome_variavel, value = valor_variavel, time = tempo)
        else:
            var=SubElement(expected,'var',name=nome_variavel,value=valor_variavel,time=tempo,menor='',maior='')
        result = SubElement(head, 'result')
        var = SubElement(result, 'var', name= "True", value = '0')
        var = SubElement(result, 'var', name= "False", value = '0')
        last_result = SubElement(head, 'last_result')
        var = SubElement(last_result, 'var', name= "True", value = '')
        last_time = SubElement(head, 'last_time')
        var = SubElement(last_time, 'var', name= "tempo", value = '-')
        #--------------gravação dos testes e identação no arquivo xml----------------#
        tree = ET.ElementTree(root)
        ET.indent(tree, space="\t", level=0)
        tree.write(xmlfile, encoding="utf-8")
        self.prencher_tabela()
```

Fonte: Autor (2022)

4.3 EXECUÇÃO DOS TESTES

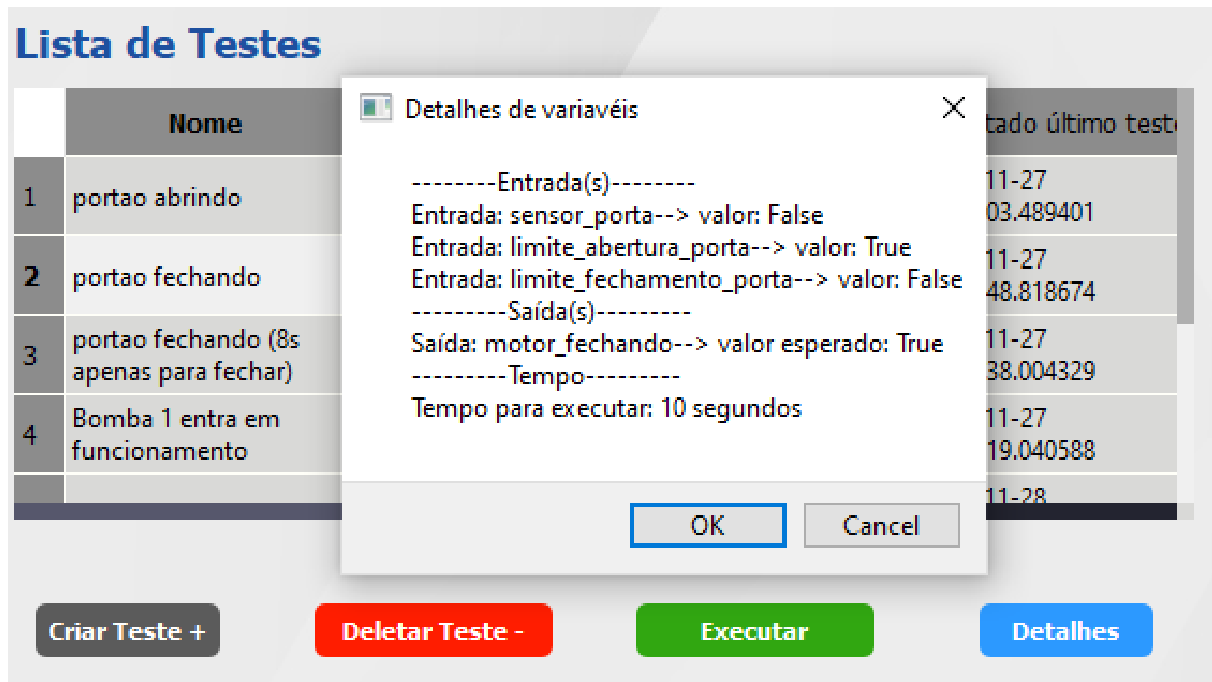
Ao abrir a aplicação o usuário tem uma tabela contemplando cada teste criado, para sua execução é necessário selecionar as linhas com testes de interesse e por fim apertar no botão de executar conforme é ilustrado na Figura 27.

Figura 27 – Tela inicial da aplicação

Fonte: Autor (2022)

Em cada teste é disponibilizado sua taxa de acerto, resultado do último teste e a data do último teste. Na tela inicial é possível criar testes, deletar testes, executar testes e ter detalhes das variáveis de cada teste como mostra a Figura 28, ao apertar em detalhes um Popup é aberto com informações de entrada, saída e o tempo de execução do teste selecionado.

