

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

Carolina de Farias

**DESENVOLVIMENTO DE UMA JIGA DE TESTES AUTOMATIZADOS PARA
COMPONENTES ELETRÔNICOS DIGITAIS UTILIZANDO O *ROBOT FRAMEWORK***

FLORIANÓPOLIS, 2023.

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELETRÔNICA**

Carolina de Farias

**DESENVOLVIMENTO DE UMA JIGA DE TESTES AUTOMATIZADOS PARA
COMPONENTES ELETRÔNICOS DIGITAIS UTILIZANDO O *ROBOT FRAMEWORK***

Trabalho de Conclusão de Curso submetido
ao Instituto Federal de Educação, Ciência
e Tecnologia de Santa Catarina como parte
dos requisitos para obtenção do título de
Engenheiro em Eletrônica.

Orientador:
Prof. Renan Augusto Starke, Dr. Eng.

FLORIANÓPOLIS, 2023.

Ficha de identificação da obra elaborada pelo autor.

DE FARIAS, CAROLINA
**DESENVOLVIMENTO DE UMA JIGA DE TESTES AUTOMATIZADOS
PARA COMPONENTES ELETRÔNICOS DIGITAIS UTILIZANDO O ROBOT
FRAMEWORK / CAROLINA DE FARIAS; orientação de RENAN
AUGUSTO STARKE. - Florianópolis, SC, 2023.**
87 p.

**Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. Bacharelado
em Engenharia Eletrônica. Departamento Acadêmico
de Eletrônica.**
Inclui Referências.

**1. JIGA DE TESTES. 2. COMPONENTES ELETRÔNICOS DIGITAIS.
3. TESTES AUTOMATIZADOS. 4. ROBOT FRAMEWORK. 5.
TESTES FUNCIONAIS. I. AUGUSTO STARKE, RENAN. II. Instituto
Federal de Santa Catarina. III. DESENVOLVIMENTO
DE UMA JIGA DE TESTES AUTOMATIZADOS PARA COMPONENTES
ELETRÔNICOS DIGITAIS UTILIZANDO O ROBOT FRAMEWORK.**

DESENVOLVIMENTO DE UMA JIGA DE TESTES AUTOMATIZADOS PARA COMPONENTES ELETRÔNICOS DIGITAIS UTILIZANDO O *ROBOT FRAMEWORK*

CAROLINA DE FARIAS

Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Eletrônica e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Eletrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 30 de junho, 2023.

Banca Examinadora:



Documento assinado digitalmente
RENAN AUGUSTO STARKE
Data: 08/08/2023 16:06:51-0300
CPF: 006.091.499-89
Verifique as assinaturas em <https://v.ifsc.edu.br>

Prof. Renan Augusto Starke, Dr. Eng.



Documento assinado digitalmente
DANIEL LOHMANN
Data: 09/08/2023 13:12:30-0300
Verifique em <https://validar.iti.gov.br>

Daniel Lohmann, Msc. Eng.



Documento assinado digitalmente
REGINALDO STEINBACH
Data: 08/08/2023 16:17:20-0300
Verifique em <https://validar.iti.gov.br>

Reginaldo Steinbach, Msc.

AGRADECIMENTOS

Em primeiro lugar, agradeço a Deus e aos amigos espirituais que foram minha força durante todos esses anos e permitiram que esse momento fosse possível.

Aos meus pais, Mariselia e Paulo, ao meu irmão Christian e meus sobrinhos. Eles que sempre foram minha base e meu ponto de apoio, sempre me incentivaram mesmo nos momentos mais difíceis e foram responsáveis por tudo que sou. Meu muito obrigado por ter a chance de compartilhar a vida com pessoas tão incríveis como vocês!

Gostaria de agradecer também a toda a minha família, por todo apoio e ajuda, que muito contribuíram em minha vida e para a realização deste trabalho. Em especial, gostaria de agradecer a minhas madrinhas Silvia e Débora, que sonharam e tornaram possível esse sonho junto comigo, muito obrigada por tudo que fizeram por mim, não existirá nada no mundo capaz de agradecer por tudo que vocês fizeram. Também gostaria de deixar um agradecimento especial aos meus padrinhos Marlene e Dorval, que sempre me apoiaram desde o início e me ajudaram em minha mudança para Florianópolis, obrigada por toda ajuda nos momentos de dificuldade.

Ao meu marido, Daniel, que acima de tudo é meu melhor amigo, sempre estive ao meu lado me ajudando a enfrentar as dificuldades e nunca desistir dos meus sonhos. Obrigada por tudo!

Meu orientador Renan, pelo ensino, paciência, apoio e ajuda durante o desenvolvimento desse trabalho e em minha jornada acadêmica.

Agradeço aos membros da banca examinadora por terem aceitado o convite.

Ao IFSC e a todos os professores do departamento de eletrônica, por terem compartilhado tantos conhecimentos.

Agradeço também a todas as amigas que fiz no IFSC, pelo companheirismo, apoio e risadas. Em especial, a meus amigos Maitê e Rayan, que compartilharam comigo todas as realizações e dificuldades durante minha jornada acadêmica.

Por fim, agradeço a todos que, de alguma forma, contribuíram para meu crescimento. Meu muito obrigada a todos vocês!

RESUMO

Este trabalho apresenta o desenvolvimento e a implementação de uma jiga de testes automatizados para componentes eletrônicos digitais utilizando o *framework* de testes *Robot Framework*. Desse modo, a interface de hardware empregada para a alimentação dos componentes em teste, geração e aquisição de sinais consistiu no dispositivo DAQ *USB-6212* da fabricante *National Instruments*. No contexto desse estudo, as portas lógicas foram os componentes eletrônicos digitais empregados para validar a eficiência da jiga de testes. Com o intuito de verificar o funcionamento correto dos componentes, foi adotado a metodologia de testes funcionais nos componentes em teste. Para armazenar os esquemáticos de ligação dos componentes, do DAQ, os dados de entrada dos testes e os resultados esperados, optou-se por utilizar o banco de dados em nuvem *MongoDB*. Essa abordagem possibilitou a geração automática dos esquemáticos de ligação entre o DAQ e os componentes em teste e também viabilizou que os testes funcionais fossem empregados para validar os componentes. Nesse sentido, de modo a facilitar a descrição clara e concisa dos casos de testes em que os componentes estão submetidos, foi empregado a utilização da sintaxe *Gherkin* no *Robot Framework*. Assim, a rastreabilidade dos componentes com defeitos e a análise de outras métricas de teste, como o tempo total de execução, podem ser realizadas por meio do relatório de testes gerado ao final da execução dos testes em cada componente.

Palavras-chave: Jiga de Testes. Componentes Eletrônicos Digitais. Testes automatizados. *Robot Framework*. Testes Funcionais.

ABSTRACT

This document presents the development and implementation of an automated test jig for digital electronic components using the *framework* for automated testing *Robot Framework*. The hardware interface used to power the components, and generate and acquire signals consisted of the DAQ device *USB-6212* from the manufacturer *National Instruments*. In this document, the logic gates were the digital electronic components used to validate the test jig efficiency. In order to verify the correct functioning of the components, the methodology of functional tests was adopted in the components under test. To store component diagrams, the diagram from DAQ, test input data, and expected results were used in the *MongoDB* cloud database. This approach enabled the automatic generation of connection schematics between the DAQ and the components under test and also enabled the functional tests to be used to validate the components. In order to facilitate the clear and concise description of the test cases in which the components under test are submitted, the use of the *Gherkin* syntax was used in the *Robot Framework*. The traceability of components with defects and the analysis of other test metrics, such as the total execution time, can be performed through the test report generated at the end of the test execution in each component.

Keywords: Test Jig. Digital Electronic Components. Automated Tests. Robot Framework. Functional Tests.

LISTA DE FIGURAS

Figura 1 – Elementos que compõem um dispositivo de testes automatizados .	16
Figura 2 – Dispositivo DAQ <i>USB-1608G Series</i> da fabricante <i>Measurement Computing</i>	18
Figura 3 – Dispositivo DAQ <i>USB-6212</i> da fabricante <i>National Instruments</i> . . .	19
Figura 4 – Esquemático de pinos do DAQ <i>USB-6212</i>	21
Figura 5 – Dinâmica de um teste funcional(teste caixa preta)	22
Figura 6 – Estrutura de projeto simples proposto pelo <i>Robot framework</i>	25
Figura 7 – Código exemplo desenvolvido utilizando o <i>Robot framework</i>	26
Figura 8 – Relatório de <i>Log</i> gerado por padrão pelo <i>Robot Framework</i>	28
Figura 9 – Exemplo de cenário de teste descrito utilizando o <i>Gherkin</i>	30
Figura 10 – Formato de um dado no <i>MongoDB</i>	31
Figura 11 – Casos de uso da jiga de testes	37
Figura 12 – Estrutura de projetos da jiga de testes	38
Figura 13 – Fluxograma contendo as ações da jiga de testes	41
Figura 14 – Caso de teste para testar um componente eletrônico digital	42
Figura 15 – Indicação da pré-condição de teste para testar um componente . .	44
Figura 16 – Ações executadas pela palavra chave que retorna o esquemático de ligação entre o DAQ e o componente	45
Figura 17 – Função desenvolvida em <i>Python</i> que retorna a divisão de pinos do componente a ser testado	45
Figura 18 – <i>Top View</i> do componente 74HC08	46
Figura 19 – Parte da função <i>Python</i> que retorna a correspondência entre o pino do DAQ e a conexão com o pino do componente em teste	47
Figura 20 – <i>Json</i> contendo o esquemático de ligação entre o DAQ e o componente 74HC08	48
Figura 21 – Ligação entre o DAQ e 74HC08 conforme o esquemático de ligação sugerido pela jiga de testes	50
Figura 22 – Verificando se o DAQ está conectado a porta <i>USB</i>	50
Figura 23 – Correspondência da palavra chave que verifica se o DAQ está conectado no diretório de <i>Resources</i>	51
Figura 24 – Função <i>Python</i> que verifica se o DAQ está conectado	51
Figura 25 – Iniciando o teste de um componente	52
Figura 26 – Correspondência na seção de <i>Keywords</i> do diretório de <i>Resources</i> para iniciar o teste de um componente	52
Figura 27 – Função <i>Python</i> que retorna os resultados esperados para um componente	53

Figura 28 – Código responsável por definir os pinos do DAQ que vão realizar a escrita e leitura nos pinos do componente	57
Figura 29 – Código responsável por criar uma correspondência entre os <i>lines</i> de escrita e leitura	58
Figura 30 – Código responsável por realizar a leitura e a escrita nos pinos do DAQ utilizando os recursos da biblioteca <i>nidaqmx</i>	59
Figura 31 – Comparando os resultados de testes	60
Figura 32 – Correspondência da <i>keyword</i> de comparação de resultados no arquivo do diretório de <i>Resources</i>	61
Figura 33 – <i>Top View</i> do componente 74HC32	63
Figura 34 – Cenário de teste para cadastrar o esquemático de ligação de um novo componente	64
Figura 35 – Cenário de teste para cadastrar os resultados esperados de um componente	68
Figura 36 – <i>Script</i> desenvolvido para validar a geração e aquisição de sinais da jiga de testes	70
Figura 37 – Sinais de entrada e saída do componente 74HC08 em pleno funcionamento	71
Figura 38 – Simulando a ocorrência da desconexão do sinal de uma das entradas do 74HC08	71
Figura 39 – Simulando a ocorrência de nível lógico baixo em um dos terminal de entrada do 74HC08	72
Figura 40 – Relatório de <i>Log</i> gerado por padrão no teste da porta lógica <i>AND</i> .	73
Figura 41 – <i>Dashboard</i> do relatório de testes para o componente 74HC08 em pleno funcionamento	75
Figura 42 – Visão mais detalhada da aba de <i>Test</i> presente no relatório customizado do componente 74HC08	76
Figura 43 – Visão mais detalhada da aba de <i>Keyword Statistics</i> presente no relatório customizado do componente 74HC08	77
Figura 44 – Formas de apresentar o resultado para o 74HC08 em falha	78
Figura 45 – Gráfico interativo que permite analisar os testes em falha	79

LISTA DE QUADROS

Quadro 1 – Especificações gerais do DAQ <i>USB-1608G Series</i>	18
Quadro 2 – Especificações gerais do DAQ <i>USB-6212</i>	19
Quadro 3 – Comparações técnicas entre os dispositivos <i>USB-6212</i> e <i>USB-1608G Series</i>	20
Quadro 4 – Objetivos de cada componente na estrutura de projeto simples sugerida pelo <i>Robot Framework</i>	25
Quadro 5 – Objetivos de cada seção presentes em arquivos do <i>Robot Framework</i>	27
Quadro 6 – Descrição resumida das funções das tecnologias utilizadas no projeto da jiga de testes	33

LISTA DE ABREVIATURAS E SIGLAS

ATDD	<i>Acceptance Test-Driven Development</i> (Desenvolvimento Orientado a Testes de Aceitação)
ATE	<i>Automatic Test Equipment</i> (Equipamento de Teste Automático)
AWS	<i>Amazon Web Services</i> (Plataforma de serviços em nuvem da Amazon)
BDD	<i>Behavior Driven Development</i> (Desenvolvimento Orientado ao Comportamento)
CI/CD	<i>Continuous Integration and Continuous Delivery</i> (Integração Contínua e Entrega contínua)
DAQ	<i>Data Acquisition</i> (Aquisição de Dados)
DUT	<i>Device Under Tester</i> (Dispositivo em Teste)
IFSC	Instituto Federal de Santa Catarina
NoSQL	<i>Not Only SQL</i> (Não apenas SQL)
RPA	<i>Robotic Process Automation</i> (Automação Robótica de Processos)
SQL	<i>Structured Query Language</i> (Linguagem de Consulta Estruturada)
TI	Tecnologia da Informação
USB	<i>Universal Serial Bus</i> (Porta Serial Universal)

SUMÁRIO

1	INTRODUÇÃO	12
1.1	Justificativa	12
1.2	Definição do Problema	13
1.3	Objetivo Geral	14
1.4	Objetivos Específicos	14
2	FUNDAMENTAÇÃO TEÓRICA	15
2.1	Jiga de testes	15
2.2	Sistemas de Aquisição de Dados (DAQ)	16
2.3	Dispositivos DAQ	17
2.3.1	<i>Measurement Computing - USB-1608G Series</i>	17
2.3.2	<i>NATIONAL INSTRUMENTS - USB-6212</i>	18
2.4	Testes automatizados por <i>software</i>	22
2.5	<i>Frameworks</i> de testes automatizados	23
2.5.1	<i>Robot framework</i>	24
2.6	<i>Gherkin</i>	29
2.7	Banco de dados <i>MongoDB</i>	31
2.8	Resumo dos dispositivos e tecnologias adotadas para desenvolver a jiga de testes	32
3	DESENVOLVIMENTO	34
3.1	Metodologia	34
3.2	Configurações necessárias para a instalação da jiga de testes automatizados	36
3.3	Casos de uso em que a jiga de testes automatizados está submetida	37
3.4	Estrutura do projeto da jiga de testes automatizados	38
3.5	Implementação do caso de uso para teste de um componente eletrônico digital	39
3.5.1	Testando a porta lógica AND 74HC08	40
3.5.1.1	<i>Diretório de Tests</i>	41
3.5.1.2	<i>Obtendo o esquemático de ligação entre o componente e o DAQ</i>	43
3.5.1.3	<i>Garantindo que o DAQ está conectado a porta USB</i>	50
3.5.1.4	<i>Executando as ações de realizar o teste em um componente</i>	52
3.5.1.5	<i>Comparando se os resultados obtidos foram iguais aos resultados esperados</i>	60
3.6	Implementação do casos de uso que envolvem cadastros de componentes e resultados esperados	61
3.6.1	Cadastrando o esquemático de ligação da porta lógica OR 74HC32	62
3.6.2	Cadastrando os resultados esperados para a porta lógica 74HC32	65
4	RESULTADOS	69
4.1	Validação do DAQ e simulação de problemas	69
4.2	Relatórios dos testes	72
5	CONSIDERAÇÕES FINAIS	81
	REFERÊNCIAS	84

1 INTRODUÇÃO

Após a fase final de fabricação de um componente eletrônico, é necessário realizar testes funcionais. Entretanto, devido à alta competitividade do mercado atual e aos custos envolvidos, nem sempre é possível testar todos os parâmetros de um componente(LANGER; HELLMUTH,2010). Por isso, torna-se evidente a necessidade de encontrar soluções capazes de otimizar o processo de testes de componentes eletrônicos.

Uma jiga de testes é composta por apenas um equipamento, que é responsável por validar o funcionamento de produtos através de testes automatizados, dessa forma, as medições são feitas de forma automática, o que diminui a necessidade da mão de obra humana e garante a realização de testes com alta qualidade e bom custo-benefício(CERTI,2023)

De acordo com (ISTQB - International Software Testing Qualifications Board,2016) alguns dos objetivos da automação de testes são o ganho de eficiência, aumento da cobertura de testes e a geração de documentação.

Nesse sentido, um equipamento capaz de realizar testes de forma automatizada pode aprimorar processos, pois é possível obter testes confiáveis de forma mais rápida do que utilizando testes manuais, que são mais demorados e suscetível a erros, além de cobrir um número maior de parâmetros de testes e obter documentação. Assim, a otimização do processo de testes pode auxiliar as companhias a melhorarem seus processos, resultando em um crescimento mais assertivo e sustentável.

A proposta desse trabalho é projetar e desenvolver uma jiga de testes automatizados para componentes eletrônicos utilizando um dispositivo capaz de fazer a geração e aquisição de sinais. Para validar o conceito, foram utilizado as portas lógicas, com o objetivo de realizar testes funcionais desse componente, por meio de um *framework* de testes automatizados, o *Robot framework*(Robot Framework Foundation,2016).

O desenvolvimento dos casos de testes foram feitos por meio da linguagem *Gherkin*, que, conforme SMART(2014), é uma ferramenta do *Behavior-Driven Development*(BDD), que visa facilitar o entendimento das partes interessadas. Assim, por meio do conceito para a descrição de cenários propostos pelo *Gherkin*, é possível produzir testes de fácil compreensão e gerar documentação do processo.

1.1 Justificativa

Considerando o atual cenário competitivo entre empresas, em que existe uma grande busca para realizar entregas mais ágeis, é essencial encontrar soluções que possam reduzir os custos, sendo que uma das etapas do processo de produção

de um produto eletrônico é a realização de testes.

Nesse sentido, desenvolver um dispositivo automatizado capaz de validar o funcionamento de componentes sem a necessidade da mão de obra humana pode reduzir custos de produção e acelerar o seu lançamento, essa abordagem também permite executar cenários de testes repetidamente e diversas vezes, de forma rápida e eficiente, o que contribui para garantir a qualidade final do componente.

Além disso, a criação de uma jiga de testes automatizados também é benéfica no contexto acadêmico do curso de Engenharia Eletrônica do Instituto Federal de Santa Catarina (IFSC), visto que existem muitos componentes disponíveis no almoxarifado da instituição, o que torna difícil a rastreabilidade de componentes queimados ou com mau funcionamento, sendo que atualmente o processo de detecção de erros é dependente da mão de obra humana, que está suscetível a erros, além de demandar muito tempo para a execução.

De acordo com (SMART,2014) a essência de se utilizar o BDD no desenvolvimento de testes é fornecer uma linguagem de fácil entendimento em inglês ou no idioma nativo de quem está construindo os testes, de forma a gerar valor para o *software* e facilitar a comunicação entre equipes, além de servir como documentação para o código. Assim, criar casos de testes utilizando o BDD permite que o usuário final compreenda facilmente o cenário de teste que está sendo executado, tornando o processo menos suscetível a erros.

Diante disso, esse trabalho tem como objetivo implementar e desenvolver uma jiga de testes utilizando testes automatizados por *software*, para que o usuário possa rastrear problemas funcionais em componentes eletrônicos de forma rápida, garantindo a qualidade do que foi testado.

1.2 Definição do Problema

De acordo com RODRIGUES(2009) o desenvolvimento de soluções que ampliem a eficácia de produção tornou-se ponto focal para o ramo industrial, sendo que a automação de processos surge como elemento central dessa crescente demanda por eficiência.

Diante da necessidade de automatizar processos, e sendo os testes uma das etapas do processo produtivo, esse trabalho visa responder, como implementar uma jiga de testes utilizando testes automatizados por *software* para realizar testes funcionais de componentes eletrônicos?

1.3 Objetivo Geral

Define-se que o objetivo deste trabalho é o desenvolvimento de uma jiga de testes utilizando testes automatizados por *software* para a realização de testes funcionais de componentes eletrônicos digitais.

1.4 Objetivos Específicos

Diante do objetivo geral pretendido, destaca-se os seguintes objetivos específicos:

- a) projetar uma jiga de testes automatizados, com o uso de um dispositivo para geração e aquisição de sinais;
- b) implementar a jiga de Testes utilizando testes automatizados por *software* através de um *framework* de testes automatizados;
- c) desenvolver os casos de testes utilizando a linguagem *Gherkin*, de forma que o cenário que está sendo executado é de fácil entendimento.
- d) validar o funcionamento da jiga de testes através dos testes funcionais de portas lógicas;
- e) gerar relatórios dos cenários de testes executados, possibilitando a análise dos resultados obtidos.

2 FUNDAMENTAÇÃO TEÓRICA

Esse capítulo tem como objetivo apresentar os principais conceitos envolvidos na criação da jiga de testes automatizados deste trabalho e apresentar as principais ferramentas utilizadas.

2.1 Jiga de testes

De acordo com (CERTI, 2023), as jigas de testes surgem como uma solução para auxiliar os processos industriais na entrega de produtos eletrônicos confiáveis, permitindo a realização efetiva da validação do funcionamento do produto a qual elas se propõe a testar.

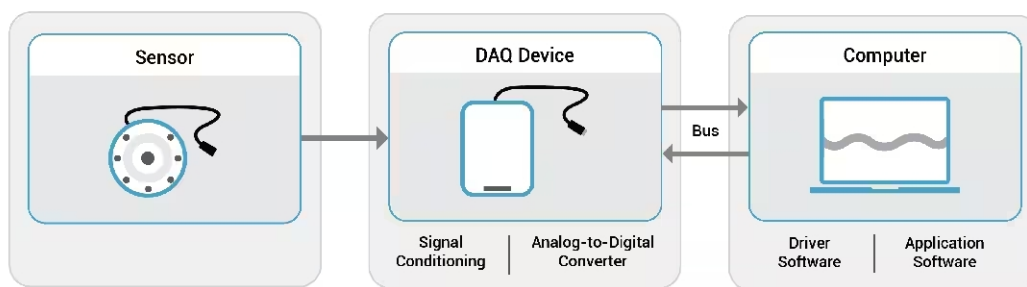
Conforme aponta (Mordor Intelligence, 2023), a demanda do mercado por dispositivos de testes automatizados tem sido impulsionada pelo aumento da indústria de eletroeletrônicos. Esse tipo de equipamento é capaz de encontrar defeitos em componentes e sistemas eletrônicos após a fabricação. Assim, nessa crescente demanda, destaca-se algumas indústrias que já aderem a esse tipo de dispositivo para realizar testes, tais como a indústria de aviação, automotiva e de tecnologia da informação (TI).

Segundo (Trenton Systems, 2021), um dispositivo de teste automatizado, é um equipamento computadorizado que realiza e avalia os resultados de testes dos mais variados tipos, como por exemplo de funcionalidade, desempenho e estresse em dispositivos e sistemas eletrônicos.

