UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Emmanuel Gonçalves Notargiacomo

Desenvolvimento e implementação de testes automatizados para garantia de qualidade de um piloto automático para máquinas agrícolas

Florianópolis
23 de julho de 2024

Emmanuel Gonçalves Notargiacomo

Desenvolvimento e implementação de testes automatizados para garantia de qualidade de um piloto automático para máquinas agrícolas

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Hector Bessa Silveira, Dr.
Supervisor: Thiago Alberici Roberto, Eng.

Florianópolis
23 de julho de 2024

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Notargiacomo, Emmanuel Gonçalves
Desenvolvimento e implementação de testes automatizados para garantia de qualidade de um piloto automático para máquinas agrícolas / Emmanuel Gonçalves Notargiacomo ; orientador, Hector Bessa Silveira, 2024.
81 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2024.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Máquinas Agrícolas. 3. Garantia de Qualidade. 4. Testes Automatizados. 5. Cucumber. I. Silveira, Hector Bessa. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Emmanuel Gonçalves Notargiacomo

Desenvolvimento e implementação de testes automatizados para garantia de qualidade de um piloto automático para máquinas agrícolas

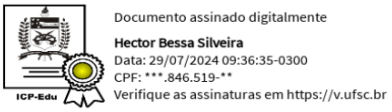
Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 23 de julho de 2024.

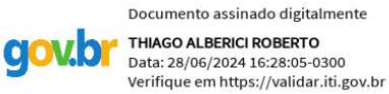
Prof. Marcelo de Lellis Costa de Oliveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Hector Bessa Silveira, Dr.
Orientador
UFSC/CTC/DAS



Thiago Alberici Roberto, Eng.
Supervisor
Hexagon Agriculture



Prof. Gabriel Thaler, Dr.
Avaliador
UFSC/CTC/DAS

Prof. Eduardo Camponogara, Dr.
Presidente da Banca
UFSC/CTC/DAS

AGRADECIMENTOS

Os principais agradecimentos para este projeto de fim de curso vão à minha família, ao meu pai Gilberto e, em especial, à minha mãe Mariângela, a quem tenho como exemplo de força e superação, sendo a principal responsável por eu estar hoje escrevendo este texto. Mãe, eu te agradeço por todas as noites mal dormidas, por todo o esforço, dedicação, paciência e esperança que teve na criação de seus 3 filhos sozinha. Saiba que sem você não seria o homem que sou hoje. Agradeço à minha irmã Tanay e ao meu irmão Daylan, peças fundamentais na minha trajetória e no meu crescimento como pessoa. A todos de minha família, meus sinceros agradecimentos pelo esforço que fizeram para que eu chegasse até aqui. Vocês foram e são essenciais para mim e não poderia dedicar inicialmente a realização deste trabalho a mais ninguém a não ser a vocês. Gostaria de agradecer à minha namorada Clara, que surgiu em minha vida em um momento crucial e me apoiou de diversas formas durante meu último ano de graduação, mostrando-me que eu era capaz e acreditando em mim o tempo todo.

Também quero agradecer aos bons amigos que fiz durante todos esses anos de graduação, em especial ao André, por todos os momentos bons e ruins que passamos juntos e, também, por ter me auxiliado a estar na empresa em que estou hoje realizando este projeto, o que fez minha vida mudar rapidamente para melhor em diversos aspectos. Agradeço ao Gian, meu companheiro de casa, pelas boas conversas, divagações e risadas durante o tempo em que moramos juntos. Aos demais amigos, agradeço pelo apoio e pelos bons momentos que compartilhamos, desde o semestre em Farmácia na UFSC em 2016, até a equipe Ampera Racing e a organização do 37º Linguição da Automação.

Obrigado, também, aos professores que marcaram minha jornada na graduação, contribuindo para meu crescimento profissional e pessoal e ampliando meus conhecimentos. Embora tenha enfrentado dificuldades que até geraram crises de ansiedade e questionamentos sobre minha continuidade no curso, a determinação e ajuda desses bons professores foram grandes motivadores para a conclusão da minha graduação.

Finalmente, agradeço à empresa Hexagon, especialmente ao time de QA do qual faço parte, por me receberem tão positivamente, por me ensinarem diariamente e por possibilitarem a realização deste projeto.

DECLARAÇÃO DE PUBLICIDADE

Florianópolis, 23 de julho de 2024.

Na condição de representante da Hexagon Agriculture na qual o presente trabalho foi realizado, declaro não haver ressalvas quanto ao aspecto de sigilo ou propriedade intelectual sobre as informações contidas neste documento, que impeçam a sua publicação por parte da Universidade Federal de Santa Catarina (UFSC) para acesso pelo público em geral, incluindo a sua disponibilização *online* no Repositório Institucional da Biblioteca Universitária da UFSC. Além disso, declaro ciência de que o autor, na condição de estudante da UFSC, é obrigado a depositar este documento, por se tratar de um Trabalho de Conclusão de Curso, no referido Repositório Institucional, em atendimento à Resolução Normativa n° 126/2019/CUn.

Por estar de acordo com esses termos, subscrevo-me abaixo.

gov.br
Documento assinado digitalmente
THIAGO ALBERICI ROBERTO
Data: 28/06/2024 16:28:05-0300
Verifique em https://validar.iti.gov.br

Thiago Alberici Roberto
Hexagon Agriculture

RESUMO

A agricultura moderna enfrenta uma série de desafios complexos, que vão desde a crescente demanda por alimentos até preocupações ambientais e a escassez de recursos. Nesse contexto, a agricultura de precisão surge como uma abordagem inovadora e promissora, visando enfrentar esses desafios por meio da aplicação de tecnologias avançadas e da otimização de processos. No âmbito específico da empresa Hexagon, que desenvolve sistemas de automação para máquinas agrícolas, a validação de novas versões de software para seus produtos é um ponto crítico. Atualmente, esse processo de validação é manual, o que representa um gargalo operacional significativo devido ao tempo prolongado de execução dos testes e à possibilidade de erros humanos. Cada ciclo de teste pode levar semanas para ser concluído, retardando o lançamento de novas versões de *software*. Diante desse cenário, a empresa tomou a iniciativa de buscar soluções inovadoras para otimizar o processo de validação, focando no desenvolvimento de testes automatizados. Essa abordagem não apenas promete acelerar o ciclo de validação, reduzindo significativamente o tempo necessário para concluir os testes, mas também melhorar a consistência e a confiabilidade dos resultados. O presente trabalho tem como objetivo desenvolver e implementar um sistema de testes automatizados para o sistema de piloto automático da Hexagon, garantindo a qualidade do produto entregue aos clientes. A validação será feita na comunicação que segue o protocolo CAN entre um computador de bordo e um motor elétrico que atua no volante das máquinas agrícolas. Além de buscar a eficiência e a robustez do sistema, o projeto também propõe integrar os testes automatizados de forma contínua ao processo de desenvolvimento de novas versões de software, garantindo uma abordagem mais ágil e iterativa. Para alcançar esses objetivos, o projeto se apoia no uso do *framework Cucumber* para a implementação dos testes automatizados, divididos em duas categorias: *Software-in-the-Loop* (SIL) e *Hardware-in-the-Loop* (HIL). Os testes SIL validam o software isoladamente, enquanto os testes HIL incluem a interação com hardware real, proporcionando uma cobertura de teste mais abrangente. Apesar dos avanços, a transição para a automação completa dos testes apresenta desafios significativos, incluindo a necessidade de desenvolver uma infraestrutura de teste robusta e garantir que os testes automatizados cubram todas as possíveis condições de operação. A implementação bem-sucedida desses testes automatizados é importante para reduzir o tempo de validação, minimizar falhas humanas e garantir a alta qualidade e segurança dos produtos da Hexagon. Neste trabalho, também são apresentados os resultados positivos da realização deste projeto, o qual resultou em uma redução de tempo de teste de algumas horas para poucos minutos para a empresa, devido a sua rapidez de execução e a integração contínua com uma plataforma que garante a realização dos testes a cada nova mudança de código, trazendo confiança e minimizando a possibilidade de erros.

Palavras-chave: Máquinas Agrícolas. Garantia de Qualidade. Testes Automatizados. Cucumber.

ABSTRACT

Modern agriculture faces a number of complex challenges, ranging from growing demand for food to environmental concerns and scarcity of resources. In this context, precision agriculture emerges as an innovative and promising approach, aiming to face these challenges through the application of advanced technologies and process optimization. In the specific scope of the company Hexagon, which develops automation systems for agricultural machinery, the validation of new software versions for its products is a critical point. Currently, this validation process is manual, which represents a significant operational bottleneck due to the prolonged test execution time and the possibility of human errors. Each testing cycle can take weeks to complete, slowing the release of new *software* versions. Faced with this scenario, the company took the initiative to seek innovative solutions to optimize the validation process, focusing on the development of automated tests. This approach not only promises to speed up the validation cycle, significantly reducing the time required to complete testing, but also improves the consistency and reliability of results. The present work aims to develop and implement an automated testing system for Hexagon's autopilot system, ensuring the quality of the product delivered to customers. Validation will be carried out in communication that follows the CAN protocol between an on-board computer and an electric motor that operates on the steering wheel of agricultural machines. In addition to seeking system efficiency and robustness, the project also proposes to integrate automated tests continuously into the development process of new software versions, ensuring a more agile and iterative approach. To achieve these objectives, the project relies on the use of *framework Cucumber* to implement automated tests, divided into two categories: *Software-in-the-Loop* (SIL) and textitHardware-in-the-Loop (HIL). SIL tests validate software in isolation, while HIL tests include interaction with real hardware, providing more comprehensive test coverage. Despite advances, the transition to full test automation presents significant challenges, including the need to develop a robust testing infrastructure and ensure that automated tests cover all possible operating conditions. Successful implementation of these automated tests is important to reduce validation time, minimize human errors and ensure the high quality and safety of Hexagon products. The positive results of carrying out this project are also presented, which resulted in a reduction in testing time from a few hours to a few minutes for the company, due to its speed of execution and continuous integration with a platform that guarantees testing. with each new code change, bringing confidence and minimizing the possibility of errors.

Keywords: Agricultural machinery. Quality assurance. Automated Tests. Cucumber.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Motivação

A agricultura de precisão tem emergido como uma solução inovadora para os desafios contemporâneos enfrentados pelo setor agrícola, caracterizado por uma crescente demanda por alimentos, a necessidade de práticas sustentáveis e a pressão por otimizar o uso de recursos naturais. Tradicionalmente, as práticas agrícolas têm sido amplamente baseadas em técnicas homogêneas que não levam em conta as variações naturais dentro de um campo, resultando em ineficiências e desperdício de recursos. A introdução de tecnologias avançadas permite uma abordagem mais detalhada e precisa na gestão agrícola, proporcionando ganhos significativos em produtividade e sustentabilidade.

Nesse contexto, a automação de processos e a utilização de sistemas integrados de controle e monitoramento tornam-se cruciais. A capacidade de coletar e analisar dados em tempo real sobre as condições do solo e das culturas permite aos agricultores tomar decisões mais embasadas e precisas. Sistemas automatizados que utilizam tecnologias como GPS, sensores de solo e drones têm o potencial de revolucionar a agricultura, mas essa revolução só será possível se esses sistemas forem desenvolvidos com um alto grau de confiabilidade e precisão. A implementação de novos produtos tecnológicos no campo agrícola deve, portanto, ser acompanhada por rigorosos processos de validação de *hardware* e *software* para garantir que os benefícios esperados sejam realmente alcançados sem comprometer a segurança e a eficácia das operações agrícolas.

A necessidade de validação de novos produtos e sistemas é um aspecto crítico nesse cenário. Com a complexidade crescente das tecnologias empregadas, é importante que os sistemas de agricultura de precisão sejam testados exaustivamente em condições controladas e reais antes de sua aplicação em larga escala. A validação envolve não apenas assegurar que os sistemas funcionem conforme projetado, mas, também, que eles possam operar de forma consistente e confiável em diversas condições ambientais e operacionais, como adversidades climáticas, oscilações de terreno e até eventos cósmicos que podem influenciar em sistemas de geolocalização terrestres. Isso inclui a realização de testes e simulações que reproduzam situações do mundo real, garantindo que qualquer falha potencial seja identificada e corrigida antes da implementação no campo. A motivação por trás deste trabalho reside, portanto, na criação e implementação de testes automatizados que garantam a qualidade e a confiabilidade de um sistema de piloto automático para máquinas agrícolas, em que um complexo sistema de controle força a máquina a restrar automaticamente uma linha guia para colher ou semear, por exemplo, dispensando o contíuno esforço manual do condutor. Isso é essencial para acompanhar o ritmo acelerado da inovação tecnoló-

gica na agricultura, assegurando que novos sistemas de automação tragam benefícios concretos e sustentáveis para o setor.

1.2 Breve Descrição do Problema

A empresa Hexagon, onde este projeto foi realizado, desenvolve, na sua divisão de agricultura, sistemas complexos que atendem diversas demandas do setor agrícola, como controladores de fertilização, de plantio e de piloto automático, que exigem a integração precisa entre software e firmware para garantir a segurança e a eficiência dos produtos. O *software* é instalado em um computador de bordo com interface ao usuário, enquanto que o *firmware* é responsável pelo controle de motores elétricos, como o *Track Controller*. O lançamento de novas versões de *software*, incluindo as atualizações de *Long Term Support* (LTS), exige uma validação rigorosa para garantir a funcionalidade e segurança dos produtos.

Atualmente, a validação dessas novas versões é realizada pelo time de *Quality Assurance* (QA) da Hexagon através de testes manuais de regressão, particularmente focados no sistema de piloto automático. Esses testes manuais são necessários para garantir que o piloto automático funcione corretamente, prevenindo possíveis falhas que poderiam resultar em acidentes e ferimentos. No entanto, esse processo manual apresenta dois problemas principais: o tempo prolongado de execução dos testes e a possibilidade de erros humanos. Cada ciclo de teste pode levar semanas para ser concluído, o que retarda o processo de lançamento de novas versões de software e firmware.

Para resolver esses problemas, a Hexagon iniciou, em 2023, o desenvolvimento de testes automatizados para validar o piloto automático, com foco na comunicação CAN entre o display e o Track Controller. Os testes automatizados são implementados utilizando o framework Cucumber e divididos em duas categorias: Software-in-the-Loop (SIL) e Hardware-in-the-Loop (HIL). Os testes SIL validam o software isoladamente, enquanto os testes HIL incluem a interação com hardware real, proporcionando uma cobertura de teste mais abrangente.

Apesar dos avanços, a transição para a automação completa dos testes apresenta desafios significativos, incluindo a necessidade de desenvolver uma infraestrutura de teste robusta e garantir que os testes automatizados cubram todas as possíveis condições de operação. A implementação bem-sucedida desses testes automatizados é crucial para reduzir o tempo de validação, minimizar falhas humanas e garantir a alta qualidade e segurança dos produtos da Hexagon.

1.3 Objetivos

O principal objetivo deste projeto é desenvolver e implementar um sistema de testes automatizados para o piloto automático, especificamente para o motor elétrico *Track Controller*. A meta é garantir a eficiência, robustez e segurança do piloto automático, um dos produtos mais utilizados e vendidos pela empresa, através de validações rigorosas e padronizadas a cada nova versão de *software* e *firmware*.

1.3.1 Objetivos Específicos

1. Redução do Tempo de Validação:
   Automatizar o processo de testes para diminuir o tempo necessário para validar o piloto automático, atualmente feito manualmente, o que pode levar semanas.

2. Minimização de Falhas Humanas:
   Eliminar a possibilidade de erros humanos durante o processo de testes, garantindo que todos os detalhes sejam verificados consistentemente.

3. Implementação de Testes Automatizados:
   Desenvolver uma infraestrutura de testes automatizados utilizando o *framework Cucumber* para a criação e execução de testes.

4. Cobertura Abrangente dos Testes:
   Garantir que os testes cubram todas as funcionalidades do *Track Controller*, tanto em cenários normais quanto em situações de falha, assegurando a robustez e precisão do sistema.

5. Escalabilidade do Sistema de Testes:
   Construir uma infraestrutura de testes flexível e adaptável, que permita a adição contínua de novos casos de teste e funcionalidades conforme o desenvolvimento progride.

6. Confiabilidade dos Resultados:
   Minimizar falsos positivos e falsos negativos nos testes, implementando mecanismos que assegurem a identificação precisa de falhas e a validação correta do sistema.

7. Integração Contínua:
   Integrar os testes automatizados em um pipeline de integração contínua (CI) para executar testes a cada commit ou pull request, garantindo a detecção e correção rápida de problemas.

8. Configuração de Ambientes de Teste SIL e HIL:
   Configurar dois tipos de ambientes de teste: Software-in-the-Loop (SIL) para

simulações de software e Hardware-in-the-Loop (HIL) para testes com hardware real, garantindo uma cobertura completa desde a simulação até a validação com hardware.

Ao alcançar esses objetivos, o projeto visa melhorar significativamente o processo de validação do piloto automático, aumentando a qualidade, segurança e confiabilidade do produto final.

1.4 Técnicas e Métodos Utilizados

Para alcançar os objetivos propostos, foi empregada uma abordagem metodológica robusta, que envolveu a aplicação de diversas técnicas e métodos. Essas metodologias foram escolhidas cuidadosamente para garantir a flexibilidade, a melhoria contínua e a validação rigorosa do sistema desenvolvido. A seguir, detalham-se as principais metodologias utilizadas.

1.4.1 Testes Automatizados

Para assegurar a qualidade e a confiabilidade do software, foram implementados testes automatizados utilizando o framework Cucumber. A integração desses testes com o C++ foi realizada afim de permitir a execução de testes de comportamento (BDD - Behavior Driven Development) de forma eficiente, ao mesmo tempo em que utiliza um tipo de escrita de testes de alto nível chamado Ghrekin. Os principais benefícios desta abordagem incluíram:

- **Verificação Contínua**: Testes automáticos executados regularmente para identificar e corrigir erros rapidamente.
- **Documentação Viva**: Os cenários de teste serviram como documentação dinâmica, facilitando a compreensão do comportamento esperado do sistema.
- **Redução de Erros**: A automação dos testes minimizou a ocorrência de erros humanos e garantiu a repetibilidade dos testes.

1.5 Organização do Documento

A estrutura deste documento é delineada conforme segue: O Capítulo 2 aborda a Fundamentação Teórica, explorando os conceitos essenciais como agricultura de precisão, metodologias ágeis e testes automatizados. No Capítulo 3 são detalhados os desafios e requisitos técnicos do problema abordado. O Capítulo 4 discorre sobre a Solução Proposta e Metodologia Utilizada, descrevendo a implementação do sistema e os métodos de validação. Os resultados obtidos são analisados no capítulo 5, onde

observa-se o cumprimento dos requisitos do projeto e a análise do impacto positivo que se teve no produto, mostrando que os testes atingem um nível satisfatório de validação do produto ao mesmo tempo em que reduzem significativamente o tempo de execução quando comparados com a validação manual. Finalmente, o Capítulo 6 expõe o sucesso atingido neste projeto e explora as perspectivas futuras de trabalho para validação do mesmo sistema de piloto automático através da comunicação utilizando uma API e da expansão da automação de testes em outros produtos fornecidos pela empresa Hexagon.

2 FUNDAMENTAÇÃO TEÓRICA

O presente capítulo aborda os principais conceitos e teorias que fundamentam este trabalho. Uma visão geral sobre agricultura de precisão é apresentada na Seção 2.1, abordando tópicos como sistemas de geolocalização e de informação geográfica. Na Seção 2.2 é apresentado o conceito de garantia de qualidade (*Quality Assurance*), explicando um pouco de sua origem e como isso se desenvolveu até se tornar uma área específica do ramo de tecnologia. A Seção 2.3 aborda os conceitos de testes automatizados, trazendo também as formas de testes via *Software-in-the-loop* (SIL) e *Hardware-in-the-loop* (HIL). O protocolo de comunicação CAN é apresentado na Seção 2.4, trazendo sua origem, funcionamento e aplicações na indústria. As Seções 2.5, 2.6, 2.7 e 2.8 abordam questões mais voltadas ao desenvolvimento de software e em específico a criação de testes automatizados, explicando inicialmente como se trabalha com versionamento de código utilizando a ferramenta GIT na Seção 2.5, abordando em seguida o conceito de desenvolvimento orientado a comportamento (BDD) na seção 2.6, apresentando a linguagem de escrita humana *Gherkin* usada nos testes automatizados na Seção 2.7 e finalizando com a apresentação do*framework cucumber* na Aeção 2.8, mostrando como essa ferramenta auxilia na criação de testes automatizados.

2.1 Agricultura de Precisão

A presente seção aborda a transição da agricultura tradicional para a agricultura de precisão, destacando a utilização de tecnologias avançadas como o Sistema Global de Navegação por Satélite (GNSS) e o Sistema de Informação Geográfica (GIS). Estas ferramentas permitem a coleta e análise de dados específicos do solo, das plantas e do ambiente, possibilitando decisões mais precisas e eficientes desde o plantio até a colheita, visando maximizar a produtividade agrícola e reduzir impactos ambientais.

