UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DEPARTAMENTO ACADÊMICO DE INFORMATICA

BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

JOSÉ HENRIQUE ROQUETTE

UMA ABORDAGEM UTILIZANDO BEHAVIOR DRIVEN DEVELOPMENT PARA GERAÇÃO DE CASOS DE TESTE: UM ESTUDO DE CASO NA ÁREA AUTOMOTIVA

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2018

JOSÉ HENRIQUE ROQUETTE

UMA ABORDAGEM UTILIZANDO BEHAVIOR DRIVEN DEVELOPMENT PARA GERAÇÃO DE CASOS DE TESTE: UM ESTUDO DE CASO NA ÁREA AUTOMOTIVA

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Bacharel, em Ciência da Computação, do Departamento de Informática, da Universidade Tecnológica Federal do Paraná.

Orientadora: Prof.ª Dra. Simone Nasser Matos

Coorientador: Prof. Dr. Max Mauro Dias Santos.

PONTA GROSSA

2018

.

TERMO DE APROVAÇÃO

UMA ABORDAGEM UTILIZANDO BEHAVIOR DRIVEN DEVELOPMENT PARA GERAÇÃO DE CASOS DE TESTE: UM ESTUDO DE CASO NA ÁREA AUTOMOTIVA

por

JOSÉ HENRIQUE ROQUETTE

Este Trabalho de Conclusão de Curso (TCC) foi apresentado(a) em treze (13) de novembro de dois mil e dezoito (2018) como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Simone Nasser Matos
Prof.(a) Orientadora

_____
Max Mauro Dias Santos
Prof.(a) Coorientador

_____
Prof. Eliana Claudia Mayumi Ishikawa
Membro titular

_____
Prof. Hugo Valadares Siqueira
Membro titular

_____
Profª. Dra. Helyane Borges
Responsável pelo Trabalho de Conclusão de Curso

_____
Prof. Dr. Saulo Jorge Beltrão de Queiroz
Coordenador do Curso

- O Termo de Aprovação assinado encontra-se na Coordenação do Curso -

AGRADECIMENTOS

Certamente estes parágrafos não irão atender a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do meu pensamento e de minha gratidão.

Agradeço aos meus professores orientadores Prof.ª Dra. Simone Nasser Matos e o Prof. Dr. Max Dias Santos pela oportunidade e sabedoria com que me guiou nesta trajetória.

Aos meus colegas que me auxiliaram em momentos de dificuldades.

Gostaria de deixar registrado também, o meu reconhecimento à minha família, principalmente a minha mãe, Rosângela Ferraça Roquette e aos meus irmãos José Maria Roquette Neto, José Guilherme Roquette e José Paulo Roquette, pois acredito que sem o apoio deles seria muito difícil vencer esse desafio.

Enfim, a todos os que por algum motivo contribuíram para a realização desta pesquisa.

RESUMO

ROQUETTE, J. H. **Uma Abordagem Utilizando do Behavior Driven Development para geração de casos de teste:** Um estudo de caso na área automotiva. 2018. 71f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2018.

As empresas automotivas adotam a metodologia em V e o método *Model-Based-Design* (MBD) para otimizar o processo de testes, assim usam uma engenharia simultânea com a implementação de um modelo ágil que possibilita a realização de testes incrementais. Mas, muitas vezes estes testes são elaborados pelas equipes de forma manual, empiricamente e não conseguem identificar todos os casos de tese do projeto, o que gera um maior custo e tempo porque a identificação de *bugs* no sistema pode ocorrer quando o software já foi embarcado. Este trabalho propõe uma abordagem que usa *Behavior Driven Development* (BDD) e outras técnicas de teste da engenharia de *software* para a criação e execução de testes em sistemas automotivos. A abordagem busca utilizar uma documentação para a automatização de geração de dados de entrada para realização de simulações de teste, trazendo benefícios ao projeto. O resultado da aplicação da abordagem proposta foi a identificação de mais casos de testes que podem ajudar na diminuição de custo e prevenção de erros futuros.

Palavras-chave: BDD. Teste de software. Testes automotivo.

ABSTRACT

ROQUETTE, J. H. **An Approach Using Behavior Driven Development to Generate Test Cases:** A Case Study in the Automotive Area. 2018. 71 p. Work of Conclusion Course Graduation in Computer Science - Federal Technology University. Ponta Grossa, 2018.

Automotive companies adopt the V-methodology and the Model-Based-Design (MBD) method to optimize the testing process, thus using simultaneous engineering with the implementation of an agile model that enables incremental testing. But often, these tests are handled empirically by teams and can not identify all cases of project thesis, which generates a greater cost and time because the identification of system bugs can occur when the software has already been shipped . This paper proposes an approach that uses Behavior Driven Development (BDD) and other software engineering test techniques for the creation and execution of tests in automotive systems. The approach seeks to use documentation for the automation of incoming data generation to perform test simulations, bringing benefits to the project. The result of the application of the proposed approach we have identified more cases of tests that can help in reducing costs and preventing future errors.

Keywords: BDD. Software testing. Automotive testing.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

BDD	*Behavior Driven Development*
DDD	*Domain Driven Development*
ECU	*Electronic Control Unit*
HIL	*Hardware-in-the-Loop*
MBD	*Model Based Design*
MIL	*Model-in-the-Loop*
PIL	*Processor-in-the-Loop*
SIL	*Software-in-the-Loop*
TDD	*Test Driven Development*
VIL	*Vehicle-in-the-Loop*

SUMÁRIO

1 INTRODUÇÃO

Dentro da engenharia de software foram estabelecidas etapas para construção de sistemas, Pressman (2011) define que o desenvolvimento de software é realizado em cinco atividades: *Comunicação, Planejamento, Modelagem, Construção* e *Emprego*. A atividade de *Comunicação* tem o objetivo de compreender os objetivos das partes interessadas no projeto e fazer o levantamento de funcionalidades e análise de requisitos. No *Planejamento* é realizado o cronograma de trabalho junto com as técnicas que vão ser utilizadas, riscos possíveis, os recursos necessários e o resultado que deve ser obtido. Na *Modelagem* são criados os modelos com o intuito de facilitar o entendimento das necessidades dos sistemas e como as partes devem se integrar. Na atividade de *Construção* ocorre a geração de código e testes necessários para identificar erros. Por fim, a última atividade é a de *Emprego,* na qual o software final é entregue ao cliente, que avalia o sistema e fornece o *feedback.*

Este trabalho tem ênfase na atividade de *Construção* que, segundo a DeveloperWorks (2012), é importante porque a descoberta de um erro em um programa durante sua fase de desenvolvimento pode trazer uma economia de gastos e aumentar a vida útil do sistema.

Na literatura existem diversas maneiras para construir testes de softwares. Uma delas é o *Behavior Driven Development* (BDD). Smart (2014) diz que a prática desta metodologia em um ambiente de construção de software pode ajudar a equipe de desenvolvimento a entregar um sistema de maior qualidade e de forma mais rápida. Essa prática utiliza o *Test-Driven Development* (TDD) e *Domain-Driven Desing* (DDD), mas tem sua principal característica em uma linguagem comum para criação de cenários, facilitando a comunicação entre os membros de um projeto.

Em BDD é comum começar identificando quais são os objetivos do programa e pensar quais funcionalidades o sistema pode ter para atingir tais objetivos. Em colaboração com o usuário são desenvolvidos cenários, automatizados em forma de especificações executáveis, os quais são usados para o desenvolvimento e documentação. O BDD em nível de código pode ajudar os desenvolvedores a escrever um código de alta qualidade, que possa ser melhor testado, tenha uma documentação abrangente sobre o que está sendo realizado, além disso pode vir a facilitar a manutenção do software.

O BDD utiliza de histórias e cenários com estruturas simples que mostram de forma clara qual é o objetivo da funcionalidade. Possuem expressões comuns do vocabulário tais como: Dado (*Given*), Quando (*When*), Então (*Then).* O BDD possui três etapas: *Dado* - descreve a precondição e prepara o teste; *Quando* - descreve a ação do teste; e *Então* - descreve o que é esperado do teste (SMART, 2014).

Utilizando o BDD em um projeto é possível que os cenários de testes sejam automatizados. Um teste automatizado pode ser construído por meio de ferramentas apropriadas. Apesar de um teste automatizado durar mais tempo em sua implementação, eles garantem a segurança na manutenção do produto e ainda podem ser executados em qualquer momento (FERNANDES, 2014).

Assim como no BDD, os cenários são usados também no setor automotivo. Banik (2017) afirma que os cenários surgiram nesta área devido a necessidade de se criar sistemas de veículos automotivos mais seguros, porém foi se estendendo a outras funcionalidades, como conforto e versatilidade. A linha de produção de um software automotivo não é diferente de um sistema commercial: o que difere os dois são os métodos de desenvolvimento e testes realizados dentro do projeto.

Uma das metodologias utilizadas para desenvolver softwares automotivos é o MBD (*Model Based Design*) que possui como centro do processo de desenvolvimento o modelo do sistema e tem objetivo de reduzir seu tempo e custo.

O MBD consiste na utilização de uma única plataforma para criação de um modelo executável em que são criados casos de testes manualmente a partir dos requisitos com o intuito de simular seu comportamento.

A execução de testes em um modelo executável se chama *Model-In-The-Loop* (MIL), após sua conclusão, é gerado um arquivo *.c* do sistema pela plataforma no qual está o modelo. Do *output* gerado na etapa anterior, tem-se o próximo passo de teste que é o *Software-In-The-Loop* (SIL) que consiste na realização de testes das funções do código em tempo real. Logo após, são realizados os testes para encontrar erros mais complexos na aplicação, esta etapa é chamada de *Processor-In-The-Loop* (PIL). Por fim, é efetuado o *download* do sistema em um *hardware* adequado e são realizados os testes que buscam erros de integração do sistema, chamado de *Hardware-In-The-Loop* (HIL) (NUNES, 2017).

Com a finalidade de melhorar o processo MBD, este trabalho propõe uma abordagem para automatizar os testes que são realizados na etapa do MIL, utilizando a estrutura de cenários do BDD junto com o Teste de Caminhos, outra

técnica de teste de software, para gerar os casos de teste de forma automatizada e encontrar mais casos de teste.

1.1 OBJETIVOS

O presente trabalho tem por objetivo geral propor uma abordagem usando BDD e a técnicas de teste de engenharia de software para geração de casos testes na área automotiva para prevenir erros futuros.

1.1.1 OBJETIVOS ESPECIFICOS

Os objetivos específicos para este trabalho são:

- Identificar os casos de testes no início do processo de teste para software automotivo.
- Criar uma ferramenta para a abordagem proposta.
- Aplicar a abordagem proposta na área automotiva.

1.2 ORGANIZAÇÃO DO TRABALHO

Este trabalho é composto por seis capítulos, os quais abrangem conceitos importantes para o desenvolvimento desta pesquisa. O capítulo 2 descreve a revisão da literatura sobre testes de software, apresentando uma breve introduçãosobre o que é teste de software e alguns métodos já utilizados para sua elaboração.

O capítulo 3 apresenta como são desenvolvidos softwares automotivos e quais tipos de testes são aplicados nesta área.

O capítulo 4 propõe a abordagem para realizar testes em software automotivos aplicando técnicas da literatura.

O capítulo 5 descreve uma aplicação da abordagem proposta em uma situação problema na área automotiva.

Por fim, o capítulo 6 faz as considerações finais sobre o trabalho e relata alguns trabalhos futuros que podem dar continuidade a esta pesquisa.

2 TESTE DE SOFTWARE E BDD

Este capítulo aborda alguns tópicos importantes para o desenvolvimento deste trabalho. A Seção 2.1 relata o porquê se deve realizar testes de software, a sua importância, vantagens e seus tipos. A Seção 2.2 descreve métodos ágeis para verificação de software e apresenta conceitos sobre testes de software automatizado. Por fim, a última seção relata as considerações finais do capítulo.

2.1 TESTE DE SOFTWARE

A etapa de teste de software busca encontrar os *bugs* do sistema antes da entrega final ao cliente e pode ser realizado de diferentes maneiras, por exemplo, testes na interface, tais como: em campos com entrada de valores diferentes do que o esperado, tipos de variáveis erradas, entre outros.