De acordo com (Trenton Systems, 2021), o dispositivo de testes automatizados, também conhecido como ATE (*Automatic Test Equipment*) é composto por um sistema de aquisição de dados (DAQ) que executa o *software* de testes. Esse *software* é responsável por controlar e gerenciar os instrumentos e as fontes de sinal do dispositivo em teste, também conhecido como DUT (*Device Under Test*).

Conforme ilustrado na Figura 1, o ATE é composto por três elementos principais: DUT, DAQ e computador. O dispositivo em teste é chamado de DUT, exemplificado pelo “Sensor”. O DAQ é responsável pela geração e aquisição de sinais, bem como realizar conversões digitais e analógicas, se necessário. E, por fim, o computador, que é onde estão os *drivers* essenciais para a comunicação com o DAQ e o *software* de testes.

Assim, essa estrutura constitui um dispositivo de testes automatizados que possibilita a execução eficiente dos testes e a análise dos resultados obtidos.

Figura 1 – Elementos que compõem um dispositivo de testes automatizados

Fonte: Omega Engineering(2023).

Alguns dos benefícios de utilizar-se um dispositivo capaz de realizar testes automatizados incluem a prevenção de erros, redução do tempo e o ganho de eficiência durante os testes, o que resulta em uma maior qualidade de testes.

2.2 Sistemas de Aquisição de Dados (DAQ)

De acordo com (ABDALLAH; ELKEELANY, 2009) “Os sistemas de aquisição de dados (DAQ) são dispositivos usados para adquirir dados do ambiente que o cerca. A aquisição de dados ajuda a obter certas informações sobre o ambiente considerado” (p.240).

Conforme (NATIONAL INSTRUMENTS CORP, 2023a), o equipamento DAQ de uso geral é um dispositivo projetado para realizar a aquisição ou a geração de sinais considerando possíveis restrições de tempo real. Assim, o *software* instalado no computador é o responsável por gerenciar as medições. O DAQ desempenha o papel de converter o sinal de entrada em um formato digital, para que o computador seja capaz de interpretar esse sinal(NATIONAL INSTRUMENTS CORP, 2023a).

Conforme aponta (NATIONAL INSTRUMENTS CORP, 2023b), um dispositivo DAQ desempenha várias funções, sendo composto por:

- Canais de entrada e saída analógicas capazes de realizar a leitura e geração de sinais analógicos.
- Canais de entrada e saída digitais capazes de realizar a leitura e geração de sinais digitais.
- Contadores e temporizadores, que permitem medir eventos digitais, como pulsos ou transições de sinais e controlar eventos com base no tempo, como gerar um sinal em uma determinada frequência ou duração.

Nesse sentido, existem dispositivos DAQ que executam apenas uma das funções citadas ou que suportam todas, como os DAQs multifuncionais (NATIONAL INSTRUMENTS CORP, 2023b).

O *software* instalado no computador é o responsável por realizar as medidas, controle, aquisição e gerenciamento de sinais. As opções de *software* variam conforme a compatibilidade suportada pelo DAQ escolhido, sendo alguns exemplos:

- a) LabVIEW (NATIONAL INSTRUMENTS, 2023): É um ambiente de programação com interface gráfica desenvolvido pela *National Instruments*. Sua abordagem é baseada em blocos que representam as interações entre os componentes do sistema e o *hardware* DAQ.
- b) MATLAB (MATHWORKS, 2023): É uma plataforma de computação numérica que fornece ferramentas de suporte para a aquisição de dados podendo ser integrado com dispositivos DAQ.
- c) DASyLab (Measurement Computing Corporation, 2023): É um *software* com interface gráfica desenvolvido pela *Measurement Computing* baseado em fluxo de dados para configurar e controlar sistemas DAQ.
- d) Python (PIERRE, 2010): É uma linguagem de programação que possui bibliotecas disponíveis para realizar a aquisição de dados, como a PyDAQmx que permite a comunicação com dispositivos DAQ.

2.3 Dispositivos DAQ

Existem diversos dispositivos DAQ que realizam a aquisição de dados para realizar testes e medições. Considerando as especificações técnicas de um DAQ, é possível garantir confiabilidade e a eficiência na aquisição e análise dos dados.

Assim, a escolha por um DAQ depende de múltiplos fatores, sendo que alguns dos parâmetros mais importantes são o tipo de sinal a ser analisado, a taxa de amostragem, resolução, o erro aceitável na medição e as restrições da aplicação (NATIONAL INSTRUMENTS CORP, 2023b).

2.3.1 Measurement Computing - USB-1608G Series

O DAQ *USB-1608G Series* desenvolvido pela *Measurement Computing* é alimentado via *USB*, sendo um DAQ multifuncional de alta velocidade e baixo custo. Os seus usos incluem aplicações de pesquisas científicas e de engenharia.

Conforme o *datasheet* desse dispositivo, ele possui entradas e saídas analógicas, com a resolução de 16 *bits* taxa de amostragem de até 500kS/s. Além disso, também inclui canais digitais, sendo 8 canais dedicados para monitorar e controlar dispositivos digitais externos. A Figura 2 ilustra o dispositivo *USB-1608G Series*.

Alguns dos *softwares* que podem ser utilizados para controlar o dispositivo são o *DASyLab*, *LabVIEW* e *MATLAB*.

Figura 2 – Dispositivo DAQ *USB-1608G Series* da fabricante *Measurement Computing*



Fonte: Measurement Computing Corporation(2018).

Desse modo, algumas das especificações técnicas do dispositivo estão descritas no Quadro 1:

Quadro 1 – Especificações gerais do DAQ *USB-1608G Series*

Número de canais de entradas analógicas	8 canais diferenciais ou 16 canais <i>single ended</i>
Resolução do conversor analógico digital	16 <i>bits</i>
Taxa de amostragem para entradas analógicas	500 kS/s
Saídas Analógicas	2 canais
Taxa de amostragem para saídas analógicas	500 kS/s
Entradas e saídas digitais	8 canais
Número de contadores/ <i>timers</i>	2
Resolução dos contadores/ <i>timers</i>	32 <i>bits</i>

Fonte: Adaptado de Measurement Computing Corporation(2018).

2.3.2 NATIONAL INSTRUMENTS - USB-6212

O modelo *USB-6212* é um DAQ multifuncional da NI que possui entradas e saídas digitais, entradas e saídas analógicas e dois contadores de 32 *bits*, sendo alimentado através da interface *USB* do computador (NATIONAL INSTRUMENTS CORPORATION, 2023a). A Figura 3 ilustra o hardware do dispositivo.

Figura 3 – Dispositivo DAQ *USB-6212* da fabricante *National Instruments*

Fonte: NATIONAL INSTRUMENTS CORPORATION(2023a).

A Figura 3 ilustra o DAQ desenvolvido pela NI. Ele suporta uma variedade de *softwares* para realizar o seu controle, como o *LabView*, *Python*, *Matlab*. O *LabView* pode ser utilizado desde que o seu ambiente esteja devidamente instalado no computador. Para as linguagens *Python* e *Matlab*, é necessário utilizar as bibliotecas suportadas pelo DAQ. Além disso, os *drivers* para a comunicação entre o DAQ e o computador devem estar corretamente instalados no computador. Os *drivers* necessários podem ser obtidos na documentação oficial da NI (CORP, 2023).

Conforme a fabricante NATIONAL INSTRUMENTS CORPORATION, algumas das especificações do dispositivo são:

Quadro 2 – Especificações gerais do DAQ *USB-6212*

Número de canais de entradas analógicas	8 canais diferenciais ou 16 canais <i>single ended</i>
Resolução do conversor analógico digital	16 <i>bits</i>
Taxa de amostragem para entradas analógicas	400 kS/s
Saídas Analógicas	2 canais
Taxa de amostragem para saídas analógicas	250 kS/s
Entradas e saídas digitais	32 canais
Número de contadores/ <i>timers</i>	2
Resolução dos contadores/ <i>timers</i>	32 <i>bits</i>

Fonte: Adaptado de NATIONAL INSTRUMENTS CORPORATION(2023b).

Os pinos digitais são multifuncionais comportando-se como entrada ou saída. Para a jiga de testes desse trabalho, foi utilizado apenas as entradas e saídas digitais, pois o objeto de estudo foram apenas os componentes digitais.

Além disso, o *USB-6212* conta com terminais para alimentação de +5V e terminais GND. Assim, os componentes digitais utilizados para realizar os testes foram alimentados com a tensão de +5V fornecida pelos pinos de alimentação do DAQ.

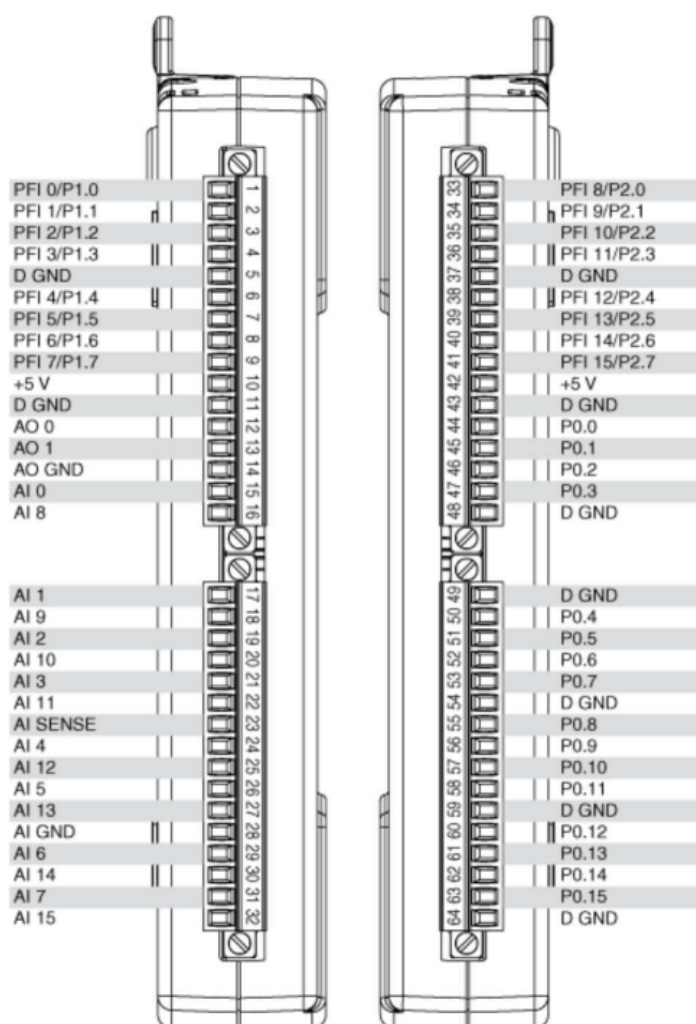
Nesse sentido, o Quadro 3 apresenta a comparação entre os dispositivos *USB-6212* e *USB-1608G Series*:

Quadro 3 – Comparações técnicas entre os dispositivos *USB-6212* e *USB-1608G Series*

Especificação	<i>USB-6212</i>	<i>USB-1608G Series</i>
Número de canais de entradas analógicas	8 canais diferenciais ou 16 canais <i>single ended</i>	8 canais diferenciais ou 16 canais <i>single ended</i>
Resolução do conversor analógico digital	16 <i>bits</i>	16 <i>bits</i>
Taxa de amostragem para entradas analógicas	400 kS/s	500 kS/s
Saídas Analógicas	2 canais	2 canais
Taxa de amostragem para saídas analógicas	250 kS/s	500 kS/s
Entradas e saídas digitais	32 canais	8 canais
Número de contadores/ <i>timers</i>	2	2
Resolução dos contadores/ <i>timers</i>	32 <i>bits</i>	32 <i>bits</i>

Fonte: Elaboração Própria (2023).

Nesse sentido, como o DAQ escolhido foi o *USB-1608G Series*, a Figura 4 representa o esquemático de ligação desse dispositivo.

Figura 4 – Esquemático de pinos do DAQ *USB-6212*

Fonte: NATIONAL INSTRUMENTS CORPORATION(2009).

Desse modo, os pinos digitais do DAQ podem ser configurados individualmente, de modo que um mesmo pino pode ser configurado tanto como entrada ou saída digital. Além disso, os pinos digitais podem ter outras funções além de entradas e saídas digitais, de modo que podem servir como *triggers* externos, contadores e sinais de temporização.

Assim, o dispositivo *USB-6212* pode ser controlado utilizando o *software* instalado no computador desde que os *drivers* necessários estejam devidamente instalados. Na jiga de testes desenvolvida por esse trabalho, a biblioteca *nidaqmx* realiza a função de escrita e leitura dos pinos digitais do DAQ. Entretanto, todas as outras funcionalidades que o DAQ *USB-6212* possui, como a escrita e leitura de pinos analógicos, também podem ser realizados utilizando a biblioteca *nidaqmx*.

2.4 Testes automatizados por *software*

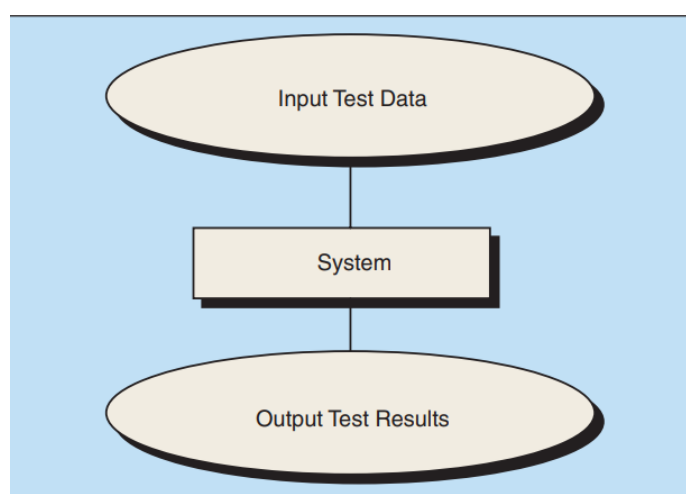
De acordo com BERNARDO; KON(2008) a definição de testes automatizados de software é:

Testes automatizados são programas ou *scripts* simples que exercitam funcionalidades do sistema sendo testado e fazem verificações automáticas nos efeitos colaterais obtidos. A grande vantagem desta abordagem, é que todos os casos de teste podem ser facilmente e rapidamente repetidos a qualquer momento e com pouco esforço. (BERNARDO; KON, 2008, p. 2).

Os testes automatizados tem como um dos objetivos confirmar o funcionamento de um projeto conforme aponta (ISTQB - International Software Testing Qualifications Board, 2014). Além disso, a abordagem de testes utilizando testes automatizados ajuda a reduzir falhas, pois as execuções podem ser repetidas e os erros podem ser encontrados antecipadamente.

Testes funcionais, também conhecidos como testes caixa preta, são uma técnica de teste que foca em testar a finalidade de um sistema utilizando seus requisitos funcionais. O objetivo dessa técnica é realizar os testes com base nos requisitos do sistema, abstraindo o entendimento do código, pois não é necessário conhecer sua implementação(FREEMAN, 2002). A Figura 5 representa a dinâmica de um teste funcional (teste caixa preta). Assim, é aplicado no sistema um conjunto de dados de entrada conhecidos pelo testador ou pelo teste automatizado e em seguida, os dados de saída resultantes da execução dos testes serão analisados e comparados com os resultados esperados pelo conjunto de dados aplicado.

Figura 5 – Dinâmica de um teste funcional(teste caixa preta)



Fonte: FREEMAN(2002).

Os testes funcionais permitem avaliar o comportamento e o desempenho de um sistema. Eles são projetados para identificar discrepâncias entre os resultados de

saída obtidos da execução dos testes e os resultados esperados, que também devem ser conhecidos pelo testador ou pelo *software*.

Segundo ISTQB - International Software Testing Qualifications Board(2016), a automação de testes utiliza as ferramentas de *software* para controlar e configurar as pré condições de teste para executar o caso de teste e realizar a comparação dos dados de saída dos testes com os resultados esperados. Assim, ao analisar os resultados de saída com os esperados, é possível detectar falhas, erros de processamento ou comportamentos inesperados.

A automação de testes também possui papel importante em *pipelines* de Integração Contínua e Entrega Contínuas(CI/CD), que conforme SETH; KHARE (2015), automatizar o processo de criação de testes é a melhor maneira de tornar a *Continuous Intregation* mais eficiente.

Entretanto, a automação de testes também possui certas desvantagens e limitações. Por exemplo, a automação só consegue verificar casos de testes conhecidos pelo computador e, além disso, o investimento inicial para configurar as condições para teste pode ser alto (ISTQB - International Software Testing Qualifications Board, 2016).

Conforme aponta (ISTQB - International Software Testing Qualifications Board, 2016) a automação de testes demanda de uma equipe especializada e erros adicionais não previstos podem ser inseridos ao automatizar testes, como falhas de configuração, falhas em dependências externas, como banco de dados, instabilidade de rede, entre outros erros.

2.5 Frameworks de testes automatizados

Um *framework* pode ser definido como um conjunto de classes que oferecem soluções para determinados problemas, de modo que suas funções podem ser utilizadas sem que o usuário conheça as implementações dos componentes das classes que compõe o *framework*(JOHNSON; FOOTE, 1988).

Conforme ISTQB - International Software Testing Qualifications Board (2016), para que um *framework* de testes seja fácil e sustentável de se utilizar, ele deve seguir algumas recomendações:

- a) Fornecer um relatório de testes com informações de qualidade sobre o *status* do teste, tais como, aprovado, reprovado, etc.
- b) Encontrar falhas nos testes e no ambiente de testes.
- c) Documentar os casos de testes, afim de oferecer informações para as partes interessadas, como testadores, gerente de projetos e desenvolvedores.

- d) Idealmente, possuir cenários de testes de fácil manutenção, assim como cenários que permitam a expansão.

Os *frameworks* para teste de *software* suportam as mais variadas técnicas de testes, tais como, testes unitários, de integração, funcionais, estresse, entre outros.

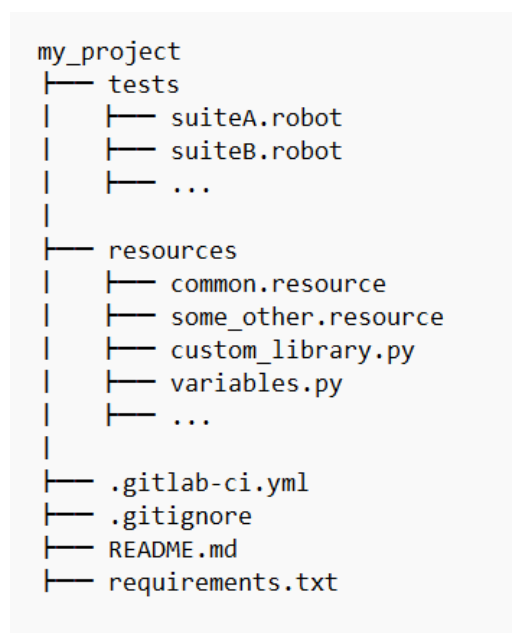
2.5.1 *Robot framework*

O *Robot framework* é um *framework open source* para a automação de testes em que a comunidade colabora para o seu desenvolvimento e aprimoramento. Segundo Robot Framework Foundation (2016), ele é desenvolvido em Python e segue uma abordagem de desenvolvimento orientada por palavras chaves.

Nesse sentido, o *Robot* pode ser utilizado em desenvolvimento orientado a testes de aceitação (ATDD), desenvolvimento orientado por comportamento (BDD) e automação RPA (Robot Framework Foundation, 2016). Objetiva-se um *framework* flexível com a necessidade de integração com diferentes tecnologias e interfaces (Robot Framework Foundation, 2016).

Desse modo, o *framework* fornece suporte para automação de testes dos mais diferentes domínios, como testes funcionais, unitários e de integração. Ele suporta a automação dos mais diversas aplicações, como *web*, *mobile*, *API* (Robot Framework Foundation, 2016).

Como o *Robot framework* foi desenvolvido para ser um *framework* de testes flexível, ele não possui uma estrutura fixa de projeto para organização de diretórios. Porém, o *framework* recomenda estrutura de projeto simples conforme apresentada abaixo (Robot Framework Guides, 2023).

Figura 6 – Estrutura de projeto simples proposto pelo *Robot framework*

Fonte: Robot Framework Guides(2023).

O Quadro 4 exemplifica o que é esperado em cada diretório conforme a documentação oficial do *Robot framework*.

Quadro 4 – Objetivos de cada componente na estrutura de projeto simples sugerida pelo *Robot Framework*

Diretório	Funcionalidade
<i>Tests</i>	Contém todos os casos de testes
<i>Resources</i>	Contém todas as palavras-chaves, variáveis e arquivos Python
<i>.gitlab-ci.yml</i>	Arquivo para integração CI/CD
<i>README.md</i>	Arquivo de configuração utilizado pelo Git
<i>requirements.txt</i>	Arquivo que define as bibliotecas necessárias para a execução do projeto

Fonte: Elaboração Própria (2023).

Os arquivos do *Robot framework* possuem a extensão “*.robot*” e seguem uma estrutura dividida por seções ilustrado pela Figura 7 e detalhado no Quadro 5. As seções são destacadas através do uso de “*****”. Não é necessário que o arquivo possua todas as seções podendo conter apenas as seções necessárias ao projeto.

Nesse sentido, o *framework* é uma estrutura de automação de testes que permite a criação de casos de teste. Assim, dentro de um arquivo “*.robot*” é possível definir um ou mais casos de teste, cada um com sua própria lógica e etapas. No *Robot framework* os casos de testes podem ser executados de duas maneiras diferentes. Os cenários de testes podem ser executados juntos de forma contínua ou executados individualmente. Um caso de teste pode ser executado isoladamente mesmo que

existam outros testes contidos dentro do mesmo arquivo “.robot”. Quando os casos de testes são executados juntos, sua execução é sequencial, o que significa que os testes são executados um após o outro, na ordem em que são definidos no arquivo de testes. Executar casos de testes juntos implica em uma execução contínua. Nesse contexto, essa abordagem de execução de testes é útil quando se deseja uma visão geral do *status* de teste do sistema. Dessa forma, para executar casos de testes juntos é necessário apenas indicar o nome do arquivo que os testes estão contidos no terminal. O comando a seguir demonstra como realizar essa execução.

```
robot nome_do_arquivo.robot
```

Para executar um caso de teste isoladamente, basta indicar o nome do cenário no comando via terminal. Essa abordagem é útil quando se pretende testar ou depurar um cenário de teste específico, mesmo que existam outros casos de teste presentes no mesmo arquivo “.robot”. Assim, o comando a seguir executa apenas o teste “Caso de Teste1” que está presente na Figura 7.

```
robot --test Caso de Teste1.robot
```

Figura 7 – Código exemplo desenvolvido utilizando o *Robot framework*

```
1 *** Settings ***
2 Documentation      Exemplo usando o formato separado por espaço.
3 Library            OperatingSystem
4 Resource           resources/arquivo.py
5
6 *** Variables ***
7 ${MENSAGEM}        Ola, mundo!
8
9 *** Test Cases ***
10 Caso de Teste1
11     [Documentation]  Exemplo de Teste.
12     Log    ${MENSAGEM}
13     Minha Palavra Chave Personalizada    diretorio/ExemplodeDiretorio
14
15 Caso de Teste2
16     Should Be Equal    ${MENSAGEM}    Ola, mundo!
17     Funcao Python Que Retorna a Mensagem Seja Bem Vindo
18
19 *** Keywords ***
20 Minha Palavra Chave Personalizada
21     [Arguments]    ${diretorio}
22     Directory Should Exist    ${diretorio}
23
24
```

Fonte: Adaptado de Robot Framework Foundation (2016)

O *Robot* utiliza o separador de dois ou mais espaços para realizar a indentação entre as palavras chaves, argumentos e instruções. Como ele é baseado

em palavras-chaves, é possível criar um código em *Robot* utilizando as palavras chaves já existentes ou personalizando conforme necessário, bem como utilizar funções *Python* como palavras chaves. As palavras chaves são executadas de forma sequencial seguindo a ordem que elas estão definidas no arquivo de teste.