Na agricultura tradicional, o recomendado é que a adição de insumos seja feita uniformemente. Isso implica que, quando adicionados ao solo, a mesma quantidade é utilizada em toda a propriedade, desconsiderando as particularidades de cada planta e parte do solo. Tal prática implica em uma maior chance de contaminação da água e do solo, podendo provocar prejuízos financeiros resultantes do excesso de uso de fertilizantes e outros insumos usuais na agricultura (MOLIN, 2006).

Contudo, é importante salientar que quando se trata de uma propriedade agrícola, o solo dificilmente apresentará a mesma composição em toda a sua área, podendo ter características físico-químicas distintas em diferentes partes, assim, necessitando de diferentes quantidades de insumo para uma ação eficiente na plantação desejada.

Para auxiliar na resolução dos problemas da agricultura tradicional, temos a

agricultura de precisão. Trata-se de uma abordagem que utiliza tecnologias avançadas de geolocalização, sensores e sistemas de informações geográficas para coletar e analisar dados específicos sobre o solo, as plantas e as condições ambientais do terreno observado. Esses dados são então utilizados para tomada de decisões em todas as etapas do processo, focando na eficiência e na qualidade, indo desde o plantio até a colheita. O objetivo principal é maximizar a produtividade da área plantada, minimizar o desperdício de insumos (sementes, adubo, fertilizantes, água, etc) e também reduzir so impactos ambientais resultantes de todo o processo, tornando a agricultura mais sustentável e rentável.

Os fundamentos para a agricultura de precisão moderna, como conhecemos hoje, vem do início do século XX, porém somente na década de 1980, na Europa e nos EUA, com o desenvolvimento de microcomputadores, sensores e softwares é que a agricultura de precisão tornou-se viável para os produtores. (SOUZA CARNEIRO GOULART E ARYSTIDES RESENDE FILHO E ALUÍZIO BORÉM, 2014).

No Brasil, observa-se que a agricultura de precisão está em uma fase inicial. No entanto, através de iniciativas tomadas por universidades como a USP e escolas agrícolas como a Esalq (Escola Superior de Agricultura Luiz de Queiroz), com a realização de simpósios sobre o assunto, bem como eventos destinados aos avanços da área, apresenta-se um cenário otimista para o crescimento deste setor dentro do país.

2.1.1 Sistema Global de Navegação por Satélite - GNSS

O GNSS, sigla para Global Navigation Satellite System (Sistema Global de Navegação por Satélite), é um sistema que permite a determinação da posição geográfica de um receptor em qualquer lugar do mundo, a qualquer momento e sob quaisquer condições meteorológicas. Ele funciona através de uma rede de satélites que emitem sinais de rádio. Esses sinais são recebidos por dispositivos GNSS, como smartphones e navegadores GPS, que calculam a posição exata do usuário com base na distância entre o receptor e pelo menos quatro satélites diferentes (EMBRATOP, 2022).

Figura 1 – Sistema de Posicionamento Global.Ilustração: Designua / Shutterstock.com

A Figura 1 representa o sistema de posicionamento global utilizado atualmente, onde ao centro há uma representação em desenho do planeta Terra e ao seu redor uma constelação de satélites que orbitam em trajetórias precisas. Esses satélites transmitem sinais continuamente, permitindo que receptores no solo calculem sua posição com alta precisão. Esse sistema é fundamental para diversas aplicações em agricultura de precisão, proporcionando dados exatos de localização que são utilizados para mapear campos, monitorar a saúde das culturas e otimizar a utilização de insumos agrícolas (DAVID CLAY, 2007).

O funcionamento do GNSS baseia-se na trilateração, que é uma técnica de medição de distâncias. Cada satélite envia um sinal com informações sobre a hora exata em que o sinal foi transmitido e a posição do satélite no momento da transmissão. O receptor GNSS capta esses sinais e calcula o tempo que o sinal levou para chegar até ele. Com essa informação, o receptor determina a distância de cada satélite e, através de cálculos complexos, determina a sua própria posição exata em termos de latitude, longitude e altitude (B. HOFMANN-WELLENHOF; WASLE, 2007).

Existem vários sistemas GNSS em operação ao redor do mundo, sendo os mais famosos o GPS (Global Positioning System) dos Estados Unidos, o GLONASS (Global Navigation Satellite System) da Rússia, o Galileo da União Europeia e o BeiDou da China. Cada um desses sistemas possui sua própria constelação de satélites e características específicas. O GPS, por exemplo, foi o primeiro a ser amplamente utilizado e é conhecido pela sua alta precisão e cobertura global. O GLONASS é similar ao GPS e oferece redundância e complementaridade ao sistema norte-americano. O Galileo, em operação completa desde 2019, oferece maior precisão e interoperabilidade com outros sistemas GNSS. Já o BeiDou, em constante expansão, busca fornecer uma alternativa robusta com cobertura global, destacando-se na região da Ásia-Pacífico.

2.1.2 Sistema de Informação Geográfica - GIS

O Sistema de Informação Geográfica (SIG), frequentemente abreviado como GIS, é uma ferramenta tecnológica poderosa que integra dados espaciais e não espaciais para análise, visualização e tomada de decisões em uma ampla gama de campos. Essa tecnologia permite a captura, armazenamento, manipulação, análise e apresentação de informações geográficas, permitindo que os usuários entendam melhor padrões e relações espaciais. Por meio de softwares especializados, os dados geográficos são representados em camadas temáticas sobre mapas, possibilitando a sobreposição de diferentes conjuntos de dados para revelar insights significativos sobre o ambiente físico, social e econômico. Além disso, o GIS tem sido cada vez mais utilizado em projetos de planejamento territorial e regional, onde a análise espacial desempenha um papel crucial na alocação de recursos e no desenvolvimento de políticas públicas.

A Figura 2 ilustra um mapa temático gerado por um Sistema de Informação Geográfica (GIS) que destaca áreas problemáticas em uma região específica. Utilizando diferentes camadas de dados sobre uma base cartográfica, a imagem sobrepõe informações críticas, como níveis de poluição, densidade populacional ou incidência de crimes, indicadas por diversas cores e símbolos. Este tipo de visualização é essencial para análises espaciais, ajudando na identificação de padrões geográficos, tendências e na tomada de decisões estratégicas para a resolução de problemas.

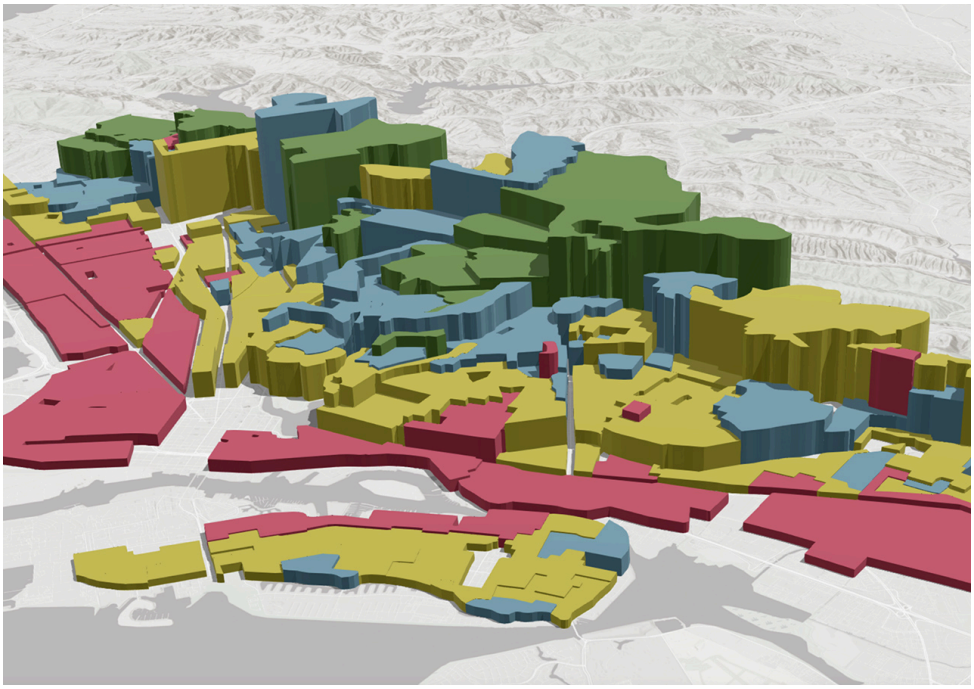


Figura 2 – Sistema de Informação Geográfica.Ilustração: Esri / esri.com

Os SIGs são amplamente utilizados em diversas áreas, incluindo planejamento urbano, gestão ambiental, agricultura, transporte, geologia, cartografia, entre outras. Desde o mapeamento de recursos naturais até o monitoramento de mudanças cli-

máticas, o GIS desempenha um papel fundamental na compreensão e resolução de desafios complexos relacionados ao espaço geográfico. Sua capacidade de integrar dados de várias fontes e analisar padrões espaciais faz dele uma ferramenta valiosa para governos, empresas e organizações que buscam otimizar recursos, planejar o desenvolvimento urbano sustentável e responder de forma eficaz a emergências e desastres naturais. Em setores como a agricultura de precisão, o GIS permite uma gestão mais eficiente das terras agrícolas, ajudando os agricultores a tomarem decisões informadas sobre irrigação, fertilização e colheita, resultando em maior produtividade e sustentabilidade ambiental (LONGLEY *et al.*, 2015).

2.2 *Quality Assurance*

A presente seção aborda a evolução histórica da Garantia de Qualidade (QA) desde suas origens na indústria de manufatura até sua consolidação no campo da tecnologia, onde se tornou essencial para garantir a qualidade e eficiência no desenvolvimento de software e sistemas computacionais, incorporando métodos proativos e abrangendo diversas etapas do ciclo de vida do produto.

A trajetória da Garantia de Qualidade (QA - Quality Assurance) na área de tecnologia remonta aos primórdios do desenvolvimento de software e sistemas computacionais. Nas décadas de 1950 e 1960, durante os estágios iniciais da programação, os processos de desenvolvimento careciam de estruturação, e a importância atribuída à qualidade não era tão destacada (COMMUNITY, 2023).

O termo "QA", surge no início do século XX, inicialmente na indústria de manufatura, sem qualquer relação com tecnologia na época. Este termo e o que ele representa passou a ser aplicado com o objetivo de melhorar os processos de produção e garantir a qualidade dos produtos fabricados, reduzindo custos relacionados a perdas. Contudo, no meio da indústria de manufatura o termo QA era mais relacionado com a inspeção e o controle de qualidade de produtos físicos, como peças de maquinaria ou produtos já acabados, mas conforme os anos passaram e a tecnologia da informação começou a se desenvolver, surgiu a necessidade de aplicar os conceitos de QA nessa fatia da indústria, ou seja, implementar o QA no processo de desenvolvimento de software.

Somente entre os anos 1950 e 1960 é que o termo "QA"começou a surgir no campo da tecnologia, ficando rapidamente consolidado entre as empresas da época. O foco inicial era apenas a detecção de erros e a validação da correção após a implementação feita pelo time responsável, ou seja, tratava-se do que hoje chamamos de "testador de software". Com o passar dos anos, houve a necessidade de evoluir a abordagem do QA para olhar além do teste de software, mas também focar em um processo de garantia de qualidade, incorporando métodos mais proativos, tendo uma visão mais geral de todo o processo e buscando prevenir defeitos já nas fases iniciais do processo de desenvolvimento de *software* (SOMMERVILLE, Ian, 2011).

O conceito de QA evoluiu significativamente desde então. Hoje em dia, a Garantia de Qualidade é uma parte essencial de quase todos os processos de desenvolvimento de software, sendo integrada desde o início do ciclo de vida do desenvolvimento até a entrega do produto final. Além disso, a abordagem do QA se expandiu para além do desenvolvimento de software, sendo aplicada em diversos outros campos, como desenvolvimento de aplicativos móveis, desenvolvimento web, análise de dados, entre outros.

Uma das principais técnicas utilizadas em QA é a automação de testes, que permite a execução de testes de forma mais eficiente e repetível, garantindo uma cobertura mais ampla do sistema. Além disso, o uso de métricas e indicadores de desempenho também desempenha um papel crucial na avaliação da qualidade do software e na identificação de áreas que precisam de melhorias.

Outro aspecto importante da Garantia de Qualidade é a garantia da conformidade com padrões e regulamentações específicas da indústria, como ISO 9000 para qualidade de produtos e processos ou normas específicas para setores como saúde e finanças. Isso ajuda as organizações a manterem a qualidade e a segurança de seus produtos e serviços, além de garantir a conformidade com requisitos legais e regulatórios (ROVEDA, 2022).

Em resumo, a Garantia de Qualidade desempenha um papel fundamental no desenvolvimento de software moderno, garantindo que os produtos sejam desenvolvidos de acordo com padrões de qualidade rigorosos e atendam às expectativas dos clientes e às necessidades do mercado (BROWN, 2020).

2.3 Testes Automatizados

Nesta seção, aborda-se a automação de testes como uma prática essencial no desenvolvimento de software, proporcionando eficiência e confiabilidade superiores em comparação aos métodos manuais. Por meio de ferramentas especializadas e frameworks, equipes de QA podem automatizar uma ampla gama de testes, desde regressão até desempenho e segurança, o que melhora significativamente a cobertura de testes e a qualidade do produto final.

A automação de testes desempenha um papel crucial no desenvolvimento de software, permitindo a execução eficiente e repetitiva de casos de teste sem intervenção humana direta. Enquanto os testes manuais são essenciais para a validação inicial de um produto, a automação oferece uma vantagem significativa em termos de eficiência e confiabilidade. Ao empregar ferramentas e frameworks dedicados, as equipes de QA podem automatizar uma ampla gama de testes, desde testes de regressão até testes de desempenho e segurança. Essa abordagem não apenas reduz o esforço manual e o tempo necessário para executar testes repetitivos, mas também aumenta a cobertura de teste e a confiabilidade dos resultados (PRESSMAN; MAXIM, 2014).

Os testes automatizados são projetados para simular interações humanas com o software, permitindo validar sua funcionalidade em diferentes cenários e condições. Por meio de scripts e casos de teste automatizados, é possível verificar sistematicamente se o software se comporta conforme o esperado, identificando assim falhas e inconsistências de forma rápida e precisa. Além disso, a automação de testes possibilita a execução de testes em larga escala e a integração contínua, promovendo uma abordagem ágil para o desenvolvimento de software e garantindo a entrega de produtos de alta qualidade de maneira eficiente (SOMMERVILLE, I., 2015).

A adoção de testes automatizados não apenas melhora a eficiência dos processos de teste, mas também contribui para a redução de custos e o aumento da satisfação do cliente. Ao detectar e corrigir defeitos precocemente no ciclo de desenvolvimento, as equipes podem evitar atrasos e retrabalhos, garantindo assim a entrega de software dentro do prazo e do orçamento estabelecidos. Além disso, a automação de testes permite que as equipes se concentrem em atividades de maior valor agregado, como a análise de requisitos e a melhoria contínua dos processos de desenvolvimento de software (SOMMERVILLE, I., 2015).

2.3.1 *Software In The Loop* (SIL)

O Software in the Loop (SIL) é uma técnica de teste automatizado que desempenha um papel crucial no desenvolvimento de sistemas embarcados e software de controle. Esta técnica permite que o código de aplicação seja executado em um ambiente de simulação que replica o comportamento do hardware real. Isso possibilita uma validação inicial e detalhada do software, facilitando a identificação de erros e falhas antes que o software seja implementado no hardware físico. A utilização de simuladores avançados permite que desenvolvedores realizem testes extensivos sem a necessidade de protótipos físicos, economizando tempo e recursos (ZHOU; LIU; WANG, 2010).

Uma das principais vantagens do SIL é a capacidade de detectar erros em estágios iniciais do desenvolvimento. Ao identificar e corrigir problemas precocemente, a técnica reduz significativamente os custos e o tempo necessários para ajustes posteriores. Além disso, o SIL permite a criação de um ambiente controlado onde cenários de teste específicos podem ser repetidos de forma consistente. Isso é particularmente útil para simular condições extremas ou raras que seriam difíceis ou perigosas de reproduzir em hardware real.

No entanto, o SIL também possui algumas limitações. A fidelidade da simulação é uma delas, pois o simulador pode não capturar todas as nuances e comportamentos do hardware real. Isso pode levar a discrepâncias entre os resultados obtidos na simulação e os testes realizados no hardware físico. Além disso, certos aspectos de desempenho, como a temporização precisa e a resposta a condições extremas, só

podem ser avaliados de maneira precisa em hardware real. Portanto, o SIL deve ser complementado com testes em hardware quando necessário para garantir a precisão e confiabilidade dos resultados.

Em termos de aplicações, o SIL é amplamente utilizado em diversas indústrias, como a automotiva, aeroespacial e de automação industrial. Na indústria automotiva, por exemplo, o SIL é essencial para a validação de sistemas de controle de veículos, como ABS e controle de tração. No setor aeroespacial, ele é empregado para testar software de controle de voo e navegação, enquanto na automação industrial, é utilizado para a verificação de sistemas de controle de processos e robótica. A versatilidade e eficiência do SIL fazem dele uma ferramenta indispensável para o desenvolvimento de sistemas críticos.

2.3.2 *Hardware In The Loop* (HIL)

O conceito de Hardware in the Loop (HIL) é amplamente utilizado em testes automatizados para simular as condições reais de operação de sistemas embarcados, particularmente no setor automotivo e aeroespacial. A técnica HIL permite a integração de componentes de hardware real dentro de um ambiente de simulação controlada, permitindo que o sistema sob teste (System Under Test - SUT) interaja com modelos virtuais que imitam o comportamento de outras partes do sistema que ainda não estão disponíveis fisicamente. Dessa forma, é possível realizar testes rigorosos e precisos antes que todos os componentes reais estejam disponíveis, acelerando o desenvolvimento e melhorando a qualidade dos produtos finais (MAHMOOD *et al.*, 2021).

No contexto de testes automatizados, o HIL é fundamental para validar o desempenho e a robustez dos sistemas embarcados sob uma variedade de cenários operacionais que seriam difíceis ou perigosos de replicar em testes físicos completos. Através da simulação de entradas e saídas, os engenheiros podem observar como o hardware real responde a diferentes condições sem a necessidade de construir um protótipo completo, economizando tempo e recursos. Por exemplo, no desenvolvimento de veículos autônomos, o HIL é usado para testar sistemas de controle de direção e frenagem em situações de emergência simuladas, garantindo a segurança e a confiabilidade do sistema (GAO *et al.*, 2017).

Além disso, o uso de HIL em testes automatizados permite a detecção precoce de falhas e a correção de erros de design, aumentando a eficiência do ciclo de desenvolvimento. As ferramentas de HIL fornecem um ambiente de teste repetível e controlável, onde é possível executar uma série de testes automatizados continuamente, identificando problemas que poderiam passar despercebidos em testes menos rigorosos. Esse nível de automação é particularmente útil na validação de software embarcado, onde pequenas mudanças no código podem ter impactos significativos no

comportamento do sistema (SCHUH *et al.*, 2014).

Em suma, o HIL desempenha um papel crucial na modernização dos testes automatizados, oferecendo uma abordagem híbrida que combina o realismo dos testes de hardware com a flexibilidade das simulações virtuais. A capacidade de realizar testes complexos e variados de forma eficiente e segura torna o HIL uma ferramenta indispensável na engenharia de sistemas embarcados contemporâneos (CUI *et al.*, 2020).

2.4 Protocolo CAN

Nesta seção discute-se o protocolo CAN (Controller Area Network), desenvolvido pela Robert Bosch GmbH nos anos 1980 para otimizar a comunicação entre sistemas eletrônicos em veículos, reduzindo cabeamento e custos, enquanto utiliza uma arquitetura de barramento que assegura a integridade dos dados em ambientes com alta interferência eletromagnética.

O protocolo CAN (Controller Area Network) foi desenvolvido pela Robert Bosch GmbH no início dos anos 1980. O objetivo inicial era atender à crescente demanda por comunicação eficiente e robusta entre diversos sistemas eletrônicos em veículos automotivos. Com a evolução dos carros, que passaram a incorporar cada vez mais componentes eletrônicos, como unidades de controle de motor, sistemas de freio, air bags, entre outros, tornou-se essencial criar um protocolo que permitisse a comunicação confiável entre esses módulos, minimizando a quantidade de cabeamento e os custos associados (CORRIGAN, 2022).

Trata-se de um sistema de comunicação serial que utiliza uma arquitetura baseada em barramento, permitindo que múltiplos dispositivos se conectem a um único canal de comunicação. Um dos principais atributos do CAN é sua capacidade de operar em ambientes com alto nível de interferência eletromagnética, mantendo a integridade dos dados transmitidos.

O CAN opera em dois modos principais: CAN padrão (ou CAN 2.0A) e CAN extendido (ou CAN 2.0B). A diferença entre esses modos está no comprimento do identificador de mensagem, que é de 11 bits no modo padrão e de 29 bits no modo extendido. Esses identificadores são cruciais para a arbitragem de mensagens no barramento, onde o protocolo emprega um método de arbitragem por contenção não-destrutiva. Esse mecanismo garante que a mensagem com o identificador mais alto tenha prioridade no barramento e que não ocorra perda de dados durante a resolução de conflitos.

As mensagens CAN são compostas por um campo de início de quadro, identificador de mensagem, controle, dados, CRC (Cyclic Redundancy Check) e campo de confirmação. O campo CRC é utilizado para verificar a integridade dos dados transmitidos, enquanto o campo de confirmação assegura que a mensagem foi recebida

corretamente pelos nós destinatários.