Figura 28 – Popup ao clicar em detalhes no teste selecionado



Fonte: Autor (2022)

O programa em python identifica cada linha associada aos testes salvos no arquivo XML e efetua por meio de um método do botão de executar teste, que ao ser clicado, fazer a chamada da função teste linha, a qual abre um terminal e executa o código que cobre testes unitários com o pytest. O código que faz a chamada para execução dos testes é apresentado na Figura 29.

Figura 29 – função teste linha

```
def teste_linha(self):
    _translate = QtCore.QCoreApplication.translate
    self.label_teste_sendo_executado.setText(_translate("MainWindow", "testes executados"))
    self.progressBar.setProperty("value", 50)
    os.system('python3 -m pytest  ')
    self.progressBar.setProperty("value", 100)
    self.prencher_tabela()
```

Fonte: Autor (2022)

A partir da biblioteca os é chamado via terminal a linha de comando

```
$ python3 –m pytest
```

É executado todas as funções dos programas em python presentes no mesmo diretório que contém a importação da biblioteca do pytest, é feito a varredura de cada função de forma que caso tenha alguma condição no final da função, está será verificada. As execuções dos testes funcionais incluem testes parametrizados pelo pytest, a Figura 30 mostra como é organizado a chamada para cada teste utilizando o método parametrize do pytest. Por meio do loop é gerenciado todos parâmetros a serem testados pela

função teste XML, cada parâmetro é uma linha selecionada pelo usuário que contém o teste de interesse.

Figura 30 – método parametrize

```
#--------------função parametrize do pytest faz a chamada para função de testes----------------#
params = []
for i in range(len(linhas)):
    params.append(linhas[i])
@pytest.mark.parametrize("n_linha", (params))
#--------------Verificação de todos testes que estão salvos no arquivo xml ----------------#
def teste_xml(n_linha):
```

Fonte: Autor (2022)

Conforme indicado na Figura 31 é mostrado o código para verificações dos testes. Primeiramente é realizado a condição para verificar se o teste correspondente a linha da tabela na aplicação é igual ao id dos testes presentes no arquivo em XML.

Figura 31 – método teste XML

```
#--------------Verificação de todos testes que estão salvos no arquivo xml ----------------#
def teste_xml(n_linha):
    tam_entrada=1
    tam_saida=1
    objects=server.getvar(client)
    for value in root.findall("./test-case"):
        id_ = int(value.attrib["id"])
        if id_ == int(n_linha) +1:
            vet_input = []
            for entrada in root.findall("./test-case[@id='"+str(id_)+"']/input"):
                tam_entrada=len(entrada)  # tam_entrada recebe o número de entradas desse teste
            for type_tag in root.findall("./test-case[@id='"+str(id_)+"']/input/var"):
                value = type_tag.get('value')
                name = type_tag.get('name')
                obj_time = type_tag.get('time')
                if(obj_time == None):
                    obj_time= 0  # zero no caso do usuário não restringir o tempo de execução
                vet_input.append(teste(name,value,obj_time,nome_teste[id_-1],0,0))
```

Fonte: Autor (2022)

Posteriormente é adicionado cada variável de entrada num vetor que inicializa uma classe para guardar cada valor de entrada. Na Figura 32 é ilustrado como é acrescentado as variáveis no vetor de entrada e o acionamento dos valores no programa do Codesys, onde é chamado o método de escrita do servidor colocando o caminho e valores das entradas correspondentes.