Cardoso (2010) descrevem em seu trabalho algumas das dificuldades em realizar testes de software, pois cada sistema possui suas próprias características como flexibilidade, complexidade, intangibilidade. A autora ainda acrescenta outros fatores que podem dificultar que as empresas executem a verificação de forma adequada, como o custo/benefício, visto que a atividade possui um custo mais elevado, além da necessidade de profissionais especializados para realizar as avaliações de software.

Segundo Gomes e Silva (2009), um teste é bem-sucedido quando encontra defeitos em um produto podendo apresentar dois resultados: i) o válido, que acontece o que é desejado que o software realize; ii) o inválido, que é aquele que o software se comporta de maneira não esperada e geralmente são executados com valores absurdos, que talvez o cliente final nunca irá exigir do sistema, mas são nesses casos que existem a maioria das falhas.

Na área de teste se tem diversas etapas, sendo elas, de *Unidade, Sistema, Aceitação, Regressão* entre outros, que podem ocorrer durante o desenvolvimento do software para validar o que está sendo programado.

Dentro de cada tipo de teste são aplicadas técnicas de caixa branca e caixa preta e são realizadas de duas maneiras: i) sem planejamento: na qual o testador irá tentar identificar erros no código de forma aleatória, e ii) planejado: em que será

realizado um plano de teste que deve acompanhar o desenvolvimento de software incluindo atividades de planejamento, projeto, execução e análise de resultados.

A seguir será detalhado a técnica de caixa branca, pois é utilizada para o desenvolvimento da abordagem proposta.

2.1.1 Teste de Caixa Branca

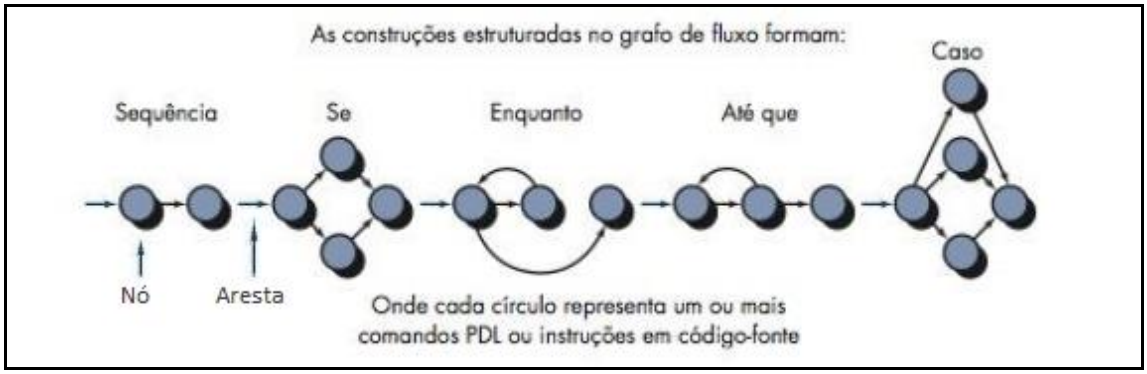
Os testes de Caixa Branca são aqueles que investigam o funcionamento interno do sistema, ou seja, esses trabalham com o código produzido pelos desenvolvedores. Segundo DeveloperWorks (2012) esses testes podem se restringir a componentes específicos ou abranger o programa como um todo.

Pressman (2011) diz que com essa técnica o testador pode criar casos de teste que garantam que todos os caminhos independentes sejam executados pelo menos uma vez, exercitem todas as decisões lógicas do sistema, executem todos os ciclos sem limites dentro de suas fronteiras operacionais e também possam exercitar estruturas de dados internas para assegurar sua validade.

Uma das estratégias de caixa branca estudada para esse trabalho foi o Teste de Caminho Básico que permite ao testador utilizar uma complexidade lógica de um projeto e essa medida define um conjunto de caminhos de execução. Segundo Pressman (2011), os casos de testes criados para praticar conjunto básico com certeza irá executar todas as instruções de um programa pelo menos uma vez.

Este tipo de teste trabalha com grafo de fluxo, que representa o fluxo de controle lógico, assim contém todas as condições de um código, por exemplo, *se... enquanto... até que.* Um exemplo de grafo de fluxo *é* ilustrado na Figura 1.

Figura 1 - Condições presentes em um grafo de fluxo



Fonte: Pressman (2011)

Em um grafo de fluxo, cada círculo representa um nó que significa um comando procedural, as setas são as arestas que mostram o fluxo que o grafo deve seguir. Por meio de um grafo de fluxo de um algoritmo consegue-se analisar todos os caminhos independentes e sua complexidade ciclomática.

Um caminho independente é qualquer caminho por meio do código que introduz um conjunto de comandos de processamentos ou uma nova condição. Um caminho independente deve incluir pelo menos uma aresta diferente de outros caminhos já definidos.

A complexidade cliclomática de um grafo de fluxo dá-se por meio de um cálculo da quantidade de arestas menos o número de nós presentes no mesmo mais o valor dois, como é exibido na Quadro 1. Com esse cálculo é possível verificar quantos caminhos independentes o grafo de fluxo possui, pois este é uma métrica de software que fornece uma medida quantitativa da complexidade lógica de um código que define a quantidade de caminhos independentes.

Quadro 1 - Fórmula da Complexidade Cliclomática

A complexidade ciclomática V(G) para um grafo de fluxo G é definida como:
$V(G) = E - N + 2$
Em que E é o número de arestas do grafo de fluxo e N é o número de nós do grafo de Fluxo

Fonte: Pressman (2011)

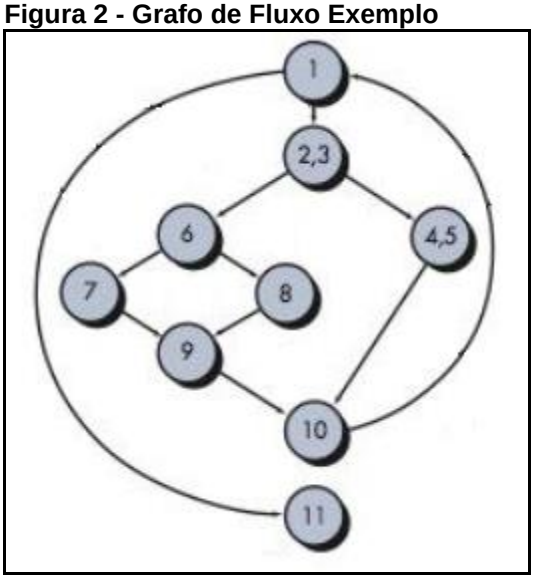
O Teste de Caminho Básico é composto por quatro etapas:

- Etapa 1: Definir o grafo de fluxo que representa o algoritmo a ser testado;
- Etapa 2: Determinar a complexidade ciclomática do grafo de fluxo;
- Etapa 3: Definir os caminhos independentes existentes;

- Etapa 4: Preparar os casos de testes que vão executar cada caminho definido na etapa anterior.

Para ilustrar melhor a segunda e terceira etapa se pode utilizar um grafo de fluxo presente no livro de Pressman (2011), exibido na Figura 2, denominado de grafo *G*.

Figura 2 - Grafo de Fluxo Exemplo



Fonte: Pressman (2011)

Depois de ter o grafo de fluxo, é possível realizar o cálculo da complexidade ciclomática do mesmo. Para isso é necessário calcular a quantidade de nós e de arestas presentes no grafo. O grafo da Figura 2 possui onze arestas e nove nós, de acordo com fórmula é realizado a subtração de arestas e se soma dois ao final, obtendo o valor quatro, como é possível verificar no Quadro 2.

Quadro 2 - Complexidade Ciclomatica do Grafo G

V(G) = Arestas – Nós + 2
V(G) = 11 – 9 + 2
V(G) = 2 + 2
V(G) = 4

Fonte: Pressman (2011)

Após ter definido a complexidade ciclomática do grafo *G*, é identificado a quantidade de caminhos independentes que o mesmo possui, que no caso é 4 (quatro). Os caminhos estão apresentados no Quadro 3

Quadro 3 - Caminhos Independentes do Grafo G

Caminho 1: 1-11
Caminho 2: 1-2-3-4-5-10-1-11
Caminho 3: 1-2-3-6-8-9-10-1-11
Caminho 4: 1-2-3-6-7-9-10-1-11

Fonte: Pressman (2011)

Observa-se que outros caminhos diferentes dos listados no Quadro 2 são combinações dos mesmos, por exemplo, o caminho: 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11, portanto, não precisa ser testado.

Tendo estes caminhos é possível construir os casos testes que executem tais comandos abrangendo todos os caminhos independentes possíveis do código, os quais são compostos pelos vértices e arestas do grafo, cujo os vértices são o estado em que o grafo se encontra e as arestas são seus caminhos, tendo um vértice de origem e um de destino.

A seguir será apresentado uma introdução sobre o que é desenvolvimento ágil e uma das suas técnicas para elaboração de testes de software.

2.2 DESENVOLVIMENTO AGIL

Segundo Pressman (2011), o desenvolvimento ágil surgiu devido a necessidade de melhorar o processo de construção de software que eram identificados dentro da engenharia de software convencional. Conforme o autor, um dos problemas é manter a fluidez dentro de um projeto de software, ou seja, dentro de um projeto é difícil ter o conhecimento de todos os requisitos logo no início ou prever alterações que o cliente deseja realizar enquanto o mesmo já está em construção.

O desenvolvimento ágil de software é uma alternativa para que a equipe que adota essa metodologia tenha a capacidade de aplicar mudanças no sistema durante seu desenvolvimento de forma que não prejudique o andamento do projeto. Pressman (2011) chama o desenvolvimento ágil como se fosse “engenharia de *software* flexível”.

Para uma equipe ser ágil, a *Agile Alliance* (2001) estabeleceu onze (11) princípios que devem ser seguidos, sendo eles:

1. Satisfazer o cliente por meio de uma entrega rápida e adiantada.

2. Atender os pedidos de alterações, mesmo atrasados.
3. Entrega de software em funcionamento frequente, dando preferência a intervalo curtos.
4. As partes interessadas e os desenvolvedores devem trabalhar em conjunto ao longo do projeto.
5. Construir projetos com uma equipe motivada, dando apoio e o ambiente necessário para que possam trabalhar.
6. A equipe deve manter uma comunicação aberta e presencial, de forma que todos saibam o que estão realizando no projeto.
7. Software em funcionamento é a principal medida de progresso.
8. Processos ágeis promovem desenvolvimento sustentável, os envolvidos no projeto devem estar capacitados para manter o mesmo ritmo por tempo indefinido.
9. Simplicidade é essencial.
10. As melhores arquiteturas, requisitos e projetos emergem de equipes que se auto organizam.
11. Intervalos regulares, a equipe de desenvolvimento se avalia para ver como se tornar mais eficiente, então ajustar seu comportamento com o que seria mais eficiente.

Dentro do desenvolvimento ágil é possível encontrar diferente técnicas, como o BDD que utiliza de cenários para produção de testes, que será explicado a seguir.

2.2.1 Behavior Driven Development (BDD)

Os conceitos teóricos apresentados nesta seção foram adaptados de Smart (2014) por isto os parágrafos não foram referenciados. Este é o principal livro sobre o assunto.

O *Behavior Driven Development* (BDD) pode ser usado quando o cliente conversa com o analista de negócio sobre as funcionalidades necessárias em sua aplicação.

Após o analista de negócio ter tais informações, ele fala com o criador de teste e o desenvolvedor e os três juntos constroem cenários que exibem o que as

funcionalidades devem executar de modo que os desenvolvedores vão se guiar por meio dos cenários.

Os testadores utilizam tais cenários como base para seus testes; tudo isso gera testes automatizados que provem aos envolvidos um grande *feedback* que ajuda na documentação da ferramenta.

Como pode-se perceber quando utilizando o BDD, as três partes precisam se relacionar para criar os cenários, que são escritos em uma linguagem comum, podendo ela ser o português, inglês ou outro idioma. Com esse trabalho em equipe é mais fácil de reduzir a perda de informação e a quantidade de mal-entendidos, o que diminui a quantidade de erros ou eventos do gênero.