Quadro 5 – Objetivos de cada seção presentes em arquivos do *Robot Framework*

Seção	Funcionalidade
<i>Settings</i>	Define as configurações gerais do arquivo de teste, aqui são importadas as bibliotecas(<i>Library</i>), os arquivos do diretório <i>Resources</i> , as pré-condições de teste
<i>Variables</i>	Define as variáveis que serão utilizadas pelo arquivo do <i>Robot</i> , podendo conter valores estáticos ou dinâmicos
<i>Test Cases</i>	Seção que define os casos de teste, cada caso é definido através das palavras-chaves, que podem ou não conter argumentos, que são as variáveis de uma palavra-chave
<i>Keywords</i>	Essa seção pode ser utilizada para definir palavras-chaves nativas ou personalizadas, definir as ações que uma palavra-chave irá executar ou realizar verificações específicas
<i>Comments</i>	Seção opcional que permite adicionar comentários

Fonte: Elaboração Própria (2023).

Desse modo, a primeira seção da Figura 7 é a *Settings*, que contém a *tag Documentation* para documentação do caso de teste e a biblioteca, indicada pela palavra *Library*, que é a *OperatingSystem*, uma biblioteca nativa da tecnologia. Além disso, também possui a indicação para o arquivo Python, “arquivo.py”, que está contido no diretório de *Resources*.

A seção *Variables* contém a *string* mensagem, que possui o valor “Ola, mundo!”. Para a seção *Test Cases* temos os cenários de testes, representados pelo “Caso de Teste1” e “Caso de Teste2”.

O “Caso de Teste1” tem como primeira ação mostrar no console qual é o valor da variável mensagem através da instrução *Log* nativa do *framework*.

A palavra chave “Minha Palavra Chave Personalizada” é uma instrução personalizada, ou seja, é um procedimento definido pelo usuário para realizar uma ação específica durante a execução do teste. Além disso, essa palavra chave possui uma variável a ser passada durante a execução, com o valor da *string* “diretorio/ExemplodeDiretorio”.

A instrução que a palavra chave deverá executar está dentro da seção *Keywords*, abaixo do nome correspondente da palavra chave personalizada, a “Minha Palavra Chave Personalizada”.

Assim, quando estiver sendo executada essa palavra chave na seção *Test Cases*, a ação que será desempenhada é a de verificar se o diretório existe, através do comando *Directory Should Exist*, contido na seção de *Keywords*.

A tag *Arguments* indica que essa palavra chave recebe uma variável, representada pela variável “diretorio”, que teve o seu valor setado na seção *Test Cases*, sendo que seu valor é a *string* “diretorio/ExplodeDiretorio”.

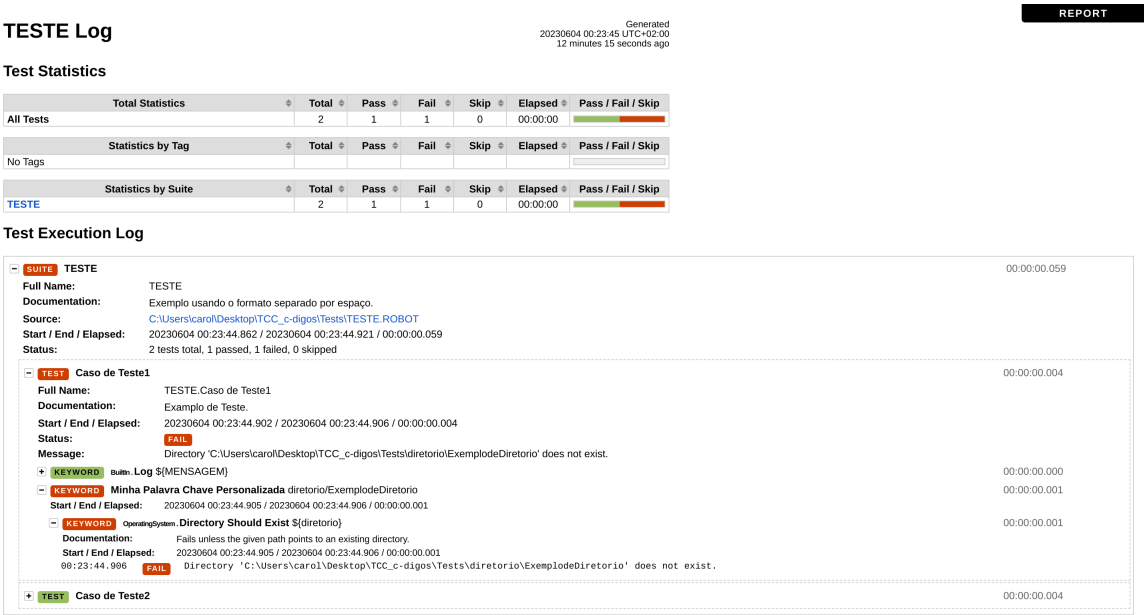
Para o “Caso de Teste2”, é executado primeiramente uma instrução nativa do *framework*, que verifica se o valor da variável mensagem é igual ao valor esperado, sendo a *string* “Ola, mundo!”.

A palavra chave personalizada “Funcao Python Que Retorna a Mensagem Seja Bem Vindo” é o nome de uma função escrita em Python que irá executar determinada ação. Nesse caso, a função Python será chamada e irá retornar a mensagem de “seja bem vindo”, sendo que essa função está dentro do arquivo “arquivo.py” no diretório de *Resources*.

Os relatórios de *Output* e *Log* são gerados por padrão após execução de cada cenário de teste ou arquivo de teste. Nesses relatórios, existem informações de quais foram os testes executados, o tempo de execução e quais palavras chaves foram executadas. Caso um caso de teste falhe, os relatórios também contém quais palavras chaves que falharam.

Desse modo, a Figura 8 mostra o relatório de *Log* padrão do *Robot Framework* que foi gerado após a execução do arquivo de testes proposto pela Figura 7. A Figura 8 apresenta o relatório de *Log* contendo quais foram os *status* dos testes, qual foi o tempo de execução e quais palavras chaves passaram e falharam.

Figura 8 – Relatório de Log gerado por padrão pelo Robot Framework



Fonte: Elaboração Própria (2023).

Para a jiga de testes desse trabalho, os relatórios padrão de *Log* e *Output* gerados pelo *framework* serão customizados através do uso da biblioteca, *rfmetrics*

(KLARCK *et al.*, 2022) e transformados em apenas um relatório, afim de propor uma interface gráfica mais intuitiva para o usuário quando for realizar a análise dos resultados de teste.

O uso do *Robot framework* para testes de *software* embarcado tem crescido. No trabalho proposto por PESONEN (2016), o *framework* atua como uma alternativa para realizar testes de aceitação em um terminal físico de máquina de cartão de crédito, sendo necessário apertar um determinado botão no terminal.

Nessa aplicação, o *framework* atua realizando os testes através do uso de uma máquina CNC controlada pelas *keywords* do *Robot* (PESONEN, 2016). Percebe-se a flexibilidade do *framework*, pois já há uma biblioteca desenvolvida pela comunidade com algumas *keywords* capazes de movimentar o eixo de uma fresadora CNC, a *CncLibrary*.

Portanto, a escolha do *Robot framework* para ser o *framework* de testes automatizados desse trabalho foi motivada pela flexibilidade oferecida e por já existirem trabalhos envolvendo testes em dispositivos de *software* embarcado.

2.6 Gherkin

O *Behavior-Driven Development* (BDD) engloba um conjunto de técnicas de engenharia de *software* com o objetivo de facilitar a criação e a entrega de *softwares* com a máxima eficiência na relação entre qualidade e agilidade (SMART, 2014).

Conforme aponta SMART (2014), o *Gherkin* é uma ferramenta do BDD, que tem como objetivo fornecer uma linguagem clara. O *Gherkin* utiliza um conjunto específico de *keywords* para executar determinadas ações (SmartBear Software, 2019).

Nesse sentido, o *Gherkin* utiliza a linguagem natural para criar uma linguagem estruturada para os cenários de teste, onde cada cenário descrito possui suas etapas (LIAO *et al.*, 2017). Há uma estrutura base contendo as palavras chave “Dado”, “Quando”, “Então”, “E” e “Mas” para criar o cenário de teste. A ordem natural de descrição do cenário é utilizando as palavras “Dado”, “Quando” e “Então” (SMART, 2014).

Assim, a palavra chave “Dado” representa a pré-condição do teste, o “Quando” indica a ação que será executada para realizar o teste e palavra chave “Então” indica o resultado esperado. Nesse sentido, as palavras chaves “E” e “Mas” servem como apoio para descrever de forma mais clara os casos de teste (SMART, 2014). A Figura 9 representa um caso de teste escrito utilizando a sintaxe essencial do *Gherkin*.

Figura 9 – Exemplo de cenário de teste descrito utilizando o *Gherkin*

```
1 Cenário: Login em uma página Web
2   Dado que eu esteja em uma determinada página web
3   Quando eu preencher credenciais inválidas de Login
4   Então devo receber a mensagem de Login Inválido
5
```

Fonte: Elaboração Própria (2023)

O cenário de teste da Figura 9 consiste em realizar o *login* em uma página *web*. A pré condição de teste indicada pela palavra chave “Dado” é estar em uma determinada página *web*. Após a pré condição ter sido executada, a palavra chave “Quando” indica que a ação de teste será realizada, assim é esperado que credenciais inválidas de *login* tenham sido preenchidas na página *web*. Por fim, a palavra chave “Então” indica que o resultado esperado é receber a mensagem de *login* inválido.

O *Gherkin* é uma linguagem de simples entendimento, em que não é necessário conhecimentos de programação para compreender quais serão os passos a serem executados. A descrição desses passos já indica o contexto da ação, qual verificação será feita e qual será o resultado esperado. Nesse sentido, o *Robot framework* não possui uma biblioteca específica para o uso do *Gherkin*, entretanto, é possível encontrar bibliotecas feitas pela comunidade do *framework* que fazem a tradução de cenários *Gherkin* em *keywords* que o *Robot* seja capaz de executar.

Alternativamente, também é possível escrever as palavras chaves personalizadas do *Robot framework* diretamente na estrutura de sintaxe do *Gherkin*, sem a necessidade de bibliotecas adicionais. Essa foi a estratégia escolhida nesse trabalho para descrever os casos de teste usando a linguagem *Gherkin* no *Robot Framework*.

Assim, a vantagem da utilização do BDD é garantir que todos entendam o objetivo de um projeto, de forma a garantir os requisitos estão sendo satisfeitos.(SMART, 2014). Por esse motivo, o *Gherkin* foi escolhido como linguagem para descrever os casos de teste desse trabalho, pois ele oferece uma linguagem simples ao usuário, de modo o usuário não precisa ter conhecimento do *Robot Framework* e consegue compreender quais passos estão sendo executados durante o teste de um componente sem dificuldades.

Entretanto, conforme aponta (SMART, 2014), o BDD apresenta desvantagens quando testes complexos precisam ser abstraídos de modo que os testes podem se tornar frágeis e de difícil manutenção.

2.7 Banco de dados *MongoDB*

Os dados de entrada do sistema e os resultados esperados de saída dos testes precisam ser conhecidos para testes funcionais serem realizados. Assim, esses dados estão armazenados no banco de dados *MongoDB* e também os esquemáticos de ligação dos componentes digitais a serem testados e o esquema de ligação dos pinos do DAQ *USB-6212*.

Conforme a documentação oficial, “O MongoDB é um banco de dados de documentos projetado para facilitar o desenvolvimento e o dimensionamento de aplicativos.”(MONGODB, 2023c). Ele é uma base de dados em nuvem, na qual os dados podem ser armazenados em regiões na AWS, Azure e *Google Cloud*(MONGODB, 2023a). Há uma versão gratuita com algumas limitações relacionadas a capacidade de armazenamento, desempenho e a disponibilidade de certas ferramentas. A base de dados utilizada nesse trabalho é da versão gratuita disponibilizada pelo *MongoDB* e ela está armazenada em regiões da AWS(*Amazon Web Services*).

O MongoDB é conhecido por ser um banco de dados não relacional, também chamado de NoSQL. Uma base de dados NoSQL é caracterizada por ser uma base não tabular, de modo que os dados são armazenados de modo diferente dos bancos de dados relacionais (MONGODB, 2023d). Desse modo, uma base de dados do *MongoDB* é formada por uma ou mais *collections* (coleções). As coleções podem ser definidas como o local onde os *documents* (documento) são armazenados, as coleções são análogas as tabelas em uma base de dados SQL (MONGODB, 2023b).

Os registros no *MongoDB* são chamados de *documents* e são compostos por chave e valor, o que torna seu formato muito parecido aos objetos *json* (MONGODB, 2023c). O *json* é um formato para a troca de dados, que tem como objetivo ser de fácil compreensão para os humanos e também ser fácil para que máquinas possam gerar e compreender esse formato, sendo que ele é baseado em padrões da linguagem *JavaScript*(JSON, 2023). A Figura 10 representa um registro do MongoDB, apresentando um *document* que tem como chave os valores nome, idade e grupos. Percebe-se também que cada chave possui seu valor correspondente, indicados pelos valores contidos dentro das aspas.

Figura 10 – Formato de um dado no *MongoDB*

```
1 {  
2   nome: "Carolina",  
3   idade: "24",  
4   grupos: ["ifsc", "engenharia eletrônica"]  
5 }  
6
```

Fonte: Adaptado de MONGODB (2023c)

As vantagens de se utilizar uma base de dados NoSQL podem ser resumidas na flexibilidade do armazenamento dos dados, estar de acordo com desenvolvimento ágil, porque as bases são projetadas para serem expansíveis, e também ter a capacidade de lidar com um grande volume de dados (MONGODB, 2023d). O NoSQL também está alinhado com paradigmas modernos de arquitetura de *software*, como o microserviços (MONGODB, 2023d).

Adicionalmente, as bases de dados *NoSQL* permitem que sejam cadastrados dados que não possuem relação entre si, de modo que no *MongoDB* o usuário pode estruturar o *Json* da maneira que desejar. Essa abordagem pode ser útil para a expansão dos *documents* que armazenam os esquemáticos de ligação dos componentes. É possível armazenar componentes que tenham um número de pinos de entrada e saídas menores ou maiores do que os presentes nos componentes já previamente cadastrados, sem prejuízos de inconsistências de dados.

Essa mudança de número de pinos dos componentes poderia ser uma preocupação em uma base de dados SQL, pois os dados devem ser definidos antecipadamente na estrutura das tabelas. Pelos motivos de flexibilidade na estrutura dos dados, estar de acordo com o desenvolvimento ágil e que os dados contidos no banco de dados tenham um formato mais legível para o entendimento do usuário, optou-se pela escolha da base de dados *NoSQL* do *MongoDB*.

2.8 Resumo dos dispositivos e tecnologias adotadas para desenvolver a jiga de testes

Este capítulo apresentou os pontos relevantes da pesquisa e do desenvolvimento deste trabalho. Foi dada ênfase ao funcionamento e comportamento das ferramentas utilizadas, bem como justificado a escolha de determinada ferramenta em detrimento de outra semelhante quando aplicável.

Desse modo, o Quadro 6 apresenta as ferramentas utilizadas no trabalho e suas respectivas funções.

Quadro 6 – Descrição resumida das funções das tecnologias utilizadas no projeto da jiga de testes

Diretório	Funcionalidade
<i>DAQ USB-6212</i>	Equipamento utilizado para realizar a geração e aquisição de sinais digitais e alimentação dos componentes digitais que serão testados.
<i>Python - biblioteca nidaqmx</i>	Biblioteca utilizada para controlar a geração e a aquisição de sinais no <i>DAQ USB-6212</i> .
<i>Testes Funcionais</i>	Técnica de teste utilizada para confirmar se o componente digital está dentro do funcionamento esperado.
<i>Robot Framework</i>	<i>Framework</i> de testes automatizados utilizado para executar os cenários de teste.
<i>Gherkin</i>	Sintaxe de linguagem utilizada para descrever os casos de teste executados pelo <i>Robot Framework</i> .
<i>MongoDB</i>	Responsável por armazenar os dados de entrada dos testes, os resultados esperados dos testes e os esquemáticos de ligação dos componentes digitais e do <i>DAQ USB-6212</i> .
<i>Formato Json</i>	Formato utilizado para armazenar os esquemáticos de ligações dos componentes, do <i>DAQ</i> e também os resultados esperados de cada componente

Fonte: Elaboração Própria (2023).

3 DESENVOLVIMENTO

Este Capítulo é responsável por abordar o desenvolvimento da jiga de testes automatizados e aprofundar como a jiga foi projetada. Primeiramente, será descrita a metodologia do projeto apresentando quais as limitações da jiga de Testes e quais foram os métodos utilizados para validá-la. Após isso, será abordado o desenvolvimento do projeto da jiga, explicando como o projeto está estruturado e descrevendo o desenvolvimento dos casos de uso que a jiga de testes automatizados está submetida.

3.1 Metodologia

Segundo (GIL *et al.*, 2002,p.41) uma pesquisa exploratória “tem como objetivo proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a constituir hipóteses. Pode-se dizer que estas pesquisas têm como objetivo principal o aprimoramento de ideias ou a descoberta de intuições”. Assim, de forma a aprimorar os testes de componentes eletrônicos, essa pesquisa propõe uma abordagem não usual para esse processo, formulando e implementando a hipótese de realizar a validação do funcionamento dos componentes por meio do uso de um *framework* de testes automatizados.

Para o desenvolvimento da jiga de testes desse trabalho com apenas um único equipamento, foi utilizado o dispositivo DAQ USB-6212 da fabricante *National Instruments*. Esse é um equipamento disponível no departamento do curso de Engenharia Eletrônica do Instituto Federal de Santa Catarina, que possui pinos de entradas e saídas digitais que são responsáveis por realizar a aquisição e geração de sinais digitais.

Os componentes que serão testados devem possuir como pré-requisito ter uma alimentação de +5V, pois será utilizado o pino do DAQ correspondente a esse valor para realizar a alimentação do componente digital em teste. Para testar componentes com outro valor de alimentação, é necessário utilizar uma fonte externa para alimentar o componente.

Para iniciar a execução de um caso de teste, é necessário conectar o DAQ a porta *USB* do computador e estar com os *drivers* do DAQ *USB-6212* devidamente instalados.

A jiga de testes foi desenvolvida para testar apenas componentes digitais que possuam a proporção de dois pinos de entrada de sinal para uma saída, devido ao formato que os dados foram estruturados no *json* que representa o esquemático de ligação dos componentes. Além disso, ela possui a capacidade de testar apenas um componente por vez e que possua no máximo 16 pinos de entrada e saída, pois somente os pinos *port0* do DAQ *USB-6212* foram utilizados, que possui como limitação

16 pinos que podem servir como entrada ou saídas digitais no DAQ *USB-6212*.

Desse modo, considerando as limitações apresentadas, as portas lógicas foram os componentes utilizados para validar a jiga de testes. Ademais, o almoxarifado do curso possui um grande número desses componentes, que são facilmente suscetíveis a apresentarem problemas.

Para o armazenamento dos esquemáticos de pinos do DAQ, dos componentes e as tabelas contendo os resultados esperados, foi utilizado o banco de dados *MongoDB*, que é uma base de dados em *cloud* com uma versão gratuita disponível. Por se tratar de um banco de dados de armazenamento em nuvem, é necessário que o usuário esteja com a conexão a *internet* disponível no momento da execução de todos os casos de uso que a jiga de testes automatizados está submetida.

O dispositivo *USB-6212* foi controlado através do uso da biblioteca *nidaqmx*. Por fim, o *Robot framework* é responsável pela execução dos casos de testes. Esse é um *framework* de automação de testes *open source* genérico e de estrutura flexível. A sintaxe de linguagem *Gherkin* foi utilizada para descrever os casos de teste desenvolvidos no *Robot framework*.

Dessa forma, os testes funcionais foram a metodologia de testes utilizada para validar se os componentes em testes estão dentro do funcionamento correto. Todas as possíveis combinações de sinais de entrada são testadas em todos os pinos de entrada do componente em teste. O valor da saída no pino do componente é lido e comparado com os resultados esperados para o determinado sinal de entrada que foi aplicado.

Ao final de execução de cada teste, é gerado um relatório de teste através do uso da biblioteca *rfmetrics*, contendo o *status* do teste e outras métricas, como o tempo de execução.

Para validar que os resultados produzidos pela jiga de testes estão corretos, foi desenvolvido um *script* separado da jiga que realiza a geração e aquisição de sinais utilizando o DAQ. O *script* foi implementado com o auxílio da biblioteca *nidaqmx* e funciona de maneira similar ao trecho de código utilizado para realizar a geração e aquisição de sinais através da jiga de Testes automatizados. Para o *script* de validação de sinais, também são testados todas as possibilidades de entrada de sinal no componente. Entretanto, esse processo é realizado apenas no primeiro conjunto de entradas do componente e o sinal de saída correspondente a esse conjunto é capturado, sendo essa uma diferença entre o *script* e a jiga, pois na jiga são testados todos os pinos do componente.

Para analisar os sinais produzidos pelo DAQ utilizando o *script* adicional, disponível na seção de resultados, foi empregado o analisador lógico (*Logic Analyzer*) de 8 canais com interface *USB*. A necessidade de desenvolver essa abordagem de

validação ocorre devido à presença de outras interações na jiga de testes além da geração e aquisição de sinais, o que dificulta a captura do momento exato em que a leitura e a escrita estão ocorrendo. Assim, como a lógica empregada para realizar a geração e aquisição de sinais é a mesma em ambos os casos, é possível garantir que a geração e aquisição de sinais utilizando os testes automatizados está correto.

3.2 Configurações necessárias para a instalação da jiga de testes automatizados

Para a execução dos casos de teste, é necessário realizar algumas configurações iniciais. A primeira etapa consiste em ter o Python instalado com versões acima da 3.8 no computador, as etapas para a instalação dessa linguagem de programação dependem do sistema operacional utilizado pelo usuário, portanto, é recomendado que se siga a documentação disponível pelo Python.

Após isso, é possível prosseguir para a instalação do *Robot framework* utilizando o comando *pip* para gerenciamento de pacotes Python, executado via terminal conforme abaixo:

```
pip install robotframework
```

Todos esses passos para a instalação do *Robot* estão disponíveis na documentação do Python e do *Robot framework*. A jiga de testes automatizados não possui uma interface gráfica disponível para o usuário, portanto é recomendado utilizar um editor de códigos, como o *Vscode*, que pode ser instalado seguindo sua documentação oficial. Entretanto, caso o usuário opte por escolher outro editor de sua preferência, é necessário garantir que o editor escolhido possui compatibilidade para a execução de códigos provenientes do *Robot framework*.

Utilizando o *Vscode*, é recomendável a instalação de uma extensão disponível, a *Robot Code*(BIEHL, 2023), essa é uma ferramenta que permite depurar os casos de testes, refatorar códigos, realizar revisões de sintaxe e outras funcionalidades. Esse passo é opcional, mas que pode ser muito útil em casos que o usuário deseja realizar algum tipo de expansão dos cenários propostos além daqueles já desenvolvidos neste trabalho.

É importante ressaltar que para a execução do cenário de teste de um componente eletrônico, é necessário que o dispositivo DAQ *USB-6212* da NI esteja conectado a porta USB do computador. Também é essencial garantir que os *drivers* do DAQ estão corretamente instalados no computador antes de prosseguir com os testes, os *drivers* podem ser obtidos na documentação oficial da NI.