Figura 32 – funções de acionamento para as entradas

```
for i in range(tam_entrada):
  if(vet_input[i].value == "True" or "False"):
    server.write_opc_var(objects,varpath, vet_input[i].name, eval(vet_input[i].value))
  else:
    server.write_opc_var(objects,varpath, vet_input[i].name, float(vet_input[i].value))
  time.sleep(int(vet_input[i].time))
for saida in root.findall("./test-case[@id='"+str(id_)+"']/expected"):
    tam_saida=len(saida)  # tam_saida recebe o número de saidas desse teste
for type_tag in root.findall("./test-case[@id='"+str(id_)+"']/expected/var"):
    value = type_tag.get('value')
    name = type_tag.get('name')
    menor = type_tag.get('menor')
    obj_time = type_tag.get('time')
    if(obj_time == None):
        obj_time= 0
    maior = type_tag.get('maior')
    if(maior != "True"):
        maior= "False"
    if(menor != "True"):
        menor= "False"
    vet_output.append(teste(name,value,obj_time,nome_teste[id_-1],maior,menor))
```

Fonte: Autor (2022)

Assim como é realizado para as variáveis de entrada é percorrido o arquivo em XML para adicionar os valores de saída no vetor de saída. Com a diferença que há as variáveis de maior e menor que comparam o valor da saída com o esperado respectivamente como maior ou menor conforme é ilustrado na Figura 33.

Figura 33 – Comparação se a saída é menor ou maior com o resultado esperado

```
#--------------teste se a saída é maior que a esperada  ----------------#
if(vet_output[j].maior=="True"):
  t_end = time.time() + int(vet_output[j].time)
  while time.time() < t_end: # testa a condição pela quantidade de segundos que o usuário determina
    data = server.read_opc_var(objects, varpath,vet_output[j].name)
    expected = float(vet_output[j].value)
    if(data>expected):
      print("teste ", id_ ," - ", nome_teste[id_ -1], " falhou")
      print("saida", vet_output[j].name, " foi = " , data)
      print("saida esperada para ", vet_output[j].name, "era menor=" , expected)
      assert data > expected
  data=1
  expected=1
#--------------teste se a saída é menor que a esperada  ----------------#
if(vet_output[j].menor=="True"):
  t_end = time.time() + int(vet_output[j].time)
  while time.time() < t_end:  # testa a condição pela quantidade de segundos que o usuário determina
    data = server.read_opc_var(objects, varpath,vet_output[j].name)
    expected = float(vet_output[j].value)
    if(data<expected):
      print("teste ", id_ ," - ", nome_teste[id_ -1], " falhou")
      print("saida", vet_output[j].name, " foi = " , data)
      print("saida esperada para ", vet_output[j].name, "era maior=" , expected)
      assert data < expected
  data=1
  expected=1
```

Fonte: Autor (2022)

Nas condições de inequações é testado se o valor é o esperado pelo tempo determinado pelo usuário, enquanto o valor do método time não for maior que o tempo selecionado em segundos são realizadas as inequações. Na Figura 34 é ilustrado como

é feita a comparação no final usando o assert do pytest. Por fim é feito a comparação entre o valor de saída e o resultado esperado para o teste. De acordo com o resultado gerado é adicionado no histórico os dados dos resultados no arquivo XML, reiniciado os valores de entrada no programa do Codesys e limpado os valores em cada vetor.

Figura 34 – Comparação entre a saída com o resultado esperado

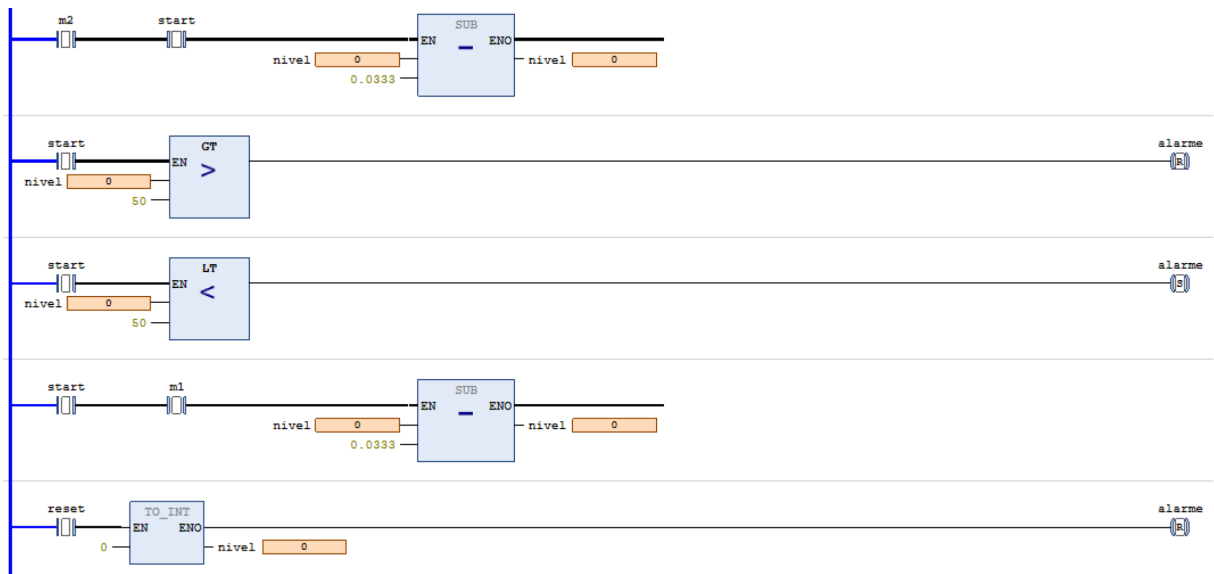
```
#--------------teste se a condição é verdadeira  ----------------#
if (data == expected):
    print("teste ", id_ ," - ", nome_teste[id_ -1], " ok!")