Para que não haja perda de tempo em manutenção, muitas organizações começaram a utilizar o BDD, que é uma prática da engenharia de *software* de ajudar times a construir e entregar um programa com mais valor e de alta qualidade da forma mais rápida. Essa prática utiliza o *Test-Driven Development (TDD) e Domain-Driven Desing (DDD)*, mas tem sua principal característica uma linguagem comum para a criação de cenários e exemplos, facilitando a comunicação entre os membros de um projeto.

Em BDD é comum começar identificando quais são os objetivos do programa e pensar nas funcionalidades do sistema. Em colaboração com o usuário são desenvolvidos exemplos para essas funcionalidades e são automatizados em forma de especificações executáveis, as quais são usadas para o desenvolvimento e documentação. O BDD na produção de código pode ajudar os desenvolvedores a escreverem um código de alta qualidade, melhor testado, documentado e mais fácil de fazer a manutenção.

Esses exemplos são escritos em formato de cenários em BDD, que possuem sua própria estrutura para que possam ser realizados os testes. Os cenários ajudam o entendimento e a comunicação da equipe no momento de resolver algum problema. Eles agem como a base para o desenvolvimento de critérios de aceitação, que futuramente tornam-se critérios de aceitação automatizados em conjunto com os testes automatizados. Os cenários ajudam os *designers* a construir interfaces funcionais para o usuário e colabora com os desenvolvedores para descobrir novos recursos para o projeto.

Na criação de cenários é importante o entendimento de quais são os objetivos do programa, pois permite determinar os recursos com um valor maior para o projeto. No desenvolvimento ágil as equipes gostam de escrever uma breve descrição do recurso que será desenvolvido como apoio na criação de cenários, seguindo um padrão, em que o primeiro escreve a intenção da ferramenta, depois a quem ela é direcionada e por final o resultado, conforme exibe o Quadro 4.

Quadro 4 - Cenário em BDD

Cenário: Título do cenário Given < passo que representa a pré-condição para um evento > When < passo que representa o que ocorre no evento > Then < passo que representa a saída de um evento >

Fonte: Adaptado Smart (2014)

Para facilitar a automatização dos critérios de aceitação, os cenários têm uma estrutura pré-determinada na linguagem que utiliza expressões “*Given*”, “*When*” e “*Then*”, traduzindo essas expressões para o português se tem Dado (*Given*), Quando (*When*), Então (*Then*), E (And) e Mas (*But*). Cada expressão possui um significado: *Dado* descreve a precondição e prepara o teste; *Quando* descreve a ação do teste; e *Então* relata o que é esperado do teste; as expressões *E* e *Mas* são utilizadas entre as expressões que se pode dizer que são as principais, conforme ilustrado no Quadro 4.

2.3 CONSIDERAÇÕES DO CAPÍTULO

Neste capítulo foi apresentado conceitos sobre teste de software, detalhando os conceitos para a realização do Teste de Caminhos, estratégia de teste de caixa branca.

Foi abordado a importância do desenvolvimento ágil e a técnica do BDD, explorando a elaboração de cenários para a realização de testes.

Os conceitos trabalhados neste capítulo são necessários para a compreensão da abordagem proposta para realização de testes na área automotiva. Abordagens de testes na área automotiva são descritas no próximo capítulo.

3 ABORDAGEM DE DESENVOLVIMENTO DE SOFTWARE AUTOMOTIVO

Este capítulo aborda alguns tópicos importantes para o entendimento deste trabalho referentes a abordagem de desenvolvimento de software automotivo que podem ser encontrados na literatura. A Seção 3.1 relata sobre o desenvolvimento de software automotivo. A Seção 3.2 descreve o desenvolvimento baseado em modelo para criação de testes automotivos. Por fim, a última seção apresenta as considerações finais do capítulo.

3.1 DESENVOLVIMENTO DE SOFTWARE AUTOMOTIVO

No desenvolvimento de software automotivo o processo ocorre da seguinte forma: é feito a coleta de requisitos, normalmente escrita de forma textual, em seguida é proposto um hardware que seja mais adequado para atender as necessidades do cliente. Após realizada a coleta de requisitos, a equipe começa a dar início a aquisição dos componentes necessários e desenvolvimento do código do processador. Nesta etapa, é comum reutilizar componentes de software de outros produtos (NUNES, 2017).

Com o protótipo finalizado, o sistema é embarcado na ECU (*Electronic Control Unit*) e são realizados testes básicos acompanhados pelos projetistas de software e hardware. Após a conclusão dos testes básicos, o protótipo é entregue ao departamento de testes que realiza a verificação mais elaboradas criando um ambiente similar ao utilizado pelo veículo (NUNES, 2017).

Esse processo pode apresentar diversos problemas, trazendo prejuízos ao projeto, pois cria barreiras entre as etapas do processo e causa uma ineficiência de tempo, recursos e validação, como é ilustrado na Figura 3 (STELLA, 2015).

Figura 3 - Abordagem tradicional de desenvolvimento automotivo

Fonte: Stella (2015)

O desenvolvimento de softwares automotivos busca por otimizações sejam elas em redução de custo de combustíveis, aumento de conforto, segurança de passageiros e custos na produção do produto final que vai ser entregue para o cliente. Para obter estas otimizações está sendo estudado o uso de produtos em ambientes agressivos o qual possui variações de temperatura, umidade, vibração, emissão eletromagnética. Por meio destes estudos os níveis de confiabilidade e disponibilidade de sistemas de automotivos possuem mais durabilidade e segurança (NUNES, 2017).

O desenvolvimento de software embarcados possui uma sequência de procedimentos que são definidos pelas práticas e ferramentas utilizadas por quem irá desenvolvê-lo. Isto é estabelecido no início do desenvolvimento, pois tem o objetivo de padronizar a documentação, o reuso de software, a portabilidade de componentes, redução do tempo de desenvolvimento, segurança das aplicações e garantia da confiabilidade.

Este processo é dividido em duas fases de acordo com o *Modelo V*: Desenvolvimento e\ Verificação e Validação. Cada uma dessas possui cinco etapas, sendo uma delas em comum (conforme ilustra a Figura 4). A Fase de Desenvolvimento é constituída pelas seguintes etapas de Hermans (2011):

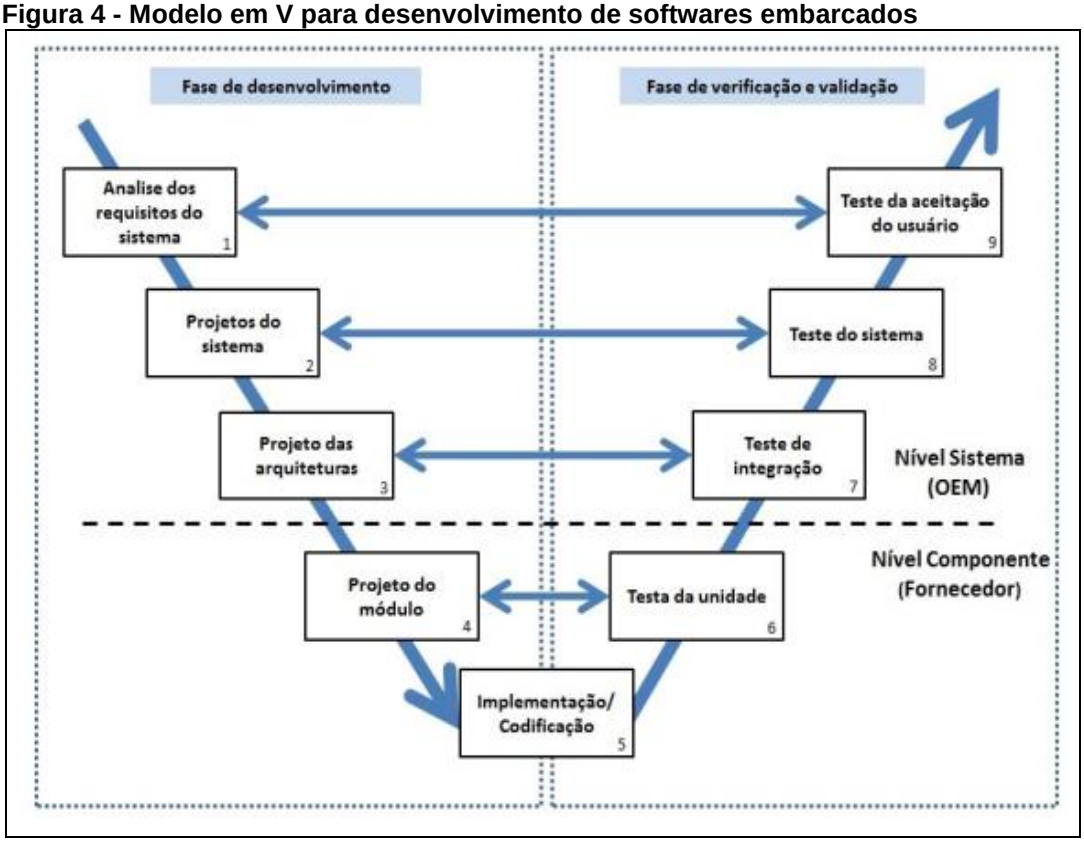
1. Análise de requisitos de sistema.
2. Projeto do sistema.
3. Projeto das arquiteturas.
4. Projeto do módulo.

5. Implementação/Codificação.

A Fase de Verificação e Validação possui etapas de testes, que são referentes com cada etapa da Fase de Desenvolvimento:

1. Implementação/Codificação.
2. Teste de unidade.
3. Teste de integração.
4. Teste de sistema.
5. Teste de aceitação do usuário.

Figura 4 - Modelo em V para desenvolvimento de softwares embarcados



Fonte: Stella (2015)

Uma das vantagens do *Modelo em V* é que o seu processo de testes não é realizado por meio de suas especificações, mas sim dos seus requisitos. Desse modo, eles são realizados com o objetivo de verificar as funcionalidades específicas pretendendo atingir um único objetivo, ao contrário de testes que são realizados a partir de suas especificações, que buscam avaliaras características de um dispositivo (STELLA, 2015).

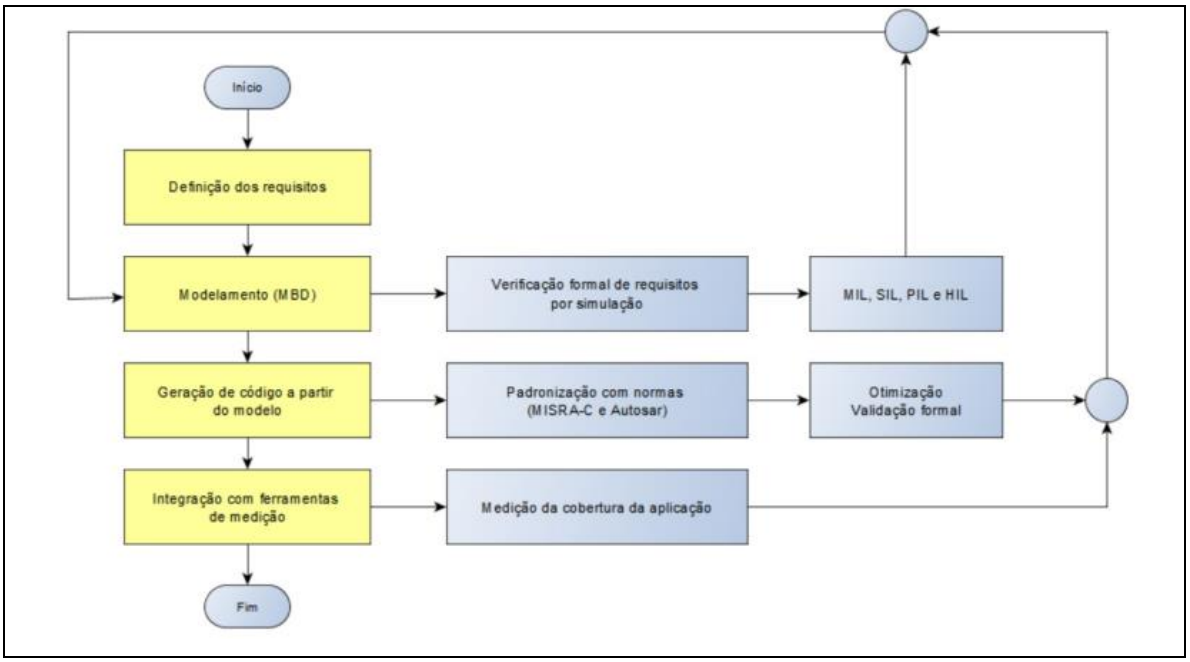
Além do Modelo em V, tem-se o projeto baseado em modelos usado na área automotiva e descrito na próxima seção.