Para a instalação das outras bibliotecas requeridas pela jiga de testes, existe um arquivo no repositório chamado de *requirements.txt*. Esse é um arquivo com todas

as bibliotecas necessárias para executar os casos de teste. As bibliotecas podem ser instaladas com o seguinte comando:

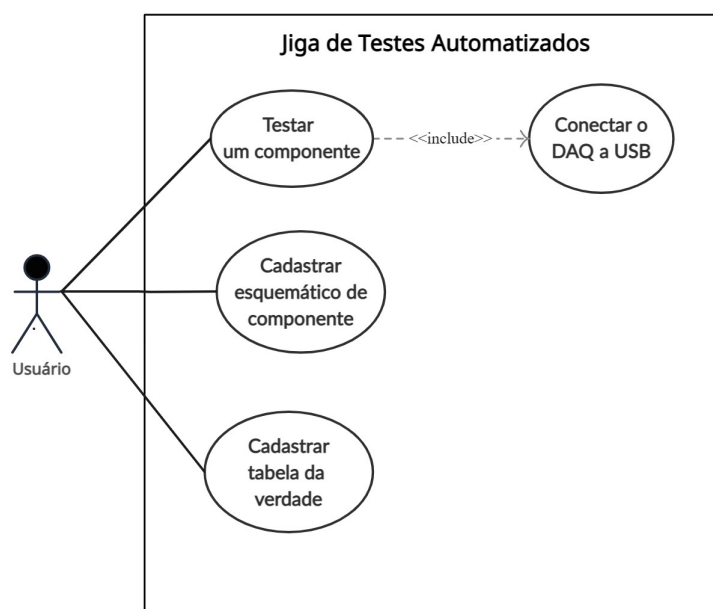
```
pip install -r requirements.txt
```

Após a realização dos passos acima, a jiga de testes está pronta para ser executada.

3.3 Casos de uso em que a jiga de testes automatizados está submetida

A jiga de testes foi projetada e implementada para ser submetida aos seguintes casos de uso apresentados na Figura 12.

Figura 11 – Casos de uso da jiga de testes



Fonte: Elaboração Própria(2023).

Para os casos de uso de cadastro do esquemático de ligação de um componente ou da tabela da verdade, não é necessário que o DAQ esteja conectado, pois essas ações independem do equipamento. Todos os casos de uso necessitam da conexão a *internet* disponível, pois em todos eles ocorrem interações com a base de dados.

Todos os casos de uso são executados através dos arquivos contidos no diretório “*Tests*”. A descrição dos cenários de testes foram desenvolvidas utilizando o *Gherkin*, o que proporciona maior clareza para o usuário de quais passos estão sendo executados durante os testes.

3.4 Estrutura do projeto da jiga de testes automatizados

A organização de diretórios da jiga de testes segue uma estrutura adaptada da estrutura de projeto sugerida pelo *Robot framework*, conforme mostrado abaixo:

Figura 12 – Estrutura de projetos da jiga de testes

```
Jiga de testes
├── Cadastros
│   ├── esquemáticoDoComponente.txt
│   └── resultadosEsperadosDoComponente.txt
├── Resources
│   └── componentesDigitais.robot
├── Results
│   ├── log.html
│   ├── output.xml
│   └── report.html
├── Scripts
│   ├── database.py
│   ├── esquemático.py
│   └── componentes_digitais.py
├── Tests
│   ├── Cadastros.robot
│   └── TestedeComponentes.robot
├── esquemático.txt
├── .gitignore
├── README.md
└── testeDeComponente.bat
```

Fonte: Elaboração Própria(2023).

O diretório jiga de testes é a pasta raiz do projeto. Assim, no diretório *Cadastros*, é possível encontrar os arquivos “.txt” que possuem um objeto *json* que cadastra esquemáticos de ligação de componentes e os resultados esperados dos testes. Nesse diretório, ocorre a interação entre o usuário e a jiga de teste. O usuário deve preencher as informações necessárias nos arquivos desse diretório para fazer a inserção de um novo esquemático de ligação de um componente ou cadastrar novos resultados esperados para um determinado componente.

Na pasta *Resources* é possível encontrar as palavras chaves do *Robot*, denominadas de *keywords*(palavras chaves) no arquivo “componentesDigitais.robot”. Esse é o arquivo onde existe a correspondência entre a *keyword Robot* e as funções desenvolvidas em Python que devem ser chamadas para executar a ação sugerida pela palavra chave, bem como também instruções nativas do *Robot framework*.

No diretório *Results* estão os relatórios de testes padrão que são gerados pelo *Robot framework* ao final da execução de cada cenário de teste. Os arquivos

“log.html”, “output.xml” e “report.html” serão customizados e transformados em apenas um relatório utilizando a biblioteca *rfmetrics*.

No diretório *Scripts* estão todos os códigos desenvolvidos em Python, são eles que executam as ações na jiga de testes. Algumas funções contidas nos arquivos dentro desse diretório são executadas quando chamadas pelas palavras chaves do diretório de *Resource*. As demais funções foram criadas para modularizar o código, buscando tornar o código mais organizado e reutilizável.

O diretório *Tests* concentra todos os casos de testes que a jiga pode executar. Esse é outro diretório que ocorre a interação do usuário e a jiga de Testes, pois os arquivos contidos dentro dele são os arquivos responsáveis por executar os cenários de testes desenvolvidos.

É no diretório *Tests* que o usuário deve executar o arquivo “TesteDeComponentes.robot”, caso queira executar o teste de um componente já previamente cadastrado. Por outro lado, se o usuário deseja cadastrar o esquemático de ligação de um componente ou os resultados esperados, ele deve executar o arquivo “Cadastros.robot”.

O arquivo README.md contém a documentação inicial do repositório. O arquivo “requirements.txt” lista todas as dependências necessárias para executar o projeto da jiga de testes automatizados. O arquivo “testeDeComponente.bat” contém os comandos para executar o caso de teste de um componente, sendo responsável por executar o arquivo de teste e gerar o relatório de testes customizado.

Por fim, o arquivo “esquematico.txt” contém o esquemático de ligação entre o DAQ e o componente em teste. Esse arquivo é gerado apenas quando o caso de teste de um componente é executado. Através do arquivo “esquematico.txt”, o usuário terá acesso às informações necessárias para realizar a correta ligação entre o DAQ e o componente em teste.

Essa estrutura de projeto foi adaptada da estrutura de projeto sugerida pelo *Robot framework*. O objetivo dessa adaptação foi fornecer maior clareza da disposição dos arquivos e dos objetivos deles para os usuários que tenham pouca ou nenhuma familiaridade com o *Robot framework*. A adaptação da estrutura de projeto também foi feita levando em consideração a importância da organização, manutenção, reutilização de código e integração com outras ferramentas. Nesse sentido, também foi considerado que boas práticas de desenvolvimento e automação de testes sejam atendidas.

3.5 Implementação do caso de uso para teste de um componente eletrônico digital

Nessa seção, serão tratados os passos necessários para realizar o teste em um componente eletrônico digital sob as condições citadas na seção de Metodologia. Será apresentado como realizar o caso de teste para a porta lógica *AND* contida no

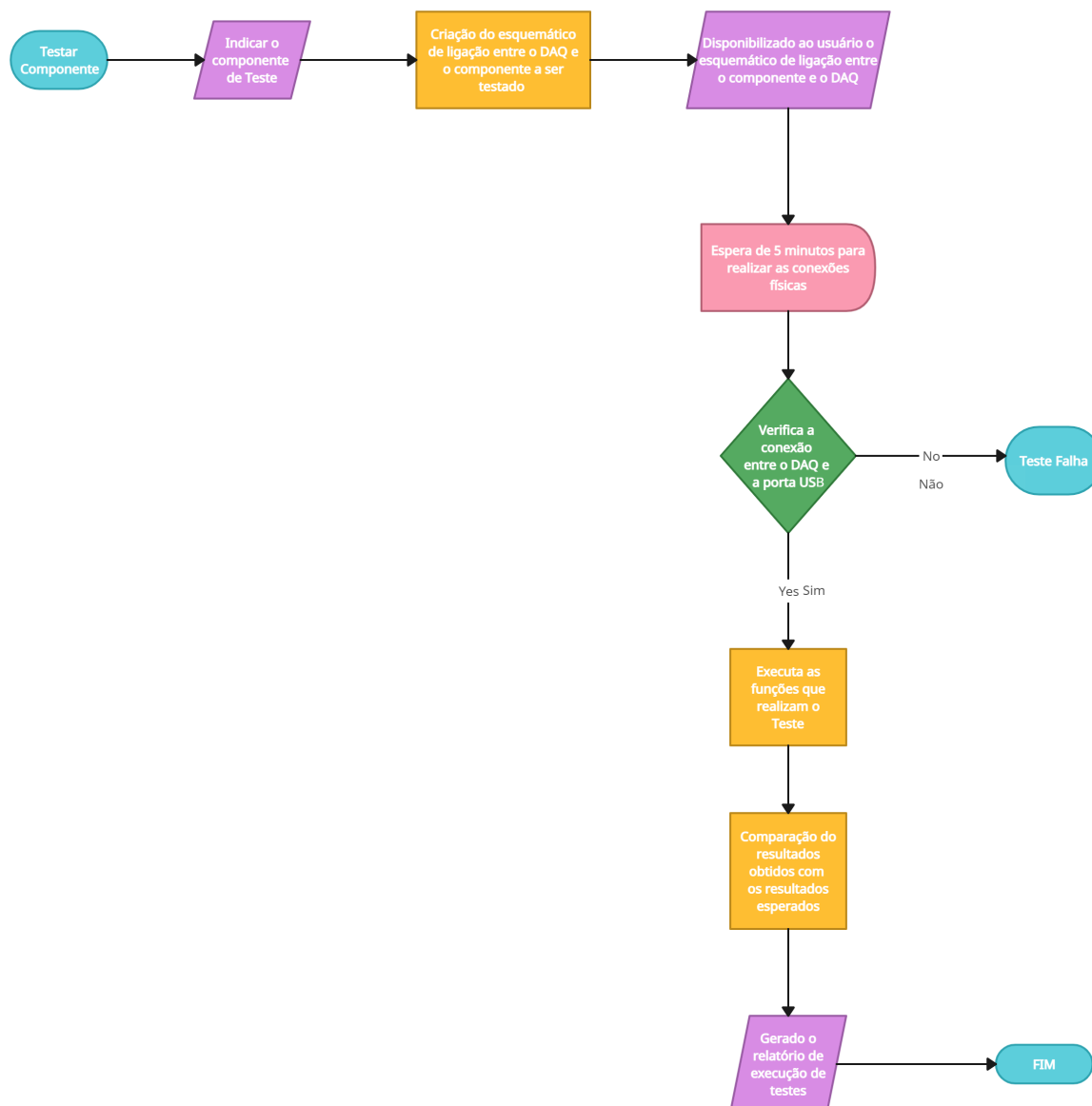
circuito integrado 74HC08.

3.5.1 Testando a porta lógica AND 74HC08

Para testar a porta lógica 74HC08, é necessário que ela já possua o esquemático de ligação de pinos cadastrados na base de dados do *MongoDB* e também que seus resultados esperados tenham sido cadastrados.

O teste irá falhar caso o esquemático do componente e os resultados esperados não tenham sido cadastrados antes da sua execução. Os cenários de testes que envolvem os cadastros serão explicados posteriormente. Para a explicação dessa seção, o esquemático de ligação do componente 74HC08 e os resultados esperados já foram cadastrados previamente.

Desse modo, a Figura 13 apresenta um fluxograma contendo as ações que são executadas para testar um componente.

Figura 13 – Fluxograma contendo as ações da jiga de testes

Fonte: Elaboração Própria(2023).

3.5.1.1 Diretório de Tests

No diretório de *Tests*, existe o arquivo “TestedeComponentes.robot”. Esse arquivo é o responsável por executar o cenário de teste que envolve os testes funcionais de um componente eletrônico digital.

No arquivo “TestedeComponentes.robot” é necessário que o usuário indique qual o componente que será testado na variável “\$componenteEscolhido”, lembrando de respeitar a indentação de pelo menos dois ou mais espaços requeridas pelo *Robot framework*. No exemplo dessa seção, o seu valor será substituído por 74HC08, entretanto, qualquer outro componente previamente cadastrado na base de dados poderia ter sido utilizado para o teste.

A variável “\$tempoDeEspera” indica o tempo de pausa no teste para que o usuário realize a conexão de ligações entre o DAQ e o componente escolhido. O valor padrão da variável “\$tempoDeEspera” é de 5 minutos, para que o usuário faça as conexões necessárias entre o DAQ e o componente em teste. Esse tempo pode ser alterado conforme escolha do usuário, lembrando de respeitar a indentação requerida. O cenário de teste descrito em *Gherkin* está mostrado na Figura 14.

Figura 14 – Caso de teste para testar um componente eletrônico digital

```

1  *** Settings ***
2  Documentation
3
4  Resource      ../../Resources/componentesDigitais.robot
5
6  Test Setup     Obtendo o esquemático de ligação entre o DAQ e o componente a ser
                  testado
7
8  *** Variables ***
9  ${componenteEscolhido}=    74HC08
10 ${tempoDeEspera}=    5 minutes
11
12 *** Keywords ***
13
14 Obtendo o esquemático de ligação entre o DAQ e o componente a ser testado
15     Retorne o esquemático de ligação do componente com o DAQ    ${
16     componenteEscolhido}
17     Sleep    ${tempoDeEspera}
18
19 *** Test Cases ***
20
21 Caso de teste para componentes eletronicos
22     [Documentation]    Caso de Testes de CI's escolhidos pelo usuário
23     Dado que o DAQ esteja conectado a porta USB
24     Quando for realizado o teste do componente escolhido
25     Entao verifique que o componente esta funcionando
26
27

```

Fonte: Elaboração Própria (2023).

Existe uma pré condição de teste para o cenário de teste de um componente. Ela é indicada pela instrução *Test Setup* do *Robot Framework*. O *Test Setup* indica que a pré-condição de teste é ter o esquemático de ligação entre o DAQ e o componente a ser testado. Essa instrução foi utilizada para indicar que existe uma ação que precisa ser executada antes do teste de um componente começar. O objetivo do *Test Setup* é preparar adequadamente o ambiente de teste.

Quando a instrução de *Test Setup* for executada, o usuário irá receber o esquemático de ligação que ele deve seguir para conectar os pinos do DAQ com os

pinos do componente. Após isso, será dado o tempo de espera de 5 minutos para que o usuário faça as conexões físicas requeridas.

Ao finalizar o tempo de pausa de 5 minutos, o teste irá iniciar. A palavra chave “Dado” do *Ghekin* indica o primeiro passo a ser executado durante o cenário de teste de um componente. Desse modo, a primeira etapa é garantir que o DAQ está conectado a porta *USB* do computador. Se for confirmado que o DAQ está conectado, o teste do componente escolhido, no exemplo, o 74HC08, será executado. A palavra chave “Quando” indica a execução do cenário de teste.

Para executar o teste, todas as possibilidades de entrada de sinal que foram cadastradas em um *document* da *collection* de resultados esperados do *MongoDB* para a porta lógica *AND* serão aplicadas nos pinos de entrada do componente. Os valores de saída serão lidos e armazenados em uma variável no código. Os valores de saída obtidos através da leitura dos pinos de saída do componente serão comparados com os resultados esperados de saída contidos no *Json* do *MongoDB* para a porta lógica *AND*.

A palavra chave “Então” do *Gherkin* indica o momento do teste que é verificado se todos os valores comparados são iguais. Se os valores forem iguais, o teste está aprovado. O *Robot framework* gera por padrão três relatórios de testes, contendo o *status* do teste, de aprovado ou reprovado e também os tempos de execução. Esses relatórios são customizados e transformados em apenas um relatório através da biblioteca *rfmetrics*. Para que o caso de teste da Figura 14 seja executado e tenha o relatório gerado automaticamente, o seguinte comando deve ser executado no terminal:

```
testeDeComponente.bat
```

O arquivo “.bat” contém os comandos necessários para executar o caso de teste de um componente e gerar automaticamente o relatório customizado. O relatório customizado terá o nome de “metrics-XX.html”, onde “XX” representa um *range* de números aleatórios gerados pela biblioteca *rfmetrics*.

O relatório customizado está em formato “.html” e pode ser aberto utilizando o navegador. Desse modo, o arquivo “TestedeComponentes.robot” do diretório *Tests* é o único arquivo em que o usuário precisa executar algum tipo de ação durante o teste de um componente.

3.5.1.2 Obtendo o esquemático de ligação entre o componente e o DAQ

Obter o esquemático de ligação entre o componente a ser testado e o DAQ é uma pré-condição de teste. A condição *Test Setup* do diretório de *Tests* indica que o arquivo de teste automatizado possui uma pré condição de teste. Assim, quando o arquivo “TestedeComponentes.robot” do diretório *Tests* é executado, a palavra chave customi-

zada da condição de *Test Setup* tem sua instrução contida na seção de *Keywords* do arquivo de teste de componente. Essas condições podem ser vistas na Figura 15.

Figura 15 – Indicação da pré-condição de teste para testar um componente

```
1 *** Settings ***
2 Documentation
3
4 Resource      ../../Resources/componentesDigitais.robot
5
6 Test Setup    Obtendo o esquemático de ligação entre o DAQ e o componente a ser
               testado
7
8 *** Variables ***
9 ${componenteEscolhido}=    74HC08
10 ${tempoDeEspera}=    5 minutes
11
12 *** Keywords ***
13
14 Obtendo o esquemático de ligação entre o DAQ e o componente a ser testado
15     Retorne o esquemático de ligação do componente com o DAQ    ${
               componenteEscolhido}
16     Sleep    ${tempoDeEspera}
17
18
19
```

Fonte: Elaboração Própria (2023).

A condição *Test Setup* realiza a chamada da palavra chave customizada “Obtendo o esquemático de ligação entre o DAQ e o componente a ser testado”. Essa palavra chave possui uma correspondência na seção de *Keywords* do arquivo de teste. Por sua vez, a ação que deverá ser executada pela palavra chave “Retorne o esquemático de ligação do componente com o DAQ \${componenteEscolhido}” está presente no diretório de *Resources*.

No arquivo “componentesDigitais.robot” do diretório de *Resources* também existe uma seção de *Keywords* do *Robot framework*. O código que deverá ser executado quando a palavra chave que retorna o esquemático de ligação é chamada está mostrado na Figura 16.

Figura 16 – Ações executadas pela palavra chave que retorna o esquemático de ligação entre o DAQ e o componente

```

1 *** Keywords ***
2
3 Retorne o esquemático de ligação do componente com o DAQ
4     [Arguments]    ${NomedoComponente}
5     Set Global Variable    ${NomedoComponente}
6     &{esquematico}=    Return Esquematico De Ligacao    ${NomedoComponente}
7     Log To Console    ${esquematico}
8     Set Global Variable    &{esquematico}
9

```

Fonte: Elaboração Própria (2023).

Desse modo, a *keyword* que retorna o esquemático de ligação tem como requisito receber o nome do componente. Então, a variável que contém o nome do componente será definida globalmente para que todo o arquivo de *Resource* possa utilizá-la em outras etapas do teste.

Com o nome do componente definido, a função desenvolvida em *Python* que retorna o esquemático de ligação é chamada. Essa função está presente no arquivo “esquematico.py” do diretório *Scripts*.

Para que o esquemático de ligação entre o DAQ e o componente seja gerado, é necessário obter os *documents* do *MongoDB* que contém a divisão de pinos do DAQ e do componente separadamente. No arquivo “*database.py*” estão as funções que retornam a divisão de pinos do DAQ e do componente a ser testado. A função que retorna o *Json* com o esquemático de pinos do componente a ser testado está mostrada a seguir na Figura 17.

Figura 17 – Função desenvolvida em *Python* que retorna a divisão de pinos do componente a ser testado

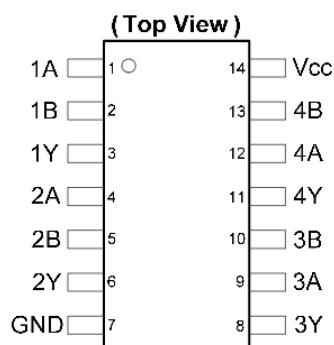
```

1 def return_component(componentName):
2     collection = connect_to_database("ComponentesDigitais")
3     cursor = collection.find({"nome": componentName})
4
5     for record in cursor:
6         return record
7

```

Fonte: Elaboração Própria (2023).

A função da Figura 17 conecta-se a base de dados e busca na *collection* “ComponentesDigitais” do *MongoDB* o componente a ser testado. No caso desse exemplo, o esquemático de ligação de pinos do 74HC08 será retornado pela função. A Figura 18 apresenta o esquemático de ligação do 74HC08.

Figura 18 – Top View do componente 74HC08

Fonte: Incorporated, Diodes (2013a).

O *document* contido na *collection* de componentes digitais da base de dados traduz a divisão de pinos do componente, mostrada na Figura 18, em um formato que seja entendível para o computador. Assim, o esquemático do componente da Figura 18 é traduzido em formato de um objeto *Json*, conforme mostrado abaixo:

```
{
  "_id": {
    "$oid": "6368012c9f264fbc148df0db"
  },
  "nome": "74HC08",
  "tipoDeComponente": {
    "portaLogica": "AND"
  },
  "entradas": {
    "numeroDoPino1": 1,
    "numeroDoPino2": 2,
    "numeroDoPino3": 4,
    "numeroDoPino4": 5,
    "numeroDoPino5": 9,
    "numeroDoPino6": 10,
    "numeroDoPino7": 12,
    "numeroDoPino8": 13
  },
  "saidas": {
    "numeroDoPino1": 3,
    "numeroDoPino2": 6,
    "numeroDoPino3": 8,
    "numeroDoPino4": 11
  },
  "alimentacao": {
    "vcc": 5,
    "pinoAlimentacao": "14"
  },
  "referencia": {
```

```
        "gnd": 0,  
        "pinoGND": "7"  
    }  
}
```

Dessa forma, o formato *Json* foi a abordagem utilizada para armazenar o esquemático de ligação de pinos dos componentes a serem testados e o esquemático de ligação do DAQ. O esquemático de ligação do DAQ está contido na *collection* “DAQ” do *MongoDB*.

O DAQ USB-6212 é um dispositivo que possui pinos digitais multifuncionais capazes de operar tanto como entrada quanto como saída de sinais digitais. Para estabelecer a conexão adequada entre o DAQ e o componente em teste é necessário apenas conhecer quais são os pinos de entrada e saída de sinal do componente a ser testado, pois dentro do *range* de pinos digitais do DAQ não existe um pino específico para cada uma dessas funções. Ademais, também é necessário conhecer o pino de alimentação do componente, pois será utilizado o pino de +5V do DAQ para realizar a alimentação e os pinos de referência do componente e do DAQ.

Nesse contexto, são feitas manipulações na função que retorna o esquemático de ligação para criar a correspondência entre o número do pino do DAQ e o número do pino do componente a ser testado. Parte da manipulação feita pela função presente no arquivo “esquematico.py” que retorna o esquemático de ligação está mostrado a seguir.

Figura 19 – Parte da função *Python* que retorna a correspondência entre o pino do DAQ e a conexão com o pino do componente em teste

```
1 def return_esquematico_de_ligacao(componente1):  
2     for pinosDAQ, pinosDoComponenteEntrada in zip(pinosDigitaisDaq, pinosDasEntradas):  
3         :  
4         if pinosDAQ in pinosDigitaisDaqKeys.values():  
5             ind = keysDAQValues.index(pinosDAQ)  
6             esquematico["Numero do pino do DAQ(" + str(keysDAQ[ind]) + ") + ":" +  
7             str(pinosDAQ)] = "Numero do pino do componente para entrada de sinal:" + str(  
8             pinosDoComponenteEntrada)  
9         if pinosDAQ in pinosUtilizadosDAQ:  
10            pinosUtilizadosDAQ.remove(pinosDAQ)  
            keysDaqUtilizadas.append(keysDAQ[ind])
```

Fonte: Elaboração Própria (2023).