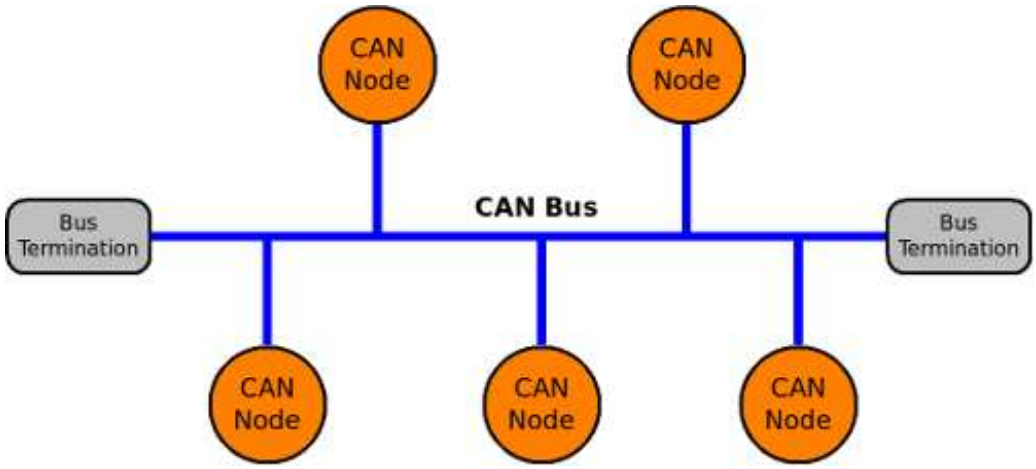


Figura 3 – Topologia da rede CAN

A Figura 3 representa a topologia de uma rede CAN, onde são visíveis dois retângulos cinzas rotulados como "Bus Termination"e várias bolas laranjas identificadas como "CAN Node". Esses elementos estão interconectados por linhas azuis denominadas "CAN Bus". A presença dos retângulos de terminação de barramento sugere que a rede CAN está configurada para minimizar reflexões de sinal e garantir a integridade da comunicação dentro do sistema. As bolas laranjas representam nós individuais na rede, cada um capaz de transmitir e receber mensagens conforme o protocolo CAN. Essa topologia é comumente utilizada em aplicações automotivas e industriais devido à sua robustez e capacidade de suportar ambientes adversos e sistemas distribuídos.

Além disso, o protocolo CAN suporta dois tipos principais de frames: frames de dados e frames de controle. Frames de dados contêm informações que são transmitidas entre os módulos, enquanto frames de controle são usados para propósitos de diagnóstico e gerenciamento de rede (BOSCH, 1991).

Um dos aspectos mais inovadores do CAN é seu mecanismo de detecção e tratamento de erros, que inclui cinco tipos diferentes de erro: erro de bit, erro de redundância cíclica, erro de preenchimento, erro de forma e erro de reconhecimento. Quando um nó detecta um erro, ele transmite um frame de erro, interrompendo a transmissão atual e iniciando o processo de retransmissão, garantindo assim a integridade da comunicação.

Em síntese, o protocolo CAN é uma solução altamente eficiente e robusta para a comunicação entre sistemas embarcados, destacando-se pela sua capacidade de operar em ambientes adversos e pela eficiência no gerenciamento de comunicação entre múltiplos dispositivos. Sua adoção ampla em diversas indústrias ressalta sua importância e versatilidade.

2.5 Gerenciamento de Versionamento de Código com Git

Na presente seção, é enfatizado o papel central do Git no versionamento de código, destacando sua capacidade robusta de gerenciar ramificações e fusões de maneira eficiente. Essa abordagem distribuída não apenas aumenta a segurança e flexibilidade no controle de versões, mas também facilita auditorias e revisões detalhadas do histórico de desenvolvimento do projeto.

O versionamento de código é uma prática essencial no desenvolvimento de software moderno, proporcionando controle sobre as modificações no código-fonte e facilitando a colaboração entre equipes de desenvolvimento. O Git, uma das ferramentas de controle de versão mais populares e amplamente adotadas, desempenha um papel crucial nesse contexto. Desenvolvido por Linus Torvalds em 2005, o Git oferece um sistema distribuído de controle de versões, permitindo que desenvolvedores gerenciem e acompanhem mudanças no código de maneira eficiente e segura (CHACON; STRAUB, 2014).

Uma das principais vantagens do Git é sua capacidade de suportar ramificações (branches) e fusões (merges) de maneira robusta. As ramificações permitem que os desenvolvedores trabalhem em funcionalidades novas ou correções de bugs de forma isolada do código principal, sem interromper o trabalho dos outros membros da equipe. Isso possibilita um fluxo de trabalho mais ágil e organizado, onde diferentes linhas de desenvolvimento podem convergir em uma versão estável do projeto. A fusão de branches no Git é facilitada por suas ferramentas avançadas de merge, que ajudam a resolver conflitos de maneira eficaz (LOELIGER; MCCULLOUGH, 2012).

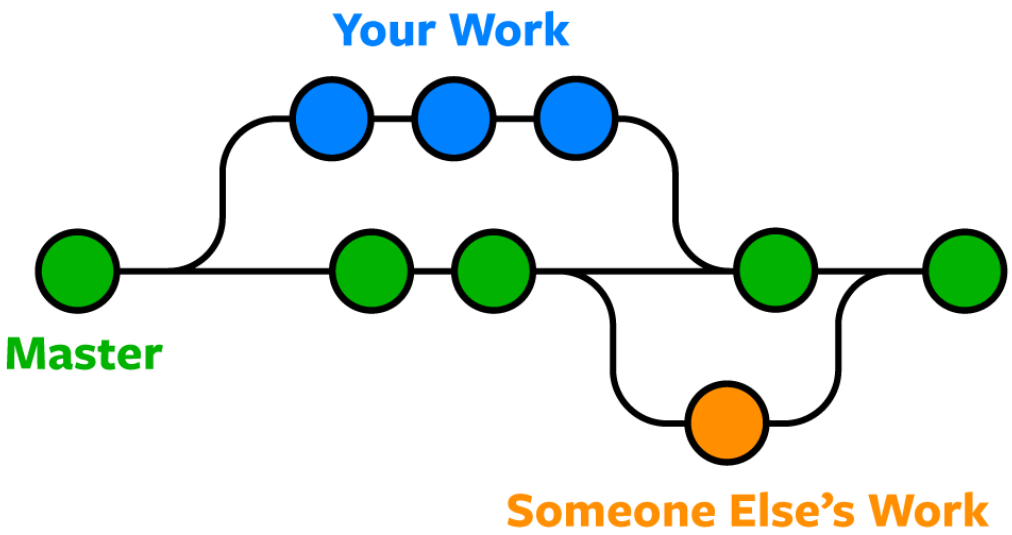


Figura 4 – Ilustração do fluxo de trabalho com git. Fonte: Github

Na Figura 4, podemos observar um fluxo de trabalho típico utilizando o Git. A imagem ilustra três ramos principais: o ramo "Master"(em verde), que representa o estado principal do projeto; o ramo "Your Work"(em azul), que simboliza o trabalho

de um desenvolvedor específico; e o ramo "Someone Else's Work"(em laranja), que representa o trabalho de outro desenvolvedor. O fluxo começa com uma ramificação do ramo "Master", onde cada desenvolvedor trabalha em suas mudanças de forma isolada. Os círculos azuis e laranjas mostram commits feitos por diferentes desenvolvedores em seus respectivos ramos. Eventualmente, esses ramos se unem novamente ao ramo "Master"após a conclusão do trabalho, através de processos de merge, integrando as mudanças de ambos os desenvolvedores no projeto principal. Este modelo de ramificação e fusão permite um desenvolvimento paralelo eficiente e uma integração controlada das modificações ao projeto central.

Além disso, o Git oferece um histórico completo e detalhado de todas as mudanças feitas no repositório, incluindo quem fez a alteração e quando. Isso é extremamente útil para auditorias e revisões de código, pois permite rastrear a origem de problemas ou identificar contribuições específicas. Comandos como git log e git blame são ferramentas poderosas nesse sentido, fornecendo uma visão clara e detalhada do histórico de desenvolvimento do projeto (MASON, 2021).

Outra característica fundamental do Git é seu modelo distribuído. Ao contrário de sistemas de controle de versão centralizados, como o Subversion (SVN), cada desenvolvedor possui uma cópia completa do repositório, incluindo todo o histórico de mudanças. Isso não só aumenta a redundância e segurança dos dados, como também permite que os desenvolvedores trabalhem offline e sincronizem suas alterações quando estiverem conectados. Esse modelo distribuído melhora a resiliência do projeto e a flexibilidade no fluxo de trabalho (BIRD *et al.*, 2010).

Em resumo, o Git se destaca como uma ferramenta indispensável para o gerenciamento de versionamento de código devido à sua eficiência, flexibilidade e robustez. Sua capacidade de lidar com ramificações e fusões complexas, junto com um histórico detalhado e um modelo distribuído, torna-o a escolha preferida de muitas equipes de desenvolvimento em todo o mundo. Ao adotar o Git, as equipes podem melhorar significativamente sua produtividade e a qualidade de seus projetos de software (SPINELLIS, 2012).

2.6 Behavior Driven Development

Nesta seção é abordada a técnica de desenvolvimento guiado por comportamento (BDD), introduzida por Dan North em 2003 para promover a colaboração eficaz na definição e validação de requisitos de software através de exemplos concretos e testes automatizados.

A técnica de desenvolvimento guiado por comportamento (do inglês Behavior Driven Development, ou apenas BDD) foi criada em 2003 por Dan North, com o objetivo de aprimorar a comunicação dentro de um time de desenvolvimento de software por meio da colaboração entre desenvolvedores, setores de qualidade e stakeholders

não técnicos (NORTH, 2006). O surgimento do BDD se deu como uma evolução do desenvolvimento orientado a testes (TDD ou Test Driven Development), visando utilizar uma linguagem mais simples e acessível para todos os envolvidos no projeto. Essa abordagem colaborativa permite um entendimento mais claro do escopo dos testes de software tanto por partes técnicas quanto não técnicas.

A abordagem do BDD é estruturada em torno de três práticas principais: descoberta, formulação e automação (CUCUMBER.IO, 2024). A descoberta envolve a colaboração entre todas as partes interessadas para identificar e entender os requisitos do sistema por meio de exemplos concretos. Esses exemplos são utilizados para esclarecer o comportamento desejado do software em situações específicas, garantindo que todos os membros da equipe possuam uma visão compartilhada das funcionalidades a serem desenvolvidas.

A formulação é o processo de transformar os exemplos identificados durante a descoberta em especificações claras e compreensíveis. Essa etapa é crucial para garantir que a documentação dos requisitos seja acessível tanto para desenvolvedores quanto para stakeholders não técnicos. Ferramentas como o Cucumber são frequentemente utilizadas para escrever essas especificações em uma linguagem simples e estruturada, como o Gherkin, que permite a descrição de comportamentos em formato "dado-quando-então".

A automação, por sua vez, envolve a conversão das especificações formuladas em testes automatizados. Esses testes são executados regularmente para validar que o software atende aos requisitos definidos, proporcionando feedback contínuo e permitindo a detecção precoce de problemas. A integração da automação com o BDD não apenas melhora a qualidade do software, mas também facilita a manutenção e a evolução do sistema ao longo do tempo. Essa abordagem colaborativa e iterativa promove uma entrega mais eficiente e confiável de software, alinhada com os objetivos de negócio.

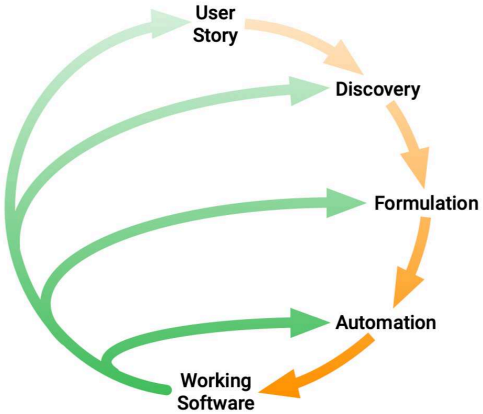


Figura 5 – Representação do processo do BDD: cucumber.io/docs/bdd/

2.7 Linguagem *Gherkin*

Na presente seção, é explorado o uso da linguagem *Gherkin* para a escrita de testes automatizados, enfatizando sua padronização e legibilidade dos cenários de teste para facilitar a compreensão universal. Também são discutidos os princípios do BDD, utilizando palavras-chave como **Dado**, **Quando**, **Então** e **E**, para detalhar passos específicos e esperados na validação das funcionalidades do software.

Para escrever os testes automatizados, pode-se utilizar a linguagem *Gherkin*, que é uma linguagem de programação criada com o propósito de descrever processos utilizando algumas palavras-chave, tendo como foco a padronização de escrita de cenários de testes, facilitando sua leitura e compreensão para qualquer pessoa (OLIVEIRA, 2020).

Para a escrita de cenários, o BDD propõe a utilização de 4 palavras chave, sendo elas **GIVEN**, **WHEN**, **THEN** e **AND** (Em português **Dado**, **Quando**, **Então** e **E**). A partir destas palavras podemos realizar a escrita de um cenário de teste dividindo-o em blocos.

```
Feature: User Authentication

    Background:
        Given the system is initialized

    Scenario: Login with valid credentials
        Given the user is on the login page
        When the user enters the username "user123" and the
           password "password123"
        And  press the login button
        Then the user is successfully authenticated
```

Figura 6 – Exemplo de código Gherkin: desenvolvido pelo autor

A Figura 6 mostra um exemplo utilizando a estrutura de escrita proposta pelo *Gherkin*, onde neste caso temos um cenário fictício de autenticação de usuário. O campo *Feature* nos indica o que o teste escrito se propõe, por exemplo "Autenticação de usuário". O passo de *background* serve para inserir passos que são comuns a mais de um cenário, evitando a reescrita do mesmo diversas vezes, como por exemplo, o sistema ser inicializado, algo que será feito para qualquer cenário proposto para a *feature* da Figura 6. Desta forma, os cenários (*Scenario*) terão apenas os passos específicos para sua execução, seguindo a lógica das 4 palavras chave descritas anteriormente. Cada cenário descreverá um fluxo específico a ser seguido para a validação da *feature* e seu resultado esperado. Por exemplo, temos um cenário de *login* com credenciais válidas, resultando em um *login* bem-sucedido. Entretanto, também poderíamos ter outro cenário que valida o fluxo de falha na tentativa de *login*, como no caso de credenciais inválidas ou de um usuário não cadastrado.

Vale ressaltar que a redação dos cenários de validação deve estar estritamente alinhada com as regras de negócio do produto a ser testado, bem como com os critérios de aceitação propostos para o time de desenvolvimento. É necessário o alinhamento de todas as partes para garantir que a validação seja realizada corretamente.

2.8 Cucumber Framework

Nesta seção, exploramos a integração do Cucumber com C++, uma implementação que possibilita aos desenvolvedores escreverem testes automatizados utilizando a metodologia Behavior Driven Development (BDD). O Cucumber-Cpp é a ferramenta central dessa integração, permitindo que especificações de comportamento sejam descritas em linguagem natural e verificadas diretamente no código C++. Essa integração se baseia no Protocolo Wire, que facilita a comunicação entre a implementação em C++ e o Cucumber em Ruby através de um servidor TCP. Esse servidor atua como uma ponte essencial, permitindo que o Cucumber execute testes e obtenha resultados diretamente da aplicação C++, promovendo uma abordagem eficiente e alinhada com as práticas ágeis de desenvolvimento de software.

Cucumber é uma ferramenta de código aberto amplamente utilizada para apoiar o desenvolvimento orientado por comportamento (BDD - Behavior Driven Development). Criada em 2008, o Cucumber foi desenvolvido para possibilitar a criação de testes automatizados que descrevem o comportamento esperado do software em uma linguagem natural e acessível para todos os envolvidos no projeto, desde desenvolvedores até stakeholders não técnicos (SMARTBEAR, 2024).

A ferramenta surgiu como uma extensão do framework RSpec para Ruby, mas rapidamente evoluiu para suportar diversas linguagens de programação, incluindo Java, JavaScript, e .NET, o que a tornou flexível e aplicável em diferentes ambientes de desenvolvimento. A principal característica do Cucumber é a utilização da linguagem Gherkin, que permite escrever especificações de comportamento em uma estrutura simples e legível. Essas especificações podem ser diretamente executadas como testes automatizados, garantindo que o software desenvolvido atenda aos requisitos definidos de maneira precisa.

O uso do Cucumber se destaca em equipes que adotam metodologias ágeis e BDD. A ferramenta facilita a comunicação e a colaboração entre todos os membros da equipe de desenvolvimento, reduzindo mal-entendidos e retrabalho. Ao permitir que testes sejam escritos em uma linguagem acessível, como "Dado, Quando, Então", Cucumber promove um entendimento compartilhado das funcionalidades do software entre desenvolvedores, testadores e partes interessadas.

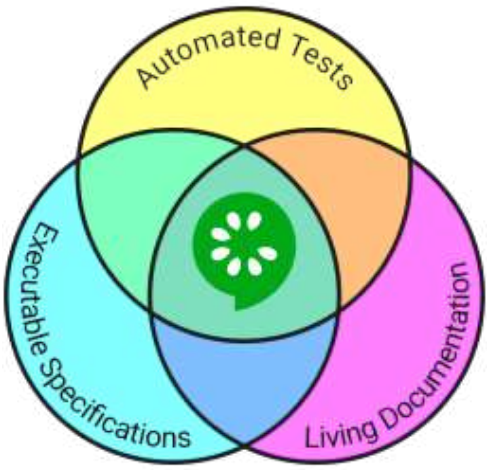


Figura 7 – Onde o cucumber se encontra: cucumber.io/docs/guides/overview

Na Figura 7, há um diagrama de Venn que representa os três principais componentes do framework Cucumber: Testes Automatizados, Especificações Executáveis e Documentação Viva. O ponto de interseção central, onde os três círculos se sobrepõem, simboliza o Cucumber em si, destacando como ele unifica esses elementos. Testes Automatizados refere-se à capacidade de executar testes de forma automatizada, Especificações Executáveis indica que as especificações podem ser executadas como testes, e Documentação Viva enfatiza que a documentação está sempre atualizada e refletindo o estado atual do sistema. Essa convergência assegura uma fonte única de verdade para todos os envolvidos no desenvolvimento de software.

No ciclo de desenvolvimento BDD, o Cucumber é implementado através de três etapas iterativas: Descoberta, Formulação e Automação. Na fase de Descoberta, a equipe discute e identifica exemplos concretos do comportamento desejado do sistema, promovendo uma compreensão clara dos requisitos. Durante a Formulação, esses exemplos são documentados em uma forma que pode ser automatizada. Finalmente, na etapa de Automação, os exemplos documentados são usados para guiar o desenvolvimento e os testes do software, garantindo que o comportamento esperado seja implementado corretamente.

Cucumber também oferece integração com diversas outras ferramentas e plataformas, como Selenium para testes de interface de usuário e sistemas de controle de versão como Git. Além disso, ele gera uma documentação viva que está sempre atualizada com o estado atual do software, facilitando a manutenção e a transparência do projeto.

A adoção do Cucumber por empresas de diferentes tamanhos tem demonstrado melhorias significativas na qualidade e velocidade do desenvolvimento de software. Por exemplo, a Flowbird utilizou o CucumberStudio, uma versão comercial do Cucumber, para um grande projeto na cidade de Helsinque, e a Ouest France empregou a ferramenta para eliminar hotfixes e melhorar a qualidade geral do seu website.

Em suma, Cucumber se consolidou como uma ferramenta essencial no desenvolvimento ágil, oferecendo meios para que as equipes entreguem software de alta qualidade alinhado com as necessidades dos negócios e dos usuários finais.

2.8.1 Integração com C++

A integração do Cucumber com C++ permite que desenvolvedores escrevam testes automatizados para aplicações desenvolvidas nessa linguagem, utilizando a metodologia Behavior Driven Development (BDD). O Cucumber-Cpp é a ferramenta que viabiliza essa integração, possibilitando que as especificações de comportamento do software sejam escritas em uma linguagem natural e verificadas diretamente no código C++.

Para compreender como o Cucumber-Cpp funciona, é importante entender alguns conceitos básicos:

1. **Protocolo Wire**:
   Este protocolo permite a comunicação entre a implementação em C++ e o Cucumber, que originalmente foi desenvolvido em Ruby. O Wire Protocol define como o Cucumber em Ruby se conecta a um servidor TCP onde o código C++ está escutando. Essencialmente, ele permite que o Cucumber-Ruby envie comandos e receba respostas da aplicação em C++.

2. **Servidor TCP**:
   Um servidor TCP (Transmission Control Protocol) é um componente de software que escuta por conexões de rede. No contexto do Cucumber-Cpp, ele permite que a aplicação em C++ receba instruções do Cucumber-Ruby, execute os testes e retorne os resultados.