3.2 DESENVOLVIMENTO BASEADO EM MODELO

O MBD (*Model-Based-Design*) é uma metodologia aplicada dentro do projeto devido sua agilidade no desenvolvimento e a quantidade de testes que é possível de se realizar no projeto. Além disto, o *Model-Based-Design* pode trazer algumas vantagens porque utiliza uma plataforma única em que o sistema pode ser desenvolvido e testado, deixando o desenvolvimento mais ágil.

Nunes (2017) elaborou um fluxograma descrevendo os passos do Desenvolvimento Baseado em Modelos orientados para a qualidade de software de uma forma de mais baixo nível, Figura 5, onde os blocos amarelos representam as metodologias e os retângulos azuis ilustram os procedimentos recomendados.

Figura 5 - Fluxograma de Desenvolvimento Baseado em Modelos orientado para qualidade de software



Fonte: Nunes (2017)

O processo se inicia com as definições de requisitos, na qual são usadas técnicas da engenharia de requisitos para definir o que o cliente deseja que seja realizado pelo produto. É importante que nessa etapa não exista ambiguidades ou mal-entendidos, pois problemas como esses podem causar perda do foco do que está sendo produzido. A geração de conflitos dentro da equipe de desenvolvimento pode ser causada por: a má comunicação, metas difíceis de serem alcançadas, falta de documentação, entre outros problemas segundo Smart (2014).

A utilização do MBD (*Model-Based-Design*), que é utilizado no processo em V, possui verificações como:

- *Model-In-the-Loop* (MIL): neste método não se tem o *hardware* envolvido. Ele consiste em criar um modelo para aplicação virtual, para testar um modelo que representa a planta física do sistema;
- *Software-In-the-Loop* (SIL): utilizado para testar o funcionamento da aplicação com funções em tempo real. É baseado na geração de um código que representa um modelo da aplicação;
- *Processor-In-the-Loop* (PIL): usado para identificar erros mais complexos. Utiliza um processador ou plataforma de desenvolvimento para atuar com a lógica da aplicação;
- *Hardware-In-the-Loop* (HIL): neste já se tem o *software* gravado no *hardware* da aplicação conectado com um outro *hardware* que simula a planta real. Este tipo de teste procura encontrar erros de integração.

A partir do modelo criado é obtido um código no qual o mesmo tem que seguir uma padronização do MISRA-C e do *Autosar*, passando por um processo de otimização (NUNES, 2017). Isso permite que o código tenha uma padronização de escrita, gaste menos recursos computacionais e o tempo de execução e segurança sejam priorizados.

Podem ser utilizadas ferramentas de validações para checar os erros comuns, por exemplo, lógica que não está sendo usada, divisão por zero entre outros. A adoção de uma ferramenta de medição de cobertura tem o objetivo de realizar uma última validação com o intuito de melhorar a qualidade final do produto.

A abordagem de Desenvolvimento Baseado em Modelos, possui como centro do processo de desenvolvimento o modelo do sistema e tem objetivo dentro do projeto de reduzir seu tempo e custo, assim utiliza uma única ferramenta que permite a criação da planta do sistema e seu controlador. Isso traz como principal vantagem a facilidade de entendimento e visualização o que diminui a possibilidade que os componentes individuais não se encaixem de maneira otimizada, além de aumentar qualidade de software (Nunes, 2017).

De acordo com o material da *Mathworks* (2018), o MBD tem algumas vantagens quando comparado com outras abordagens mais tradicionais de desenvolvimento, sendo elas:

- Um ambiente de projeto que integra funções e componentes;
- Possibilidade de localizar e corrigir problemas no início do projeto;
- Reutilização de projetos para atualizações ou para sistemas derivativos;
- Geração automática de código.

O *Model-Based-Design* é uma forma possível para realizar a construção de um sistema, utilizando uma única plataforma de desenvolvimento. Seu processo se inicia com a coleta de requisitos, porém ao invés de serem descritos de forma textual como em abordagens tradicionais, os requisitos são utilizados para desenvolver uma máquina de estado do sistema. Após sua construção, são elaborados testes das funções com o intuito de encontrar e remover erros.

A partir da máquina de estado elaborada o código *.c* é gerado automaticamente a partir da ferramenta que está sendo usada para desenvolver o projeto. Em seguida, o código gerado é testado e verificado (NUNES, 2017). Concluída esta etapa o sistema é embarcado com o modelo.

Cada uma dessas etapa do processo é testada de forma continua, utilizando o que foi elaborado na parte de desenvolvimento para validar os testes. Os testes que são realizados ainda no modelo, são compostos por dados de entrada que irão servir para realizar uma simulação da máquina de estado proposto, gerando assim um *output* de como o sistema iria se comportar naquela situação. Essa etapa é o que chamamos de *Model-in-the-Loop* e será detalhada na próxima seção porque a abordagem proposta pretende melhor a criação de casos de teste na fase de modelo.

3.2.1 *Model-in-the-loop (MIL)*

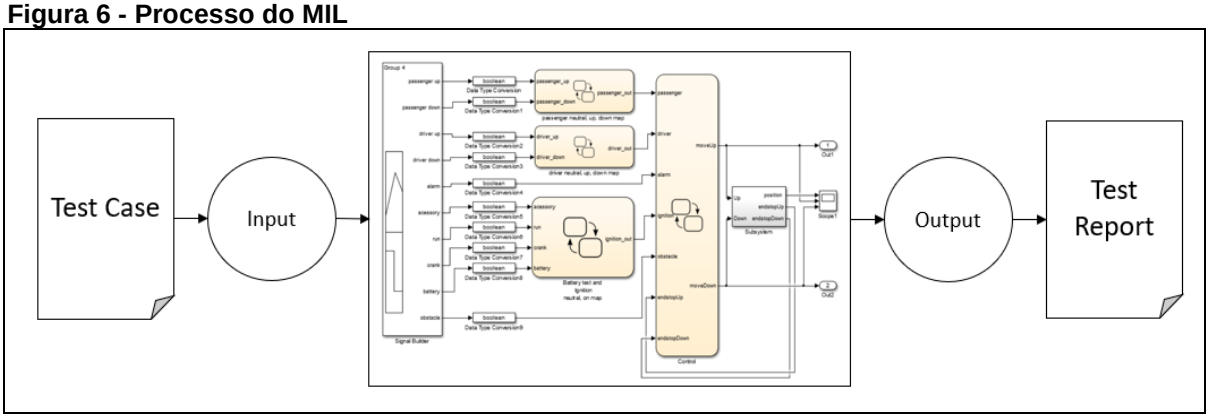
Testes MIL podem ser definido como uma simulação básica na qual são realizados na fase inicial do projeto, a fim de analisar o modelo do controlador juntamente com o modelo da planta do sistema através de uma simulação (SHOKRY; HINCHEY; 2009).

Geralmente o controlador é implementado em ferramentas como o Matlab / Simulink (2018) e a planta modelo é construída no Simulink (2018) no qualexiste

uma conexão entre o controlador e a planta, conforme relatam Vidanapathirana e Dewasurendra (2013).

Por meio dessas ferramentas, é possível atribuir entradas de sinal que correspondem aos casos de teste que devem ser verificados pelos testadores. Os casos servem como entradas para a execução da simulação que gera as saídas e chamada de relatório de teste, conforme apresentado na Figura 6.

Figura 6 - Processo do MIL



Fonte: Autoria própria.

Os relatórios de teste devem ser analisados para verificar se a simulação está sendo executada como esperado. Se a saída gerada for um valor inesperado, provavelmente ocorreu algum problema com o ambiente o qual o software está sendo desenvolvido ou provavelmente existiu um erro sintaxe no momento da construção do modelo (SHOKRY; HINCHEY, 2009).

Devido a este tipo de teste ser realizado de forma manual e exigir experiência e conhecimento do profissional, erros podem ser cometidos tais como: nem todos os casos de testes foram detectados para validar o sistema ou a documentação não é completa, o que causa mal-entendimento dentro da equipe.

3.3 CONSIDERAÇÕES DO CAPITULO

Neste capítulo foi apresentado conceitos sobre o desenvolvimento de software automotivo, focando no desenvolvimento baseado em modelos (MBD).

Relatou-se que os testes automotivos podem ser realizados em fase inicial da criação do software que será embarcado posteriormente, ou seja, no modelo MIL. Este modelo não contempla a identificação de todos os casos de testes e a documentação não é completa, o que acarreta maior custo e tempo de projeto. Por

isto, a abordagem proposta oferece uma solução para o problema e será descrita no próximo capítulo.

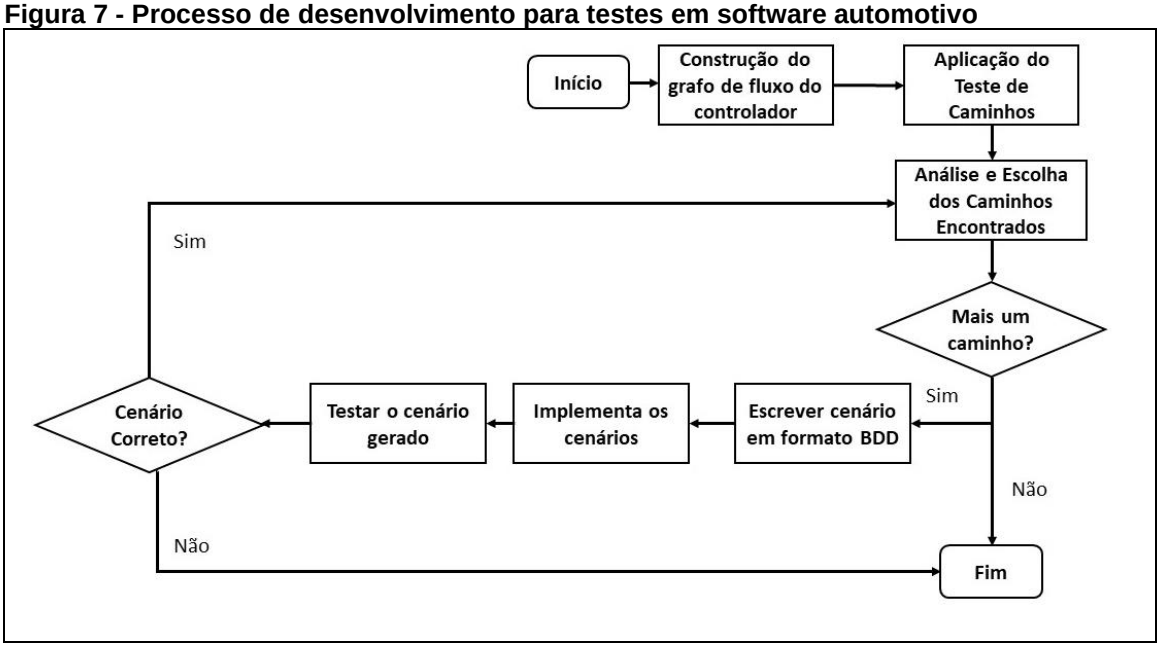
4 ABORDAGEM PROPOSTA

Este capítulo descreve a abordagem proposta para minimizar os problemas relacionados aos testes automotivos que foram descritos no capítulo anterior. A seção 4.1 apresenta a visão geral da abordagem proposta. A seção 4.2 descreve passo-a-passo quais procedimentos devem ser adotados para a execução de teste de software automotivo por meio da abordagem. Por fim, a seção 4.3 relata as considerações finais do capítulo.

4.1 VISÃO GERAL DA ABORDAGEM

Esta abordagem busca solucionar os problemas sobre a identificação da quantidade de casos de testes que serão produzidos na área automotiva, além de criar uma documentação maior sobre o que está sendo realizado dentro do projeto. A documentação é elaborada no formato de cenários no padrão utilizado no BDD, podendo ser usada para a geração de dados de entrada para realização de teste MIL em plataformas como o Simulink (2018). A visão geral da abordagem é exibida na Figura 7.