    for elm_ in root.findall("./test-case[@id='"+str(id_)+"']/last_result/var"):
      elm_.attrib= {'name' : 'True' , 'value' : "1" }
      tree.write('output.xml')
    for elm in root.findall("./test-case[@id='"+str(id_)+"']/result/var"):
      value = elm.get('value')
      name = elm.get('name')
      value_ = int(value)
      if name == "True" :
          value_ = value_ + 1
          elm.attrib= {'name' : 'True' , 'value' : str(value_) }
          tree.write('output.xml')

else:
    for elm_ in root.findall("./test-case[@id='"+str(id_)+"']/last_result/var"):
      elm_.attrib= {'name' : 'False' , 'value' : "1" }
      tree.write('output.xml')
    for elm in root.findall("./test-case[@id='"+str(id_)+"']/result/var"):
      value = elm.get('value')
      name = elm.get('name')
      value_ = int(value)
      if name == "False" :
          value_ = value_ + 1
          elm.attrib= {'name' : 'False' , 'value' : str(value_) }
    print("teste ", id_ ," - ", nome_teste[id_ -1], " falhou")
    print("saida", vet_output[j].name, " foi =" , data)
    print("saida esperada para ", vet_output[j].name, "era=" , expected)
    tree.write('output.xml')
#### RESET DOS VALORES DE ENTRADA NO CODESYS #####
for i in range(tam_entrada):
  server.write_opc_var(objects,varpath, vet_input[i].name, False)
vet_input.clear()  # limpa o vetor de entrada
vet_output.clear() # limpa o vetor de saída
assert data == expected
```

Fonte: Autor (2022)

Utilizando dois casos de testes para o exemplo do tanque de criogenia é verificado os eventos de acertos e falhas do código. Para o teste de reset, onde é necessário que o tanque esteja vazio rapidamente para causar o efeito de quench no equipamento, foi utilizado como entrada a variável reset do programa feito no Codesys como True e a saída esperada para o nível ser zero. Na Figura 35 é ilustrado o teste do tanque feito no Codesys, no código é possível identificar que ao acionar o reset. ele por sua vez ativa o bloco que abaixa o nível para zero e reinicia o alarme no final.

Figura 35 – tanque criogênico no Codesys



Fonte: Autor (2022)

Na Figura 36 é apresentado o resultado para o teste de reset realizado pelo pytest.

Figura 36 – Resultado do teste de reset

Fonte: Autor (2022)

Conforme esperado a saída do nível vai para zero e é mostrado no terminal que o teste passou em 1,64 segundos, os dois casos que foram pulados são funções de inicialização e término de comunicação com o programa do Codesys, que não são de interesse para o teste final. Ainda na Figura 36 é possível ver o histórico de cada teste, com sua taxa de acerto, resultado do último teste e a data que ocorreu o último evento.

Utilizando um exemplo que irá falhar, é determinado que o nível não ultrapasse 90 porcento do seu valor final. Na Figura 37 é apresentado a configuração de entrada e saída no arquivo XML feito para testar a falha.