3 DESCRIÇÃO DO PROBLEMA E REQUISITOS TÉCNICOS

O presente capítulo aborda a descrição do problema e os requisitos técnicos associados ao desenvolvimento e à validação de sistemas automatizados da Hexagon Agriculture. Inicialmente, a Seção 3.1 apresenta a empresa e seu histórico, destacando sua especialização em soluções para a agricultura de precisão. Na Seção 3.2 são descritos os produtos principais, como o sistema de orientação e piloto automático, que utiliza tecnologia de posicionamento por satélite e navegação inercial para garantir precisão nas operações agrícolas.

São detalhados os desafios enfrentados no desenvolvimento do software e firmware desses sistemas, bem como o processo de validação realizado pela equipe de Quality Assurance. O capítulo enfatiza a importância dos testes de regressão para garantir a segurança e a robustez do piloto automático, além dos esforços para automatizar esses testes, a fim de reduzir o tempo de validação e minimizar falhas humanas.

3.1 Sobre a Empresa

Nesta seção, é apresentado um panorama sobre a Hexagon Agriculture, destacando sua especialização em soluções inovadoras para aumentar a eficiência e produtividade na agroindústria, além de sua trajetória desde a fundação como Arvus Tecnologia em 2007 até a integração com a Hexagon AP.

Hexagon Agriculture é uma divisão da empresa global Hexagon A&P (*Autonomy & Positioning*), especializada no desenvolvimento de soluções inovadoras que aumentam a eficiência e a produtividade através da automação e melhor utilização de recursos para clientes da agroindústria. Suas tecnologias otimizam processos ao longo de todo o ciclo de produção agrícola e florestal, adaptando-se às necessidades específicas de cada cliente.

Originalmente conhecida como Arvus Tecnologia, a empresa foi fundada em 2007 por três alunos do curso de Engenharia de Controle e Automação. Em 2014, a Hexagon consolidou sua presença no mercado agrícola ao integrar a Arvus Tecnologia, ILab Sistemas e a vertical de agricultura da Hexagon Geosystems, criando assim sua divisão de Agricultura.

A Hexagon Agriculture estabeleceu-se como um nome de referência em soluções de Agricultura de Precisão, oferecendo produtos para o gerenciamento e controle de processos em toda a cadeia de plantio na indústria agrícola. Seus produtos atendem tanto culturas de grãos, como soja e milho, quanto segmentos especializados, como cana-de-açúcar e silvicultura.

Entre seus produtos mais comercializados destaca-se o sistema de orientação e piloto automático, reconhecido por sua precisão de aproximadamente 2 centímetros. Esse sistema garante um alinhamento exato e reduz a sobreposição durante operações

de plantio, aplicação de insumos e tratos culturais.

Entre os diversos clientes da Hexagon Agriculture, destacam-se parceiros estratégicos como a MP Agro, fabricante de implementos agrícolas e parceira OEM (Original Equipment Manufacturer). A colaboração específica com a MP Agro envolve a venda de computadores de bordo da Hexagon, que são integrados às máquinas da MP Agro, resultando em soluções completas e integradas. Além disso, a Hexagon Agriculture atende empresas de renome como Raízen e Klabin, oferecendo soluções personalizadas para atender às necessidades específicas de cada cliente.

Para resolver problemas técnicos enfrentados pelos clientes, seja em relação ao desempenho dos produtos ou ao mau funcionamento de componentes, a Hexagon dispõe de diversas equipes de suporte técnico distribuídas pelo Brasil e pela Europa, especializadas em agricultura de plantios padrão, florestal e cana-de-açúcar.

3.2 Computador de Bordo

Na presente seção, detalha-se o papel crucial do computador de bordo *Display* da Hexagon Agriculture, que atua como controlador central integrando Agricultura de Precisão com o sistema das máquinas agrícolas e supervisionando múltiplos processos simultâneos, como fertilização e plantio.

Para gerenciar todos os processos dos produtos disponibilizados pela empresa, a Hexagon utiliza um computador de bordo, instalado geralmente no painel da cabine do trator, de modo que fique de fácil acesso ao operador da máquina. Através de portas de comunicação CAN, o computador manda os sinais de controle para o sistema hidráulico do trator.

O computador de bordo comercializado pela Hexagon Agriculture, denominado *Display*, desempenha a função de controlador central para os processos agrícolas executados no sistema trator-implemento. De agora em diante, referenciaremos esse dispositivo como *Display*. Ele integra os princípios da Agricultura de Precisão com as capacidades do sistema hidráulico do trator, aliando-se às especificidades do implemento utilizado para monitorar, controlar e fornecer uma interface supervisória ao usuário para uma variedade de processos de campo, como preparo de solo, plantio, fertilização, aplicação de defensivos agrícolas e colheita realizados no campo.

Além disso, o *Display* sincroniza o funcionamento de máquinas em situações específicas em que várias delas são utilizadas simultaneamente, como no processo de alocação de transbordo durante a colheita de cana-de-açúcar. Nesse cenário, um trator principal colhe a plantação e transfere o produto para outro trator durante o processo de colheita, e o *Display* realiza o monitoramento do nível do trator de transbordo.

O *Display* também gerencia informações operacionais, como a taxa atual de aplicação de defensivos agrícolas, ou fornece dados sobre áreas já tratadas, os quais são enviados para o sistema de Cloud oferecido pela Hexagon Agriculture. Dessa

forma, as informações ficam acessíveis para análises subsequentes.

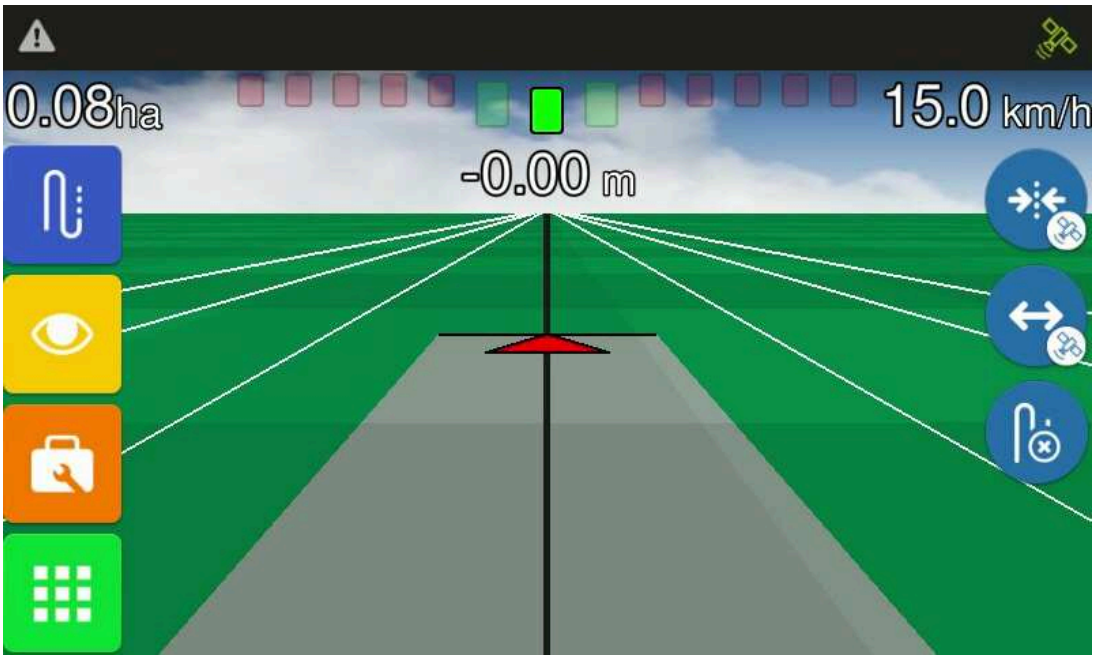


Figura 8 – Tela inicial do *Display* Ti5

Na Figura 8 é exposta a tela de operação do *Display*. Nela, o trator é representado pela seta vermelha, as linhas pretas que tocam o horizonte são as guias para o piloto automático, as quais seão geradas a partir do movimento do veículo em conjunto com a determinação de alguns pontos no mapa. As guias podem ser linhas retas, curvas ou traçados totalmente adaptados ao local feito, sendo gerada uma guia principal em preto e 3 paralelas brancas de cada lado do veículo, sendo geradas novas paralelas a medida que o veículo muda sua guia principal. O objetivo das guias é servir como referência a ser seguida pelo veículo no momento em que o piloto automático é ativado, possibilitando que sejam exploradas também as paralelas da guia gerada. A área pintada de cinza é a informação de onde já foi aplicado o fertilizante, o número no topo direito indica a velocidade do trator, o número no centro logo acima do horizonte indica o erro de seguimento de guia.

O *Display* é oferecido em três modelos diferentes: Ti5, Ti7 e Ti10. A diferença entre eles é o tamanho da tela, poder de processamento e quantidade de portas de comunicação USB e CAN. O valor do produto se destaca por ser um ponto central na operação local da agroindústria. O *Display* detém a propriedade de automatizar qualquer processo, seja ele o preparo de solo, o plantio, a aplicação de insumos, a colheita, entre outros, e fornecer ao usuário informações relevantes em tempo real sobre o mesmo.

3.3 Piloto Automático

Na presente seção, destaca-se o sistema HxGN AgrOn Piloto Automático da Hexagon, que oferece precisão excepcional na navegação de máquinas agrícolas e florestais, minimizando sobreposições e compactação do solo para otimizar as operações no campo. Utilizando tecnologia avançada de posicionamento por satélite e navegação inercial, o Piloto Automático garante um controle contínuo e preciso dos movimentos das máquinas, aumentando significativamente a produtividade ao manter alinhamento exato e eficiente dos equipamentos, adaptando-se bem a terrenos desafiadores e condições adversas de visibilidade.

3.3.1 HxGN AgrOn Piloto Automático

O HxGN AgrOn Piloto Automático da Hexagon, que neste documento será referenciado de agora em diante como *Auto Steering*, é uma solução inovadora para a navegação de máquinas agrícolas e florestais. Este sistema proporciona uma precisão excepcional, minimizando sobreposições e compactação do solo, aspectos essenciais para a eficiência das operações no campo. Utilizando uma combinação de posicionamento por satélite e navegação inercial, o *Auto Steering* garante um controle preciso e contínuo dos movimentos das máquinas (HEXAGON, 2024a).

Figura 9 – Uso real do *Auto Steering* operando com o motor *Track Controller*.

Na Figura 9 podemos notar o *display* Ti5 no canto superior direito operando o produto *Auto Steering* da Hexagon juntamente com o motor do piloto elétrico acoplado ao volante do veículo (*Track Controller*), mostrando um caso real do uso destes dois produtos em conjunto.

Um dos maiores benefícios deste sistema é o aumento da produtividade nas operações agrícolas. Com o HxGN AgrOn, os operadores podem manter um alinhamento exato das máquinas, resultando em uso eficiente dos recursos e redução dos custos operacionais. O sistema é projetado para operar em diversas condições, oferecendo desempenho superior mesmo em terrenos desafiadores e com baixa visibilidade.

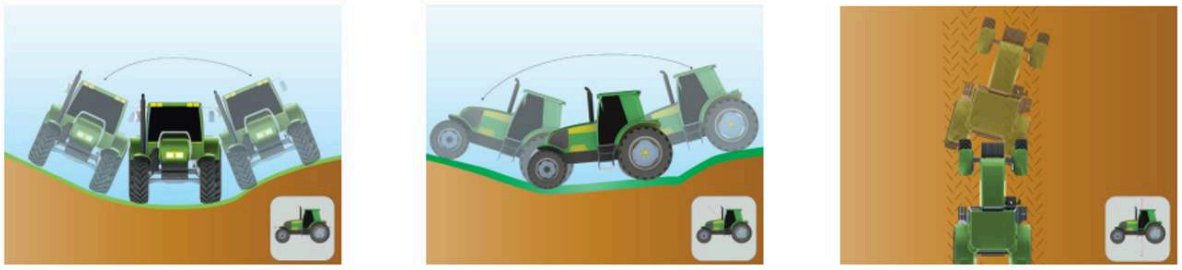


Figura 10 – Posicionamento do *Auto Steering*.Ilustração: Hexagon

A Figura 10 representa a capacidade de identificação de posicionamento que o sistema é capaz de detectar, incluindo 3 eixos de rotação que são combinados com posicionamento via satélite e com outros sensores presentes na máquina, com o objetiv de melhorar a atuação do *Auto Steering* em campo.

Além disso, o *Auto Steering* se destaca pela facilidade de uso e integração com outras funcionalidades dos equipamentos. Com várias opções de *Display* e acesso remoto para suporte técnico, a resolução de problemas e manutenção do sistema tornam-se tarefas simples e rápidas. A precisão do HxGN AgrOn é garantida por diferentes configurações de GNSS, adaptando-se às necessidades específicas de cada operação.

A tecnologia do *Auto Steering* representa um avanço significativo na agricultura moderna, oferecendo uma combinação de precisão, eficiência e praticidade. Este sistema reafirma o compromisso da Hexagon em fornecer soluções tecnológicas avançadas que aumentam a produtividade e a sustentabilidade nas operações agrícolas.

3.3.2 HxGN AgrOn Track Controller

O *Track Controller* da Hexagon é uma solução avançada destinada ao controle de veículos agrícolas, visando otimizar operações de campo. Este sistema oferece uma precisão robusta no direcionamento, o que é fundamental para maximizar a eficiência e a produtividade das atividades agrícolas. A precisão do Track Controller permite a redução de sobreposições e espaços não tratados, o que se traduz em uma utilização mais eficiente dos insumos e em uma maior uniformidade nas operações (HEXAGON, 2024b).

Figura 11 – Motor do Track Controller com volante.Ilustração: Hexagon

A Figura 11 mostra o motor do *Track Controller* em conjunto com o volante acoplado a ele, sendo este o motor que substitui o sistema de volante original do veículo para realizar o controle das rodas.

Uma das principais características do Track Controller é sua compatibilidade com diferentes tipos de veículos agrícolas, desde tratores até colheitadeiras. Esta flexibilidade torna o sistema uma escolha atraente para diversas operações agrícolas, independentemente do tipo ou tamanho do maquinário utilizado. Além disso, o sistema pode ser integrado com outras soluções da Hexagon, proporcionando uma gestão mais holística das operações de campo.

O Track Controller também se destaca pela facilidade de uso. Com uma interface intuitiva, os operadores podem rapidamente configurar e ajustar o sistema conforme necessário. Esta simplicidade de operação reduz a curva de aprendizado e minimiza o tempo de inatividade, permitindo que as atividades agrícolas continuem de forma eficiente e ininterrupta. A interface amigável é um grande diferencial, especialmente em um ambiente onde a rapidez e a precisão são essenciais.

A conectividade é outro ponto forte do Track Controller. Ele permite a comunicação em tempo real com outros sistemas e dispositivos, facilitando a troca de dados e o monitoramento contínuo das operações. Esta conectividade é crucial para a agricultura de precisão, onde a coleta e análise de dados são fundamentais para a tomada de decisões informadas. A capacidade de monitorar e ajustar as operações em tempo real contribui significativamente para a eficiência e a eficácia das atividades agrícolas.

Por fim, o Track Controller contribui para a sustentabilidade das operações agrícolas. Ao melhorar a precisão e a eficiência, o sistema ajuda a reduzir o desperdício de insumos como sementes, fertilizantes e pesticidas. Isso não apenas reduz os custos operacionais, mas também minimiza o impacto ambiental da agricultura. A combinação de eficiência operacional e sustentabilidade torna o Track Controller uma solução

valiosa para os agricultores que buscam melhorar suas práticas e resultados.

3.4 O Problema Tratado

Nesta seção, são detalhados o funcionamento integrado do *Display*, *Auto Steering* e *Track Controller*, com ênfase na operação do software TiX no *Display* e do firmware no *Track Controller*, cruciais para o piloto automático e suas atualizações via sistema de gerenciamento. Adicionalmente, discute-se a adoção de testes automatizados iniciada pela Hexagon em 2023 para melhorar a eficiência e reduzir falhas nos testes de regressão do *Auto Steering*, focando na validação do *Track Controller* através de testes em SIL e HIL, visando garantir a integridade da comunicação CAN e a operação correta do sistema em condições reais.

Como mencionado anteriormente, para que tudo funcione, o *Display* opera com a ativação do *Auto Steering* juntamente com o motor elétrico *Track Controller* para garantir o funcionamento do piloto automático no que diz respeito ao seguimento de referência de diversos tipos de trajetos. Tanto o *Display* como o *Track Controller* possuem códigos próprios que possibilitam sua operação. Neste documento trataremos o sistema que opera no *Display* como **software TiX** e o código referente ao *Track Controller* como **firmware**, sendo o *software TiX* desenvolvido e instalado em um sistema embarcado com interface ao usuário, enquanto o *firmware* é desenvolvido para uma placa externa de controle do motor elétrico.

Dentro da empresa, o *software TiX* é lançado de forma anual em um formato estável e com todas as novidades desenvolvidas no ano anterior, recebendo o título de LTS, onde a sigla significa "Long Term Support"ou "Suporte de Longo Prazo"em português, sendo uma versão de software que recebe suporte e atualizações por um período mais longo do que as versões padrão. Contudo, a empresa também lança versões novas do seu *software TiX* com o objetivo de introduzir novas funcionalidades dentro destas versões e também corrigir eventuais problemas que podem ser relatados externamente por clientes ou internamente pelo time de *Quality Assurance*. Cada versão nova de *software TiX* pode carregar consigo uma versão nova do *firmware* de alguma placa que opere algum dos diversos produtos disponibilizados, estando a versão do *firmware* disponível com a nova versão do *software TiX* internamente dentro do *display*.

Para realizar a atualização do software, o usuário deve baixá-la em um dispositivo USB e inserí-la em seu *display*, acessando uma tela específica chamada "Gerenciador de Sistema", onde com o dispotivido USB plugado poderá atualizar o *display*. Uma vez que o *display* está atualizado, para atualizar a placa referente ao seu produto (no caso, referente ao *track controller*), o usuário fará essa atualização dentro do próprio *display*, o qual identificará no momento da inicialização, que a placa conectada possui uma nova versão do *firmware* para ser instalado, avisando ao usuário e

perguntando se o mesmo gostaria de realizar a atualização ou não.

Dentro do processo de desenvolvimento de novas versões de *software TiX* e *firmware*, o time de *Quality Assurance* atua realizando testes destas novas versões de sistema afim de validar que as correções foram feitas de forma correta e que os novos produtos operam de acordo com o planejado. Os testes são divididos em dois tipos:

1. Testes de Integração: Validam novas funcionalidades dentro do *software TiX*.
2. Testes de Regressão: Validam funcionalidades já existentes de forma completa.

O escopo deste trabalho está nos testes de regressão, em específico do *Auto Steering*, por se tratar de um dos produtos mais utilizados e vendidos da empresa, sendo essencial que ele seja sempre validado quando uma atualização ou correção de *bug* é lançaca, afim de garantir sua eficiência, robustez e segurança. Como é um produto que controla uma máquina, a questão da segurança é uma das partes mais importantes na validação, pois um comportamento inadequado de um piloto automático pode ocasionar a batida de um veículo e consequentemente o ferimento do operador ou de alguém próximo, por isso o teste de regressão se mostra essencial para este produto a cada novo lançamento de versão de *software TiX* e *firmware*.

O time de *Quality Assurance* fica então responsável pela realização dos testes de regressão do *auto steering* a cada lançamento de versão. Para isso, os testes são documentados utilizando a linguagem *Gherkin*, conforme explicado na Seção 2.7, sendo sempre reutilizados para evitar que cada pessoa que realize o teste o faça de forma diferente, garantindo que um padrão seja sempre seguido e que contemple a validação do produto na sua totalidade.

Até então estes testes eram feitos de forma manual na Hexagon, ou seja, ativamente feitos por um humano validando o comportamento do piloto. Isto acarreta em dois pontos principais:

1. A duração dos testes;
2. A possibildade de falhas humanas.

O primeiro ponto é crucial, pois o processo de validação exige aproximadamente quatro horas de trabalho direto da pessoa responsável. No entanto, essa atividade é apenas uma parte de uma série de tarefas que a equipe realiza semanalmente. Além disso, caso algum problema seja encontrado, ele é reportado ao time de desenvolvimento, o qual possui suas demandas semanais e irá priorizar a correção de qualquer problema de acordo com suas outras demandas. Portanto, o processo completo de validação do Auto Steering pode levar de 2 a 3 semanas, necessitando de novas validações para corrijir erros caso tenham sido encontrados, aumentando ainda mais o

tempo necessário para a validação do *auto steering* durante o processo de lançamento de uma nova versão de *software TiX*.

Já o segundo ponto está diretamente relacionado com o anterior, pois o time de QA possui outras demandas semanais e o fator humano sempre está propício a falhas, e por mais que exista um processo de teste documentado, sempre existe a possibilidade de pequenos detalhes passarem despercebidos pelo responsável do teste, o que pode resultar em um problema quando o cliente instala uma nova versão do *software TiX* e realiza os testes em campo.

Pensando em reduzir o tempo de validação e evitar falhas humanas, a Hexagon vem investindo desde 2022 no estudo e desenvolvimento de testes automatizados. A ideia é que estes testes automatizados validem de forma satisfatória um produto ao ponto de não necessitar de uma pessoa realizando os mesmos testes diversas vezes a cada lançamento de uma nova versão de *software TiX* ou *firmware*.