Figura 7 - Processo de desenvolvimento para testes em software automotivo



Fonte: Autoria própria

A entrada da abordagem são os requisitos do sistema e o controlador do sistema, suas etapas são a *Construção do grafo de fluxo, Aplicação do teste de*

caminhos, Análise dos caminhos encontrados, Escrever o cenário em formato BDD, Implementar o cenário e *Testar o cenário* que serão detalhadas na próxima seção.

4.2 ETAPAS DA ABORDAGEM

As seções seguintes descrevem cada etapa da abordagem proposta.

4.2.1 Construção do Grafo de Fluxo

Nesta etapa, deve-se verificar as condições propostas pela análise de requisitos a qual já deve estar elaborada e realizar um estudo sobre o controlador construído.

Por meio do conhecimento sobre os requisitos do sistema é possível definir como deve ser o funcionamento do mesmo. Realizando a abstração dos requisitos e do controlador dados como entrada, pode-se identificar quem atua dentro do sistema e as suas variáveis. Desta forma, o grafo de fluxo apresenta quem está acionando cada variável.

Variáveis normalmente quando acionadas realizam algum evento, estes por sua vez devem ser encontrados e identificados conforme a variável que pode ocasionar aquela ação. Um evento pode gerar outros e como consequência se tem subeventos a partir dos quais pode-se criar uma lista com quem atua no sistema, quais são suas variáveis, que eventos elas podem ocasionar e quais acontecimentos podem vir ocorrer em seguida. Com isso, é possível criar um grafo de fluxo de como o sistema pode se comportar.

As informações podem definir as condições de *loops,* prioridades do sistema, "*if's",* que devem estar presentes no grafo de fluxo. A análise de requisitos permite definir as arestas do grafo dando um significado para cada uma.

Ao analisar o controlador programado, que é uma máquina de estados, é possível identificar quais são os estados que estão conectados com outros. Com este entendimento, pode-se abstrair os vértices do grafo (nós), sendo cada um deles o estado atual o qual o sistema está executando.

Após compreender o funcionamento do sistema através de seus requisitos e seu controlador, deve-se elaborar o grafo de fluxo que contenha um estado inicial e

terminal, sendo que as arestas são oriundas dos requisitos e os nós (estados) são obtidos pelo controlador.

4.2.2 Aplicação do Teste de Caminhos

Após a criação do grafo de fluxo, pode-se obter os fluxos que o sistema, deve seguir aplicando o teste de Caminhos no grafo.

Esta técnica permite percorrer o grafo de fluxo buscando os caminhos independentes do mesmo. Esses caminhos dentro do grafo são representados por arestas e cada caminho independente deve incluir pelo menos uma aresta diferente dos outros caminhos identificados.

Esta etapa por sua vez pode ser realizada de forma manual, a qual o testador irá percorrer e anotar todos os caminhos do grafo de fluxo. Outra solução é automatizar a busca de caminhos em um grafo usando algum algoritmo escolhida pelo usuário, como o algoritmo Dag All Paths (FELCHAR; ROQUETTE; 2018), que busca todos os caminhos dentro de um grafo.

Com os caminhos identificados, tem-se a representação dos casos de teste do sistema.

4.2.3 Análise e Escolha dos Caminhos Encontrados

Nesta etapa o testador analisa todos os caminhos independentes encontrados, verificando o conjunto de ações presentes em cada um. Durante a análise se deve verificar se existem caminhos impossíveis de acontecer ou redundantes ao sistema, pois estes são descartados.

Na fase de análise, é possível classificá-los utilizando algum grau de liminaridade entre os caminhos encontrados, este que por sua vez devem ser escolhidos pela equipe. Por exemplo, um problema que temos vários caminhos que representam a ação do motorista dentro do carro, podemos classificar todos esses cenários pertencentes a um grupo Motorista, com isto a equipe terá uma documentação mais abrangente sobre o que está sendo trabalhado dentro do projeto.

Concluído a análise e interpretação dos caminhos encontrados, deve-se escolher individualmente um delespara escrevê-lo em formato de cenário em formato BDD.

4.2.4 Escrever o Cenário em Formato BDD

O primeiro passo é definir o título do cenário, ele servirá para descrever o que está acontecendo naquele cenário. Após o título estabelecido é necessário verificar quais são as situações para que o evento daquele caminho ocorra e estas devem ser escritas na etapa *Dado* do cenário.

Na próxima etapa do cenário em BDD, tem-se o *Quando*, que descreve a ação do sistema que será testada. Por último, utilizando os requisitos é estabelecido como o sistema deve se comportar após realizar tal evento, isto é escrito na parte *Então* do cenário.

Com o cenário concluído, tem-se a geração de dados de entrada para cada caso de teste encontrado, porém, escritos de forma textual. O cenário construído representa a documentação do caso de teste.

4.2.5 Implementar o Cenário

Dado um cenário, tem-se de forma clara e concisa do que está ocorrendo, em qual situação o sistema estava para ele ocorrer e como é esperado que seja seu comportamento.

A partir destas informações o testador deve ser capaz de implementar o ambiente de teste, determinando as variáveis de entrada que o controlador deve receber para executar o cenário que deve ser testado.

Esta implementação pode ser realizada de forma manual, a qual o testador prepara o ambiente de teste manualmente, se adotado o MBD como metodologia, o ambiente de teste é desenvolvido na mesma plataforma que o controlador foi construído. Outra forma é usar uma ferramenta automatizada que implementa e integra a documentação e a geração de testes. Neste trabalho, criou-se uma ferramenta (*Basalt*) que possibilita esta integração e é explicada no próximo capítulo.

Com o ambiente de teste definido é possível realizar a simulação de teste do controlador com os dados de entrada.

4.2.6 Testar o Cenário

Estabelecido os preparativos para que o teste possa ser executado, o testador deve rodar uma simulação para a verificação do controlador com aquele determinado ambiente. O resultado desta simulação é como o controlador se comportou dado o ambiente definido.

Dado o *output* gerado pela simulação realizada, quem o realiza deve analisar os dados de saída obtidos e compará-los com a etapa *Então* do cenário utilizado para a elaboração daquele caso de teste. Se o resultado encontrado não corresponder com o que foi definido dentro do cenário, este por sua vez deve ser atribuído como “reprovado”, indicando que houve algum erro com o sistema. Caso o contrário, se o comportamento do controlador corresponder com o que foi definido dentro do cenário receberá o valor “aprovado” e o testador deve voltar a etapa de *Analise e Escolha de Um Caminho* e repetir o processo até que não haja mais cenários para serem testados ou todos tenham sido aprovados.

4.3 CONSIDERAÇÕES DO CAPÍTULO

Neste capítulo foi apresentado a abordagem proposta por este trabalho, descrevendo passo a passo cada etapa. Seguindo cada etapa, o usuário deve ser capaz de realizar teste em software automotivos desde que possua os requisitos do sistema e o controlador.

Os resultados da abordagem proposta para a área automotiva estão descritos no próximo capítulo.

5 RESULTADOS

Este capítulo descreve os resultados obtidos por este trabalho. A seção 5.1 relata uma situação problema na área automotiva em que a abordagem foi aplicada para a geração de testes. A seção 5.2 apresenta a ferramenta implementada para a automatização da geração dos casos. A seção 5.3 descreve a aplicação da abordagem na situação problema. A seção 5.4 relata uma análise comparativa da abordagem proposta com trabalho de literatura. Por fim, a última seção narra as considerações finais do capítulo.

5.1 SITUAÇÃO PROBLEMA

Um sistema de janela elétrica de um carro é um subsistema que pertence a um veículo. Ele interage com uma série de componentes mecânicos para realizar atividades como abrir/fechar a janela, detecção de objetos pesados, controle de interruptores e de unidade.

Considerando somente a janela do lado do passageiro do veículo, o motorista e o passageiro podem acionar o interruptor para fechar, abrir ou parar a janela e qual o sinal do motorista tem prioridade sobre o do passageiro. Durante a movimentação de fechar a janela, o vidro pode encontrar algum obstáculo que impeça de concluir seu percurso de fechar.

Seguindo a análise de requisitos elaborada pelos autores Santos *et al.* (2015), foram identificados dez requisitos que deveriam estar presentes no sistema, sendo eles:

- **Requisito 1**: A janela deve estar completamente fechada ou aberta em 4 segundos.
- **Requisito 2**: O vidro deve começar a se movimentar 200 ms depois do comando ter sido selecionado.
- **Requisito 3**: Após 4 segundos em movimentação em uma única direção, sem nenhuma interrupção, o motor deve desligar.
- **Requisito 4**: O sistema de controle deve operar entre uma voltagem de 12,5V e 14,5V.

- **Requisito 5**: Se o comando de fechar ou abrir a janela continuar pressionado entre 200ms e 1 segundo a janela deve estar aberta ou fechada completamente.
- **Requisito 6**: Se um obstáculo for detectado, o vidro deve parar e abaixar aproximadamente 10 cm.
- **Requisito 7**: O sistema de detecção de obstáculos tem prioridade sobre os sinais do motorista e do passageiro.
- **Requisito 8**: O sinal do motorista tem prioridade sobre o sinal do passageiro.
- **Requisito 9**: Quando acionado o sistema de segurança, a janela deve fechar completamente ao menos que tenha algum obstáculo no percurso.
- **Requisito 10:** O sistema de controle só pode ser operado se na ignição estiver posicionado em "*On*", junto com o *Acessory* e *Run*.

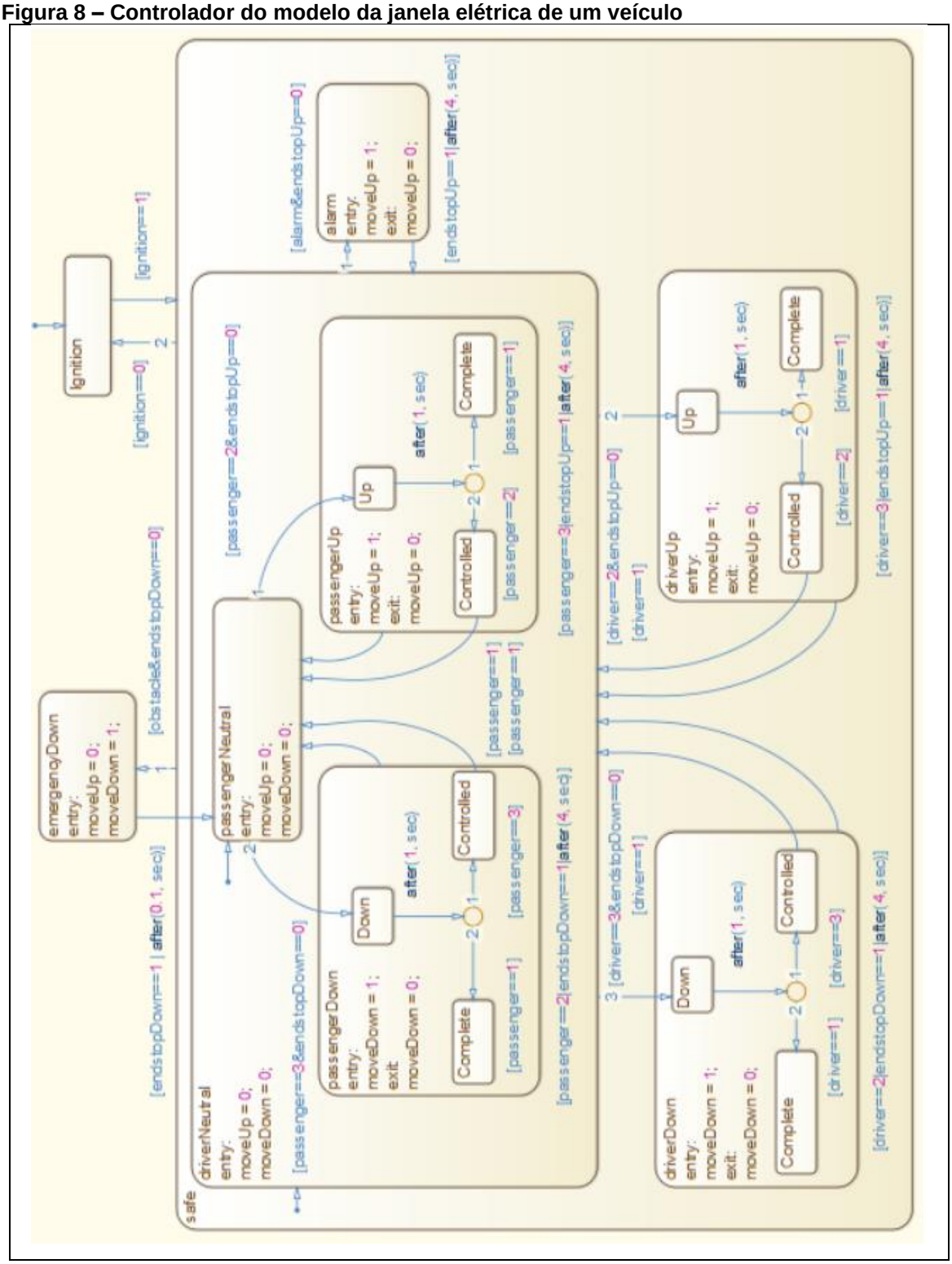
O sistema da janela foi modelado por Santos *et al*. (2015) atendendo os requisitos identificados e implementado na ferramenta do *Simulink*. A Figura 8 mostra o controlador que foi desenvolvido pelos autores e representado por uma máquina de estado.