Figura 37 – Arquivo XML do teste de falha

```
<test-case name="teste maximo falho(nivel 90) por 60s" id="7">
    <input>
        <var name="start" value="True" />
    </input>
    <expected>
        <var name="nivel" value="90" maior="True" time="60" />
    </expected>
```

Fonte: Autor (2022)

A partir desses valores de entrada e saída o código em python verifica por 60 segundos se a condição é verdadeira como é mostrado na Figura 38. Se o atributo de maior for true no arquivo XML é testado se a saída é maior que o esperado, onde testa a condição enquanto o tempo determinado pelo usuário em segundos não acabar.

Figura 38 – Verificação do teste de falha

```
#---------------teste se a saída é maior que a esperada -----------------#
if(vet_output[j].maior=="True"):
  t_end = time.time() + int(vet_output[j].time)
  while time.time() < t_end: # testa a condição pela quantidade de segundos que o usuário determina
    data = server.read_opc_var(objects, varpath,vet_output[j].name)
    expected = float(vet_output[j].value)
    if(data>expected):
      print("teste ", id_ ," - ", nome_teste[id_ -1], " falhou")
      print("saida", vet_output[j].name, " foi =" , data)
      print("saida esperada para ", vet_output[j].name, "era menor=" , expected)
      assert data > expected
```

Fonte: Autor (2022)

A figura 39 exibe a saída no terminal do teste de falha, que segundo o modelo de caso proposto o resultado foi como o esperado, é informado qual teste falhou, sua saída no momento que falhou, o tempo executado no teste e como era o resultado da variável esperada.

Figura 39 – Resultado via terminal para o teste de falha

```
                if(data>expected):
                  print("teste ", id_ ," - ", nome_teste[id_ -1], " falhou")
                  print("saida", vet_output[j].name, " foi =" , data)
                  print("saida esperada para ", vet_output[j].name, "era menor=" , expected)
>                 assert data < expected
E                 assert 90.02542114257812 < 90.0

test_unit.py:136: AssertionError
------------------------------------------------------------ Captured stdout call ------------------------------------
teste  7  -  teste maximo falho(nivel 90) por 60s  falhou
saida nivel  foi = 90.02542114257812
saida esperada para  nivel era menor= 90.0
============================================================ short test summary info =================================
FAILED test_unit.py::teste_xml[6] - assert 90.02542114257812 < 90.0
========================================================= 1 failed, 2 skipped in 11.33s ==============================
```

Fonte: Autor (2022)

4.4 VALIDAÇÃO DOS REQUISITOS

Utilizando o primeiro requisito não funcional da Tabela 4 para a porta automatizada, onde a porta deve permanecer aberta depois do sensor desativar ser de 10 segundos ou mais. As variáveis para o caso de teste gravadas no XML são ilustradas na Figura 40.

Figura 40 – Variáveis para o teste do motor fechando

```
<test-case name="portao fechando" id="2">
    <input>
        <var name="sensor_porta" value="False" />
        <var name="limite_abertura_porta" value="True" time="10" />
        <var name="limite_fechamento_porta" value="False" />
    </input>
    <expected>
        <var name="motor_fechando" value="True" />
    </expected>
```

Fonte: Autor (2022)

Para validar também o primeiro requisito funcional da Tabela 3, que deve ter o motor atuando para abrir a porta quando o sensor for ativado. Suas variáveis no arquivo XML são apresentadas na Figura 41.

Figura 41 – Variáveis para o teste do motor abrindo

```
<test-case name="portao abrindo" id="1">
    <input>
        <var name="sensor_porta" value="True" />
        <var name="limite_fechamento_porta" value="False" />
        <var name="motor_fechando" value="False" />
    </input>
    <expected>
        <var name="motor_abrindo" value="True" />
    </expected>
```

Fonte: Autor (2022)

O resultado para os seguintes testes unitários é demonstrado na Figura 42, onde tiveram suas saídas correspondentes as esperadas, passando nos testes em 11,50 segundos.

Figura 42 – Testes para o motor atuar

Fonte: Autor (2022)

As entradas de cada caso de teste são setadas no programa do Codesys, o qual seu servidor informa os dados de saída que são verificados no cliente do programa de testes. No caso do teste do motor fechando o código em ladder tem uma função de bloco que funciona como timer, assim como script de testes unitários há um delay utilizando a biblioteca time com o método time.sleep, neste teste de dez segundos, que após passar esse tempo compara a saída com o resultado esperado.