Tendo isso em mente, em 2023 a empresa decidiu que focaria no desenvolvimento de testes automatizados para a validação do *Auto Steering*, mais precisamente para o motor elétrico *Track Controller*, validando toda a sua comunicação CAN com o *Display* e garantindo assim seu correto funcionamento. Com isso, o time de Controle da empresa desenvolveu uma infraestrutura dentro do código existente do sistema de *Auto Steering* para comportar o *framework cucumber*, o qual, conforme dito na Seção 2.8, é uma ferramenta desenvolvida para possibilitar a criação de testes automatizados, sendo utilizado internamente para a validação do *Track Controller*. Juntamente com esta infraestrutura, foi disponibilzado pelo time de Controle um documento que especifica todas as chamdas possíveis de serem feitas para o **track controller** via CAN e como isso se comporta dentro do código, afim de possibiliar ao time de *Quality Assurance* escrever a estrutura do código necessário para a realização dos testes automatizados.

Os testes automatizados se dividem em duas vertentes:

- Testes em **SIL** (*Software-in-the-loop*);
- Testes em **HIL** (*Hardware-in-the-loop*).

Esses testes são fundamentais para garantir a qualidade e a confiabilidade dos sistemas desenvolvidos, sendo que ambos buscam validar principalmente a comunicação CAN entre o display e uma placa que controla um determinado produto.

Os testes em SIL podem ser executados toda vez que um desenvolvedor sobe uma nova alteração de código. Este processo é crucial, pois garante que, caso algum erro ocorra, essa nova versão seja impedida de entrar na versão oficial. Assim, o desenvolvedor tem a oportunidade de observar qual foi o erro e buscar uma solução antes que o código seja integrado ao produto final. Esse mecanismo de feedback imediato é essencial para garantir que nenhuma versão de software seja gerada sem que

esses testes em SIL tenham sido rodados e aprovados, o que contribui significativamente para a estabilidade e a qualidade do software desde o início do processo de desenvolvimento.

Contudo, nem todos os testes podem ser validados apenas no ambiente SIL, pois este é limitado a validações gerais de *software*, não abrangendo aspectos de *hardware* ou integração mais amplos. Para testes que envolvem interação com o *hardware* real, são implementados os testes em HIL. Os testes em HIL buscam criar uma estrutura com os componentes reais montados e conectados a um computador. A comunicação é feita entre o computador e as placas, sensores e atuadores, de forma a simular o comportamento do produto em um ambiente real, mas de maneira automatizada. Este processo permite que os testes sigam o mesmo princípio de validação do SIL, porém aplicados a um contexto mais próximo das condições de operação reais do produto.

Os testes em HIL são especialmente importantes para detectar problemas que só se manifestam quando o software interage com o hardware. Por exemplo, problemas de temporização ou de comunicação que podem não ser evidentes em um ambiente puramente de software. Assim, a implementação de testes em HIL complementa os testes em SIL, proporcionando uma cobertura de teste abrangente que abrange tanto os aspectos de software quanto de hardware do sistema. Dessa forma, garante-se que o produto final funcione corretamente em todas as condições esperadas de operação, aumentando a confiabilidade e a robustez do produto desenvolvido.

3.5 Requisitos Técnicos

Na presente seção, são delineados os requisitos técnicos fundamentais para a implementação dos testes automatizados no sistema *Auto Steering*, abrangendo a necessidade de cobertura completa das funcionalidades do *Track Controller* via CAN, a escalabilidade da infraestrutura de testes para incorporar novos casos e a garantia da confiabilidade dos resultados. Além disso, são discutidos os procedimentos de integração contínua no pipeline de desenvolvimento e a configuração dos ambientes de teste SIL e HIL, essenciais para assegurar a robustez e a precisão do sistema ao longo do ciclo de desenvolvimento.

Esta seção delineia os requisitos técnicos exigidos para implementar os testes automatizados no sistema *Auto Steering*. Inclui a cobertura necessária dos testes, escalabilidade do sistema de testes, confiabilidade dos resultados, integração contínua no pipeline de desenvolvimento e configuração dos ambientes de teste, tanto em simulações de software quanto com hardware real. Essas diretrizes são cruciais para assegurar a robustez e precisão do sistema durante seu ciclo de desenvolvimento.

3.5.1 Cobertura de Testes

O sistema de testes deve cobrir todas as funcionalidades disponíveis na documentação das chamadas CAN do *Track Controller*. Para garantir a robustez e a precisão do sistema, os testes automatizados devem atender a diversos aspectos funcionais. Estes testes incluirão cenários que simulam as operações normais do piloto automático, como calibração de zona morta e seguimento de referência de posição, além de situações de falhas, como perda de sinal do sensor. Cada chamada de método e recebimento de mensagens via rede CAN ajudará a assegurar que o sistema está respondendo corretamente e mantenha o comportamento esperado dentro das regras de negócio.

3.5.2 Escalabilidade

À medida que o projeto evolui, o sistema de testes deve ser capaz de incorporar novos casos de teste e funcionalidades. Isso significa que a infraestrutura de testes deve ser flexível e adaptável, permitindo a adição contínua de novos elementos de teste à medida que são desenvolvidas novas funcionalidades ou identificadas novas necessidades de teste. Tal flexibilidade é crucial para acompanhar o ritmo de desenvolvimento e garantir que todas as novas funcionalidades sejam testadas de forma abrangente.

3.5.3 Confiabilidade

A confiabilidade dos testes é essencial para garantir que os resultados satisfaçam os requisitos desejados. O sistema de testes deve minimizar a ocorrência de falsos positivos e falsos negativos, garantindo que os erros reais sejam detectados e que os testes bem-sucedidos sejam corretamente reconhecidos. Para isso, devem ser implementados mecanismos de identificação de falhas que permitam a conclusão dos testes e gerem uma informação do que foi validado e o que apresentou problemas.

3.5.4 Integração Contínua

Os testes automatizados serão integrados a um pipeline de integração contínua (CI), permitindo a execução automática dos testes em cada aprovação de *pull request*, que é uma solicitação para revisar e possivelmente integrar mudanças de código de um *branch* para outro, conforme explicado na Seção 2.5. Esta *pipeline* deve ser configurada para executar testes SIL e HIL, garantindo que cada alteração no código seja verificada quanto à sua funcionalidade e integração com o hardware real. Esta abordagem assegura que quaisquer problemas sejam detectados e corrigidos rapidamente, mantendo a qualidade do software em níveis satisfatórios.

3.5.5 Ambientes de Teste

Dois tipos de ambientes de teste serão configurados: ambientes SIL e HIL. O ambiente SIL será utilizado para simulação de software, permitindo a validação do comportamento do software em condições simuladas, como falhas de sensor e variáveis ambientais. O ambiente HIL envolverá hardware real, incluindo motor e encoder, para testar a integração do software com o hardware e validar a resposta a eventos do mundo real e a precisão do controle de motor. A configuração desses ambientes garantirá uma cobertura completa dos testes, desde a simulação até a validação com hardware real.

4 SOLUÇÃO PROPOSTA E IMPLEMENTAÇÃO

O presente capítulo aborda a solução proposta e a metodologia utilizada para o desenvolvimento dos testes automatizados para o *Track Controller*. Primeiramente, na Seção 4.1 descreve a proposta de solução, trazendo um fluxo de trabalho junto com seus objetivos.Na Seção 4.2 é descrita estrutura de escrita dos testes, que são definidos em arquivos com extensão .feature utilizando a linguagem Gherkin e a implementação dos cenários de teste em C++, demonstrando como cada etapa definida nos arquivos .feature é traduzida para funções executáveis. Esta Seção também aborda cada teste feito nos ambientes de SIL e HIL, descrevendo os cenários desenvolvidos, suas motivações e a entrega de valor para cada cenário.

4.1 Descrição da Solução Proposta

A presente Seção apresenta a solução proposta para buscar solucionar o problema tratado na Seção 3.4 buscando o cumprimento dos requisitos técnicos apresentados na Seção 3.5. Será apresentado um fluxo de trabalho onde cada etapa terá sua motivação e objetivo definido, afim de ao final propor uma solução que possa ser realizada independente da forma de implementação escolhida.

Como mencionado na Seção 3.4 deste documento, houve o desenvolvimento de uma documentação por parte do time de Controle da Hexagon, a qual apresenta todas as mensagens via protocolo CAN que são enviadas e recebidas entre o *Display* e o *Track Controller*. Os testes automatizados serão divididos em SIL e HIL, então é necessário definir quais testes serão destinados a cada tipo de validação, sendo um teste apenas validado em HIL caso sua validação em SIL não seja possível por limitações do próprio conceito de validação SIL. A Figura 12 apresenta uma proposta de fluxo de desenvolvimento e implementação que busca atingir os objetivos propostos neste trabalho.

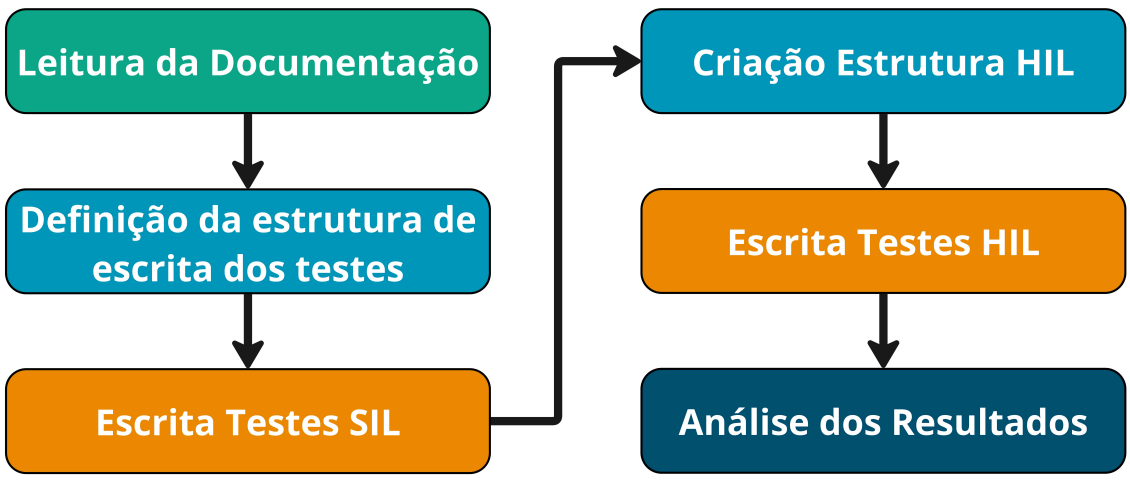


Figura 12 – Fluxo de trabalho para solução proposta: Elaborado pelo Autor

A primeira etapa consiste na leitura por parte do time de QA da documentação desenvolvida pelo time de Controle com as mensagens CAN. Isto é crucial pois o correto entendimento dessas mensagens ajudam o time de QA a entender como é esperado que cada aspecto do *Track Controller* se comporte, além de auxiliar no levantamento de dúvidas sobre as regras de negócio do produto e também sobre o levantamento de demandas para o time de Controle, podendo ser estas demandas por exemplo a criação de novas mensagens que contemplem outras partes do *Track Controller* ou a correção de eventuais problemas encontrados. A leitura da documentação também auxilia na definição de quais chamadas serão feitas via SIL e HIL, pois neste momento a equipe discute sobre cada tópico presente no documento e levanta da possibilidade de validação em SIL, para caso verifique-se que esta abordagem não validará uma determinada mensagem, ela entra na lista de chamadas para o HIL.

Uma vez que a leitura da documentação foi finalizada, é crucial definir a estrutura de escrita dos testes, ou seja, qual o padrão de escrita será seguido, como será a ponte entre a linguagem *Gherkin* de alto nível e a estrutura de baixo nível do código, o que pode variar de acordo com a linguagem utilizada, até a própria organização das pastas de cada etapa dos testes.

Com as etapas de leitura da documentação e definição da estrutura dos testes, inicia-se a escrita de código para o ambiente SIL. A motivação para isso deve-se ao fato de validações em SIL não necessitarem de ambientes reais para funcionarem, ou seja, são menos custosos no sentido de temporal e financeiro. Logo, começar pelos testes em SIL se mostra uma boa abordagem, pois possibilita validar algumas chamadas e levantar problemas da documentação enquanto dispensa a necessidade de montagens físicas.

Tendo os testes em SIL implementados, é necessário desenvolver a estrutura física para os testes em HIL, isso inclui o levantamento de todos os componentes necessários, sua instalação em um local apropriado para os testes e a integração entre o sistema físico e um computador.

Apenas após a criação e validação da estrutura dos testes em HIL é que inicia-se a escrita dos testes para este ambiente. Desta forma, valida-se não apenas a comunicação das mensagens CAN mas também a própria configuração da estrutura criada, que deve ser robusta, confiável e simular o máximo possível um ambiente real.

Por fim, a análise dos resultados ajuda a identificar a efetividade dos testes desenvolvidos nos dois ambientes, através da geração de relatórios, levantamento de problemas encontrados e o quanto os testes desenvolvidos abrangem o escopo proposto do produto.

4.2 Implementação

Esta Seção aborda a implementação da solução proposta na Seção anterior ao escopo deste trabalho, trazendo inicialmente na Subseção 4.2.1 a definição da estrutura de escrita dos testes, explorando todos os aspectos que envolvem a criação dos códigos referentes aos testes automatizados, até as Subseções 4.2.2 e 4.2.3, que abordam os testes desenvolvidos para os ambientes SIL e HIL respectivamente, abordando para cada teste os cenários propostos, suas motivações e objetivos.

4.2.1 Definição da Estrutura de Escrita dos Testes

Os testes são definidos em arquivos com a extensão .feature, que contêm a descrição das funcionalidades e cenários de uso do sistema utilizando a linguagem Gherkin. Cada arquivo .feature pode incluir várias funcionalidades, onde cada funcionalidade é composta por um ou mais cenários. Esses testes são especialmente úteis em ambientes de desenvolvimento ágil, onde a clareza e a facilidade de manutenção são fundamentais.

A escolha de utilizar arquivos .feature na linguagem Gherkin com o framework Cucumber.cpp para escrever testes automatizados oferece várias vantagens significativas. Primeiramente, a linguagem Gherkin é uma linguagem de domínio específico que utiliza uma sintaxe simples e de fácil compreensão, baseada em frases estruturadas em linguagem natural. Isso facilita a leitura e o entendimento dos testes não apenas para desenvolvedores, mas também para outras partes interessadas, como gerentes de projeto, analistas de negócios e clientes. A clareza proporcionada por Gherkin permite que todos os envolvidos no projeto possam entender exatamente o que está sendo testado e como os testes são executados, promovendo melhor comunicação e colaboração dentro da equipe.

Além disso, os arquivos .feature servem como uma excelente forma de documentação. Cada arquivo descreve detalhadamente as funcionalidades e os cenários de uso do sistema, funcionando como uma especificação executável. Isso significa que a documentação está sempre atualizada e sincronizada com o código, eliminando a necessidade de manutenção de documentos separados que muitas vezes se desatualizam rapidamente. A documentação gerada por arquivos .feature é precisa e confiável, refletindo o estado atual do sistema e suas funcionalidades.

No desenvolvimento, cada arquivo .feature pode foca em um tipo específico de teste. Por exemplo, "Calculadora.feature"contém todos os cenários necessários para validar as quatro operações básicas de uma calculadora. A Figura 13 mostra como um arquivo .feature pode ser estruturado para essa finalidade, detalhando os diferentes cenários usados em todo o teste. Isso facilita a organização e a manutenção dos testes, garantindo que cada aspecto crítico do sistema seja adequadamente verificado.

A seguir, será apresentada a estrutura do código de um projeto fictício de uma calculadora. A escolha deste exemplo genérico deve-se à necessidade de preservar a confidencialidade do código proprietário da empresa. No entanto, ele serve para ilustrar como a organização e a implementação do código são realizadas, refletindo a metodologia aplicada a todos os testes conduzidos. A estrutura abordará desde a definição dos requisitos até a execução das funcionalidades principais, evidenciando as práticas e padrões seguidos ao longo do desenvolvimento.

```
1  @CalculadoraExemplo
2  Feature: Testes de conta para uma calculadora
3    Rule: A calculadora realiza as quatro operacoes basicas
4
5    @Soma
6    Scenario: Valida a soma
7      When Defino que A recebe o valor '2'
8      And Defino que B recebe o valor '3'
9      Then O resultado da soma de A com B deve ser '5'
10
11
12   @Subtracao
13   Scenario: Valida a subtracao
14     When Defino que A recebe o valor '4'
15     And Defino que B recebe o valor '3'
16     Then O resultado da subtracao de A com B deve ser '1'
17
18   @Multiplicacao
19   Scenario: Valida a multiplicacao
20     When Defino que A recebe o valor '2'
21     And Defino que B recebe o valor '3'
22     Then O resultado da multiplicacao de A com B deve ser '6'
23
24   @Divisao
25   Scenario: Valida a divisao
26     When Defino que A recebe o valor '10'
27     And Defino que B recebe o valor '2'
28     Then O resultado da divisao de A com B deve ser '5'
```

Figura 13 – Exemplo da estrutura de escrita dos testes: Elaborado pelo Autor

Dentre os aspectos técnicos da estrutura de escrita dos testes temos:

- Cada "Feature"(ou funcionalidade) descreve um conjunto de comportamentos relacionados. Neste caso, trata-se de uma calculadora.
- O "Background"define o contexto comum para todos os cenários. Por exemplo, estabelece-se que vamos nos conectar a um dispositivo específico antes de iniciar os testes.
- Os "Scenarios"(ou cenários) descrevem situações específicas que queremos testar. Por exemplo, um cenário valida a soma entre dois números enquanto outro valida a subração.

Após a elaboração dos cenários de teste nos arquivos .feature, é fundamental compreender que essas descrições, embora sejam cruciais para representar as funcionalidades e casos de uso do sistema em um nível alto de abstração, não são executáveis por si só. Os arquivos .feature, estruturados com palavras-chave como 'Given', 'When' e 'Then', delineiam as etapas de preparação, ação e verificação, respectivamente, fornecendo uma visão clara e compreensível dos testes propostos.

No entanto, essa representação de alto nível requer uma tradução concreta para que os testes possam ser executados e validados. Esta tradução envolve a criação de implementações específicas para cada etapa definida nos arquivos .feature, a um nível mais baixo de abstração, utilizando a linguagem de programação C++.

Tomando como exemplo o teste realizado na Figura 13, para que estes passos sejam executados é necessário desenvolver um arquivo específico em C++ que efetivamente estabelece a conexão entre o código de alto nível com a aplicação. Cada etapa descrita nos arquivos .feature precisa ser mapeada para uma função correspondente no arquivo de declaração dos passos (*steps*), que no caso da calculadora pode se chamar **CalculadoraSteps.cpp**. Essa abstração é essencial para transformar a descrição conceitual dos testes em ações tangíveis que possam ser executadas e validadas pelo sistema.

```cpp
#include <gtest/gtest.h>

#include <cucumber-cpp/autodetect.hpp>

#include "sil/context/CalculadoraContext.h"

using cucumber::ScenarioScope;

ScenarioScope<CalculadoraContext> context;

WHEN("^Defino que A recebe o valor '(.*)'$") {
    REGEX_PARAM(int, A);

    context->SetA(A);
}

WHEN("^Defino que B recebe o valor '(.*)'$") {
    REGEX_PARAM(int, B);

    context->SetB(B);
}

THEN("^O resultado da soma de A com B deve ser '(.*)'$") {
    REGEX_PARAM(int, result);

    EXPECT_EQ(context->SomaAB(), result);
}
```

Figura 14 – Exemplo da estrutura de escrita dos passos para os testes: Elaborado pelo Autor

Na Figura 14, é apresentado um trecho de código em C++ que ilustra a utilização do framework Cucumber-Cpp para definir testes automatizados. O código representa a implementação de cenários de teste para operações básicas de uma calculadora, como soma, subtração, multiplicação e divisão.

O código inclui os cabeçalhos necessários do Google Test (gtest) e do Cucumber-Cpp, além do cabeçalho do contexto da calculadora CalculadoraContext.h. Em seguida, define-se um escopo de cenário utilizando a macro ScenarioScope do Cucumber-Cpp, o qual cria uma instância do contexto da calculadora para ser utilizada nos testes.

Cada passo de teste é definido utilizando a macro WHEN ou THEN do Cucumber-Cpp, que correspondem a expressões regulares para capturar valores específicos. Por exemplo, a macro WHEN captura os valores de A e B, e os atribui utilizando as funções SetA e SetB do contexto da calculadora.

Os passos THEN verificam o resultado das operações aritméticas, comparando o resultado esperado com o resultado obtido das funções SomaAB, SubtracaoAB, DivisaoAB e MultiplicacaoAB do contexto da calculadora (sendo as 3 últimas funções não apresentadas na Figura 14). A verificação é feita utilizando a macro EXPECT_EQ

do Google Test para assegurar que os resultados são iguais aos esperados.

É válido dizer que o arquivo de definição dos passos está em uma camada intermediária de abstração, mas ainda assim dentro de cada definição dos passos são chamadas funções externas. Tais funções estão dentro de um arquivo de contexto (*context*), o qual possui a implementação prática das funções que são chamadas nos passos. No exemplo da calculadora, o passo "Defino que A recebe o valor '(.*)'"recebe um valor inteiro que denomina de "A"e ao final invoca uma função "SetA()"que recebe como argumento a variável "A"criada dentro do passo. O contexto implementa o que esta função "SetA()"faz ao ser chamada.