A parte do código mostrada pela Figura 19 verifica quais são os pinos de entrada do componente a ser testado e cria uma correspondência entre os pinos do DAQ disponíveis. Além disso, também é retirado os pinos do DAQ que foram utilizados, para garantir que não exista o uso de um pino do DAQ que já está ocupado.

Para criar a correspondência entre os pinos de saída do componente e o DAQ é realizado o mesmo processo. Os pinos de alimentação e de GND também passam pelo mesmo processo de criação de correspondências. Essa implementação garante que os pinos do DAQ sejam alocados de forma eficiente e que não haja conflitos de uso.

O arquivo “esquematico.txt” contém o objeto *Json* retornado para o usuário com o esquemático de ligação entre o DAQ e o componente em teste. Esse arquivo é gerado somente quando o teste de um componente está sendo realizado. O esquemático de ligação entre o DAQ e o 74HC08 que foi gerado está mostrado na Figura 20. O usuário deve realizar as conexões físicas conforme indicado pelo arquivo.

Figura 20 – *Json* contendo o esquemático de ligação entre o DAQ e o componente 74HC08

```

1 "nomeDoComponente": "74HC08",
2 "tipoDeComponente": "AND",
3 "Numero do pino do DAQ(p0_0):44": "Numero do pino do componente para entrada de
  sinal:1",
4 "Numero do pino do DAQ(p0_1):45": "Numero do pino do componente para entrada de
  sinal:2",
5 "Numero do pino do DAQ(p0_2):46": "Numero do pino do componente para entrada de
  sinal:4",
6 "Numero do pino do DAQ(p0_3):47": "Numero do pino do componente para entrada de
  sinal:5",
7 "Numero do pino do DAQ(p0_4):50": "Numero do pino do componente para entrada de
  sinal:9",
8 "Numero do pino do DAQ(p0_5):51": "Numero do pino do componente para entrada de
  sinal:10",
9 "Numero do pino do DAQ(p0_6):52": "Numero do pino do componente para entrada de
  sinal:12",
10 "Numero do pino do DAQ(p0_7):53": "Numero do pino do componente para entrada de
  sinal:13",
11 "Numero do pino do DAQ(p0_8):55": "Numero do pino do componente para saida de sinal
  :3",
12 "Numero do pino do DAQ(p0_10):57": "Numero do pino do componente para saida de
  sinal:6",
13 "Numero do pino do DAQ(p0_12):60": "Numero do pino do componente para saida de
  sinal:8",
14 "Numero do pino do DAQ(p0_14):62": "Numero do pino do componente para saida de
  sinal:11",
15 "Numero do pino do DAQ para alimentacao(+5V):42 ou 10":
16 "Numero do pino do componente para alimentacao de 5V:14",
17 "Numero do pino GND do DAQ(DGND):64": "Numero do pino do componente para de GND0:7"
18

```

Fonte: Elaboração Própria (2023).

Assim, o *Json* contém a correspondência de ligação entre o número do pino do DAQ e o número do pino do componente que deverá ser conectado. Para realizar a alimentação do componente, é sugerido dois pinos do DAQ que são capazes de fornecer a alimentação de +5V, que são os pinos 10 ou 42. O usuário pode escolher

qual dos pinos deseja conectar.

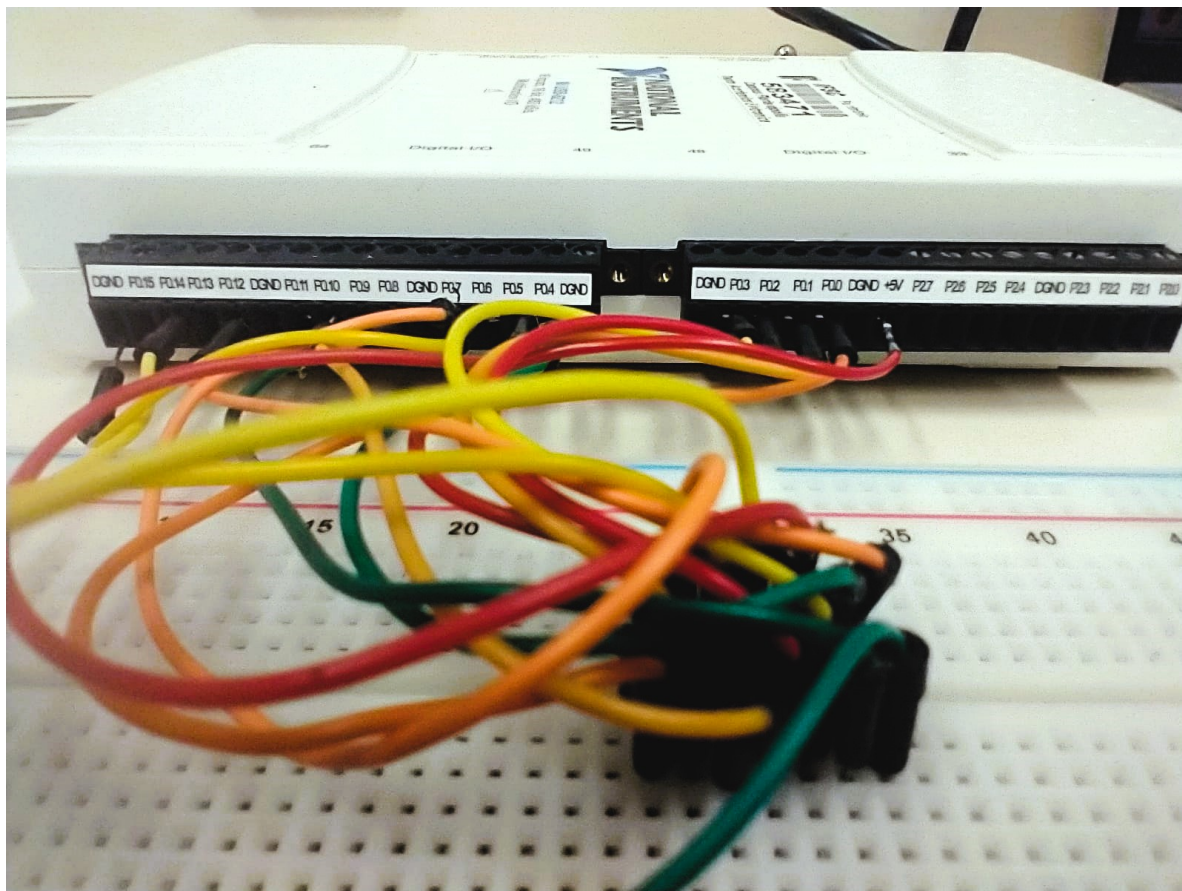
Além disso, como o DAQ USB-6212 possui diversos pinos, foi adicionado ao *Json* qual é o *port* correspondente ao número do pino do DAQ que deverá ser conectado. Assim, além de poder se guiar pelo número do pino, o usuário também pode se guiar pelo nome do pino, procurando o *port* correspondente.

A lógica para realizar a ligação entre o DAQ e o 74HC08 ocorre da seguinte maneira, exemplificando para o pino número 44 do DAQ, que corresponde ao *port* “p0_0”, o usuário deverá conectar ao pino número 1 do 74HC08, que serve como entrada de sinal no componente. O pino *port* “p0_8”, que corresponde ao pino número 55 do DAQ deve ser conectado ao pino número 3 do 74HC08, esse pino corresponde ao um pino de saída de sinal no componente em teste. Essa lógica de ligação deve ser aplicada ao demais pinos.

Após ter sido gerado o *Json* com o esquemático de conexões entre os pinos do DAQ e os pinos do 74HC08, o teste irá entrar em pausa. Nesse momento, existe um *delay* de 5 minutos no teste para que o usuário realize as conexões físicas requisitadas. O teste do componente propriamente dito só irá iniciar após se passarem os 5 minutos. Esse tempo de espera pode ser alterado conforme desejado no arquivo de teste de componente do diretório de *Tests*.

Desse modo, a Figura 21 apresenta a ligação entre o DAQ e o 74HC08 após seguir o esquemático de ligação sugerido pela jiga de testes.

Figura 21 – Ligação entre o DAQ e 74HC08 conforme o esquemático de ligação sugerido pela jiga de testes



Fonte: Elaboração Própria (2023).

3.5.1.3 Garantindo que o DAQ está conectado a porta USB

Quando a pré-condição de teste for cumprida, ou seja, que tenha se passado os 5 minutos de espera para que o usuário realize as conexões físicas entre os pinos do DAQ e do componente a ser testado, o caso de teste será iniciado.

Nesse momento, o sistema está pronto para começar a realizar o teste do componente. A palavra-chave “Dado” do *Gherkin*, contida no arquivo de teste de componente localizado no diretório *Tests*, indica a primeira etapa do teste. A Figura 22 mostra a primeira ação de teste a ser executada.

Figura 22 – Verificando se o DAQ está conectado a porta USB

```
1 *** Test Cases ***
2
3 Caso de teste para componentes eletronicos
4   [Documentation]    Caso de Testes de CI's escolhidos pelo usuário
5   Dado que o DAQ esteja conectado a porta USB
6
7
```

Fonte: Elaboração Própria (2023).

A palavra chave “Dado que o DAQ esteja conectado a porta USB” tem sua correspondência na seção de *Keywords* do arquivo “componentesDigitais.robot” do diretório de *Resources*. O código que a palavra chave personalizada irá executar está mostrado na Figura 23.

Figura 23 – Correspondência da palavra chave que verifica se o DAQ está conectado no diretório de *Resources*

```

1 *** Keywords ***
2
3 Dado que o DAQ esteja conectado a porta USB
4     ${deviceConnected}=    Check Device Connected
5     Should Be True         ${deviceConnected}
6     Log To Console         ${deviceConnected}
7
8

```

Fonte: Elaboração Própria (2023).

A função “*Check Device Connected*” é desenvolvida em Python e utiliza dos recursos da biblioteca *nidaqmx* para verificar se o dispositivo está conectado, conforme mostrado na Figura 24. Essa função pode ser encontrada no arquivo de teste de componente do diretório *Scripts*.

Figura 24 – Função Python que verifica se o DAQ está conectado

```

1 def check_device_connected():
2     local = nidaqmx.system.System.local()
3     for device in local.devices:
4         print(f'Device Name: {device.name}, Product Type: {device.product_type}')
5         print('Input channels:', [chan.name for chan in device.ai_physical_chans
6         ])
7         print('Output channels:', [chan.name for chan in device.ao_physical_chans
8         ])
9         return True
10        return False

```

Fonte: Elaboração Própria (2023).

Através do apoio da biblioteca *nidaqmx*, é verificado se existe um dispositivo DAQ conectado e qual é o nome desse dispositivo. No caso do DAQ *USB-6212* o nome do dispositivo por padrão é *Dev1*. Conhecer o nome do dispositivo é um requisito na hora de realizar a leitura e escrita nos pinos do DAQ, pois serve como referência para indicar qual é o dispositivo DAQ que deve realizar essa ação.

Caso o DAQ não esteja conectado, a variável “local” não irá retornar parâmetro nenhum, causando a falha no teste. A abordagem de verificar se o DAQ está

conectado é importante para não causar um falso negativo. De modo a não atribuir a falha nos testes somente a um componente que não está no funcionamento correto. Além disso, a conexão do DAQ com o computador depende do usuário, portanto, é uma ação que está suscetível a erro humano. Nesse sentido, a abordagem de indicar que o DAQ deve estar conectado, também serve para lembrar ao usuário que ele deve realizar essa ação para que o teste seja executado.

3.5.1.4 Executando as ações de realizar o teste em um componente

A ação de realizar o teste do componente em si integra todas as aplicações do projeto. No arquivo de teste de componente do diretório de *Tests*, a palavra chave “Quando” indica que o teste do componente irá iniciar, conforme indicado na Figura 25.

Figura 25 – Iniciando o teste de um componente

```
1 Caso de teste para componentes eletronicos
2 [Documentation] Caso de Testes de CI's escolhidos pelo usuário
3 Dado que o DAQ esteja conectado a porta USB
4 Quando for realizado o teste do componente escolhido
5
```

Fonte: Elaboração Própria (2023).

A palavra chave “Quando for realizado o teste do componente escolhido” possui sua correspondência na seção *Keywords* do arquivo de diretório de *Resources* conforme demonstrado na Figura 26.

Figura 26 – Correspondência na seção de *Keywords* do diretório de *Resources* para iniciar o teste de um componente

```
1 *** Keywords ***
2 Quando for realizado o teste do componente escolhido
3     @{results}=    Make The Test For Component    ${esquemático}
4     Set Global Variable    @{results}
5
```

Fonte: Elaboração Própria (2023).

A função “*Make The Test For Component*” foi desenvolvida utilizando o *Python* e está disponível no arquivo de teste de componente no diretório de *Scripts*. A função precisa receber como argumento o esquemático de ligação. O parâmetro do esquemático já foi gerado pelo teste anteriormente, na seção de *Test Setup*.

A primeira etapa a ser realizada pela função é fazer a descompactação do esquemático de ligação entre o DAQ e o componente, que está em formato *Json*. Nesse momento, é obtido qual é o tipo do componente em teste, quais pinos do DAQ foram utilizados e quais são os pinos de entrada e saída do componente. Além

de descompactar o esquemático de ligação entre o DAQ e o componente em teste, também é retornado o *document* que está armazenado no banco de dados contendo os resultados esperados para o componente em teste. A Figura 27 representa a função que retorna os resultados esperados de um componente.

Figura 27 – Função Python que retorna os resultados esperados para um componente

```
1 def return_expected_result(componentName):  
2     collection=connect_to_database("ResultadosEsperados")  
3     cursor = collection.find({"componente":componentName})  
4     for record in cursor:  
5         return record  
6
```

Fonte: Elaboração Própria (2023).

No caso do 74HC08, será pesquisado qual é o *document* que corresponde ao componente *AND*. A *collection* de resultados esperados do *MongoDB* irá retornar os resultados para a porta lógica AND em formato *Json* conforme abaixo.

```
{  
  "_id": {  
    "$oid": "63741a88dd01631beca327fd"  
  },  
  "componente": "AND",  
  "case1": {  
    "entrada1": 1,  
    "entrada2": 1,  
    "saida": 1  
  },  
  "case2": {  
    "entrada1": 1,  
    "entrada2": 0,  
    "saida": 0  
  },  
  "case3": {  
    "entrada1": 0,  
    "entrada2": 1,  
    "saida": 0  
  },  
  "case4": {  
    "entrada1": 0,  
    "entrada2": 0,  
    "saida": 0  
  }  
}
```

Para realizar os testes funcionais de forma adequada, é preciso conhecer os dados de entrada do teste e também os resultados de saída esperados. Os valores que serão escritos em todos os pinos de entrada do componente são os valores indicados pela chaves “entrada1” e “entrada2” de cada um dos *cases* do *Json* de resultados esperados. Seguindo a mesma lógica, os valores de saída que serão comparados aos valores de saída lidos nos pinos de saída do componente são os valores indicados pela chave “saida” de cada um dos *cases* contidos no *Json* de resultados esperados.

Inicia-se o processo de escrita nos pinos de entrada do componente e a função “*write_digital_ports*” é responsável por realizar todas as ações que envolvem esse processo. Para obter quais são os pinos digitais do DAQ, é preciso conhecer o esquemático de ligação do dispositivo. O esquemático de pinos do DAQ está na *collection* DAQ do *MongoDB*. Assim, o esquemático apresentado pela Figura 4 foi transformado em um formato *Json*. Os valores dos pinos digitais, analógicos e de alimentação do DAQ *USB-6212* estão em chaves separadas dentro do objeto *Json* e o valor de cada chave é o número do pino. O esquemático do DAQ em *Json* está mostrado abaixo.

```
{
  "_id": {
    "$oid": "640fb50c73529e938de34028"
  },
  "nomeEquipamento": "Dev1",
  "pinosDigitais": {
    "p0_0": 44,
    "p0_1": 45,
    "p0_2": 46,
    "p0_3": 47,
    "DGND": 64,
    "p0_4": 50,
    "p0_5": 51,
    "p0_6": 52,
    "p0_7": 53,
    "p0_8": 55,
    "p0_9": 56,
    "p0_10": 57,
    "p0_11": 58,
    "p0_12": 60,
    "p0_13": 61,
    "p0_14": 62,
    "p0_15": 63,
    "p1_0": 1,
    "p1_1": 2,
    "p1_2": 3,
    "p1_3": 4,
    "p1_4": 6,
```

```

    "p1_5": 7,
    "p1_6": 8,
    "p1_7": 9,
    "p2_0": 33,
    "p2_1": 34,
    "p2_2": 35,
    "p2_3": 36,
    "p2_4": 38,
    "p2_5": 39,
    "p2_6": 40,
    "p2_7": 41
  },
  "pinosanalógicos": {
    "ao0": 12,
    "ao1": 13,
    "aoGND": 14,
    "ai0": 15,
    "ai8": 16,
    "ai1": 17,
    "ai9": 18,
    "ai2": 19,
    "ai10": 20,
    "ai3": 21,
    "ai11": 22,
    "aiSENSE": 23,
    "ai4": 24,
    "ai12": 25,
    "ai5": 26,
    "ai13": 27,
    "aiGND": 28,
    "ai6": 29,
    "ai14": 30,
    "ai7": 31,
    "ai15": 32
  },
  "alimentacao": {
    "+5Vpino1": 42,
    "+5Vpino2": 10
  }
}

```

A função `"return_allLinesDaq_digitalVersion"` contida dentro da função `"write_digital_ports"` retorna o nome e o número de todos os pinos digitais do DAQ através da chave de `"pinosDigitais"` presente no *Json* da *collection* DAQ do *MongoDB*.

Para escrever e ler nos pinos digitais do DAQ *USB-6212* é preciso conhecer o *line* de cada pino. O *line* pode ser definido como o número de identificação de um dos *ports* do DAQ, ou seja, são os pinos que o *port0*, *port1* e *port2* possuem. Exemplificando

para o *port0* o *line* é número que está depois do *underline* no *Json*, então, para o *port0_0*, o *line* é o número 0 e o número do pino do *port0_0* é o 44 no esquemático de ligação do DAQ.

A função “*get_DAQlines_to_write*” contida dentro da função “*write_digital_ports*” define quais serão os *lines* do DAQ que serão utilizados para realizar a escrita nos pinos do componente. Para descobrir qual *line* deverá ser utilizado, é verificado no *Json* de esquemático de ligação quais os números dos pinos do DAQ que foram utilizados para os pinos de entrada do componente e comparados com o número dos pinos contidos no *document* de esquemático do DAQ do *MongoDB*. Quando o número do pino no esquemático for igual ao número do pino contido no registro da base de dados, o *line* deverá ser utilizado.

O mesmo processo é feito para a função “*get_DAQlines_to_read*” que verifica quais serão os *lines* de leitura do DAQ. Então, é verificado quais foram os números dos pinos do DAQ utilizados para os pinos de saída do componente em teste e comparados com aqueles que ainda estão disponíveis após passarem pela função que seleciona os *lines* de escrita. A função que obtém os *lines* de leitura já foi projetada para fornecer suporte para outros *ports* além do *port0*. O código que define os *lines* está apresentado na Figura 28.

Figura 28 – Código responsável por definir os pinos do DAQ que vão realizar a escrita e leitura nos pinos do componente

```

1 def get_DAQlines_to_write(record,portasUtilizadas):
2     allLines = []
3     ports=[0,0,0]
4     for lines,linePinsNumber in record.items():
5         if str(linePinsNumber) in portasUtilizadas:
6             if 'p0' in lines:
7                 ports[0]=1
8                 allLines.append(lines[3])
9
10    return ports,allLines
11
12 def get_DAQlines_to_read(record, lines, portasUtilizadas):
13     readLines = []
14     ports = [0,0,0]
15     for liness,valoresLines in record.items():
16         portLine = liness
17         if portLine != 'DGND':
18             if str(valoresLines) in portasUtilizadas:
19                 lineNaoUtilizados = portLine.replace('p','').split('_')
20                 if lineNaoUtilizados[1] not in lines:
21                     readLines.append(lineNaoUtilizados[1])
22                 if 'p0' in portLine:
23                     ports[0]=1
24                 elif 'p1' in portLine:
25                     ports[1]=1
26                 elif 'p2' in portLine:
27                     ports[2]=1
28                 else:
29                     continue
30
31    return readLines
32
33 def write_digital_ports(portasUtilizadas, values, componentName,expected_results,
34     pinosSaidaUtilizados):
35     record = return_allLinesDaq_digitalVersion()
36     ports,allLines = get_DAQlines_to_write(record,portasUtilizadas)
37     lines = []
38     result = []
39     for alllines in allLines:
40         lines.append(alllines)
41     readLines = get_DAQlines_to_read(record,lines,pinosSaidaUtilizados)
42

```

Fonte: Elaboração Própria (2023).

No presente trabalho, apenas os pinos *port0* estão sendo utilizados para realizar a leitura e escrita nos pinos digitais. Desse modo, existem 16 *lines* disponíveis para realizar esse processo. No caso do componente utilizado como exemplo, o 74HC08, estão sendo utilizados os *lines* 0,1,2,3,4,5,6,7 do *port0* para servirem como pinos de escrita do DAQ e os *lines* 8,10,12 e 14 para servirem como pinos de leitura do *port0* do

DAQ.

A comparação entre os pinos do DAQ presentes no esquemático de ligação que foi entregue ao usuário e os pinos que existem na *database* é importante pois o arquivo de esquemático é um arquivo texto, que é passível de edição por parte do usuário. Em caso de ocorrer alguma edição acidental no arquivo, garante-se que não será utilizado nenhum pino inexistente do DAQ ou um pino que não seja um pino digital. Além disso, essa verificação permite que caso o usuário faça alguma alteração no esquemático para utilizar outros pinos dentro da faixa de pinos digitais do DAQ e contidos no *port0*, seja possível que o teste continue sendo realizado sem problemas, mesmo que os pinos do esquemático gerado para o usuário realizar as ligações tenham sido alterados.

A leitura e a escrita nos pinos de entrada e saída do componente não são realizadas ao mesmo tempo para todos os pinos. Nesse sentido, o processo de escrita ocorre primeiro nas duas primeiras entradas do componente e então a saída correspondente a essas entradas é lida e assim por diante com os demais pares de duas entradas para uma saída, até que todas as saídas do componente tenham tido seu valor lido e comparado com o resultado esperado para as entradas que foram aplicadas. Foi preciso criar uma correspondência entre os *lines* de escrita do DAQ e os *lines* de leitura, de modo a agrupar em pares os dois *lines* de escrita com o seu *line* de saída correspondente. Essa correspondência foi feita na função da Figura 29.

Figura 29 – Código responsável por criar uma correspondência entre os *lines* de escrita e leitura

```
1 def mix_read_and_write_lines(writeDaqLines,readDaqLines,componentName):
2     j=0
3     final = {}
4     for i in range(0,len(readDaqLines)):
5         item = readDaqLines[i]
6         final[item] = [writeDaqLines[j],writeDaqLines[j+1]]
7         j+=2
8
9     return final
10
```

Fonte: Elaboração Própria (2023).