4.5 LIMITAÇÕES

A aplicação foi feita para rodar testes em qualquer linguagem para CLP, porém pode-se haver necessidade de mudanças no código e no caminho das variáveis para acesso via OPC-UA. Os testes a serem executados são limitados a testes funcionais, o projeto consegue executar testes baseados na entrada, saída e no tempo de execução.

Os testes unitários são feitos de forma isolada do restante do sistema, cujo

objetivo é assegurar a qualidade das unidades de forma individual e não o sistema todo. No projeto são realizados testes unitários em métodos e funções das classes de cada caso, em virtude disso para verificar o comportamento do sistema todo é necessário adicionar testes de mais detalhados em todo conjunto.

Se ocorrer o evento de falha na configuração de rede ou o programa no Codesys não esteja conectado, o código de testes unitários irá falhar antes dos testes serem computados. No entanto, caso o usuário deixe o programa no Codesys parado, os testes terão falhas devido ao sistema ter conexão e passar nas funções de conectividade.

4.6 CONSIDERAÇÕES FINAIS

Notam-se melhorias consideráveis de tempo e investimento nos testes com a implementação da automação para testes funcionais. Caso o usuário deseje realizar testes de regressão, onde é necessário realizar verificações em funcionalidades testadas anteriormente para comparar se as mesmas não foram impactadas, será necessário automatizar apenas uma vez e aplicá-la quando for necessário a validação.

A qualidade e eficácia de testes são dependentes do critério avaliador e se há uma ferramenta de teste que o suporte. Sem a automatização de testes com auxílio de ferramentas, a aplicação pode se tornar propensa a erros.

A partir do modelo de casos de testes é possível realizar qualquer teste unitário com base nas entradas, saídas e o tempo para executá-lo. Os casos utilizados durante o projeto podem ser obtidos para revalidação do software após uma modificação, porém a automação dos testes deverá ser focada em tarefas repetitivas e que geram reiteração.

5 CONCLUSÕES

Este trabalho teve como intuito apresentar o conceito de testes de software, sua importância no mercado e contribuir para uma contínua evolução dos ambientes industriais. A disponibilidade de ferramentas de teste permite a transferência de tecnologia para os ambientes de produção, fator indispensável para a criação de software de excelência. A existência de ferramentas automatizadas auxiliam a adquirir os conceitos básicos na comparação, seleção e de técnicas de teste.

Entre os objetivos do projeto encontra-se a realização de uma investigação dos livros e artigos, por meio de um estudo e revisões da literatura sobre testes de software para automação industrial.

A parir do capítulo 3.1 conclui-se mais um dos objetivos específicos, onde foi discutido como realizar a comunicação OPC UA. Há exito na conexão entre o Codesys e a ferramenta criada por CLP virtual, porém não foi utilizado um CLP real que teria seu funcionamento conforme o simulado.

Nos capítulos 3.2 e 3.4 são cumpridos mais dois requisitos do projeto, onde é elaborado o desenvolvimentos de dois casos de testes de caixa-preta e como foi elaborada a interface no PyQt, respectivamente. O último objetivo específico é testar a abordagem desenvolvida presente nos capítulos 4.3 e 4.4, que foi limitado a testes simples do tipo caixa-preta, verificando as saídas conforme a variação das entradas e o tempo de execução.

Para desenvolver o software foi necessário fazer a busca para métodos de comunicação para CLPs. Com a escolha do uso do codesys, o qual tem um servidor integrado para o procolo OPC UA, foram feitas pesquisas de bibliotecas com diferentes linguagens para atuar como cliente, onde foram priorizadas as que fossem de C++, python e que fossem de código aberto. Com a comunicação entre cliente e servidor em funcionamento foram realizados casos de testes simples, que serviram para testes funcionais. A aplicação final visa criar um software para automação de testes que tenha integração com programas de CLPs.

Ao utilizar a aplicação proposta é possível criar um teste, fazer a comunicação entre cliente e servidor via OPC UA e testá-la automaticamente, utilizando um script de teste unitário. Cada verificação de método é aferida de modo a avaliar as entradas definidas e resultados esperados.