O contexto é dividido em dois arquivos, um com extensão .h e ou tro com extensão .cpp. Arquivos .h (ou cabeçalhos) e .cpp (ou arquivos de implementação) são utilizados em C++ para separar a declaração de funções, classes, e variáveis (no arquivo .h) da sua implementação (no arquivo .cpp). Esta separação melhora a organização do código, facilitando a manutenção e a legibilidade, permitindo que múltiplos arquivos de implementação utilizem as mesmas declarações sem redundância.

```
1  #ifndef TEST_BEHAVIOR_SIL_CONTEXT_CALCULADORACONTEXT_H_
2  #define TEST_BEHAVIOR_SIL_CONTEXT_CALCULADORACONTEXT_H_
3
4  class CalculadoraContext {
5   public:
6      CalculadoraContext();
7
8      void SetA(int A);
9      void SetB(int B);
10     int SomaAB();
11
12  private:
13     int A_;
14     int B_;
15 };
16
17 #endif  // TEST_BEHAVIOR_SIL_CONTEXT_CALCULADORACONTEXT_H_
```

Figura 15 – Exemplo da estrutura de escrita do arquivo de contexto .h: Elaborado pelo Autor

O código de cabeçalho CalculadoraContext.h, mostrado na Figura 15, define a classe CalculadoraContext, que tem como objetivo fornecer um contexto básico para operações de uma calculadora simples. A classe inclui um construtor público para inicializar instâncias e dois métodos SetA(int A) e SetB(int B) para definir os valores dos atributos privados A_ e B_, respectivamente. Além disso, há um método SomaAB() que realiza a soma desses dois atributos e retorna o resultado.

A classe utiliza dois atributos privados, A_ e B_, que armazenam os valores dos números a serem somados. O uso de guardas de inclusão

- #ifndef TEST_BEHAVIOR_SIL_CONTEXT_CALCULADORACONTEXT_H_

- #define TEST_BEHAVIOR_SIL_CONTEXT_CALCULADORACONTEXT_H_
- #endif

garante que o arquivo de cabeçalho não seja incluído mais de uma vez durante o processo de compilação, evitando possíveis erros de redefinição. Essa estrutura simples e eficiente permite que a classe CalculadoraContext seja facilmente utilizada para operações básicas de adição, mantendo a integridade e modularidade do código.

```
1  #include "sil/context/CalculadoraContext.h"
2
3  CalculadoraContext::CalculadoraContext() : A_(0), B_(0) {
4  }
5
6  void CalculadoraContext::SetA(int A) {
7      A_ = A;
8  }
9
10 void CalculadoraContext::SetB(int B) {
11     B_ = B;
12 }
13
14 int CalculadoraContext::SomaAB() {
15     return A_ + B_;
16 }
```

Figura 16 – Exemplo da estrutura de escrita do arquivo de contexto .cpp: Elaborado pelo Autor

Na Figura 16, CalculadoraContext.cpp implementa as funções definidas na classe CalculadoraContext, descrita no arquivo de cabeçalho CalculadoraContext.h. O construtor da classe, CalculadoraContext::CalculadoraContext(), inicializa os atributos privados A_ e B_ com o valor zero, garantindo que a calculadora comece com valores padrão.

O método SetA(int A) define o valor do atributo A_ com o valor passado como parâmetro, e o método SetB(int B) faz o mesmo para o atributo B_. O método SomaAB() realiza a soma dos dois atributos A_ e B_, retornando o resultado. Esses métodos fornecem a funcionalidade básica para definir valores e calcular a soma dos números armazenados, permitindo o uso da classe CalculadoraContext para operações simples de adição.

Uma vez que o step é estabelecido, sua finalidade inclui a troca de mensagens através do barramento CAN. Esta funcionalidade permite a verificação da integridade das operações realizadas via código durante a execução de testes automatizados. A validação da alteração de variáveis internas do motor do *track controller* é um exemplo desse processo. Além disso, em testes realizados em HIL, essa validação pode

ser conduzida de maneira mais visual, onde comandos são enviados via CAN para rotacionar o motor, enquanto a posição do encoder é lida e verificada.

A etapa de preparação e troca de mensagens via CAN é feita por uma biblioteca própria desenvolvida pela Hexagon. Por se tratar de um código sigiloso e o foco deste projeto não ser na forma como o sistema estabelece a comunicação CAN e sim na validação do produto, este código não será mostrado neste projeto. Contudo, é importante salientar que esta biblioteca desenvolvida garante que os steps criados enviem e recebam dados via CAN, validando assim os testes criados de forma segura e efetiva.

Dentro da estrutura do código, afim de facilitar a identificação dos testes, os mesmos serão divididos em pastas, uma específica para HIL e outra para SIL, tendo dentro de cada uma uma pasta específica para os arquivos .feature, uma para os arquivos *contexts* e outra para os arquivos de *steps*, conforme a Figura 17.

Figura 17 – Estrutura de organização dos arquivos: Elaborado pelo Autor

Quando os testes são executados, existe um *feedback* referente ao sucesso ou falha de cada passo dos testes, gerando desta forma um relatório. Isso permite identificar de forma visual onde um teste falhou e qual foi o motivo de sua falha, gerando assim um relatório completo. A Figura 18 mostra um caso em que todos os testes foram executados com sucesso, indicados pela cor verde.

Figura 18 – Exemplo de execução de testes com sucesso: Elaborado pelo Autor

Os testes executados com sucesso aparecem com a cor verde no relatório gerado, mostrando cada cenário e dentro de cada cenário qual passo foi executado. Também é possível observar no canto superior direito a quantidade de cenários que foram executados, bem como a quantidade total de passos validados na execução do teste e o tempo total que a execução levou.

Caso algum passo falhe durante a execução, por qualquer motivo, o cenário em questão aparecerá em vermelho e haverá uma identificação específica do passo em que a falha ocorreu, mostranod também o motivo dela ter ocorrido, como pode ser visto na Figura 19.

Pode-se observar que o relatório indica a quantidadde de cenários e passos que falharam. No caso da Figura 19, o cenário que falhou é o que valida a subtração de dois números, no passo em que verifica o resultado. O relatório indica que o resultado real da subtração foi "2"enquanto o resultado esperado era "1", ou seja, o valor indicado na chamada do passo "O resultado da subtracao de A com B deve ser ’X’"passou um valor esperado 1, mas na prática a execução retornou outro resultado devido ao valor das variáveis "A"e "B", passadas nos passos anteriores, gerando o erro e consequentemente a falha do teste.

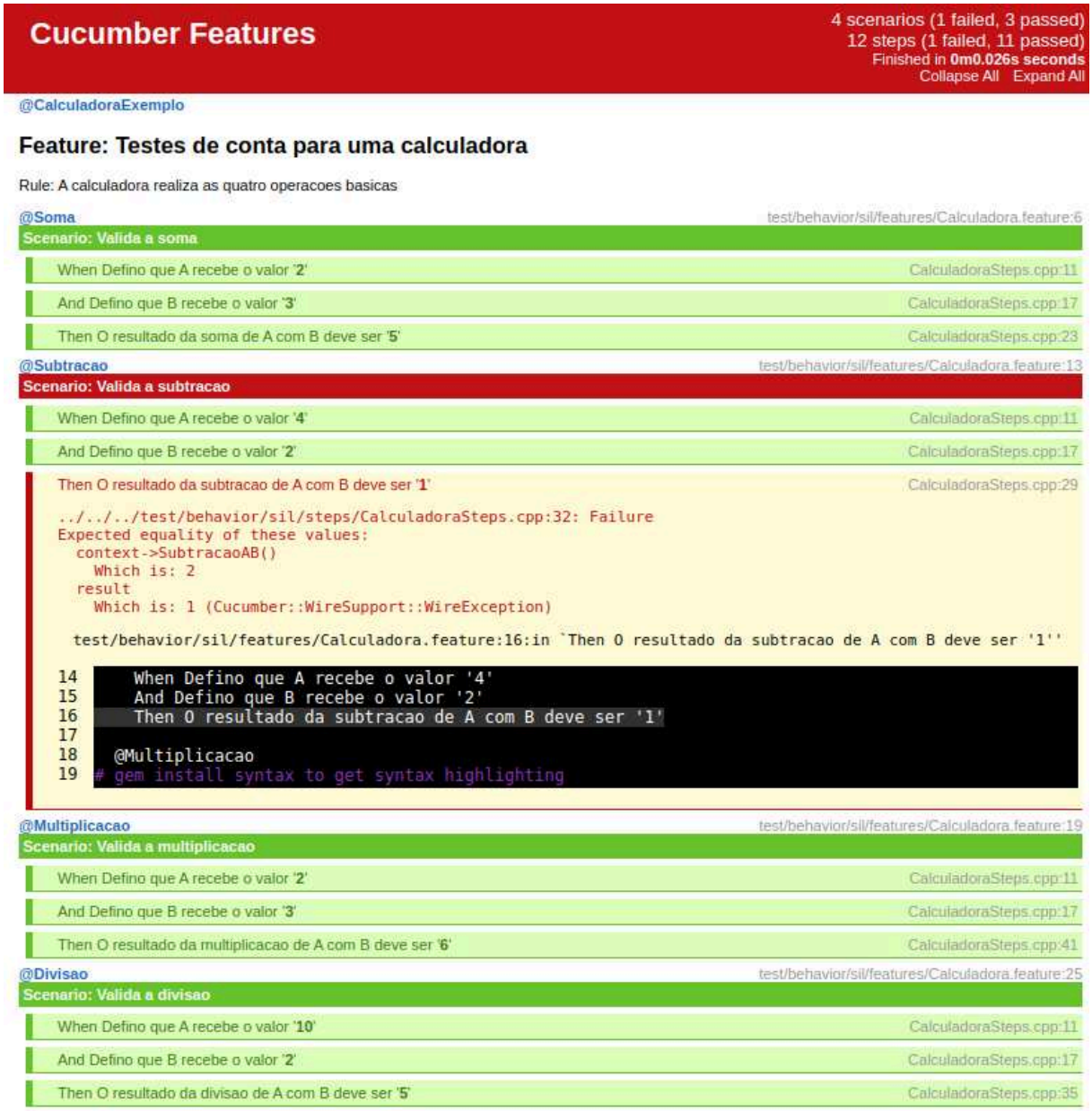


Figura 19 – Exemplo de execução de testes com falha: Elaborado pelo Autor

Vale ressaltar também que a falha de um cenário não influencia na execução dos demais cenários. Isso é importante pois a ideia é que os testes automatizados não parem toda vez que uma falha ocorrer, pois isso poderia torná-los muito demorados, o que vai de encontro a proposta. O ideal é justamente que a execução de cada cenário seja independente, indicando apenas ao final quais foram executados com sucesso e quais falharam.

4.2.2 Testes em SIL

Na presente Subseção, exploraremos os testes realizados em ambiente de SIL para a validação da comunicação do Track Controller. Esses testes foram conduzidos sem a necessidade de implementações físicas, pois recorremos a simulações para

reproduzir o ambiente de comunicação CAN. Um script foi empregado para simular uma rede CAN virtual, proporcionando um ambiente controlado para a avaliação do Track Controller.

Ao conduzir os testes no ambiente SIL, destacam-se quatro categorias principais de testes, cada uma direcionada a aspectos específicos do Track Controller:

- Tipo de Atuador
- Tipo de Encoder
- Modo de Operação
- Status de Segurança

A seguir, vamos abordar a justificativa por trás de cada tipo de teste, examinar os cenários específicos que eles avaliam e ressaltar a importância de validar sua execução de forma adequada.

4.2.2.1 Tipo de Atuador

A identificação correta do tipo de atuador é crucial para garantir que o motor do *Track Controller* funcione conforme o esperado. Os atuadores convertem sinais elétricos em movimento mecânico e sua correta caracterização permite otimizar o controle do sistema. Testes automatizados que validem o tipo de atuador asseguram que os sinais CAN transmitidos e recebidos são interpretados corretamente pelo sistema, evitando falhas de comunicação e mal funcionamento. Isso é especialmente importante em sistemas críticos como pilotos automáticos, onde a precisão e a confiabilidade são fundamentais para a segurança e o desempenho. Dentre os cenários possíveis para validar o tipo de atuador, temos:

1. Comportamento Padrão do Tipo de Atuador:
   O objetivo é verificar o tipo de atuador padrão e o modo de operação. Este teste assegura que, na ausência de configurações específicas, o sistema se comporta conforme o esperado com as definições de fábrica.

2. Tipo do Atuador - Controle de Velocidade:
   O foco está em configurar o atuador para um motor brushless trifásico e definir o modo de operação como controlador de velocidade. A precisão no controle de velocidade é fundamental para a estabilidade e a resposta dinâmica do motor, especialmente em situações onde a manutenção de uma velocidade constante é crucial para o desempenho do *Track Controller*.

3. Tipo do Atuador - Controle de Posição:
   Envolve a configuração do atuador como motor brushless trifásico e a definição

do modo de operação para controle de posição.Ele é crítico em aplicações onde a exatidão na movimentação é imprescindível, como em ajustes finos de superfície de controle ou na manutenção de posições específicas durante a operação.

Cada um desses cenários de teste desempenham um papel importante na validação do *Track Controller* no que se refere a configuração e verificação dos tipos de atuador disponíveis atualmente para este motor. Os testes validam não apenas a condição inicial mas também se as variáveis que são alteradas para modificar o tipo do atuador são efetivamente alteradas no código, apresentando falha na execução caso isso não seja verdadeiro.

4.2.2.2 Tipo de Encoder

Através desses testes, é possível garantir que o sistema está configurando e gerenciando corretamente o tipo de codificador (*Encoder*) e a frequência das mensagens de feedback, aspectos críticos para o funcionamento preciso e confiável do *Track Controller*. Estes elementos são fundamentais para assegurar que o *Auto Steering* opere de maneira eficaz, respondendo corretamente às variáveis do ambiente de trabalho agrícola e mantendo um alto nível de desempenho.

1. Verificação do Tipo Inicial de Encoder:
   O objetivo é assegurar que, ao iniciar a aplicação Track Controller utilizando o socket CAN, o tipo de encoder e a frequência das mensagens de feedback estejam configurados de maneira adequada. Especificamente, é verificado se o tipo de encoder inicial é 'Three Phase Hall' e se a frequência das mensagens de feedback do codificador é de 1 Hz. Este cenário é importante porque estabelece uma linha de base para as configurações iniciais do sistema, garantindo que ele inicie em um estado conhecido e previsível.

2. Configuração e Verificação do Tipo de Encoder:
   O cenário é delineado para testar a capacidade do sistema de configurar e validar diferentes tipos de *encoders* e suas frequências de mensagens de *feedback*. Cada combinação é configurada e, em seguida, verificada para assegurar que o sistema está reconhecendo e aplicando corretamente as configurações. Este teste é crucial pois valida a flexibilidade e robustez do Track Controller em lidar com diversas configurações de *encoder*, que podem ser necessárias para diferentes condições operacionais. Além disso, garante que as mensagens de feedback sejam enviadas na frequência correta, o que é essencial para o desempenho e a segurança do sistema.

A realização desses testes é fundamental para a validação do Track Controller, pois eles asseguram que o sistema pode iniciar com configurações padrão adequa-

das e ajustar-se corretamente a diversas configurações de encoder. Isso é vital para garantir que o *Auto Steering* opere de maneira confiável e precisa, adaptando-se a diferentes necessidades operacionais e mantendo a eficiência e segurança.

4.2.2.3 Modo de Operação

Especificamente, este teste concentra-se na configuração e verificação do modo de operação do controlador, crucial para garantir a comunicação eficaz entre o Track Controller e os sistemas externos. A seguir são apresentados os cenários que validam os modos de operação do *Track Controller*:

1. Verificar se o modo de controle inicial é indefinido:
   O teste verifica se o modo de controle inicial está corretamente configurado como indefinido. Isso é importante para garantir que o Track Controller comece em um estado conhecido e previsível antes de qualquer operação. A razão dele começar neste estado é para evitar que ele atue no momento em que o sistema é iniciado, sendo necessária a sua ativação de forma intencional pelo usuário.

2. Configuração e Verificação do Modo de Operação:
   Configura explicitamente o modo de controle para "CAN bus controller"e verifica se a configuração foi aplicada corretamente. Este é um passo crítico, pois o Track Controller precisa operar em modo CAN bus para estabelecer comunicação eficaz com outros dispositivos e sistemas externos, como sensores e atuadores.

Os testes que validam o modo de operação são importantes para garantir a segurança do usuário ao iniciar o sistema, impedindo que em um cenário real o *Track Controller* comece ativado e possa movimentar a máquina em que está instalado de uma forma brusca e indesejada pelo piloto. Validando que o status inicial é indefinido garantimos que o sistema não colocará em risco nenhum equipamento e principalmente nenhuma vida. Da mesma forma, verificamos também que a configuração do modo de operação está funcionando corretamente ao realizar a verificação da variável dentro do código.

4.2.2.4 Status de Segurança

Estes testes buscam validar se alarmes de segurança do sistema foram ou não disparados, verificando o estado incial do sistema ao mesmo tempo que força determinados alarmes a serem disparados e verifica se sua variável foi alterada. A seguir são apresentados os cenários que validam os alarmes de segurança do sistema:

1. Nenhum Alarme Disparado:
   Avalia se não há alarmes acionados após o *engage* (ativação) do Track Controller.

Ao verificar a ausência de alarmes específicos, como **controle ausente**, **sobreposição de atuador** ou **falha de alimentação**, este teste garante que o sistema esteja funcionando conforme o esperado durante o *engage*, o que é vital para a segurança do operador e a integridade da máquina.

2. Alarme de Perda de Mensagens de Controle:
   Avalia se o alarme de controle ausente é acionado quando o sistema deixa de receber mensagens por um período prolongado após o *engage*. Isso é crucial para identificar falhas de comunicação que podem ocorrer durante a operação, garantindo que os operadores sejam alertados caso a comunicação com o Track Controller seja interrompida, evitando potenciais riscos e danos. É essencial para o funcioanmento que o sistema receba constantemente mensagens de controle, pois uma demora das mensagens pode acarretar em um aumento do erro em relação a uma guia, exigindo uma correção brusca do controlador quando a próxima mensagem chegar, tornando mais arriscada a operação e piorando a eficiência do sistema de controle.

4.2.3 Testes em HIL

Nesta Subseção serão apresentados os testes feitos em HIL para a validação do da comunicação *Track Controller*. Para a realização desses testes, foi desenvolvida uma estrutura física fixada na parede da sala de testes que existe na empresa, a qual pode ser vista nas figuras 20, 21 e 22.

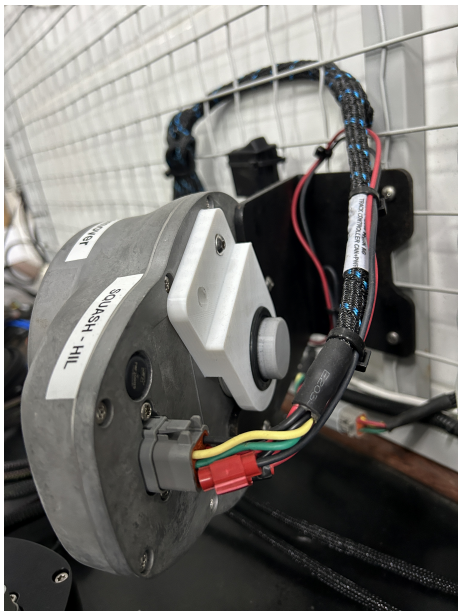


Figura 20 – Estrutura HIL - 1

Figura 21 – Estrutura HIL - 2

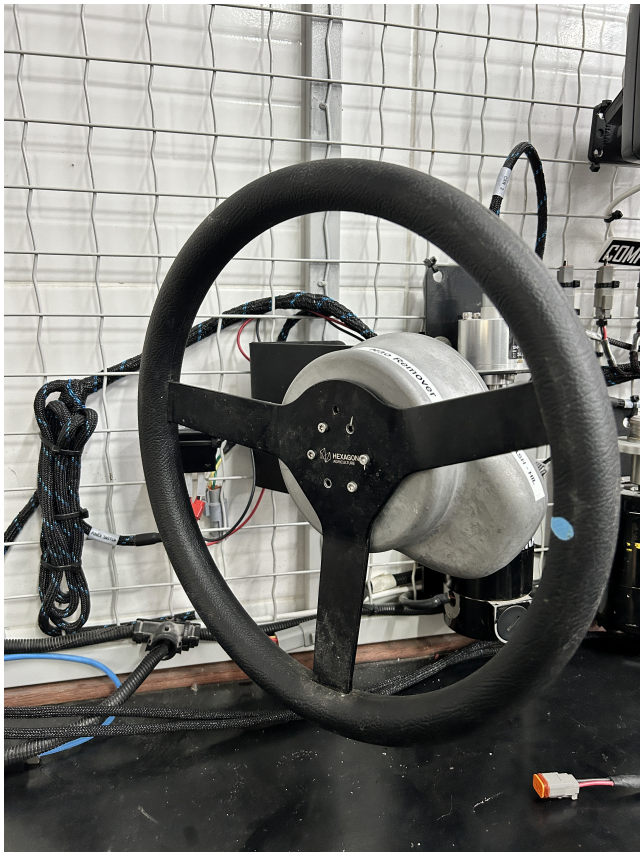


Figura 22 – Estrutura HIL - 3

A estrutura consiste em um motor *Track Controller* específico para os testes, o qual permanecerá fixo no local. Este motor está acoplado a uma base de fixação metálica, especialmente desenvolvida para servir de suporte adequado ao motor. A base, por sua vez, está parafusada a uma grade cinza visível ao fundo da imagem. Na parte traseira do motor (Figura 20), encontra-se fixado ao eixo do motor um redutor, o qual impõe uma certa resistência ao movimento do eixo. Adicionalmente, a ligação elétrica que alimenta o motor também envia e recebe informações de sua placa interna. Em seu cabeamento também existe um interruptor de segurança (ao fundo na esquerda na Figura 22), que interrompe a alimentação do motor em casos de falhas ou emergências. Este cabeamento é levado até um computador dedicado aos testes em HIL, que fica ligado o tempo todo e está preparado para executar os testes sempre que necessário.