O controlador é composto por uma série de variáveis que podem ter dois tipos de valores 0 ou 1, sendo que 0 representa que a variável está desliga e 1 ligada. As variáveis utilizadas são:

- *Passenger Up* e *Passenger Down* – representa o sinal do passageiro para movimentar a sua janela para cima e para baixo.
- *Driver Up* e *Driver Down* – representa o sinal do motorista para movimentar a janela do passageiro para cima e para baixo.
- *End Stop* – limite o qual o vidro pode chegar, sendo ele para cima ou para baixo.
- *Obstacle* – detecção de objetos.
- *Alarm* – *status* do sistema de segurança.
- *Battery* – atribui o valor da bateria do carro.
- *Crank* – representa a manivela.

- *Accessory* – representa se algum dos acessórios do carro está ligados
- *Run* – necessária para ligar o automóvel.

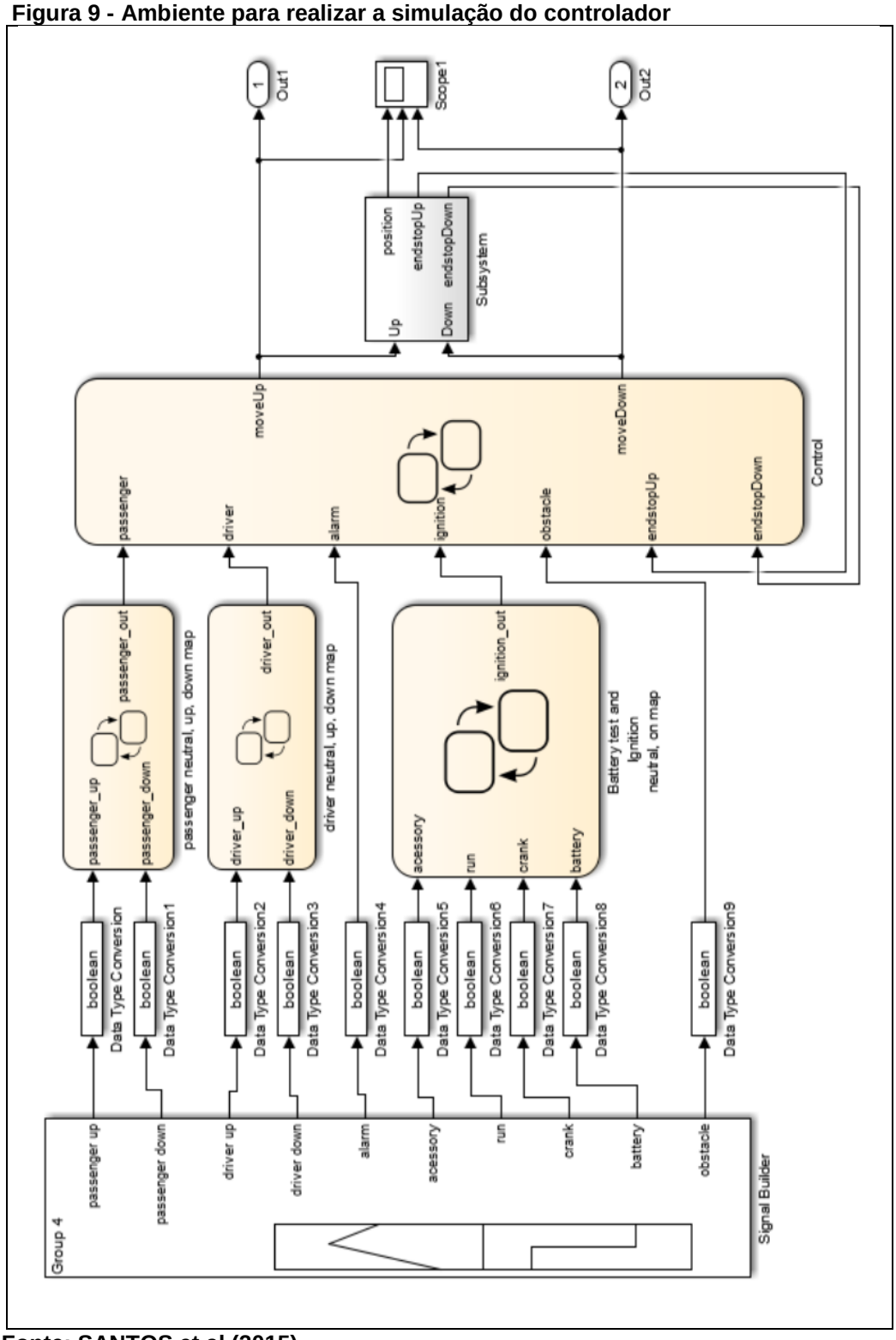
Figura 8 – Controlador do modelo da janela elétrica de um veículo



Fonte: Santos *et al*. (2015)

O controlador implementado no *Simulink*, além de permitir a construção do controlador, também possibilita que seja criado o ambiente de teste para a realização de simulações. A Figura 9 apresenta o ambiente para realização de simulações.

Figura 9 - Ambiente para realizar a simulação do controlador



Fonte: SANTOS et al (2015)

Com os requisitos e o controlador descritos nesta seção, pode-se realizar a aplicação da abordagem proposta.

5.2 BASALT: AUTOMOTIVE TESTING TOOL

Basalt Automotive testing tool é uma ferramenta proposta por este trabalho que é capaz de automatizar a produção de casos de teste por meio de cenários semelhante ao BDD.

Esta ferramenta tem o objetivo de simplificar a programação de *inputs* utilizando cenários, reduzindo o tempo e recursos durante a fase de testes de um projeto.

Os cenários em *Basalt* seguem a mesma estrutura de etapas dos cenários de BDD, mostrando o propósito de cada funcionalidade. Portanto, o cenário suportado pela ferramenta também possui a etapa *Dado.* Para descrever o evento que será testado tempo a etapa *Quando,* e a por último a etapa *Então* que representa a saída esperada.

Os cenários em *Basalt* contém detalhadamente cada etapa, estipulando valores de tempo dentro do cenário.

Diferente dos cenários em BDD, é preciso declarar as variáveis que são utilizadas naquele cenário, dessa forma, todas as variáveis que não são alteradas durante a simulação devem ser declaradas no cenário com valores estáticos. Já as variáveis que são utilizadas dentro dos cenários podem ser fixas e com valores tabulares.

Cenários com valores fixos possuem um valor estático, ou seja, não são alteradas. As variáveis que estão presentes no texto antes do seu nome devem ser acompanhadas pelo símbolo “$” e os seus valores de tempo que serão dois valores, o primeiro é o instante que a variável é acionada e o segundo o valor de tempo o qual a variável volta seu valor a zero, e devem vir acompanhadas de “%”. O quadro 5 apresenta um exemplo de um cenário escrito com valores fixos.

Quadro 5 - Cenário em Basalt com valores fixos

```
time = 10;
passenger_up = 0;
passenger_down = 0;
driver_down = 0;
alarm = 0;
accessory = 0;
run = 0;
crank = 0;
battery = 1;

Título: Motorista envia um sinal de subida e encontra um obstáculo
Dado que a $drive_up é acionada no intervalo de %2 até %5
Quando a $obstacle é acionada no intervalo de %6 até %7
Então o vidro automático deve alternar a direção de seu movimento
```

Fonte: Autoria própria

Basalt também aceita cenários com valores tabulares, na qual as variáveis são colocadas de forma genérica no meio do cenário e é construída uma tabela abaixo do cenário com os valores para cada variável. O Quadro 6 apresenta um exemplo com cenários tabulares.

Quadro 6 - Cenário em Basalt com valores tabulares

```
time = 10;
passenger_up = 0;
passenger_down = 0;
driver_down = 0;
alarm = 0;
accessory = 0;
run = 0;
crank = 0;
battery = 1;

Título: Motorista envia um sinal de subida e encontra um obstáculo
Dado que a $variavel1 é acionada no intervalo de %tempo1 até %tempo2
Quando a $variavel2 é acionada no intervalo de %tempo3 até %tempo4
Então o vidro automático deve alternar a direção de seu movimento

|variavel1  |variavel2  |tempo1 |tempo2|tempo3|tempo4|
|drive up   |obstacle   |2      |5     |4     |6     |
|drive up   |obstacle   |3      |5     |6     |8     |
|drive up   |obstacle   |2      |5     |1     |4     |
```

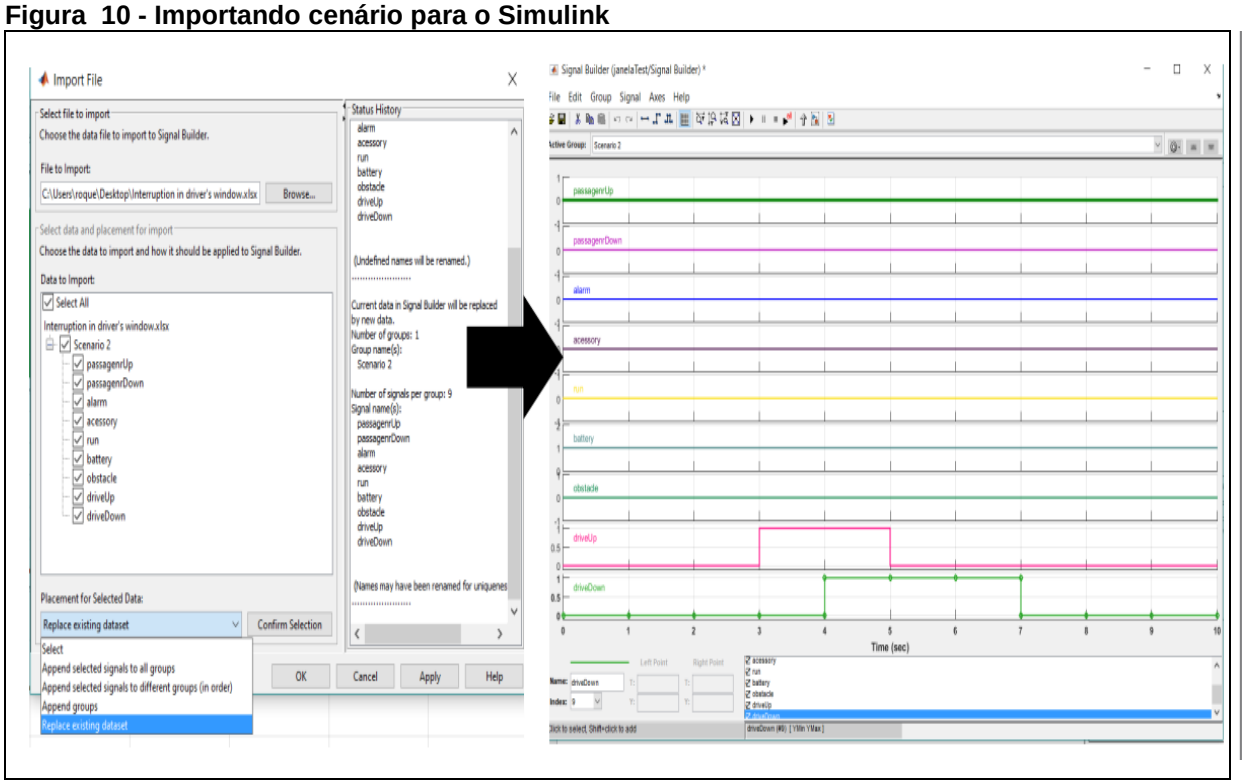
Fonte: Autoria própria

Percebe-se que as variáveis são declaradas da mesma maneira das que não usam tabela, porém seus nomes são genéricos e tem valores correspondentes para cada linha da tabela, ou seja, cada linha da tabela gera um caso de teste diferente.