Assim, cada *line* de leitura recebe seus dois *lines* de escrita. Para o 74HC08, o *line* 8, que corresponde ao pino 55 do DAQ, é o *line* de leitura e os *lines* 0 e 1, correspondentes aos pinos 44 e 45 do DAQ, serão seus *lines* de escrita. Quando for escrito algum dado nos *lines* 0 e 1, sabe-se que o valor que foi produzido por essas duas entradas deve ser lido pelo *line* 8. Essa lógica segue-se para os demais *lines*.

Para escrever e ler nos pinos do DAQ, utiliza-se os recursos da biblioteca *nidamx*. Todos os valores de entrada contidos nos *cases* do *Json* de resultados espera-

dos serão testados para todos os pinos de entrada do componente em teste e todos os valores de saída lidos serão comparados com os valores de saída para cada *case*. Os códigos responsáveis por realizar a escrita e leitura nos pinos do DAQ utilizando os recursos da *nidaqmx* estão demonstrados na Figura 30

Figura 30 – Código responsável por realizar a leitura e a escrita nos pinos do DAQ utilizando os recursos da biblioteca *nidaqmx*

```

1 def write_digital_ports(portasUtilizadas, values, componentName, expected_results,
2   pinosSaídaUtilizados):
3     with nidaqmx.Task() as task:
4         task.do_channels.add_do_chan(
5             "Dev1/port0/line" + lines[keyValue][0] + ":" + lines[keyValue][1],
6             line_grouping=LineGrouping.CHAN_PER_LINE)
7
8         try:
9             task.write([value1, value2], auto_start=True)
10            time.sleep(2)
11            results = read_digital_ports(keyValue, value1, value2, expected_results)
12            result.append(results)
13        except nidaqmx.DaqError as e:
14            print(e)
15
16 def read_digital_ports(line, value1, value2, expected_results):
17     allresults = []
18     expectedResultForThisCase = get_the_expected_results_for_the_correct_case(
19         value1, value2, expected_results.items())
20     with nidaqmx.Task() as task:
21         task.di_channels.add_di_chan("Dev1/port0/line" + line,
22             line_grouping=LineGrouping.CHAN_PER_LINE)
23
24         time.sleep(10)
25         data = task.read()
26         values = {'entrada1': value1, 'entrada2': value2, 'saida': data}
27         allresults2 = compare_the_values(expectedResultForThisCase, values)
28         allresults.append(allresults2)
29     return allresults

```

Fonte: Elaboração Própria (2023).

Assim, o fluxo para realizar a leitura e escrita ocorre da seguinte maneira. A variável *task* define quais serão os *lines* que vão ser utilizados para a escrita. Os valores da entrada 1 e entrada 2 de um dos *cases* do *Json* de resultados esperados são escritos nos *lines* definidos pela variável *task*. Após isso, existe um *delay* de 2 segundos antes de realizar a leitura. Esse *delay* foi utilizado para garantir que não existiriam problemas de sincronização entre a leitura e a escrita. Definido qual é o *line* de leitura que corresponde aos *lines* que foram utilizados como escrita, é realizado uma segunda pausa de 10 segundos e então o dado é lido e enviado para a função que compara os valores.

Exemplificando para o caso do 74HC08 e considerando o *case1* do *Json*

contendo os resultados esperados, os *lines* 0 e 1 foram definidos como *lines* de escrita anteriormente pelo código e os valores das entradas no *case1* são de nível lógico alto para as duas entradas. Será aplicado nível lógico alto para nos pinos 44 e 45 do DAQ, que correspondem aos *lines* 0 e 1 respectivamente e estão conectados aos pinos 1 e 2 do 74HC08 conforme o *Json* de esquemático de ligação. Então, após o *delay* de 10 segundos, será lido no pino 55 do DAQ, que correspondente ao *line* 8 e está conectado ao pino de número 3 do 74HC08. O valor lido será comparado ao valor da saída do *case1* do *Json*. Assim, espera-se que o valor lido seja nível lógico alto, o que confirma que para esse caso, o componente estava em pleno funcionamento.

Esse processo, é feito para os demais pinos do componente e é testado todos os casos possíveis em todos os pinos. Portanto, todos os pinos irão receber os dados de entrada do *case1* e todos os dados de saída serão lidos e comparados com o valor esperado para o *case1*. Somente depois de todos os pinos de entrada do componente terem recebido os valores de entrada contidos no *case1*, e o valor nos pinos de saída do componente que correspondente aos pinos de entrada em questão tenha sido lido, é que os outros *cases* contidos no *Json* serão testados, seguindo o mesmo fluxo.

Como dito anteriormente, a comparação dos resultados vai sendo feita a cada dado lido e esse valor é armazenado para posterior utilização, que é quando o teste irá verificar se os resultados obtidos correspondem aos resultados esperados. Caso ocorra alguma falha durante a escrita nos pinos do DAQ, o código irá retornar uma exceção e o teste irá falhar.

3.5.1.5 Comparando se os resultados obtidos foram iguais aos resultados esperados

A palavra chave “Então” do *Gherkin* presente no arquivo de teste de componente do diretório de *Tests* indica que os resultados de teste serão comparados, conforme a Figura 31.

Figura 31 – Comparando os resultados de testes

```
1 *** Test Cases ***
2
3 Caso de teste para componentes eletronicos
4   [Documentation]    Caso de Testes de CI's escolhidos pelo usuário
5   Dado que o DAQ esteja conectado a porta USB
6   Quando for realizado o teste do componente escolhido
7   Entao verifique que o componente esta funcionando
8
```

Fonte: Elaboração Própria (2023).

A correspondência dessa palavra chave está na seção de *keywords* do diretório de *Resources*, sendo demonstrado pela Figura 32.

Figura 32 – Correspondência da *keyword* de comparação de resultados no arquivo do diretório de *Resources*

```
1 *** Keywords ***
2
3 Entao verifique que o componente esta funcionando
4     ${endResult}=    Check Results    ${results}
5     Should Be True    ${endResult}
6     IF    ${endResult}
7         Log    Testes Funcionaram
8     END
9
```

Fonte: Elaboração Própria (2023).

A função “*Check Results*” é uma função desenvolvida em *Python* que verifica os resultados. Toda vez que um dado é lido pelo DAQ, o seu resultado é armazenado e comparado ao resultado esperado para a determinada entrada que foi aplicado. Se os valor do resultado lido e do resultado esperado são iguais, é armazenado o valor *True* em uma lista do *Python*, se não forem, é armazenado *False*, esse comparação vai sendo feita durante a escrita e leitura nos pinos digitais do DAQ.

A função “*Check Results*” apenas verifica se todos os valores que foram armazenados são verdadeiros e retorna *True* se forem. Essa verificação é feita somente após todos os casos de teste terem sido testados, ou seja, o processo de escrita e leitura já foi finalizado e o teste está pronto para verificar se o componente está funcionando ou não.

Assim, o *Robot framework* sabe que se o retorno da função que verifica os resultados for verdadeiro, o teste passou e retorna no *console* uma mensagem de testes funcionando. Do contrário, o testes receberam a condição de falha e nenhuma mensagem será exibida no console.

3.6 Implementação do casos de uso que envolvem cadastros de componentes e resultados esperados

Nessa seção, será tratado a implementação dos casos de teste de cadastro de um novo esquemático de ligação de componente e cadastro de resultados esperados. Atualmente, a base de dados no *MongoDB* já possui alguns esquemáticos de ligação de alguns modelos de portas lógicas e seus resultados esperados cadastrados.

Além disso, por se tratar de uma base de dados em *cloud*, é necessário que o usuário esteja com conexão a *internet* disponível para que os cenários de testes sejam executados com sucesso, já que eles precisam inserir registros no banco de dados.

A execução dos cenários de teste de cadastro de esquemático ou de cadas-

tro de resultados esperados não dependem do DAQ, ou seja, não é necessário que ele esteja conectado a porta *USB*.

3.6.1 Cadastrando o esquemático de ligação da porta lógica OR 74HC32

Para iniciar o processo, o usuário deve acessar o diretório de Cadastros da jiga de testes e preencher os valores das chaves que estão em branco no arquivo “esquematicoDoComponente.txt”, o qual está estruturado no formato JSON, conforme mostrado abaixo:

```
{
  "nome": "",
  "tipoDeComponente": {
    "portaLogica": ""
  },
  "entradas": {
    "numeroDoPino1": ,
    "numeroDoPino2": ,
    "numeroDoPino3": ,
    "numeroDoPino4": ,
    "numeroDoPino5": ,
    "numeroDoPino6": ,
    "numeroDoPino7": ,
    "numeroDoPino8":
  },
  "saidas": {
    "numeroDoPino1": ,
    "numeroDoPino2": ,
    "numeroDoPino3": ,
    "numeroDoPino4":
  },
  "alimentacao": {
    "vcc": ,
    "pinoAlimentacao": ""
  },
  "referencia": {
    "gnd": ,
    "pinoGND":
  }
}
```

Desse modo, o usuário deve preencher o campo “nome” com o modelo do componente desejado, no caso do exemplo dessa seção, o 74HC32.

A chave “tipoDeComponente” espera qual o tipo do componente que será utilizado, assim, o valor da chave “portaLogica” será preenchido com o valor “OR”, conforme abaixo:

```
"tipoDeComponente": {
  "portaLogica": "OR"
}
```

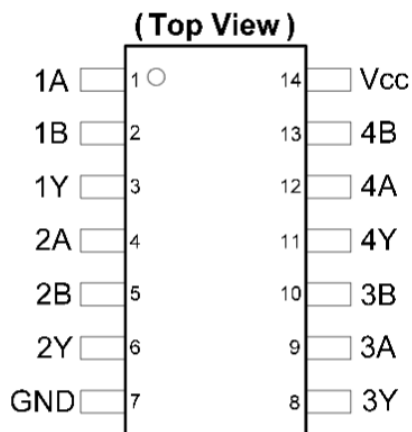
Caso seja adicionado outro tipo de componente digital além das portas lógicas, basta substituir a chave “portaLogica” pelo nome correspondente ao componente e o seu valor.

Não existe nenhum efeito contrário em realizar essa mudança dentro do dicionário “tipoDeComponente”, pois os dados do *MongoDB* são flexíveis, de modo que não é necessário que exista relação entre os diferentes *documents* cadastrados em uma mesma *collection*.

Após isso, o usuário deve cadastrar os números dos pinos de entrada do componente dentro do dicionário “entradas”. Para a chave “numeroDoPinoXX”, onde “XX” representa o número do pino em questão, o valor a ser preenchido é o número do pino correspondente a essa entrada.

O mesmo processo deve ser aplicado para o dicionário “saidas”. A Figura 33 apresenta a divisão de pinos do componente 74HC32.

Figura 33 – Top View do componente 74HC32



Fonte: Incorporated, Diodes (2013b).

Considerando a divisão de pinos do 74HC32, o documento *Json* ficaria com os seguintes valores para entradas e saídas:

```
"entradas": {
  "numeroDoPino1": 1,
  "numeroDoPino2": 2,
  "numeroDoPino3": 4,
  "numeroDoPino4": 5,
  "numeroDoPino5": 9,
  "numeroDoPino6": 10,
```

```

    "numeroDoPino7": 12,
    "numeroDoPino8": 13
  },
  "saidas": {
    "numeroDoPino1": 3,
    "numeroDoPino2": 6,
    "numeroDoPino3": 8,
    "numeroDoPino4": 11
  }
}

```

Caso o componente a ser cadastrado tenha um número maior de pinos do que os disponíveis no arquivo para preencher, é necessário apenas adicionar os pinos faltantes seguindo o padrão “numeroDoPinoXX”, onde “XX” representa o número do pino em questão.

Por fim, é necessário cadastrar o valor de alimentação e o número do pino correspondente a alimentação, e realizar o mesmo para o pino de referência, conforme abaixo, entretanto, o *json* pode ser estruturado conforme a especificação do componente a ser cadastrado.

```

1  "alimentacao": {
2    "vcc": 5,
3    "pinoAlimentacao": "14"
4  },
5  "referencia": {
6    "gnd": 0,
7    "pinoGND": "7"
8  }

```

Com o arquivo “esquematicoDoComponente.txt” preenchido e salvo, o caso de teste para cadastrar um novo esquemático de ligação no diretório *Tests* deve ser executado. O arquivo “Cadastros.robot” contém o cenário de teste de cadastro de componente, que possui a seguinte descrição de teste da Figura 34,

Figura 34 – Cenário de teste para cadastrar o esquemático de ligação de um novo componente

```

1  *** Test Cases ***
2
3  Caso de teste para adicionar esquematico de um componente
4    [Documentation]    Caso de Testes para adicionar o esquemático de um componente
5    Dado que o arquivo esteja no diretório    ./Cadastros/esquematicoDoComponente.
        txt
6    Quando o arquivo estiver preenchido com os dados do esquematico do componente
7    Então adicione na base de dados
8
9

```

O cenário descrito em *Gherkin* indica que dado que o arquivo “esquemático-DoComponente.txt” exista no diretório Cadastros, e esteja preenchido, então o *Json* será adicionado na base de dados.

Para o cenário de teste de cadastro de um componente, são utilizadas bibliotecas nativas do *Robot* para validar que o arquivo está no diretório correto e está preenchido.

Esse caso de uso não precisa obrigatoriamente ser executado junto do cenário de teste de cadastro de resultados esperados. É possível utilizar um resultado esperado já cadastrado previamente para realizar os testes do novo componente cadastrado, desde que o tipo do componente indicado no objeto *json* seja o mesmo.

Para executar o cenário de teste de cadastro de um componente separado do caso de teste de inserção dos resultados esperados, pode-se executar no terminal o comando abaixo ou executar pela interface utilizando o *vscode* caso ele seja o editor escolhido pelo usuário.

```
robot --test "Caso de teste para adicionar esquemático de um componente"  
Cadastros.robot
```

Alternativamente, os casos de testes podem ser executados juntos, utilizando o comando a seguir no terminal. Assim, os casos de testes serão executados um após o outro conforme estão dispostos no arquivo de testes de cadastros do diretório de *Tests*.

```
robot Cadastros.robot
```

Se o cenário de teste de cadastro de um componente passou, o arquivo contendo o objeto *Json* que foi preenchido pelo usuário será adicionado na *collection* “ComponentesDigitais” do *MongoDB*.

3.6.2 Cadastrando os resultados esperados para a porta lógica 74HC32

Os resultados esperados são um dos parâmetros mais importantes de teste. Os valores contidos nos registros da *collection* de “ResultadosEsperados” do *MongoDB* são comparados aos valores lidos nas saídas do componente. Além disso, os valores de sinal de entrada indicados dentro dos *documents* contidos na *collection* de “ResultadosEsperados” são utilizados para definir qual será o sinal de entrada aplicado nos pinos de entrada do componente.

Os passos para cadastrar os resultados esperados são os mesmo dos realizados para cadastrar um novo esquemático de ligação. No caso das portas lógicas, os resultados esperados são o análogo as tabelas da verdade.

No diretório de Cadastros existe o arquivo “resultadosEsperadosDoComponente.txt” em formato *Json*, que segue a estrutura abaixo:

```
{
  "componente": "",
  "case1": {
    "entrada1": ,
    "entrada2": ,
    "saida":
  },
  "case2": {
    "entrada1": ,
    "entrada2": ,
    "saida":
  },
  "case3": {
    "entrada1": ,
    "entrada2": ,
    "saida":
  },
  "case4": {
    "entrada1": ,
    "entrada2": ,
    "saida":
  }
}
```

Desse modo, deve-se preencher os espaços vazios das chaves do *Json* com as informações do componente que terá seus resultados esperados cadastrados.

A chave “componente” espera que se cadastre o tipo do componente, por exemplo, no caso da porta lógica 74HC32, o valor esperado é *OR*.

O *Json* está ligado ao tipo do componente e não ao nome do modelo do componente. Essa estrutura permite deixar os registros de resultados esperados mais flexíveis. O mesmo registro de resultado esperado pode ser utilizado para componentes de diferentes fabricantes, desde que eles tenham o mesmo propósito. Por exemplo, as portas lógicas 74HC32 e SN74LS32, poderiam compartilhar o mesmo registro de resultado esperado, pois ambas são portas lógicas OR.

Após preencher o tipo do componente, é necessário preencher todos os sinais de entrada que serão aplicados nos pinos de entrada do componente e qual o valor de saída esperado para as entradas que foram aplicadas. Para o componente 74HC32, os parâmetros de casos de testes são os valores correspondentes a sua tabela da verdade, conforme abaixo:

```
{
```

```
"componente": "OR",
"case1": {
  "entrada1": 0,
  "entrada2": 0,
  "saida": 0
},
"case2": {
  "entrada1": 0,
  "entrada2": 1,
  "saida": 1
},
"case3": {
  "entrada1": 1,
  "entrada2": 0,
  "saida": 1
},
"case4": {
  "entrada1": 1,
  "entrada2": 1,
  "saida": 1
}
}
```

O “case1” indica que deve ser aplicado os sinais de nível lógico baixo em ambas as entradas do componente e o resultado esperado será também nível lógico baixo. Assim, os demais *cases* também devem ser preenchidos com as requeridas entradas e saídas.

O objeto *Json* em questão também permite a expansão ou mudanças, de modo que o usuário pode adicionar mais casos de testes ou então diminuir o número de casos. Durante o teste, todos os casos de testes possíveis indicados pelos dicionários *cases* serão testados.

Os valores contidos nas chaves “entrada1” e “entrada2” dos dicionários “case” serão aplicados nos pinos de entrada do componente em teste. Os valores lidos na saída dos pinos do componente serão comparados com os valores contidos na chave de “saida” de cada *case* do objeto *Json*.

Para realizar a inserção do *Json* de resultados esperados de um componente, é preciso preencher todo o arquivo “resultadosEsperadosDoComponente.txt” e salva-lo.

Após isso, o caso de teste de inserção de resultados esperados no arquivo “Cadastros.robot” do diretório de *Tests* está pronto para ser executado. O cenário de teste é apresentado na Figura 35.

Figura 35 – Cenário de teste para cadastrar os resultados esperados de um componente

```
1 *** Test Cases ***
2
3 Caso de teste para adicionar os resultados esperados para um componente
4 [Documentation]    Caso de Testes para adicionar os resultados esperados
5 Dado que o arquivo esteja no diretório    ./Cadastros/
6 resultadosEsperadosDoComponente.txt
7 Quando o arquivo estiver preenchido com os dados dos resultados do componente
8 Então adicione na base de dados
```

Fonte: Elaboração Própria (2023)

Esse cenário de teste segue a mesma lógica do cenário de cadastro de um componente. Primeiramente, é verificado se o arquivo está no diretório correto. Quando o arquivo de resultados esperados estiver preenchido, o *Json* será adicionado a base de dados.

O cenário de teste de cadastro de resultados esperados pode ser executado separadamente do cenário de cadastro de um componente, utilizando o comando a seguir no terminal ou executando via editor de códigos.

```
robot --test "Caso de teste para adicionar os resultados esperados
para um componente" Cadastros.robot
```

Alternativamente, também é possível executar esse caso de teste junto do cenário de teste de cadastro de esquemático de ligação de um componente, utilizando o seguinte comando:

```
robot Cadastros.robot
```

Após o cenário de teste ter sido executado e caso ele tenha sido aprovado, os resultados esperados preenchidos pelo usuário são inseridos na *collection* de “ResultadosEsperados” do *MongoDB*.

4 RESULTADOS

Essa Capítulo tem como objetivo realizar a apresentação dos resultados do desenvolvimento da jiga de testes automatizados, apresentando o modo como a geração e aquisição de sinais foi validado. Além disso, também será apresentado informações sobre o relatório final de testes gerado.

4.1 Validação do DAQ e simulação de problemas

Para validar a aquisição e geração de sinais realizados pelo DAQ *USB-6212*, foi necessário desenvolver um *script* separado da jiga de testes. Essa abordagem foi necessária devido às várias interações entre diferentes tecnologias que existem na jiga, como Python e Robot, além da necessidade de comunicação com o banco de dados. Ademais, na jiga de testes, a geração e aquisição de sinais não ocorrem simultaneamente em todos os pinos. Em vez disso, a geração de sinal é realizada em conjuntos específicos de pinos. Devido a isso, há momentos em que a geração de sinal não é realizada em um pino específico, pois está sendo executada em outro conjunto de pinos. Isso resulta em um *delay* até que o próximo sinal seja gerado no pino em questão, dificultando a captura do sinal. Por conta disso, a abordagem utilizando o *script* adicional facilita a validação da geração e aquisição de sinais. O *script* desenvolvido pode ser visto na Figura 36.

Figura 36 – Script desenvolvido para validar a geração e aquisição de sinais da jiga de testes

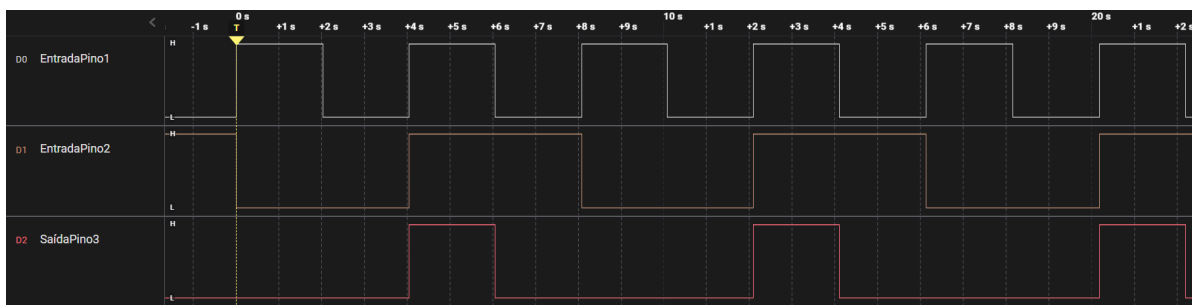
```
1 import time
2 import nidaqmx
3
4 with nidaqmx.Task() as task, nidaqmx.Task() as read_task:
5     task.do_channels.add_do_chan("Dev1/port0/line0:1", line_grouping=nidaqmx.
6         constants.LineGrouping.CHAN_PER_LINE)
7     read_task.di_channels.add_di_chan("Dev1/port0/line8", line_grouping=nidaqmx.
8         constants.LineGrouping.CHAN_PER_LINE)
9
10    while(True):
11        task.stop()
12        task.write([True, True], auto_start=True)
13        time.sleep(2)
14        data = read_task.read()
15        task.write([False, True], auto_start=True)
16        time.sleep(2)
17        data = read_task.read()
18        task.write([True, False], auto_start=True)
19        time.sleep(2)
20        data = read_task.read()
21        task.write([False, False], auto_start=True)
22        time.sleep(2)
23        data = read_task.read()
```

Fonte: Elaboração Própria (2023).

Para o *script* da Figura 36, os pinos *p0_0* e *p0_1* realizam a escrita de sinal nos pinos de entrada do componente. A leitura do sinal de saída no pino do componente foi realizada pelo pino *p0_8* do DAQ. Esse conjunto de pinos do DAQ de entrada e saída também fazem parte do conjunto de pinos que o teste apresentado para o 74HC08 da Figura 14 utiliza. O *script* ficará escrevendo todos os casos de testes apresentados até que sua execução seja interrompida.

Para validar os sinais, foi utilizado o analisador lógico *Logic Analyzer* de 8 canais com conexão *USB*. A taxa de amostragem do DAQ *USB-6212* é de 400k/S por segundo. Portanto, a taxa de amostragem utilizada para o *logic analyzer* foi de 1M/s. O componente utilizado para validar os sinais foi a porta lógica *AND 74HC08*.