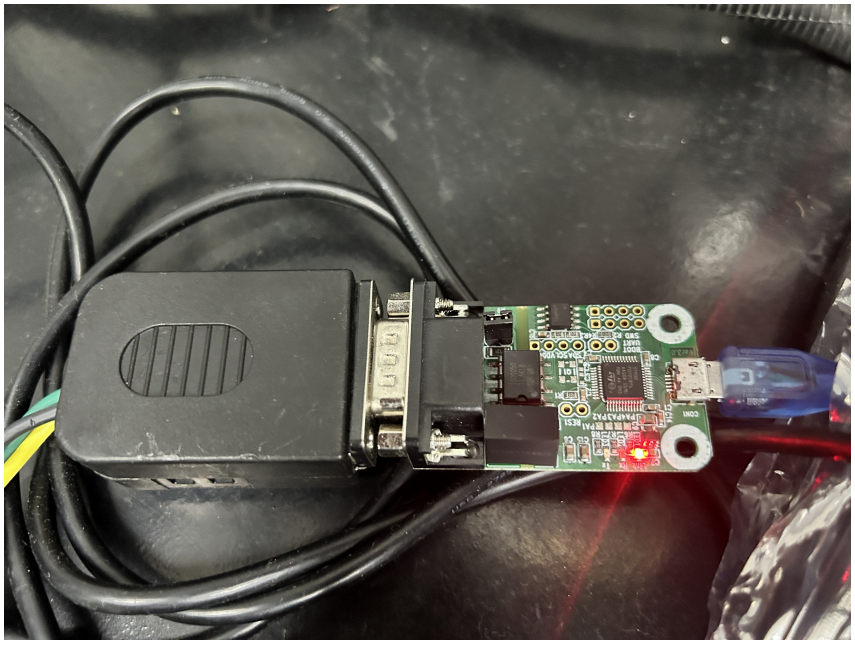


Figura 23 – Adaptador USB-CAN

Para realizar a comunicação entre o motor e o computador, foi utilizado um dispositivo de conversão USB para o protocolo CAN (Figura 23), que permite enviar e receber as informações necessárias do motor. Com a estrutura validada, foi feito o levantamento dos tipos de testes que seriam feitos utilizando a estrutura do HIL, sendo eles:

- Controle de Referência
- Calibração da Zona Morta
- Desativação por Atuação Manual
- Status de Segurança

A seguir será explicada a motivação de cada um dos tipos de testes citados, os cenários que os englobam e porquê sua validação é importante.

4.2.3.1 Controle de Referência

Tendo em vista que o projeto busca validar a comunicação do *Track Controller*, a garantia de que ele consegue seguir uma referência de posição de maneira precisa e consistente é de suma importância. Esta validação é essencial para assegurar a qualidade, a segurança e a eficiência do sistema como um todo. A precisão no seguimento da referência de posição é um critério crítico para o desempenho adequado do *Track Controller*, impactando diretamente na operacionalidade e na eficácia das máquinas as quais ele pode ser instalado. Portanto, a implementação de testes que confirmem a capacidade do *Track Controller* de seguir referências de posição com exatidão é

indispensável para o sucesso do projeto. A seguir são apresentados os cenários que validam o seguimento de referência do *Track Controller*:

1. Verificação da Posição Inicial do Encoder:
   O teste verifica a posição inicial do *encoder* no momento em que o motor é ligado. Pela documentação é padronizado que a posição inicial seja '32768' unidades do *encoder*, sendo este valor então a "posição zero". Ao verificar a posição inicial, a tolerância esperada para a leitura do valor é zero, sendo então o valor lido exatamente '32768' para garantir que o motor não tenha se movimentado no momento em que foi ligado. Também é feita uma verificação se ao iniciar o Track Controller não está controlando, sendo este teste é crucial para assegurar que o controlador inicie a partir de um estado conhecido e esperado.

2. Verifica o estado do controlador durante a operação:
   Para este teste é esperado que ao ligar o motor do *Track Controller*, uma vez que o controle seja habilitado (*engaged*) com uma posição de referência, o motor irá girar até esta referência e uma variável indicará que o sistema "está controlando", sendo esta variável a que é verificada no momento em que o motor está em funcionamento. A ideia é que esta variável que indica se está controlando ou não esteja apenas no estado "está controlando"(*controlling*) no momento em que uma referência foi dada ao motor e ele está se movimentando para chegar a esta referência, sendo que uma vez que o motor esteja na posição desejada a variável é alterada para o estado "não está controlando".

3. Verifica o estado do controlador quando não está operando:
   Este teste é muito similar ao anterior, mas a diferença está que aqui verifica-se a variável que indica se o sistema está ou não controlando o motor no momento em que não há mais ação de controle. Ou seja, aqui verificamos a alteração da variável para o status "Não está controlando"(*not controlling*). Para validar isso, o motor é ligado, é dado o *engage* com uma posição de referência, o motor movimenta-se para esta posição, em seguida um certo tempo é esperado e é dado o *disengage* (desativação) do *Track Controller*, para que neste momento a variável seja verificada e indique o status "*not controlling*". Isso ajuda a garantir que o sistema está alterando a variável que indica se o sistema de controle está ativado ou não, evitando que isso possa influenciar em outras partes do sistema que dependem desta informação.

4. Configurar e verificar Múltiplas Posições de Referência:
   Para este teste a ideia é verificar se o *Track Controller* consegue seguir uma série de múltiplas referências em sequência. Para isso é utilizada uma estrutura de

escrita de testes que faz com que um mesmo teste seja repetido quantas vezes forem necessárias alterando uma variável específica, sendo isso chamado de *Scenario Outline*. Isso permite o envio de 'n' posições para o motor, sendo cada posição validada realizando a leitura do *encoder*, considerando sempre que o valor da posição inicial é '32768' e que o motor pode girar no sentindo horário (somando o valor inicial) e anti-horário (subtraindo o valor inicial). Este teste é importante para validar a robustez do sistema, uma vez que realiza 'n' vezes uma mudança e seguimento de referência, fazendo o motor girar e parar e realizando a leitura, funcionando com um teste de *stress*.

4.2.3.2 Calibração da Zona Morta

A realização de um teste de calibração da zona morta de um motor elétrico é crucial para assegurar o funcionamento preciso e eficiente do sistema de controle do motor. A zona morta refere-se ao intervalo de entrada em que não há resposta na saída do motor, ou seja, pequenas variações na entrada não resultam em movimento ou mudança significativa na operação do motor. Esse fenômeno pode causar problemas de precisão, especialmente em aplicações que exigem controle fino e resposta rápida. Calibrar essa zona morta permite ajustar o controlador do motor para compensar essa inatividade, garantindo que o motor responda adequadamente a todos os comandos, resultando em uma operação mais precisa e eficiente.

Além disso, a calibração da zona morta é importante para a longevidade e a eficiência energética do motor. Quando um motor opera com uma zona morta inadequada, ele pode experimentar oscilações e variações desnecessárias que aumentam o desgaste mecânico e o consumo de energia. Com uma calibração correta, o sistema de controle pode minimizar esses efeitos, promovendo uma operação mais suave e economizando energia. Assim, a calibração da zona morta não apenas melhora o desempenho e a precisão do motor, mas também contribui para a durabilidade e a sustentabilidade do sistema como um todo. A seguir são apresentados os cenários que validam a calibração da zona morta:

1. Verificação do valor inicial de zona morta:
   A verificação do valor inicial de zona morta, que no sistema é definido por uma variável de código e inicialmente tem o valor zero, é crucial para entender o ponto de partida do *Track Controller*. Esse valor inicial permite identificar se o motor está operando dentro dos parâmetros esperados ou se há necessidade de correção. Além disso, conhecer o valor inicial ajuda a medir a eficácia do processo de calibração, comparando o valor antes e depois da calibração para garantir que a zona morta tenha sido ajustada adequadamente.

2. Iniciar a calibração da zona morta e sair antes da finalização:

Este teste busca validar se o sistema consegue entrar no estado de calibração da zona morta, ou seja, se o comando enviado via CAN de fato inicia o processo desejado, ao mesmo tempo em que valida o envio da mensagem que para a calibração da zona morta antes de sua finalização. Além disso, também funciona para validar uma regra de produto, ao verificar o valor da variável de zona morta do sistema após interromper o processo de calibração. O objetivo é, dado que o processo de calibração foi interrompido, então o valor da variável de zona morta deve permanecer como o seu estado anterior, sendo zero pois este teste é feito após o motor ser ligado.

3. Iniciar a calibração da zona morta e definir a variável com o resultado:
   O propósito deste cenário é similar ao anterior, porém, além de assegurar o início da calibração da zona morta, aqui o procedimento de calibração é executado até sua conclusão. A variável de zona morta do sistema é então atualizada com o resultado da calibração, e o valor exato é verificado pelo usuário, sendo registrado (*logged*) no sistema.

4.2.3.3 Desativação por Atuação Manual

Em um sistema como o *Track Controller*, a presença de um teste que valide a desativação do *Auto Steering* diante da intervenção manual do condutor no volante é crucial para garantir a segurança e a confiabilidade do sistema. Este teste serve como uma garantia contra situações em que o *Auto Steering* pode não reconhecer prontamente a intervenção humana, seja devido a falhas técnicas ou ações impróprias do condutor. Ao validar a resposta do sistema à intervenção manual, assegura-se que o controle do veículo seja imediatamente devolvido ao condutor em caso de necessidade, evitando potenciais situações de perigo ou colisão.

A ideia é que o sistema opere de tal forma que o piloto desarme quando houver uma atuação manual, mas existe um limiar aceitável afim de evitar que toda e qualquer atuação desative o sistema de *Auto Steering*, tornando-o muito sensível, algo que também não é desejado. Isso é valivado através da leitura da corrente do motor, pois quando o sistema está operando, ao corrigir a tragetória, uma certa corrente é imposta ao motor, mas se há algo impedindo o motor de girar, a corrente aumenta para gerar a força suficiente, afim de movimentar o volante para que permaneça na rota desejada. Existe então um valor de corrente definido que diz para o sistema que caso a corrente do motor seja igual ou superior a 'x' o piloto automátido será desarmado. Este valor de corrente pode ser alterado por uma variável do sistema dentro de uma faixa de valor bem definida, sendo que um valor muito baixo deixará o sistema sensível demais enquanto um valor muito alto tornará praticamente impossível causar o desarme. Existem também um alarme específico em caso de desarme por atuação manual que é

validado nos cenários de testes, os quais são apresentados a seguir:

1. Configurar um valor alto de corrente para desativação por atuação manual:
   A importância deste cenário está em validar que nenhum alarme de desativação manual é disparado quando o limiar de corrente é configurado para um valor alto, estando acima da corrente normal de operação do motor. Logo, verifica-se se a variável de limiar de corrente para desativação foi configurada como esperado, aciona o controle de posição do motor para uma determinada referência e então observa-se a não existência de um alarme de desarme do *Auto Steering* por atuação manual, pois o limiar de corrente está acima do que é demandado para a atuação do motor.

2. Configurar um valor baixo de corrente para desativação por atuação manual:
   Muito similar com o cenário anterior, mas a ideia agora é contrária. Almeja-se validar que ao determinar um valor de corrente muito baixo como limiar para desarme do *Auto Steering*, deixando-o muito sensível a qualquer movimentação, o mesmo será desarmado e o respectivo alarme será disparado. A ideia é colocar um valor de corrente tão baixo que a própria atuação do motor para seguir uma referência ultrapasse esse valor de corrente, ocasionando o desarme e o alarme.

3. Limpar todos os alarmes de desativação por atuação manual Para este cenário, é preciso que um alarme tenha sido disparado, então aproveita-se de parte do cenário anterior, onde é definido um valor muito baixo de limiar de desarme para o *Auto Steering*. Desta forma, mandando-o para uma posição a corrente fica acima do limiar, ocasionando o desarme e disparando o alarme. Uma vez que o alarme foi dispadado, é enviado um comando que "limpa"todos os alarmes existentes. Para validar que este comando de limpar alarmes funciona, verifica-se logo em seguida a não existência do alarme de desativação do *Auto Steering* por atuação manual, validando assim a funcionalidade de limpar os alarmes.

4.2.3.4 Status de Segurança

Da mesma forma que o tópico similar na etapa de validação em SIL, estes testes buscam validar se alarmes de segurança do sistema foram ou não disparados, verificando o estado incial do sistema ao mesmo tempo que força determinados alarmes a serem disparados e verifica se sua variável foi alterada. Contudo, aqui são verificados outros alarmes que necessitavam de uma estrutura real para serem disparados, causando esta "repetição"entre os cenários de SIL e HIL, embora ambos se complementem para uma validação única dos status de segurança do sistema. Assim, os cenários validados por este teste automatizado são:

1. Nenhum alarme disparado ao dar *engage* no *Track Controller*:
   Verifica que nenhum alarme de segurança é acionado após o *engage* do *Track Controller* com um setpoint de posição qualquer.A ideia é verificar que não há alarmes de "controle ausente", "sobrescrição de atuador"ou "fonte de alimentação". Esse teste é fundamental para confirmar que o controlador não gera alarmes falsos durante a operação normal, o que poderia levar a interrupções desnecessárias e comprometer a operação.

2. Verificar alarme de mensagem de controle ausente:
   o teste verifica se o alarme de controle ausente é acionado quando o *Track Controller* não recebe uma mensagem de controle por um período prolongado após o *engage*. Este teste é crucial para garantir que o sistema pode detectar e alertar sobre a perda de comunicação de controle, uma condição que poderia comprometer a segurança e eficiência do sistema.

5 ANÁLISE DOS RESULTADOS

O presente capítulo aborda uma análise aprofundada dos resultados alcançados no decorrer do projeto, focalizando a conformidade com os requisitos técnicos estabelecidos. Veremos que a implementação de ambientes de teste robustos para Software-in-the-Loop (SIL) e Hardware-in-the-Loop (HIL) assegurou uma validação precisa do sistema, abarcando todos os aspectos da comunicação CAN e aderindo estritamente às diretrizes estabelecidas pelas regras de negócio. Além disso, a estrutura dos testes demonstrou escalabilidade, possibilitando a inclusão simplificada de novos testes conforme exigido, enquanto metodologias rigorosas garantiram a confiabilidade dos resultados, incluindo verificações detalhadas a nível de código.

Embora a integração contínua dos testes em SIL tenha sido eficazmente realizada, persistiram desafios na integração dos testes em HIL, em virtude da necessidade de interação com hardware físico e da complexidade inerente à automação. Todavia, o impacto positivo dos testes automatizados no produto foi inegável, reduzindo significativamente o tempo de teste e permitindo que a equipe de QA direcionasse recursos para outras áreas vitais de validação. Tais resultados enfatizam a importância fundamental dos testes automatizados para garantir a qualidade e a confiabilidade do *Track Controller*.

5.1 Cumprimento dos Requisitos Técnicos

Para começar a analisar o que foi desenvolvido no decorrer deste projeto é necessário relembrar os requisitos técnicos citados anteriormente que nortearam o que foi executado. Em resumo, os requisitos técnicos, conforme a Seção 3.5, eram:

1. Cobertura dos testes:
    - Cobrir todas as chamadas da documentação disponibilizada;
    - Incluir testes que simulam operações reais;
    - Garantir o seguimento das regras de negócio.
2. Escalabilidade:
    - Comportar a expansão dos testes com possibilidade de adição de novos cenários.
3. Confiabilidade
    - Minimizar a ocorrência de falsos positivos e falsos negativos;
    - Identificar casos de sucesso e falhas.
4. Integração Contínua:

- Adicionar a execução dos testes em uma Pipeline para que sejam executados a cada nova alteração do código.

5. Ambiente de testes
    - Desenvolver testes no ambiente simulado (SIL) e no ambiente real (HIL).

A seguir serão apresentados em detalhes cada

5.1.1 Ambiente de Testes

Um dos objetivos principais do projeto era desenvolver ambientes de testes robustos tanto para Software-in-the-Loop (SIL) quanto para Hardware-in-the-Loop (HIL). Considerando a importância de ambos os tipos de teste para a validação completa do sistema, foram investidos tempo e recursos significativos para criar ambientes de teste que fossem funcionais, eficientes e confiáveis. Esse processo envolveu a seleção e configuração de ferramentas adequadas, além de um planejamento cuidadoso para garantir que ambos os ambientes pudessem atender às necessidades específicas dos testes.

Foi possível alcançar esse objetivo com sucesso. O ambiente de testes SIL foi cuidadosamente configurado para permitir uma execução contínua e precisa dos testes, garantindo a verificação detalhada das mudanças no código. Paralelamente, desenvolveu-se um ambiente de testes HIL que, embora exija a inicialização manual dos testes, está totalmente equipado para interagir com o hardware necessário e realizar os testes previstos. Este desenvolvimento constitui um avanço importante, estabelecendo uma base sólida para futuras melhorias e possibilitando a automação completa do processo de testes em HIL. Dessa forma, o requisito técnico de desenvolver ambos os ambientes de teste foi atingido.

5.1.2 Cobertura dos Testes

Dentro da documentação desenvolvida para a validação do *Track Controller*, as chamadas de comunicação CAN estavam divididas em 7 tópicos principais:

- Modo de Operação
- Tipo de *Encoder*
- Controle de Referência
- Tipo de atuador
- Atuação Manual
- Calibração de zona morta

- Status de segurança

Para cada um destes tópicos, foi desenvolvido um arquivo de testes que valida diferentes cenários, sempre focados em validar usos reais do produto. Estes 7 tópicos se dividiram entre a validação via *software* (SIL) e a validação via *hardware* (HIL), sempre focando em testar no ambiente HIL apenas aquilo que, por limitações do ambiente SIL, necessitaria da validação neste tipo de ambiente. Os testes de "Status de Segurança" foram os únicos dentro dos 7 tópicos que se dividiram entre os dois ambientes de testes, deixando os demais divididos entre:

- "Tipo de Atuador", "Tipo de Encoder"e "Modo de Operação": **SIL**;
- "Controle de Referência", "Calibração da Zona Morta"e "Atuação Manual": **HIL**.

Portanto, com a abordagem detalhada acima, o requisito técnico de cobertura dos testes foi atingido devido ao desenvolvimento de arquivos de testes específicos para cada um dos sete tópicos principais, garantindo que todos os cenários reais de uso do produto fossem validados. A divisão estratégica entre validação via *software* (SIL) e *hardware* (HIL), aplicada conforme as limitações de cada ambiente, assegurou a eficácia dos testes. Em particular, o tópico de "Status de Segurança"foi validado em ambos os ambientes, reforçando a integridade e a segurança do sistema. Dessa forma, a metodologia empregada proporcionou uma cobertura abrangente e confiável dos testes, cumprindo integralmente o requisito técnico estabelecido.

5.1.3 Escalabilidade

A estrutura de criação dos testes utilizada neste trabalho, a qual pode ser vista na Seção 4.2.1, ppermite a criação de novos testes de maneira simples e eficiente, bastando criar os arquivos necessários. Para tanto, deve-se redigir o cenário em linguagem natural no arquivo .feature. Caso seja necessário implementar um novo passo, basta adicioná-lo no arquivo correspondente na pasta "Steps". Se for preciso incluir um novo contexto, o arquivo na pasta "Context"também pode ser editado conforme a necessidade.

Desta forma, os testes são totalmente escaláveis caso novas mensagens de comunicação CAN para o *Track Controller* sejam desenvolvidas ou surja uma nova regra de negócio que precise ser validada. Portanto, é válido afirmar que este requisito também foi atingido.

5.1.4 Confiabilidade

Uma das questões principais na elaboração dos testes automatizados era a confiabilidade deles, ou seja, o quanto a empresa poderia confiar nesses testes para

garantir que os aspectos cobertos não apresentem problemas que possam impactar o usuário final. Isso envolve avaliar a ocorrência de falhas nos testes, como falsos positivos e falsos negativos. Falsos positivos ocorrem quando o teste indica a presença de um problema inexistente, enquanto falsos negativos acontecem quando o teste não detecta um problema real. Minimizar a taxa de falsos negativos é crucial para evitar que problemas passem despercebidos e cheguem ao usuário final, ao mesmo tempo em que é importante reduzir a taxa de falsos positivos para evitar desperdício de recursos com correções desnecessárias.