Dado como entrada um cenário seguindo os padrões da ferramenta, ela irá gerar como saída uma planilha com os valores estabelecidos dentro do cenário. Esta

planilha possui uma estrutura que permite importá-la para o *Simulink* e gerar a produção de caso de teste, como ilustrado na Figura 10.

Figura 10 - Importando cenário para o Simulink



Fonte: Autoria própria

A seguir se apresenta a uso da ferramenta proposta na elaboração de cenários de testes para a área automotiva.

5.3 APLICAÇÃO DA ABORDAGEM PROPOSTA

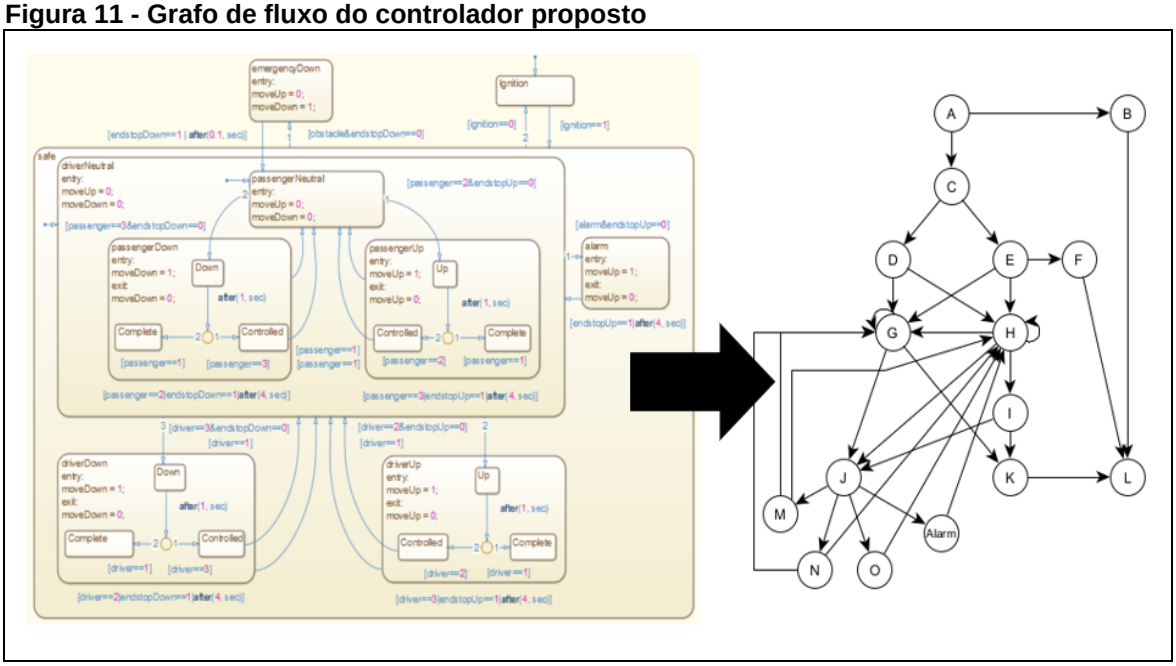
Dado o controlador, Figura 8, utilizado como entrada para aplicação da abordagem proposta, é possível realizar sua análise verificando suas operações lógicas e requisitos para a elaboração de um grafo de fluxo que atenda os caminhos que o sistema pode seguir. A Figura 11 apresenta o grafo de fluxo que representa o controlador proposto de solução.

Para sua construção abstraímos os atuadores do sistema relacionando com suas variáveis: o atuador "*Motorista"* responsável pelas variáveis "*Driver Up"* e *"Driver Down", "Passageiro" pode acionar* "*Passenger Up"* e *"Passenger Down"* e por último "*Ninguém"* que controla as variáveis "*Obstacle*" e "*Alarm*".

Em seguida, verifica-se quais eventos podem ocorrer no sistema. Foram identificados os seguintes eventos: *“envio de um sinal durante mais de um segundo para abrir ou fechar a janela”, “envio de um sinal durante menos de um segundo para abrir ou fechar a janela”, “a o vidro automático encontrou um obstáculo”, “interrupção na movimentação do vidro”* e “*o alarme do carro foi disparado”.*

Ao relacionar os eventos com suas respectivas variáveis é possível definir o grafo de fluxo do controlador, conforme ilustra a Figura 11.

Figura 11 - Grafo de fluxo do controlador proposto



Fonte: Autoria própria

O grafo da Figura 11 tem seu estado inicial em A, ou seja, é o vidro parado. O primeiro “*if”* define se a bateria está ligada ou não, se não irá para o estado B e terminar o processo em L. Se irá para o estado C, cujo o *if* indica que motorista mandou ou não um sinal. Se sim, prossegue-se para D, senão para E, em E tem-se a mesma situação, porém, agora para o passageiro. Se ele mandar o sinal vai para o estado G ou H, senão para F, que significa que ninguém mandou sinal então indo para o estado L.

Nos estados D e E, pode-se seguir para os estados G e H, onde respectivamente representam o movimento de subir e descer do vidro. Ambos os estados podem ter o mesmo destino I e K. O segundo representa que o sinal enviado foi maior que 1 segundos, sendo movimentado até a duração do seu sinal e parando (L). Quanto ao estado I, este significa que seu sinal durou menos que 1 segundo e deve continuar seu movimento (J), o nó M representa que o sistema não

encontrou nenhuma interrupção ou obstáculo, N significa interrupções que podem ocorrer, sendo elas para cima e para baixo. O nó O significa o encontro de um obstáculo pela janela e por último o nó referente ao *Alarm*.

Com o grafo de fluxo construído é possível aplicar o Teste de Caminho para identificar os casos de teste necessários. Neste trabalho foi utilizado um código elaborado por Felchar e Roquette (2018), que recebe um grafo de entrada e como saída retorna os caminhos encontrados desse grafo, porém o grafo precisa seguir algumas regras, não são permitidos ciclos ou *loops* dentro do grafo.

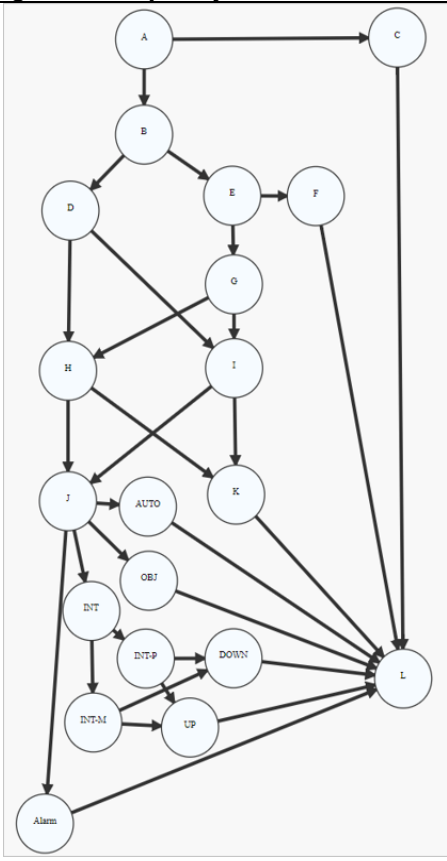
Visto que o grafo construído no passo anterior possui ciclos e *loops*, foi necessário a remoção dos mesmos criando novos nós que satisfizesse as condições estabelecidas pelo programa utilizado.

Os estados necessários foram “OBJ” que significa que o objeto foi encontrado e suas medidas foram tomadas, “AUTO” simbolizando que o sistema funcionou de forma automática até o final de sua atividade, dois estados INT-M e INT-P referentes a interrupções do motorista e passageiro e “DOWN” e “UP”, representando uma interrupção para baixo ou para cima.

Após aplicar o teste de caminho no grafo, foram encontrados 24 (vinte e quatro) caminhos independentes, Figura 12, sendo eles divididos em três categorias: *Neutral*, no qual ninguém envia um sinal, *Driver*, são os caminhos onde o motorista envia o primeiro sinal e *Passenger* representam os caminhos que em que o passageiro envia o primeiro sinal.

Possuindo os caminhos independentes do sistema, foi possível encontrar pelo menos 32 possíveis casos de teste a serem implementados, porém ao analisar pode-se identificar alguns caminhos que não precisam ser convertidos em cenários, isto por serem situações impossíveis de acontecer. Por exemplo, tem-se o caminho "A"->"B"->"E"->"G"->"I"->"J"->"OBJ"->"L", ao analisar verifica-se que é um caminho cujo o passageiro envia um sinal para o vidro abrir, ou seja, para baixo, segundo o cenário no meio do seu percurso a janela encontraria um obstáculo, porém como a janela já está se movimentando para baixo, não é possível que ela encontre um obstáculo no trajeto, sendo assim um cenário impossível de acontecer.

Figura 12 - Aplicação do teste de caminhos do grafo de fluxo



0: "A"->"B"->"D"->"H"->"J"->"AUTO"->"L“
1: "A"->"B"->"D"->"H"->"K"->"L“
2: "A"->"B"->"D"->"H"->"J"->"OBJ"->"L“
3: "A"->"B"->"D"->"H"->"J"->"INT"->"INT-M"->"UP"->"L”
4: "A"->"B"->"D"->"H"->"J"->"INT"->"INT-M"->"DOWN"->"L“
5: "A"->"B"->"D"->"H"->"J"->"INT"->"INT-P"->"UP"->"L“
6: "A"->"B"->"D"->"H"->"J"->"INT"->"INT-P"->"DOWN"->"L“
7: "A"->"B"->"D"->"I"->"J"->"AUTO"->"L“
8: "A"->"B"->"D"->"I"->"K"->"L“
9: "A"->"B"->"D"->"I"->"J"->"OBJ"->"L“
10: "A"->"B"->"D"->"I"->"J"->"INT"->"INT-M"->"UP"->"L"
11: "A"->"B"->"D"->"I"->"J"->"INT"->"INT-M"->"DOWN"->"L"
12: "A"->"B"->"D"->"I"->"J"->"INT"->"INT-P"->"UP"->"L"
13: "A"->"B"->"D"->"I"->"J"->"INT"->"INT-P"->"DOWN"->"L“
14: "A"->"B"->"E"->"G"->"H"->"J"->"AUTO"->"L“
15: "A"->"B"->"E"->"G"->"H"->"K"->"L“
16: "A"->"B"->"E"->"G"->"H"->"J"->"OBJ"->"L“
17: "A"->"B"->"E"->"G"->"H"->"J"->"INT"->"INT-M"->"UP"->"L“
18: "A"->"B"->"E"->"G"->"H"->"J"->"INT"->"INT-M"->"DOWN"->"L“
19: "A"->"B"->"E"->"G"->"H"->"J"->"INT"->"INT-P"->"UP"->"L"
20: "A"->"B"->"E"->"G"->"H"->"J"->"INT"->"INT-P"->"DOWN"->"L“
21: "A"->"B"->"E"->"G"->"I"->"J"->"AUTO"->"L“
22: "A"->"B"->"E"->"G"->"I"->"K"->"L“
23: "A"->"B"->"E"->"G"->"I"->"J"->"OBJ"->"L“
24: "A"->"B"->"E"->"G"->"I"->"J"->"INT"->"INT-M"->"UP"->"L"
25: "A"->"B"->"E"->"G"->"I"->"J"->"INT"->"INT-M"->"DOWN"->"L"
26: "A"->"B"->"E"->"G"->"I"->"J"->"INT"->"INT-P"->"UP"->"L"
27: "A"->"B"->"E"->"G"->"I"->"J"->"INT"->"INT-P"->"DOWN"->"L“
28: "A"->"B"->"E"->"F"->"L“
29: "A"->"C"->"L“
30: "A"->"B"->"D"->"H"->"J"->"Alarm"->"L“
31: "A"->"B"->"E"->"G"->"H"->"J"->"Alarm"->"L"

Fonte: Autoria própria

Ao analisar todos os caminhos, foram encontrados (dois) cenários com situações impossíveis, igual a descrita a cima.