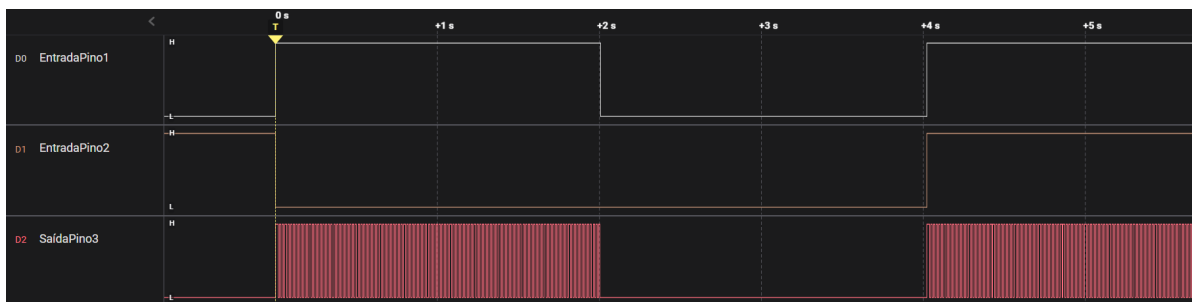
A Figura 37 apresenta a execução do *script* da Figura 36 para o componente 74HC08. A “EntradaPino1” é o valor de sinal da entrada do pino número 1 da *AND*, sendo esse o sinal de saída do pino *p0_0* do DAQ. A “EntradaPino2” representa o valor de sinal do pino de entrada número 2 do 74HC08, sendo esse o sinal de saída do pino *p0_1* do DAQ. E o valor de saída representado por “SaídaPino3” foi obtido do pino número 3 do 74HC08 e tem seu valor lido pelo pino *p0_8* do DAQ.

Figura 37 – Sinais de entrada e saída do componente 74HC08 em pleno funcionamento

Fonte: Elaboração Própria(2023)

Analisando os sinais da Figura 37 é possível perceber que a *AND* está em pleno funcionamento, pois o sinal de saída está se comportando conforme a tabela da verdade para o determinado sinal de entrada que foi aplicado. Por exemplo, em 4s ambas as entradas se encontravam em nível lógico alto, o sinal de saída obtido também foi de nível lógico alto. Em 6s, temos a “EntradaPino1” em nível lógico baixo e a “EntradaPino2” em alto, resultando em um sinal de saída de nível lógico baixo. Em 8s a “EntradaPino1” está em nível lógico alto e a “EntradaPino2” em baixo, obtendo nível lógico baixo na saída. Por fim, em 10s as duas entradas estão em nível lógico baixo, resultado também em nível lógico baixo na saída. Todos esses resultados estão conforme o esperado para a *AND*.

Para simular que o componente não está funcionando, foi utilizado duas abordagens. A primeira delas foi a de retirar o pino de entrada de sinal de uma das entradas do componente. A Figura 38 representa os resultados dessa simulação. Nesse momento, “EntradaPino1” está conectada ao pino número 1 do 74HC08, que é o sinal de saída do pino *p0_0* do DAQ. A “EntradaPino2” é o valor de saída do pino *p0_1* do DAQ, entretanto, o pino 2 do componente se encontra em aberto, sem nenhuma entrada de sinal. E o “SaídaPino3” é o valor de saída no pino de número 3 do 74HC08, sendo o sinal de entrada do pino *p0_8* do DAQ.

Figura 38 – Simulando a ocorrência da desconexão do sinal de uma das entradas do 74HC08

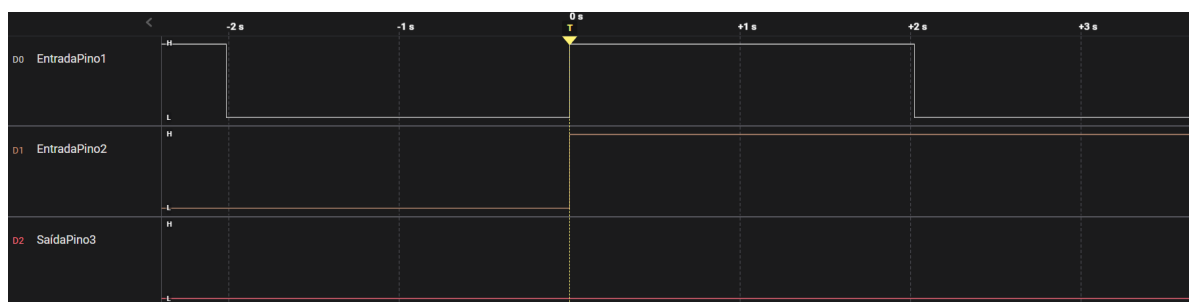
Fonte: Elaboração Própria(2023)

Nesse momento, é possível observar que a *AND* não estaria se comportando conforme o esperado. De 0 a 2s, a “EntradaPino1” está em nível lógico alto e a

“EntradaPino2” em baixo. O resultado esperado para essas combinações de sinal de entrada seria de nível lógico baixo, entretanto, o valor obtido na “SaídaPino3” é de nível lógico alto. Nesse momento, com um dos terminais de entrada do 74HC08 aberto, o terminal se comporta como uma antena. Dessa forma, o terminal aberto está suscetível a ser afetado por ruídos externos, o que influencia no estado de saída da porta lógica. Assim, a saída da porta lógica tem oscilações muito rápidas de sinal, resultado em flutuações e instabilidade no sinal de saída. Quando esse mesmo teste for realizado na jiga de testes, espera-se obter o *status* de reprovado no teste.

Para validar outro comportamento não esperado dentro dos requisitos funcionais do 74HC08, foi simulado um comportamento como se uma das entradas estivesse sempre em nível lógico baixo. Portanto, a Figura 39 representa esse caso teste. A “EntradaPino1” está conectada ao pino *p0_0* do DAQ, a “EntradaPino2” está conectada ao *p0_1* do DAQ. Entretanto, o pino de número 2 do 74HC08 está conectado a nível lógico baixo a todo momento. A “SaídaPino3” é o sinal de saída lido pelo pino *p0_8* do DAQ.

Figura 39 – Simulando a ocorrência de nível lógico baixo em um dos terminal de entrada do 74HC08



Fonte: Elaboração Própria(2023)

Conforme a Figura 39, é possível perceber que quando ambas as entradas estão em nível lógico alto, a saída obtida é de nível lógico baixo, o que não corresponde aos resultados esperados para a AND nesse caso de teste. Ademais, percebe-se que o sinal de saída está sempre em nível lógico baixo, independente da entrada de sinal aplicada.

Com base nos resultados obtidos pelo *logic analyzer*, é possível confirmar que a geração e aquisição de sinais utilizando o DAQ está se comportando da maneira correta. Dessa forma, as demais funções da jiga de testes podem ser validas através da depuração de código.

4.2 Relatórios dos testes

O relatório de teste é gerado após a execução dos arquivos de teste do diretório de *Tests*. O *Robot framework* gera por padrão três relatórios, o de *Log*, o

Output e o *Report*. Esses relatórios podem ser encontrados no diretório de *Results*. Entretanto, a interface gráfica dos relatórios não é tão amigável, conforme pode ser visto no relatório de *Log* da Figura 40 que apresenta a porta lógica *AND* em pleno funcionamento.

Figura 40 – Relatório de *Log* gerado por padrão no teste da porta lógica *AND*

JigadeTestes Log

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	1	1	0	0	00:02:45	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
No Tags						

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
JigadeTestes	1	1	0	0	00:02:46	
JigadeTestes . Tests	1	1	0	0	00:02:46	
JigadeTestes . Tests . TestedeComponentes	1	1	0	0	00:02:46	

Test Execution Log

SUITE JigadeTestes

Full Name: JigadeTestes
Source: C:\Users\carol\Desktop\JigadeTestes
Start / End / Elapsed: 20230602 20:57:19.866 / 20230602 21:00:06.300 / 00:02:46.434
Status: 1 test total, 1 passed, 0 failed, 0 skipped

SUITE Tests

Full Name: JigadeTestes.Tests
Source: C:\Users\carol\Desktop\JigadeTestes\Tests
Start / End / Elapsed: 20230602 20:57:19.900 / 20230602 21:00:06.291 / 00:02:46.391
Status: 1 test total, 1 passed, 0 failed, 0 skipped

SUITE TestedeComponentes

Full Name: JigadeTestes.Tests.TestedeComponentes
Source: C:\Users\carol\Desktop\JigadeTestes\Tests\TestedeComponentes.robot
Start / End / Elapsed: 20230602 20:57:19.905 / 20230602 21:00:06.285 / 00:02:46.380
Status: 1 test total, 1 passed, 0 failed, 0 skipped

TEST Caso de teste para componentes eletronicos

Full Name: JigadeTestes.Tests.TestedeComponentes.Caso de teste para componentes eletronicos
Documentation: Caso de Testes de CI's escolhidos pelo usuário
Start / End / Elapsed: 20230602 20:57:21.587 / 20230602 21:00:06.279 / 00:02:44.692
Status: PASS

SETUP Obtendo o esquemático de ligação entre o DAQ e o componente a ser testado

KEYWORD componentesDigitais. Dado que o DAQ esteja conectado a porta USB

KEYWORD componentesDigitais. Quando for realizado o teste do componente escolhido

KEYWORD componentesDigitais. Entao verifique que o componente esta funcionando

Generated
20230602 21:00:06 UTC-03:00
46 days 22 hours ago

Fonte: Elaboração Própria(2023)

Assim, quando o teste de um componente é realizado, os relatórios são customizados utilizando a biblioteca *rfmetrics*. Para os testes que envolvem cadastros, o relatório não é customizado automaticamente ao final da execução dos testes de cadastros. Portanto, caso o usuário deseje verificar o relatório dos testes de cadastro, deve visualizar os relatórios presentes no diretório de *Results*.

Desse modo, o objetivo de customizar os relatórios padrão do *Robot* é oferecer uma interface gráfica mais amigável, de modo que o relatório pode ser útil tanto ao time de gestão quanto ao time de desenvolvimento, trazendo melhores informações das métricas de teste. Nesse sentido, ao final da execução do caso de teste de um componente, o relatório customizado é gerado automaticamente desde que o usuário

tenha executado o seguinte comando no terminal para executar o caso de teste de um componente.

```
testeDeComponente.bat
```

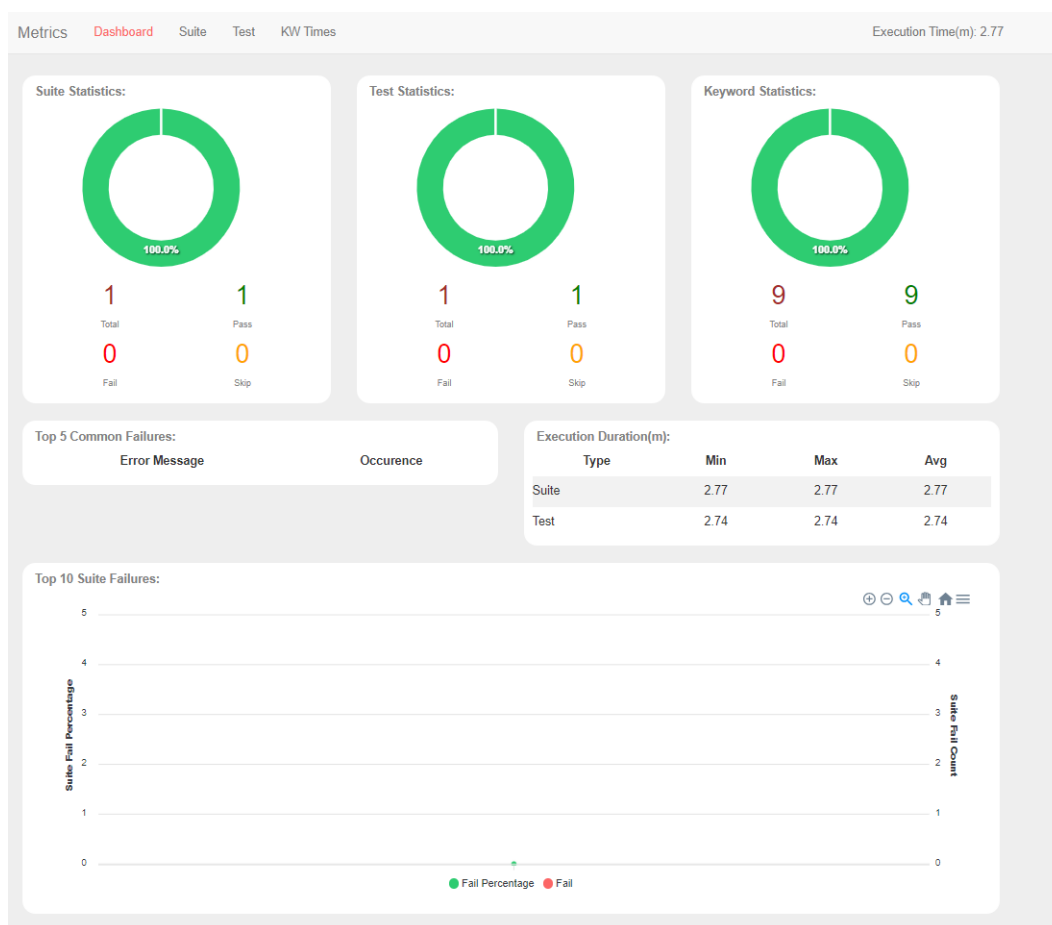
Alternativamente, também é possível gerar o relatório de testes customizado sem executar o arquivo de teste de componente do diretório *Tests* com o comando a seguir. Caso essa seja a opção utilizada para gerar o relatório de teste customizado, o relatório gerado será feito a partir dos relatórios da última execução do *Robot*, ou seja, o relatório customizado será feito a partir dos relatórios existentes no diretório de *Results*.

```
robotmetrics --inputpath ./Results/ --output output.xml
```

Nesse contexto, caso os últimos relatórios de testes presentes no diretório de *Results* sejam relatórios gerados a partir dos testes de cadastro, o relatório gerado pela *rfmetrics* quando executado o comando acima, será em relação as métricas do teste de cadastro e não dos testes de um componente. Portanto, é recomendável que se o usuário deseja obter as métricas do teste de um componente, ele execute no terminal o comando que gera o relatório customizado automaticamente.

O relatório gerado pelo teste da porta lógica *AND 74HC08* é apresentado na Figura 41. A geração e aquisição de sinais foi feita considerando o componente em pleno funcionamento, conforme a representação dos sinais que pode ser vista na Figura 37. O relatório gerado recebe o nome de “metrics-XX.html”, onde XX é um *range* de números aleatórios gerados pela biblioteca *rfmetrics* e pode ser aberto utilizando o *browser*.

Figura 41 – *Dashboard* do relatório de testes para o componente 74HC08 em pleno funcionamento



Fonte: Elaboração Própria(2023)

Nesse sentido, após executar o caso de teste de um componente para o 74HC08, o resultado produzido pelos testes automatizados foi de teste aprovado, o que está de acordo com o analisado pela Figura 37. O *dashboard* da Figura 41 do relatório customizado indica que os testes passaram.

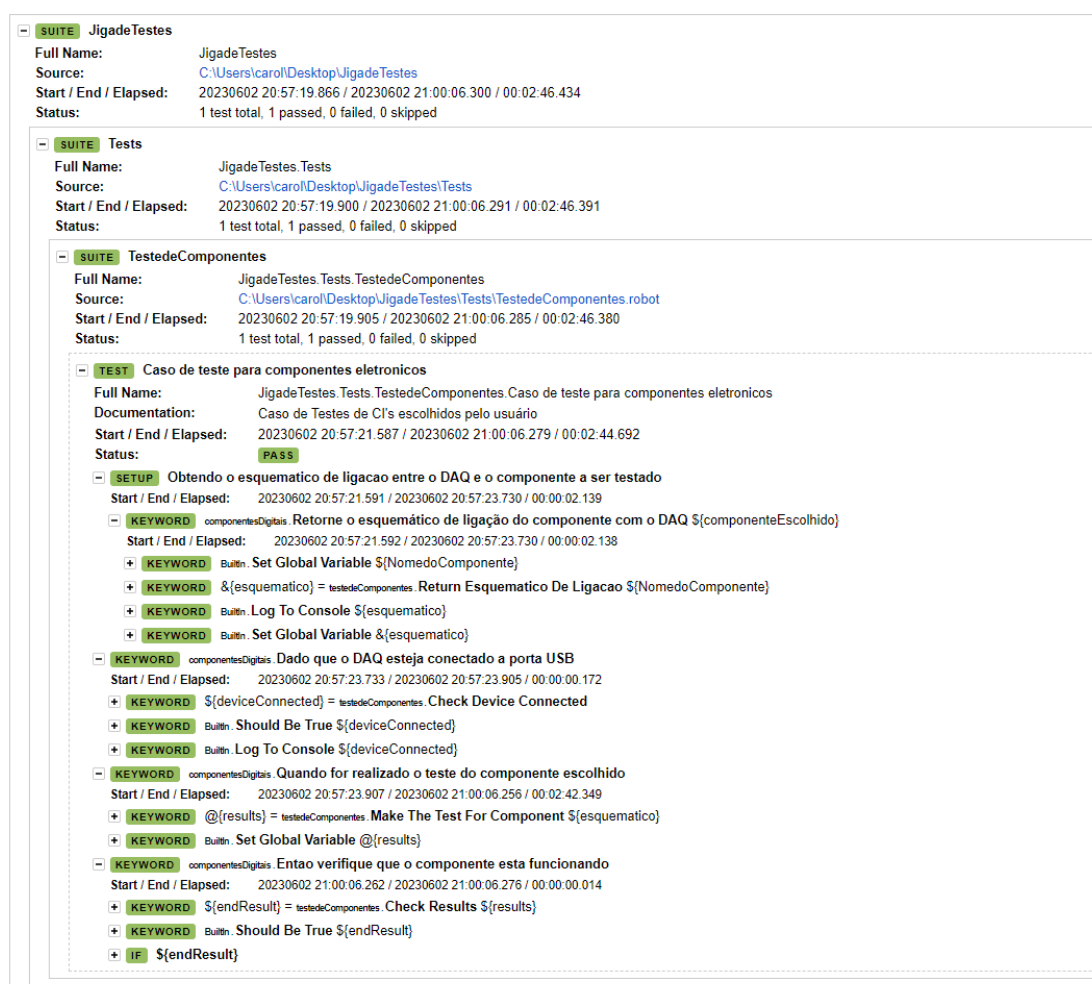
O gráfico de *Suite Statistics* indica que todos os testes presentes no arquivo de testes de componentes do diretório de *Tests* foram aprovados. O gráfico de *Test Statistics* avalia individualmente cada caso de teste presente em um arquivo de teste, no arquivo de teste de componente do diretório *Tests*, só existe um caso de teste. Portanto, o gráfico de *Test Statistics* se comporta de maneira similar ao de *Suite Statistics* indicando que todos os testes foram aprovados.

Para o gráfico de *Keyword Statistics* existem todas as *keywords* do *Robot* framework que foram executadas. Assim, é indicado que no arquivo de teste de componente foram executadas 9 *keywords* e todas elas foram aprovadas no caso do 74HC08. Ademais, o relatório customizado também contém o tempo médio de execução do teste. Para o 74HC08 analisado, o tempo médio total de execução do teste foi de 2 minutos e 77 segundos. Esse tempo pode ser otimizado com modificações de performance

no código. Ademais, não existem informações nos campos de falhas, pois o *status* de teste foi de aprovado.

Além disso, o relatório também fornece informações mais detalhas de cada um dos gráficos presentes no *dashboard*, contando com as abas *Suite*, *Test* e *KW Times*. As abas *Suite* e *Test* redirecionam ao relatórios padrão gerados pelo *Robot Framework*, proporcionando uma análise mais técnica dos resultados de teste. Nessas abas, é possível obter a informação das mensagens de *logs* mostradas no console e também os valores contidos nas variáveis do testes. A Figura 42 apresenta uma parte da visão mais detalhada do relatório de teste do 74HC08 para a aba de *Test Statistics*.

Figura 42 – Visão mais detalhada da aba de *Test* presente no relatório customizado do componente 74HC08



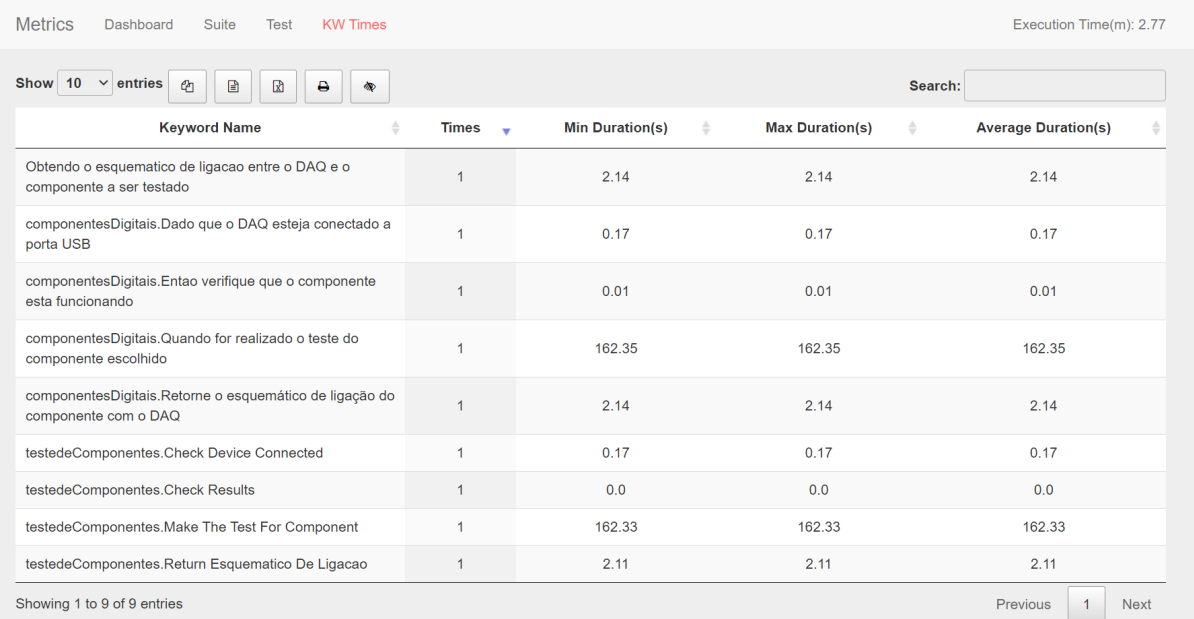
Fonte: Elaboração Própria(2023)

Desse modo, a análise dos resultados da aba *Test* do relatório permite obter todas as variáveis de teste, como por exemplo, obter qual foi o esquemático de ligação gerado, qual foram os valores obtidos na lista de resultados e também o tempo que demorou para que essas variáveis individualmente tenham os seus valores atribuídos. Portanto, a seção de *Test* é uma seção mais voltada ao time de automação, pois

consegue fornecer as métricas de tempo detalhadas de cada passo do teste e também fornece suporte para que seja feito a depuração do teste. As demais funcionalidades da jiga de testes automatizados foram validadas através dessa seção do relatório.

Para a aba de *KW Times* do relatório, existe as métricas de tempo para cada *keyword* que foi executada. Essa aba é uma descrição mais detalhada do gráfico de *Keyword Statistics*. A Figura 43 representa a aba *KW Times* do teste executado para o 74HC08.