Tendo isso em mente, durante todo o desenvolvimento dos testes, a ideia foi validar a nível de código cada chamada feita, verificando e comparando as variáveis do sistema para garantir que suas mudanças surtam nos efeitos previstos. Também houve apoio do time de desenvolvimento para ter um entendimento maior da estrutura do código, afim de garantir o total entendimento por parte do time para saber onde os testes estavam atuando, garantindo que uma chamada estava executando o que era previsto.

Por exemplo, para garantir que um passo que inicia a calibração da zona morta de fato está enviando esta mensagem e não outra qualquer, foi necessário entender onde a chamada desenvolvida pela empresa atuava dentro do código que controla todo o sistema do *Track Controller*, para então haver a confirmação de que o passo desenvolvido estava executando corretamente o comando de iniciar a calibração da zona morta.

Com a geração dos relatórios seguindo os modelos mostrados nas Figuras 18 e 19, é possível identificar sempre os casos de sucesso e falhas dos testes. Especificamente nos casos de falhas é possível ainda identificar o motivo da falha, possibilitando a averiguação por parte do time responsável do motivo da falha e corrigindo o problema encontrado.

Tendo em vista os pontos apresentados, conclui-se que o requisito de confiabilidade dos testes também foi atingido. As Figuras 18 e 19 ilustram a eficácia dos testes automatizados ao mostrar claramente os cenários de sucesso e as falhas encontradas. Esse nível de detalhe permite que a equipe de QA rastreie a causa exata da falha e tome as medidas corretivas necessárias. Dessa forma, os relatórios não apenas confirmam a execução correta dos testes, mas também proporcionam uma ferramenta eficaz para a identificação e correção de erros, garantindo que problemas não passem despercebidos e sejam resolvidos antes de impactar o usuário final. Portanto, a capacidade de detectar e detalhar falhas específicas reforça a confiabilidade dos testes realizados.

5.1.5 Integração Contínua

Um dos critérios fundamentais para o trabalho era implementar uma integração contínua que abrangesse tanto os testes em Software-in-the-Loop (SIL) quanto em Hardware-in-the-Loop (HIL). Enquanto a integração contínua dos testes em SIL foi bem-sucedida e está atualmente operando na pipeline sempre que há uma mudança no código, a integração dos testes em HIL ainda não foi alcançada. Esta limitação impede a plena verificação do sistema em condições que envolvem a interação direta com o hardware, essencial para garantir a robustez e a confiabilidade do produto final em ambiente real.

Os testes em HIL são notoriamente mais difíceis de integrar na pipeline de integração contínua devido a várias complexidades inerentes ao seu processo. Ao contrário dos testes em SIL, que são executados inteiramente em software e podem ser facilmente escalonados e automatizados em qualquer ambiente de desenvolvimento, os testes em HIL exigem a interação com hardware físico. Isso significa que o ambiente de teste precisa de hardware específico configurado corretamente, disponível e operante para cada execução de teste. A disponibilidade limitada de hardware e a necessidade de configuração manual tornam o processo mais suscetível a atrasos e erros, dificultando a automação completa.

Além disso, a automação dos testes em HIL é dificultada pela necessidade de iniciar esses testes através de comandos manuais, um processo que atualmente não está integrado no fluxo de integração contínua de código. Para integrar completamente os testes HIL na *pipeline*, que é uma sequência automatizada de processos e ferramentas que compila e testa o código, seria necessário desenvolver soluções de automação que possam controlar remotamente o hardware, iniciar os testes e coletar os resultados de forma confiável. Esse desenvolvimento exige um esforço significativo e recursos adicionais para criar um ambiente de teste robusto que possa operar de maneira automatizada e integrada com o restante do sistema de integração contínua. Esses desafios técnicos e logísticos são os principais obstáculos que tornam a integração de testes em HIL uma tarefa mais complexa e demorada em comparação aos testes em SIL.

Portanto, considera-se que o requisito de integração contínua foi atingido parcialmente, uma vez que os testes em SIL já estão rodando na *pipeline*.

5.2 Problemas Encontrados

No decorrer do desenvolvimento deste projeto alguns problemas ocorreram, os quais trouxeram a necessidade de pesquisa dentro do time de QA afim de buscar uma solução para contorná-los. Uma das dificuldades encontradas foi o fato de algumas variáveis importantes para a realização completa dos testes não estarem disponíveis

para acesso via código. Isso exigiu uma investigação mais aprofundada do código do sistema e também conversas com o time de desenvolvimento com o objetivo de entender como seria possível tornar essas variáveis acessíveis. Essa necessidade gerou demandas do time de controle e desenvolvimento, que embora causaram algum atraso na criação dos testes automatizados, foram essenciais para garantir que os testes escritos estivessem funcionando na totalidade do que se esperava. Esses atrasos por conta de falta de acesso a variáveis impactou no desenvolvimento dos testes tanto em SIL quanto em HIL.

Além disso, a própria montagem do cenário em HIL também se mostrou um desafio. O time iniciou o desenvolvimento dos testes em SIL, justamente por não necessitarem de qualquer estrutura real, mas em determinado momento não havia como continuar implementando os testes sem ter uma estrutura física montada. Tendo isso em mente, as dificuldades estavam em como prender o motor de forma adequada que possibilitasse seu acesso e que não atrapalhasse os demais testes realizados na sala, ao mesmo tempo que era preciso se preocupar com a alimentação do motor e a comunicação com o computador utilizado pela equipe para a realização dos testes. O time conseguiu um adaptador USB-CAN para usar no HIL e também desenvolveu um script para a utilização deste equipamento, tendo então um cenário montado e funcional.

Outro problema enfrentado durante os testes em HIL foi que para simular um cenário mais próximo da realidade, foi inserido um redutor acoplado ao eixo do motor. A ideia desse redutor é causar uma dificuldade na rotação, adicionando uma carga similar ao que aconteceria na instalação em uma máquina real. Com a adição desse redutor, os testes de controle de referência e qualquer outro teste que exigisse do motor chegar a uma determinada posição começaram a perder a robustez, apresentando falhas a um nível considerável. Isso se deu pois os testes não levavam em conta qualquer tipo de tolerância, exigindo para o sucesso que um valor absoluto de posição do *encoder* fosse atingido na sua totalidade, mas com um redutor isso não era possível e um erro sempre estava presente.

Para verificar a taxa de acerto de posicionamento do motor com o redutor adicionado, foi executada uma bateria de testes onde a tolerância de leitura do encoder era zero, ou seja, dado que uma posição X foi passada para o *Track Controller*, após a sua movimentação até a posição de referência a leitura do *encoder* é feita, onde espera-se receber como posição o mesmo valor X, sem possibilidade de alteração. Para isso, conforme a Tabela 1, foram executados um total de 100 cenários, onde cada cenário representa uma posição de referência enviada e lida, onde ao final verificou-se que com uma tolerância zero na leitura do *encoder* havia uma taxa de acerto de apenas 30%.

Afim de contornar esse problema, o qual impactaria qualquer teste em HIL que

exigisse posicionamento do *Track Controller*, conversou-se com o time de controle para entender o que seria uma tolerância aceitável dentro do sistema. No início cogitou-se utilizar uma tolerância percentual, mas isso se mostrou mais oneroso do ponto de vista de desenvolvimento, então o uso de uma tolerância usando valores absolutos do *encoder* foi seguido. Nas conversas com o time responsável, percebeu-se que um valor de 10 posições de tolerância seria aceitável dentro da proposta do projeto, uma vez que esse valor representaria em erros reais aproximadamente 0.1 graus de erro de posicionamento de roda no veículo, um valor que não traria nenhuma falha significativa ao teste e ao sistema de *Auto Steering*.

Com a adição da tolerância na verificação de referência de posição do *Track Controller*, os mesmos testes de carga foram rodados, como visto na Tabela 1, onde verificou-se que agora a cada 100 posições enviadas e lidas, 100% delas apresentaram sucesso. Ou seja, agora todo e qualquer teste que dependesse de girar o *Track Controller* para uma posição específica usando o redutor acoplado teriam mais confiança e robustez, sem perder o foco na qualidade e garantindo que o resultado seja coerente com o esperado.

Tabela 1 – Testes de posicionamento com e sem tolerância de leitura.

Cenários Totais: 100	**Sucesso**	**Falha**	**Taxa de Acerto**
Sem Tolerância	30	70	30%
Com Tolerância (10 unidades de posição)	100	0	100%

5.3 Impacto no Produto

Uma das motivações para o desenvolvimento dos testes automatizados é a redução do tempo de testes em conjunto com o fato da automação permitir aos membros do time de QA focar em outras validações também importantes para a empresa. Para validar isso, retoma-se os aspectos abordados na Seção 3.4:

- Duração dos testes;
- Possibilidade de falhas humanas.

Afim de avaliar esses pontos, foram rodados os testes de SIL e HIL de forma completa, obtendo-se os resultados apresentados na Tabela 2, a qual mostra informações reais da execução dos testes em SIL e HIL.

Tabela 2 – Resultados de todos os testes em SIL e HIL.

Ambiente de Testes	**Cenários**	**Steps**	**Tempo**
SIL	17	100	41,005s
HIL	20	135	1m 42,883s

Pela Tabela 2, percebe-se que ao todo existem 17 cenários, com um total de 100 passos executados, sendo que a execução de todos esses cenários rodando em máquina local levou 41,005s. Vale ressaltar que o tempo de execução pode ser maior quando um desenvolvedor subir uma nova versão de código no repositório, pois a validação presente em uma *pipeline* valida muito mais do que apenas os testes automatizados. Contudo, mesmo levando em conta esse tempo a mais para a execução na *pipeline*, ainda assim os testes em SIL se mostram muito mais rápidos do que o que inicialmente era validado de forma manual pelo time de QA, que levava de 1 a 2 semanas para uma validação completa devido ao ciclo de desenvolvimento em *sprints*.

A análise demonstra que os testes desenvolvidos em SIL proporcionaram resultados significativos. Além de alta confiabilidade, eles asseguram que nenhuma alteração de código causará problemas na validação do *Track Controller*, graças à integração contínua.

Os testes em HIL, também mostrados na Tabela 2, mostram um total de 20 cenários executados, contendo um total de 135 passos e um tempo de execução total de 1m e 42,883s, um tempo muito inferior quando comparado com as 4h contínuas levadas de forma manual. Embora os testes em HIL ainda dependam da intervenção inicial de uma pessoa para serem executados, seu valor é substancial, uma vez que o processo de execução é simplificado a um único comando, facilitando a automação e garantindo eficiência no ciclo de teste do sistema. Após a execução, basta observar o relatório gerado e verificar o resultado, possibilitando ao membro do time de QA executar outras tarefas em paralelo enquanto os testes são executados no HIL.

Em resumo, o desenvolvimento dos testes automatizados para o *Track Controller* trouxeram resultados positivos bastante significativos para o produto, uma vez que a comunicação CAN existente foi validada de forma satisfatória, garantindo a qualidade do produto, impedindo alterações de código que prejudicem a eficiência e segurança, ao mesmo tempo em que possibilitam ao time de QA de trabalharem em outras validações tão importantes quanto para a empresa.

Isso resulta em uma redução de custos para a Hexagon neste produto, uma vez que diminui substancialmente o tempo necessário para a validação pelo time de QA. Além de otimizar os recursos da empresa. Evitar esses problemas não só previne a frustração do cliente com o produto, mas também garante que ele seja entregue ao mercado de forma mais robusta e confiável, fortalecendo a reputação da Hexagon e sua competitividade no setor.

6 CONSIDERAÇÕES FINAIS

Este trabalho abordou o problema de desenvolver e implementar testes automatizados para validar o piloto automático desenvolvido pela Hexagon, aplicado na agricultura de precisão. A necessidade surgiu da demanda por processos mais eficientes e seguros na validação de novas versões de *software TiX* e *firmware*, buscando mitigar os desafios enfrentados pelos métodos tradicionais de testes manuais.

Durante o desenvolvimento do projeto de automação e eficiência nos testes do Track Controller, foram enfrentados desafios significativos que demandaram soluções criativas e ajustes contínuos. Inicialmente, a falta de acesso a variáveis essenciais para os testes automatizados exigiu investigações detalhadas e colaboração estreita com o time de desenvolvimento para garantir a integridade dos testes. Além disso, a montagem do cenário HIL apresentou obstáculos, que foram superados com a aquisição de um adaptador USB-CAN e o desenvolvimento de um script dedicado. A introdução de um redutor ao eixo do motor nos testes em HIL inicialmente comprometeu a robustez dos testes de controle de referência, levando à implementação de tolerâncias aceitáveis para garantir a precisão dos resultados. Apesar desses desafios, os resultados foram altamente positivos: os testes em SIL conseguiram reduzir significativamente o tempo de validação, enquanto os testes em HIL melhoraram a eficiência operacional e a qualidade do produto. Essas melhorias não apenas otimizaram os recursos da empresa, mas também fortaleceram sua reputação e competitividade no mercado, assegurando a confiabilidade do Track Controller em situações reais de operação.

Como trabalhos futuros sugere-se a validação da comunicação com o *Track Controller* via API e a implementação de testes automatizados para demais produtos desenvolvidos pela empresa. Este avanço permitirá uma integração mais robusta e flexível com outros sistemas, facilitando a troca de informações e comandos de maneira mais eficiente. A implementação de uma API para comunicação com o *Track Controller* não só melhorará a interoperabilidade do sistema, mas também possibilitará uma maior escalabilidade e personalização, atendendo a demandas específicas de diferentes clientes e cenários operacionais.

A implementação de testes automatizados para os demais produtos da empresa trará diversos benefícios. Primeiramente, garantirá uma detecção mais rápida e precisa de falhas, aumentando a qualidade e a confiabilidade dos produtos. Além disso, permitirá uma validação contínua das funcionalidades, assegurando que novas alterações não comprometam o sistema existente. Isso resultará em maior eficiência no desenvolvimento e manutenção dos produtos, além de proporcionar maior confiança aos clientes e parceiros na estabilidade das soluções oferecidas pela empresa.

REFERÊNCIAS

B. HOFMANN-WELLENHOF, H. Lichtenegger; WASLE, E. **GNSS – Global Navigation Satellite Systems: GPS, GLONASS, Galileo, and more**. 1. ed. New York: Springer, 2007. Disponível em: https://link.springer.com/book/10.1007/978-3-211-73017-1. Acesso em: 17 jun. 2024.

BIRD, Christian; RIGBY, Peter C.; BARR, Earl T.; HAMILTON, David J.; GERMÁN, Daniel M.; DEVANBU, Premkumar. The Promises and Perils of Mining Git. *In*: PROCEEDINGS of the 6th Working Conference on Mining Software Repositories. [*S.l.*: *s.n.*], 2010. P. 1–10.

BOSCH, Robert. **Controller Area Network: Architectures, Protocols, and Standards**. Upper Saddle River, NJ: Prentice Hall, 1991.

BROWN, Alice. Quality Assurance Best Practices in Software Development. *In*: PROCEEDINGS of the International Conference on Software Engineering. [*S.l.*: *s.n.*], 2020. P. 112–125.

CHACON, Scott; STRAUB, Ben. **Pro Git**. [*S.l.*]: Apress, 2014.

COMMUNITY, DIO. **QA (Quality Assurance): qual a importância dessa área na tecnologia?** [*S.l.*], 2023. Disponível em: https://www.dio.me/articles/qa-quality-assurance-qual-a-importante-dessa-area-na-tecnologia. Acesso em: 31 jan. 2024.

CORRIGAN, Steve. **Introduction to the Controller Area Network (CAN)**. [*S.l.*], 2022. Disponível em: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf?ts=1716285300160&ref_url=https%253A%252F%252Fwww.google.co.uk%252F#:~:text=A%20controller%20area%20network%20(CAN,of%20a%20TI%20CAN%20transceiver.. Acesso em: 21 mai. 2024.

CUCUMBER.IO. **Behavior-Driven Development**. [*S.l.*: *s.n.*], 2024. Acesso em: 24 maio 2024. Disponível em: https://cucumber.io/docs/bdd/.

CUI, X. *et al.* Advances in Hardware-in-the-Loop Simulation for Autonomous Vehicles. **IEEE Access**, 2020. DOI: 10.1109/ACCESS.2020.2975710. Disponível em: https://ieeexplore.ieee.org/document/9025941. Acesso em: 24 mai. 2024.

DAVID CLAY, Francis J. Pierce e. **Precision Farming: Basics and Applications**. 2. ed. Madison, WI: ASA-CSSA-SSSA, 2007. Disponível em: https://www.researchgate.net/publication/265632148_Precision_Farming_Basics_and_Applications. Acesso em: 17 jun. 2024.

EMBRATOP. **O que é GPS?** [*S.l.*], 2022. Disponível em: https://www.embratop.com.br/noticias/o-que-e-gps/. Acesso em: 13 mai. 2024.

GAO, Y. *et al.* Applications of Hardware-in-the-Loop Simulation in Automotive and Aerospace Industries. **SAE International Journal of Commercial Vehicles**, 2017. DOI: 10.4271/2017-01-0580. Disponível em: https://www.sae.org/publications/technical-papers/content/2017-01-0580/. Acesso em: 24 mai. 2024.

HEXAGON. **HxGN AgrOn Piloto Automático: Sistema para a navegação automatizada de tratores**. [*S.l.*], 2024a. Disponível em: https://hexagon.com/pt/products/hxgn-agron-auto-steering. Acesso em: 28 mai. 2024.

HEXAGON. **HxGN AgrOn Track Controller: Sistema de direção eletromecânico para veículos agrícolas**. [*S.l.*], 2024b. Disponível em: https://hexagon.com/pt/products/hxgn-agron-track-controller. Acesso em: 28 mai. 2024.

LOELIGER, Jon; MCCULLOUGH, Matthew. **Version Control with Git**. [*S.l.*]: O'Reilly Media, 2012.

LONGLEY, Paul A.; GOODCHILD, Michael F.; MAGUIRE, David J.; RHIND, David W. **Geographic Information Systems and Science**. 4. ed. New York: Wiley, 2015.

MAHMOOD, A. *et al.* Hardware-in-the-Loop Simulation for Automotive Systems. **IEEE Transactions on Industrial Informatics**, 2021. DOI: 10.1109/TII.2021.3064650. Disponível em: https://ieeexplore.ieee.org/document/9381234. Acesso em: 24 mai. 2024.

MASON, Ryan. **Git Essentials**. [*S.l.*]: Packt Publishing, 2021.

MOLIN, José Paulo. **Agricultura de Precisão**. 1. ed. Piracicaba: Escola Superior de Agricultura "Luiz de Queiroz", 2006. Disponível em:

https://www.teses.usp.br/teses/disponiveis/11/11132/tde-20072006-113156/publico/Agricultura_de_Precisao.pdf. Acesso em: 17 jun. 2024.

NORTH, Dan. **Introducing BDD**. [*S.l.*], 2006. Disponível em: https://dannorth.net/introducing-bdd/. Acesso em: 13 mai. 2024.

OLIVEIRA, Larissa Taw Rumiana de. **Estudo de Caso da Utilização de Gherkin Para Automatização de Testes e Obtenção de Métrica de Cobertura de Requisitos**. Florianópolis, 2020. Disponível em: https://repositorio.ufsc.br/handle/123456789/218486. Acesso em: 13 mai. 2024.

PRESSMAN, R. S.; MAXIM, B. R. **Software Engineering: A Practitioner's Approach**. [*S.l.*]: McGraw-Hill Education, 2014.

ROVEDA, Ugo. **TESTES AUTOMATIZADOS: O QUE SÃO E COMO UTILIZÁ-LOS**. [*S.l.*], 2022. Disponível em: https://www.kenzie.com.br/blog/testes-automatizados. Acesso em: 1 fev. 2024.

SCHUH, G. *et al.* **Model-Based Design of Embedded Systems with Hardware-in-the-Loop Testing**. [*S.l.*]: Springer, 2014. DOI: 10.1007/978-3-642-45430-7. Disponível em: https://link.springer.com/book/10.1007/978-3-642-45430-7. Acesso em: 24 mai. 2024.

SMARTBEAR. **Cucumber Documentation**. [*S.l.*], 2024. Disponível em: https://cucumber.io/docs/cucumber/. Acesso em: 21 mai. 2024.

SOMMERVILLE, I. **Software Engineering**. [*S.l.*]: Pearson, 2015.

SOMMERVILLE, Ian. **Software Engineering**. 9th. Boston, MA: Addison-Wesley, 2011.

SOUZA CARNEIRO GOULART E ARYSTIDES RESENDE FILHO E ALUÍZIO BORÉM, Antonio Carlos de. **Agricultura de Precisão: Resultados de um novo olhar**. 1. ed. Brasília: Embrapa, 2014. Disponível em: https://www.embrapa.br/busca-de-publicacoes/-/publicacao/1019098/agricultura-de-precisao-resultados-de-um-novo-olhar. Acesso em: 17 jun. 2024.

SPINELLIS, Diomidis. Git: Version Control for Everyone. **IEEE Software**, v. 29, n. 3, p. 100–101, 2012.

ZHOU, G.; LIU, J.; WANG, X. Software-in-the-Loop Simulation of Control Systems. **IEEE Transactions on Industrial Electronics**, v. 57, n. 4, p. 1266–1273, 2010. DOI: 10.1109/TIE.2009.2032201. Disponível em: https://ieeexplore.ieee.org/document/5205541. Acesso em: 24 mai. 2024.