Como trabalhos futuros é possível realizar a implementação de novos critérios para os casos de testes, utilizar um CLP físico com testes práticos para adquirir conhecimentos usando um modelo real.

Novas funcionalidades podem ser adicionadas conforme a necessidade do

usuário, com o código atual pode-se desenvolver novos recursos, ajustes no layout para torná-lo mais intuitivo, com mais opções para validações dos testes e criação de relatórios mais completos para cada teste. Por fim, possibilitar testar outros tipos de dados, melhorar a dinâmica de atrasos dos sinais, testar com programas com múltiplas tarefas.

REFERÊNCIAS

CASTRUCCI, P.; MORAES, C. **Engenharia de automação industrial**. 1. ed. Rio de Janeiro: LTC, 2001.

CASTRUCCI, P.; MORAES, C. **Engenharia de automação industrial**. 2. ed. Rio de Janeiro: LTC, 2006.

CHARBACHI, P.; EKLUND, L. **Pairwise testing for PLC embedded**. 2016. Master's Degree Thesis in Bachelor of Science in Computer Science) — School of Innovation Design and Engineering, Malardalen University, Vasteras, 2016.

DELAMARO, M. et al. **Introdução ao teste de software**. São Paulo: Elsevier Brasil, 2000.

ELIPSE. **ELIPSE E3**. 2021. Disponível em: <https://www.elipse.com.br/en/produto/elipse-e3/>. Acesso em: 15 set. 2021.

FIGMA. **Figma**. 2022. Disponível em: <https://www.figma.com>. Acesso em: 24 out. 2022.

FOUNDATION, O. **OPC Unified architecture specification**. 2021. Disponível em: <https://opcfoundation.org/developer-tools/specifications-unified-architecture>. Acesso em: 15 set. 2021.

FRANCO, I. et al. Utilização do protocolo de comunicação ole for process control em processos industriais. **Exacta**, v. 8, n. 3, p. 319–329, fev. 2011.

GROUP, C. **CODESYS**. 2022. Disponível em: <https://www.codesys.com>. Acesso em: 24 out. 2022.

JAVARONI, T. **Desenvolvimento de uma metodologia para realização de testes em programas de CLP na indústria de petróleo e gás**. 2014. Dissertação (Mestrado em Engenharia de Automação e Sistemas) — Centro Tecnológico, Universidade Federal de Santa Catarina, Florianópolis, 2014.

KORMANN, B.; TIKHONOV, D.; HEUSER, B. Automated PLC software testing using adapted UML sequence diagrams. In: 4TH IFAC SYMPOSIUM ON INFORMATION CONTROL PROBLEMS IN MANUFACTURING. **Proceedings [...]**. Bucharest, Romania: IFAC Proceedings Volumes, 2012. v. 45, p. 1615–1621. Disponível em: <https://bityli.com/Bxh0nv>. Acesso em: 15 ago. 2021.

LABICHE, Y. On testing object-oriented programs. 1997.

LEHNHOFF, S. et al. IEC 61850 based OPC UA communication: The future of smart grid automation. In: 17TH POWER SYSTEMS COMPUTATION CONFERENCE, 2011. **Proceedings [...]**. Stockholm, Sweden: Power Systems Computation Conference (PSCC), 2011. p. 22–26. Disponível em: <https://www.semanticscholar.org/paper/IEC-61850-based-OPC-UA-Communication-The-Future-of-Lehnhoff-Mahnke/844064a5c57f2604db396e5804f52187d9cf7eb0>. Acesso em: 8 set. 2021.

LIANG, Q.; LI, L. The study of soft plc running system. **Procedia Engineering**, v. 15, p. 1234–1238, 2011.

MAHNKE, W.; LEITNER, S.-H.; DAMM, M. **OPC unified architecture**. Berlin: Springer, 2009.

MARCAL, L.; GUIMARAES, M.; RESENDE, A. Automatização de uma termoformadora visando melhorias no processo produtivo de uma empresa fabricante de peças termoplásticas para o setor automobilístico. In: XXXIII ENCONTRO NACIONAL DE ENGENHARIA DE PRODUCAO. **Anais [...]**. Salvador: ABEPRO, 2013. Disponível em: <http://www.abepro.org.br/biblioteca/enegep2013_TN_STO_177_013_22735.pdf>. Acesso em: 19 ago. 2021.