Figura 43 – Visão mais detalhada da aba de *Keyword Statistics* presente no relatório customizado do componente 74HC08



Keyword Name	Times	Min Duration(s)	Max Duration(s)	Average Duration(s)
Obtendo o esquemático de ligação entre o DAQ e o componente a ser testado	1	2.14	2.14	2.14
componentesDigitais.Dado que o DAQ esteja conectado a porta USB	1	0.17	0.17	0.17
componentesDigitais.Entao verifique que o componente esta funcionando	1	0.01	0.01	0.01
componentesDigitais.Quando for realizado o teste do componente escolhido	1	162.35	162.35	162.35
componentesDigitais.Retorne o esquemático de ligação do componente com o DAQ	1	2.14	2.14	2.14
testedeComponentes.Check Device Connected	1	0.17	0.17	0.17
testedeComponentes.Check Results	1	0.0	0.0	0.0
testedeComponentes.Make The Test For Component	1	162.33	162.33	162.33
testedeComponentes.Return Esquemático De Ligacao	1	2.11	2.11	2.11

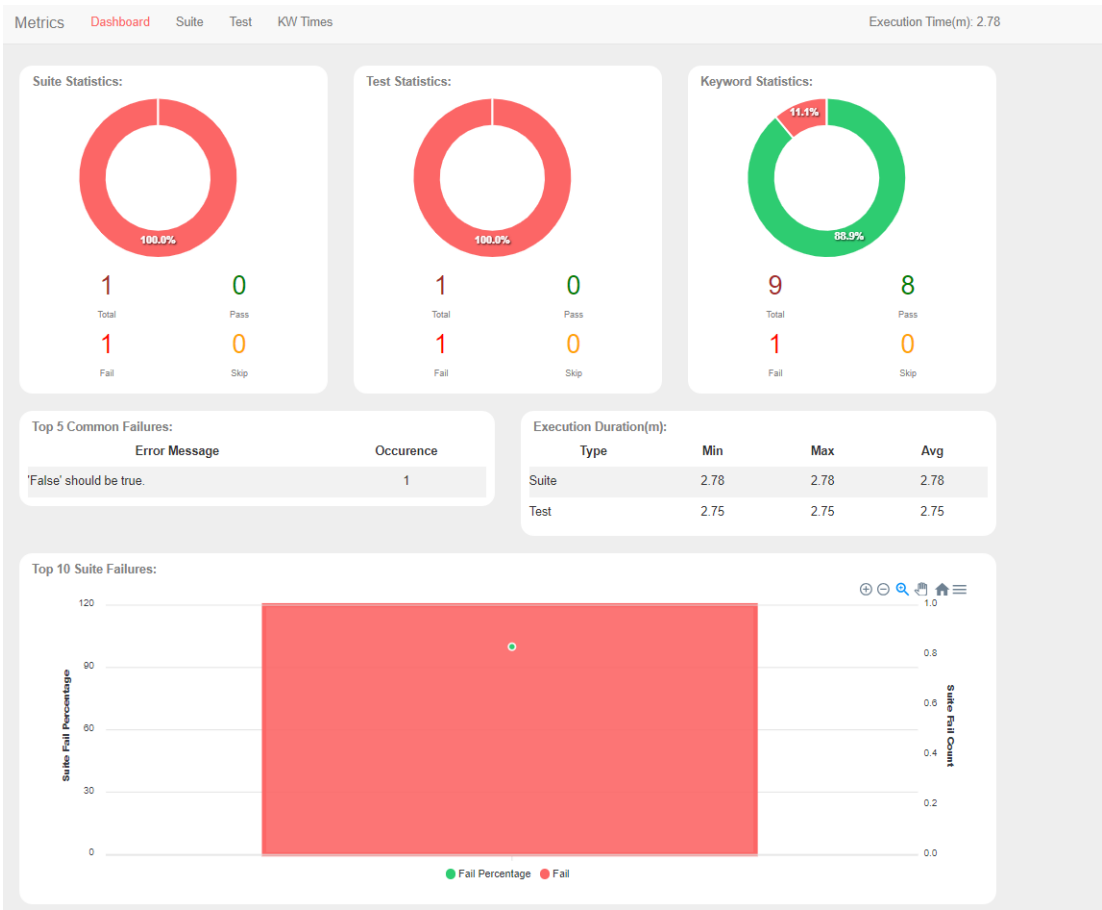
Fonte: Elaboração Própria(2023)

Assim, a Figura 43 apresenta uma descrição detalhada contendo quantas vezes a palavra chave foi executada, qual o tempo mínimo e máximo que ela ficou sendo executada e qual a média de tempo da execução da *keyword*. Nesse sentido, analisando a Figura 43 a *keyword* que mais demorou para ser executada foi a da palavra chave “Quando” do *Gherkin*, que realiza efetivamente o teste do componente, demorando um total de 2 minutos e 71 segundos para ser executada. Após isso, a *keyword* mais demorada é a que obtém o esquemático de ligação, demorando pouco mais de 2 segundos para ser executada. A métrica de tempo total do teste pode ser vista pela variável *Execution Time(m)* da Figura 43, indicando que o tempo total para a realização do teste foi de 2 minutos e 77 segundos.

Nesse sentido, também foram realizados testes com o 74HC08 simulando um componente em falha. A simulação utilizada foi a mesma da Figura 38, em que um dos terminais de entrada do componente está em aberto. Sabendo que o resultado produzido pela jiga deve ser o mesmo do resultado na Figura 38, o arquivo de teste de

componente foi executado, gerando o relatório customizado com o seguinte *dashboard* que pode ser visualizado na Figura 44.

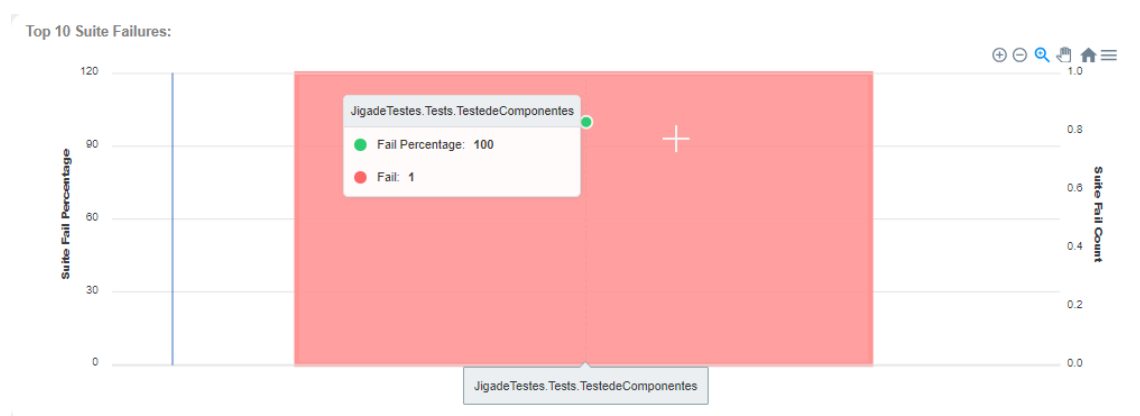
Figura 44 – Formas de apresentar o resultado para o 74HC08 em falha



Fonte: Elaboração Própria(2023)

Assim, o relatório indica claramente que o componente está em falha. Os gráficos de *Suite Statistics* e *Test Statistics* indicam que o teste está em falha.

Além disso, quando ocorre uma falha, a seção *Top 5 Common Failures* da Figura 44 indica qual foi a mensagem de erro mostrada no *console* e qual a ocorrência dela quando houve a falha no teste. A seção *Top 10 Suite Failures* apresenta um gráfico interativo para a análise de todos os testes que tiveram falha, conforme pode ser visto na Figura 45.

Figura 45 – Gráfico interativo que permite analisar os testes em falha

Fonte: Elaboração Própria(2023)

Desse modo, o gráfico apresentado pela Figura 45 é um gráfico interativo, em que o usuário através do cursor do *mouse* consegue interagir com o gráfico e obter quantos foram as *suites* de teste que falharam, ou seja, quais foram os arquivos de teste que falharam. No caso do 74HC08 analisado, existe a falha em apenas um arquivo. Além disso, também se obtém a porcentagem de falha, que é de 100% no caso do 74HC08 analisado.

Analisando o gráfico de *Keywords Statistics*, é indicado que oito das nove *keywords* passaram, sendo que apenas uma das palavras chave apresentou falha. Entretanto, é importante ressaltar que para que o teste seja aprovado, todas as *keywords* devem passar. A porcentagem das *keywords* que passaram foi de 88.9% e a *keyword* reprovada representa 11.1% do total de estatísticas das *keywords* de teste. A *keyword* que foi apresentada falha foi a de comparação dos resultados. Todas as possibilidades de sinal foram aplicadas aos terminais de entrada do componente e os resultados de saída foram obtidos. Assim, mesmo que um componente esteja em falha, todos os casos de testes são executados. Apresentar falha na *keyword* de comparação de resultados fornece para o usuário uma visão clara de onde está o problema, indicando que os resultados não estão conforme o esperado, mas que o teste em si do componente não obteve problemas para ser realizado, mesmo que um dos terminais esteja desconectado. O tempo total médio para a execução do teste em falha foi de 2.78 segundos.

Dessa forma, é perceptível que a customização dos relatórios padrão do *Robot framework* agrega valor ao relatório de testes disponibilizado ao usuário. O relatório customizado oferece maior clareza para a visualização, análise e interpretação dos resultados dos testes. Além disso, o relatório tem o foco de apresentar de forma clara os principais indicadores de teste, que são os testes que foram executados, as *keywords* executadas e quais delas apresentaram falhas ou não e também os tempos de execução para cada uma. O relatório customizado da *rfmetrics* consegue oferecer

métricas de teste tanto para o time de desenvolvimento, com abordagens mais técnicas em certas abas e também métricas ao time de gestão, trazendo gráficos que excluem a parte do código e fornecem apenas as informações mais importantes e relevantes sobre os testes.

Assim, ao fornecer para o usuário um relatório de teste, é oferecido uma visão geral dos resultados dos testes funcionais realizados nos componentes. Essa visão geral ajuda a identificar os componentes em falha e ainda fornece métricas de teste. Os relatórios contém os passos que foram executados, o que possibilita rastrear o fluxo da execução dos testes e identificar erros.

5 CONSIDERAÇÕES FINAIS

Neste trabalho apresentou-se o desenvolvimento de uma jiga de testes automatizados controlada pelo *framework* de teste de *software Robot framework* para componentes eletrônicos digitais. Essa é uma abordagem não usual para o teste de componentes eletrônicos digitais e que após o desenvolvimento da jiga de testes, tornou-se uma alternativa viável para os testes de componentes eletrônicos.

Empregar o *Robot framework* para realizar os testes funcionais de componentes eletrônicos apresentou diversas vantagens. Através do uso dos testes automatizados por *software*, foi possível garantir maior eficiência nos testes, resultados confiáveis e identificação de erros de forma ágil. Além disso, a utilização do *framework* para os testes permitiu a geração de relatórios de testes em que é possível realizar a rastreabilidade dos componentes com defeito bem como outras análises métricas.

Percebe-se também que o uso dos testes automatizados se sobrepõe aos testes realizados de forma manual e o uso de um *framework* de testes de *software* permite desenvolver testes de fácil compreensão para o usuário. Portanto, essa abordagem contribui para tornar os testes de componentes eletrônicos digitais mais robustos.

A jiga de testes automatizados desse trabalho tinha como objetivo ser desenvolvida utilizando um dispositivo para a geração e aquisição de sinais, o que foi possível através da utilização do DAQ *USB-6212* da fabricante *National Instruments*. Esse dispositivo permitiu que toda a geração e aquisição de sinais fosse feita utilizando somente ele, bem como possibilitou alimentar os componentes em testes sem a necessidade de utilizar uma fonte externa para isso. Entretanto, qualquer dispositivo capaz de realizar a aquisição e geração de sinais, como um gerador de funções e um osciloscópio ou um gerador combinado com um osciloscópio, poderia ter sido utilizado.

O desenvolvimento dos cenários de testes através da sintaxe *Gherkin* no *Robot framework* possibilitou que os casos de testes fossem escritos de forma clara e concisa. Isso facilita o entendimento para usuários menos familiarizados com o *framework* de testes automatizados e também com a realização de testes em componentes eletrônicos digitais.

A validação da jiga de testes utilizando as portas lógicas foi uma abordagem benéfica, pois as portas lógicas são componentes de fácil compreensão e que possuem um comportamento bem definido e conhecido, o que possibilitou simplificar o processo de validação. Ademais, esses componentes existem em grande número no almoxarifado do curso de Engenharia Eletrônica do IFSC e, portanto, são componentes de fácil acesso e que permitem a reutilização de sua estrutura de testes automatizados quando outros componentes mais complexos forem acrescentados a jiga de testes.

Dessa maneira, esse trabalho alcançou seus objetivos propostos, apresen-

tando uma nova abordagem para os testes de componentes eletrônicos digitais. Assim, foi desenvolvido uma jiga de testes automatizados utilizando um único dispositivo, o DAQ *USB-6212*, capaz de realizar a aquisição e geração de sinais. A implementação dos casos de testes foi realizada através do *framework* de testes automatizados *Robot framework*, aproveitando a sintaxe *Gherkin* para descrever os cenários de testes de forma clara e compreensível. Dessa forma, a utilização do *Gherkin* para a descrição dos casos de testes também funciona como documentação, pois o comportamento esperado dos testes é facilmente compreendido.

A utilização das portas lógicas como componente em teste permitiu a validação da geração e aquisição de sinais da jiga. Além disso, o uso das portas lógicas também permitiu que fossem realizados testes funcionais nesse componente através da jiga de testes, o que também possibilitou validar todas as outras integrações existentes na jiga de testes automatizados, como as ações em que envolvem o uso do banco de dados em *cloud MongoDB*.

Por meio do uso do *MongoDB*, foi possível armazenar todos dados necessários para os testes em nuvem, eliminando a necessidade do usuário gerenciar um banco de dados local. Além disso, o formato no qual os dados são armazenados no *MongoDB*, o *JSON*, permitiu armazenar os esquemáticos de ligação dos componentes, do DAQ e os dados de teste de forma simples, facilitando o desenvolvimento da funcionalidade que disponibiliza o esquemático de ligação entre o DAQ e o componente em teste. Dessa forma, o usuário só precisa seguir as recomendações para realizar as ligações requeridas. Além disso, a abordagem utilizando o *MongoDB* também facilita que o usuário cadastre novos esquemáticos e dados de testes, pois o *Json* é um formato amplamente conhecido e utilizado, o que proporciona maior autonomia para o usuário em caso de dificuldades nesse processo. Assim, é possível que novos componentes e dados de testes sejam incorporados a Jiga de forma autônoma pelo usuário.

Ao final da execução dos testes, foram gerados relatórios contendo métricas relevantes para a análise dos resultados obtidos. Esses relatórios proporcionaram informações importantes sobre o desempenho e a qualidade dos testes realizados. Dessa forma, os relatórios são capazes de refletir qual é o estado atual de um componente com base nos testes funcionais que foram realizados, de modo a facilitar a rastreabilidade de componentes com defeitos ou não. O repositório contendo o projeto da jiga de testes pode ser encontrado no repositório do *github* da autora (FARIAS, 2023) <https://github.com/Caroles12/JigadeTestes>.

Assim, a abordagem de utilizar um *framework* de testes de automatizados juntamente com um dispositivo para geração e aquisição de sinais permitiu rastrear componentes com defeito de forma rápida, simples e confiável.

Para futuros trabalhos, sugere-se as seguintes oportunidades de melhoria:

- a) Expandir os casos de testes para suportar outras metodologias de testes além dos testes funcionais;
- b) Expandir a infraestrutura de testes para permitir os testes de componentes eletrônicos mais complexos;
- c) Aumentar o número de componentes que podem ser testados durante uma única execução de um caso de teste;
- d) Implantação dos testes dos componentes em um *pipeline* de testes automatizados, de modo que seja possível realizar a integração contínua e arquivar os resultados de testes, permitindo que eles sejam consultados e avaliados posteriormente.

REFERÊNCIAS

- ABDALLAH, M.; ELKEELANY, O. **A survey on data acquisition systems DAQ**. In: IEEE. *2009 International Conference on Computing, Engineering and Information*. [S.l.], 2009. p. 240–243. 16
- BERNARDO, P. C.; KON, F. **A importância dos testes automatizados**. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54–57, 2008. 22
- BIEHL, D. **RobotCode - Language support for Robot Framework for Visual Studio Code**. 2023. Disponível em: <https://github.com/d-biehl/robotcode>. Acesso em: 30 maio 2023. 36
- CERTI, F. **JIGA DE TESTES**. 2023. Disponível em: <https://certi.org.br/pt/competencias-gestao-da-integridade-jiga-de-testes>. Acesso em: 01 mai 2023. 12, 15
- CORP, N. I. **NI Device Drivers**. 2023. Disponível em: <https://www.ni.com/pt-br/support/downloads/drivers/download.ni-device-drivers.html#327643>. Acesso em: 03 junho 2023. 19
- FARIAS, C. d. **JigadeTestes**. 2023. Disponível em: <https://github.com/Caroles12/JigadeTestes>. Acesso em: 21 junho 2023. 82
- FREEMAN, H. **Software testing**. *IEEE instrumentation & measurement magazine*, IEEE, v. 5, n. 3, p. 48–50, 2002. 22
- GIL, A. C. *et al.* **Como elaborar projetos de pesquisa**. [S.l.]: Atlas São Paulo, 2002. v. 4. 34
- Incorporated, Diodes. **74HC08**. 2013. Disponível em: <https://pdf1.alldatasheetpt.com/datasheet-pdf/view/608279/DIODES/74HC08.html>. Acesso em: 31 maio 2023. 46
- Incorporated, Diodes. **74HC32**. 2013. Disponível em: <https://pdf1.alldatasheet.com/datasheet-pdf/view/608276/DIODES/74HC32.html>. Acesso em: 31 maio 2023. 63
- ISTQB - International Software Testing Qualifications Board. **Syllabus Agile Tester**. [S.l.]: ISTQB, 2014. 22
- ISTQB - International Software Testing Qualifications Board. **Syllabus Test Automation Engineer**. [S.l.]: ISTQB, 2016. 12, 23
- JOHNSON, R. E.; FOOTE, B. **Designing reusable classes**. *Journal of object-oriented programming*, v. 1, n. 2, p. 22–35, 1988. 23
- JSON. **Introducing JSON**. 2023. Disponível em: <https://www.json.org/json-en.html>. Acesso em: 22 maio 2023. 31
- KLARCK, P. *et al.* **Robot Framework Metrics Report**. 2022. Disponível em: <https://github.com/adiralashiva8/robotframework-metrics>. Acesso em: 04 maio 2023. 29
- LANGER, A.; HELLMUTH, C. **Device and method for testing electronic components**. [S.l.]: Google Patents, 2010. US Patent 7,673,520. 12

- LIAO, C.-F. *et al.* **Toward a service platform for developing smart contracts on blockchain in bdd and tdd styles**. In: IEEE. *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*. [S.l.], 2017. p. 133–140. 29
- MATHWORKS, T. **MATLAB**. 2023. Disponível em: <https://www.mathworks.com/products/matlab.html>. Acesso em: 03 junho 2023. 17
- Measurement Computing Corporation . **DASYLab - Data Acquisition System Laboratory**. 2023. Disponível em: <https://www.mccdaq.com/dasylab-index>. Acesso em: 03 junho 2023. 17
- Measurement Computing Corporation. **USB-1608G Series**. 2018. Disponível em: <https://www.mccdaq.com/PDFs/specs/USB-1608G-Series-data.pdf>. Acesso em: 20 maio 2023. 18
- MONGODB. **Database. Deploy a multi-cloud database**. 2023. Disponível em: <https://www.mongodb.com/docs/manual/>. Acesso em: 22 maio 2023. 31
- MONGODB. **Databases and Collections**. 2023. Disponível em: <https://www.mongodb.com/docs/manual/core/databases-and-collections/>. Acesso em: 03 junho 2023. 31
- MONGODB. **MongoDB Documentation**. 2023. Disponível em: <https://www.mongodb.com/docs/manual/>. Acesso em: 22 maio 2023. 31
- MONGODB. **What is NoSQL?** 2023. Disponível em: <https://www.mongodb.com/nosql-explained>. Acesso em: 22 maio 2023. 31, 32
- Mordor Intelligence. **AUTOMATED TEST EQUIPMENT MARKET SIZE & SHARE ANALYSIS - GROWTH TRENDS & FORECASTS (2023 - 2028)**. 2023. Disponível em: <https://www.mordorintelligence.com/industry-reports/global-automated-test-equipment-market-industry>. Acesso em: 20 maio 2023. 15
- NATIONAL INSTRUMENTS. **LabVIEW**. 2023. Disponível em: <https://www.ni.com/pt-br/support/downloads/software-products/download.labview.html#477380>. Acesso em: 03 junho 2023. 17
- NATIONAL INSTRUMENTS CORP. **DAQ de uso geral**. 2023. Disponível em: https://www.ni.com/docs/en-US/bundle/labview/page/lvmeasconcepts/general_purpose_daq.html. Acesso em: 20 maio 2023. 16
- NATIONAL INSTRUMENTS CORP. **How to Choose the Right DAQ Hardware for Your Measurement System**. 2023. Disponível em: <https://www.ni.com/pt-br/shop/data-acquisition/how-to-choose-the-right-daq-hardware-for-your-measurement-system.html>. Acesso em: 20 maio 2023. 16, 17
- NATIONAL INSTRUMENTS CORPORATION. **Bus-Powered M Series Multifunction DAQ for USB - 16-Bit, up to 400 kS/s, up to 32 Analog Inputs, Isolation**. 2009. <https://www.ni.com/docs/en-US/bundle/usb-621x-features/resource/371931f.pdf>. Acesso em 28 maio 2023. 21
- NATIONAL INSTRUMENTS CORPORATION. **USB-6212 - Multifunction I/O Device**. 2023. Disponível em: <https://www.ni.com/pt-br/support/model.usb-6212.html>. Acesso em: 28 maio 2023. 18, 19

- NATIONAL INSTRUMENTS CORPORATION. **USB-6212 Specifications**. 2023. Disponível em: <https://www.ni.com/docs/en-US/bundle/usb-6212-specs/page/specs.html>. Acesso em: 28 maio 2023. 19
- Omega Engineering. **A Complete Guide to Data Acquisition (DAQ) Systems**. 2023. Disponível em: <https://www.omega.com/en-us/resources/daq-systems>. Acesso em: 20 maio 2023. 16
- PESONEN, A. S. **An Open and General Numerical Control and Machine Vision Based Architecture for Payment Terminal Acceptance Test Automation**. Master's Thesis, 2016. 29
- PIERRE, C. **Welcome to PyDAQmx's documentation!** 2010. Disponível em: <https://pythonhosted.org/PyDAQmx/>. Acesso em: 03 junho 2023. 17
- Robot Framework Foundation. **Robot Framework User Guide**. 2016. Disponível em: <https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#getting-started>. Acesso em: 15 abril 2023. 12, 24, 26
- Robot Framework Guides. **Estrutura do Projeto**. 2023. Disponível em: https://docs.robotframework.org/docs/examples/project_structure. Acesso em: 06 maio 2023. 24, 25
- RODRIGUES, C. E. B. **Jiga automática para testes de fontes chaveadas**. 2009. 13
- SETH, N.; KHARE, R. **ACI (automated Continuous Integration) using Jenkins: Key for successful embedded Software development**. In: IEEE. *2015 2nd International Conference on Recent Advances in Engineering & Computational Sciences (RAECS)*. [S.l.], 2015. p. 1–6. 23
- SMART, J. **BDD in Action: Behavior-driven development for the whole software lifecycle**. [S.l.]: Simon and Schuster, 2014. 12, 13, 29, 30
- SmartBear Software. **Gherkin Reference**. 2019. Disponível em: <https://cucumber.io/docs/gherkin/reference/>. Acesso em: 20 maio 2023. 29
- Trenton Systems. **What Is Automatic Test Equipment (ATE)?** 2021. Disponível em: <https://www.trentonsystems.com/blog/automatic-test-equipment-overview>. Acesso em: 20 maio 2023. 15