MURNANE, T.; REED, K. On the effectiveness of mutation analysis as a black box testing technique. In: 2001 AUSTRALIAN SOFTWARE ENGINEERING CONFERENCE. **Proceedings [...]**. Canberra: IEEE Computer Society, 2001. p. 12–20. Disponível em: <https://ieeexplore.ieee.org/document/948492>. Acesso em: 15 mai. 2021.

PERRY, D. E.; KAISER, G. E. Adequate testing and object-oriented programming. **J. Object Oriented Program.**, v. 2, n. 5, p. 13–19, jan. 1990.

PFROMMER, J. et al. Open source opc ua pubsub over tsn for realtime industrial communication. In: 23RD INTERNATIONAL CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION (ETFA), 2018, Turin. **Proceedings [...]**. IEEE, 2018. p. 1087–1090. Disponível em: <https://ieeexplore.ieee.org/document/8502479>. Acesso em: 15 set. 2021.

POGACEAN, C.; BROSCHEI, S.; SüSS, G. **Implementing Deterministic OPC UA Communication**. Munich, Germany, 2018. Disponível em: <https://ca.atevatech.com/wp-content/uploads/sites/3/2019/01/OPCUAPublisherSubscriber_W_EN_160407_100.pdf>. Acesso em: 17 abr. 2021.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. European. [S.l.]: McGraw-Hill, 1997. Adapted by Darrel Ince.

RIBEIRO, M. A. **Automação industrial**. 4. ed. Salvador: Tek Treinamento & Consultoria, 2001.

ROCHA, D. **Sistemas de supervisão e controlo de autómatos: soluções baseadas em OPC E IEC 60870-5-104**. Dissertação (Mestrado) — Instituto Politécnico do Porto. Instituto Superior de Engenharia do Porto, 2013.

ROCKWELLAUTOMATION. **CompactLogix 5480 Controllers**. 2021. Disponível em: <https://www.rockwellautomation.com/en-us/products/hardware/allen-bradley/programmable-controllers/small-controllers/compactlogix-family/compactlogix-5480-controllers.html>. Acesso em: 15 set. 2021.

SILVA, E. **Introdução às linguagens de programação para CLP**. 1. ed. São Paulo: Blusher, 2016.

SOUZA, T. Testes de desempenho de software: teoria e prática. In: V ESCOLA REGIONAL DE SISTEMAS DE INFORMAÇÃO DO RIO DE JANEIRO, 2018, Rio de

Janeiro. **Anais [...]**. 2018. p. 139–171. Disponível em: <https://sol.sbc.org.br/livros/index.php/sbc/catalog/download/7/13/39-1?inline=1>. Acesso em: 15 set. 2021.

SUDHEERBABU, G. **A formal toolchain for model-based testing of PLC systems**. 2021. Master's degree Programme in Information Technology, Computer Engineering — Åbo Akademi University, Finland, 2021.

UAEXPERT. **unified-automation**. 2022. Disponível em: <https://www.unified-automation.com>. Acesso em: 24 out. 2022.

WEG. **Controlador Lógico Programável PLC300**. 2021. Disponível em: <https://www.weg.net/catalog/weg/BR/pt/Automaç~ao-e-Controle-Industrial/Controle-de-Processos/Controladores-Lógicos-Programáveis/Controlador-Lógico-Programável-PLC300/Controlador-Lógico-Programável-PLC300/p/MKT_WDC_BRAZIL_PROGRAMMABLE_LOGIC_CONTROLLER_PLC_PLC300>. Acesso em: 15 set. 2021.

ZOUBEK, B.; ROUSSEL, J.-M.; KWIATKOWSKA, M. Towards automatic verification of ladder logic programs. In: IMACS-IEEE CESA 3 COMPUTATIONAL ENGINEERING IN SYSTEMS APPLICATIONS, 2003, Lille, France. **Proceedings [...]**. HAL, 2003. Disponível em: <https://hal.archives-ouvertes.fr/hal-00354468/file/2003_cesa_BZ_JMR_MK.PDF>. Acesso em: 15 set. 2021.