Tabela 1 - Análise dos caminhos encontrados

Cenários	Quantidade
Situações impossíveis	2
Referentes ao motorista	13
Referentes ao passageiro	13
Neutro	4
TOTAL	32

Fonte: Autoria própria

Desta forma sobraram 30 caminhos que devem ser escritos em formato de cenário BDD e testados, eles foram classificados usando como métrica quem executa o evento. Desse modo, classificamos os caminhos em três categorias: cenários referentes ao motorista, ao passageiro e neutro, como ilustrado na Tabela 1. Cenários classificados como neutros, são os cenários em que o próprio sistema é o atuador naquele caminho, ou seja, não são referentes ao motorista ou passageiro.

No primeiro grupo de cenários, referente ao motorista, foram encontrados 13 caminhos, sendo eles presentes na Quadro 7.

Quadro 7 - Caminhos referentes ao motorista

Test Case ID	Caminho	Título
0	A->B->D->H->J->AUTO->L	Motorista enviou sinal para cima por menos de 1 segundo
1	A->B->D->H->K->L	Motorista enviou sinal para cima por mais de 1 segundo
2	A->B->D->H->J->OBJ->L	Motorista enviou sinal para cima e encontrou um objeto
3	A->B->D->H->J->INT->INT-M->UP->L	Motorista enviou sinal para cima e teve interrupção do motorista para cima
4	A->B->D->H->J->INT->INT-M->DOWN->L	Motorista enviou sinal para cima e teve interrupção do motorista para baixo
5	A->B->D->H->J->INT->INT-P->UP->L	Motorista enviou sinal para cima e teve interrupção do passageiro para cima
6	A->B->D->H->J->INT->INT-P->DOWN->L	Motorista enviou sinal para cima e teve interrupção do passageiro para baixo
7	A->B->D->I->J->AUTO->L	Motorista enviou sinal para baixo por menos de 1 segundo
8	A->B->D->I->K->L	Motorista enviou sinal para baixo por mais de 1 segundo
9	A->B->D->I->J->OBJ->L	Motorista enviou sinal para baixo e encontrou um objeto
10	A->B->D->I->J->INT->INT-M->UP->L	Motorista enviou sinal para baixo e teve interrupção do motorista para cima
11	A->B->D->I->J->INT->INT-M->DOWN->L	Motorista enviou sinal para baixo e teve interrupção do motorista para baixo
12	A->B->D->I->J->INT->INT-P->UP->L	Motorista enviou sinal para baixo e teve interrupção do passageiro para cima
13	A->B->D->I->J->INT->INT-P->DOWN->L	Motorista enviou sinal para baixo e teve interrupção do passageiro para baixo

Fonte: Autoria própria

Os caminhos encontrados referentes ao passageiro estão descritos na Quadro 8.

Quadro 8 - Caminhos referentes ao motorista

Test Case ID	Caminho	Título
14	A->B->E->G->H->J->AUTO->L	Passageiro enviou sinal para cima por menos de 1 segundo
15	A->B->E->G->H->K->L	Passageiro enviou sinal para cima por mais de 1 segundo
16	A->B->E->G->H->J->OBJ->L	Passageiro enviou sinal para cima e encontrou um objeto
17	A->B->E->G->H->J->INT->INT-M->UP->L	Passageiro enviou sinal para cima e teve interrupção do motorista para cima
18	A->B->E->G->H->J->INT->INT-M->DOWN->L	Passageiro enviou sinal para cima e teve interrupção do motorista para baixo
19	A->B->E->G->H->J->INT->INT-P->UP->L	Passageiro enviou sinal para cima e teve interrupção do passageiro para cima
20	A->B->E->G->H->J->INT->INT-P->DOWN->L	Passageiro enviou sinal para cima e teve interrupção do passageiro para baixo
21	A->B->E->G->I->J->AUTO->L	Passageiro enviou sinal para baixo por menos de 1 segundo
22	A->B->E->G->I->K->L	Passageiro enviou sinal para baixo por menos de 1 segundo
23	A->B->E->G->I->J->OBJ->L	Passageiro enviou sinal para menos e encontrou um objeto
24	A->B->E->G->I->J->INT->INT-M->UP->L	Passageiro enviou sinal para baixo e teve interrupção do motorista para cima
25	A->B->E->G->I->J->INT->INT-M->DOWN->L	Passageiro enviou sinal para baixo e teve interrupção do motorista para baixo
26	A->B->E->G->I->J->INT->INT-P->UP->L	Passageiro enviou sinal para baixo e teve interrupção do passageiro para cima
27	A->B->E->G->I->J->INT->INT-P->DOWN->L	Passageiro enviou sinal para baixo e teve interrupção do passageiro para baixo

Fonte: Autoria própria

Por último, tem-se os caminhos classificados como neutros, exibido na Quadro 9.

Quadro 9 - Caminhos referentes ao neutro

Test Case ID	Caminho	Titulo
28	A->B->E->F->L	O carro está ligado, mas ninguém envia sinal
29	A->C->L	o carro está desligado
30	A->B->D->H->J->Alarm->L	Alarm interrompe o sinal do motorista para subida
31	A->B->E->G->H->J->Alarm->L	Alarm interrompe o sinal do motorista para subida

Fonte: Autoria própria

Com a análise de todos os caminhos identificados pelo Teste de Caminhos, deve-se escolher um dos caminhos para escrevê-lo no formato de cenário em BDD. Neste trabalho será exibido o caso de teste “#16 - passageiro mandou o sinal de subida por mais de um segundo e encontrou um objeto”.

O caso de teste #16 é uma situação que a janela do passageiro está aberta, o passageiro aciona seu sinal para realizar a movimentação da janela para cima e no meio do percurso identifica um objeto, tendo assim que parar e descer aproximadamente 10 cm, segundo os requisitos estabelecidos. O Quadro 10 ilustra o cenário BDD para o caso de teste #16.

Quadro 10- Caminho escrito no formato BDD

Título: passageiro enviou o sinal de subida por mais de um segundo e encontrou um objeto ***Dado*** que o vidro do passageiro está aberta ***Quando*** o vidro encontrar um obstáculo ***Então*** a janela deve parar e descer 10 cm

Fonte: Autoria própria

Com o cenário escrito em formato de BDD, deve-se iniciar a implementação do mesmo. Neste trabalho foi utilizado da ferramenta *Basalt*, em que se modificou um pouco o cenário escrito no Quadro 10, definindo as variáveis necessárias para a simulação funcionar e o intervalo de tempo das variáveis que serão testadas naquele cenário. O cenário modificado é apresenta o Quadro 11.

Quadro 11 - Caminho escrito no formato BDD para Basalt

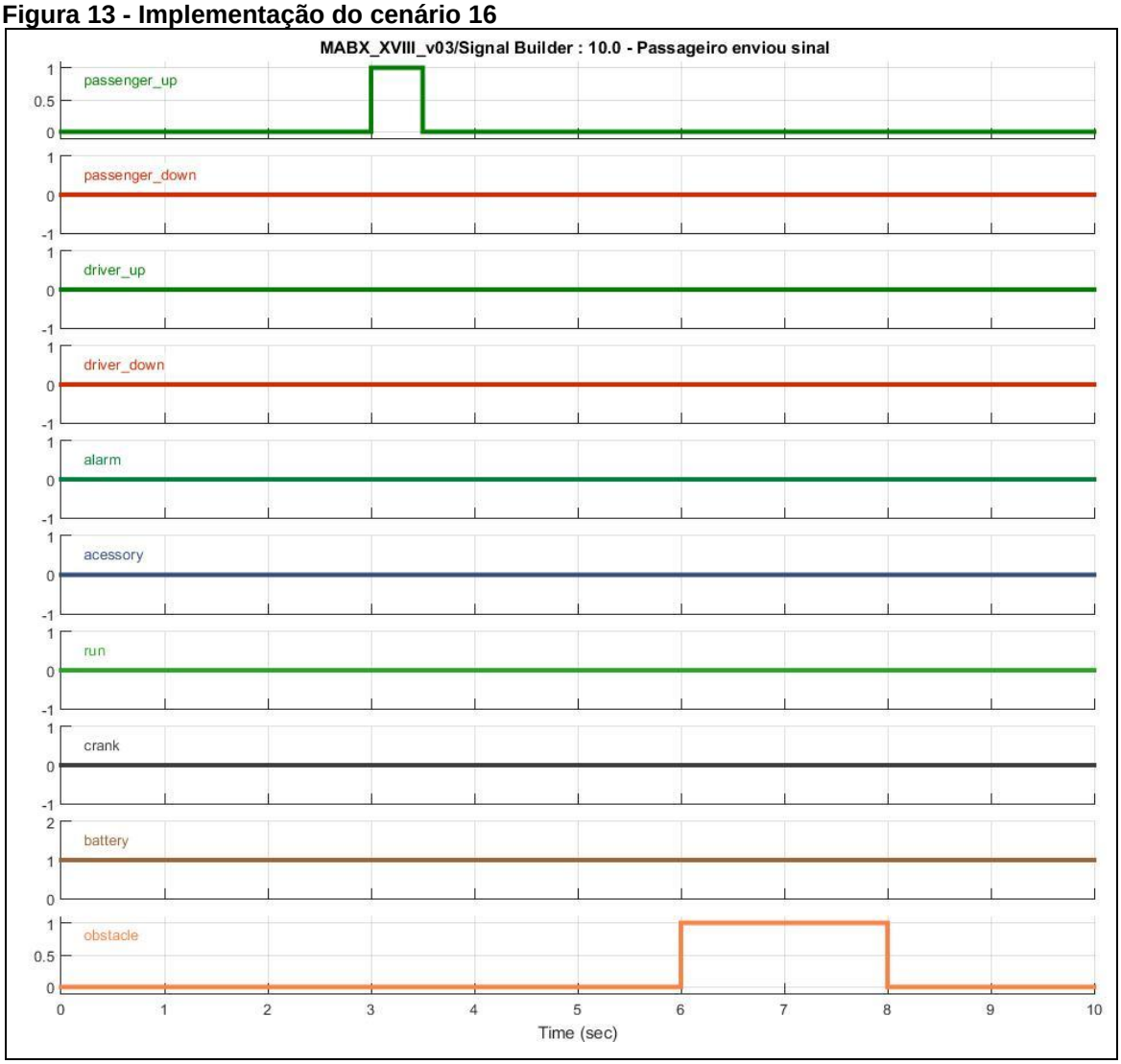
```
time = 10;
passenger_up = 0;
passenger_down = 0;
driver_up = 0;
driver_down = 0;
alarm = 0;
acessory = 0;
run = 0;
crank = 0;
battery = 1;
obstacle = 0;

Titulo: Passageiro enviou sinal para cima e encontrou um objeto
Dado que a janela do passageiro está aberta
Quando $variavel1 for acionado no intervalo %tempo1 até %tempo2 segundos
E encontrar um $variavel2 durante o intervalo %tempo3 até tempo$ segundos
Então o vidro deve reconher o obstaculo e descer aproximadamente 10 cm

|variavel1       |variavel2  |tempo1|tempo2|tempo3|tempo4|
|passenger_up |obstacle   |3        |4        |6        |8        |
|passenger_up |obstacle   |1        |4        |6        |8        |
```

Fonte: Autoria própria

Para a escrita deste cenário, utilizou-se de uma tabela para definir valores para as variáveis que estão presentes no cenário, com isto buscou-se economizar tempo na construção de cenários, pois ao escrever neste formato se utiliza apenas um cenário para gerar a quantidade de caso de teste igual o número de linhas da tabela.

Ao utilizar este cenário como entrada no *Basalt*, o programa irá retornar como saída uma planilha, está por sua vez apresenta a programação do ambiente de teste pronto, sendo necessário somente importar o arquivo para o *Simulink*. Dessa forma, tem-se o caso de teste #16 já programado e pronto para ser testado conforme apresenta a Figura 13.

Figura 13 - Implementação do cenário 16

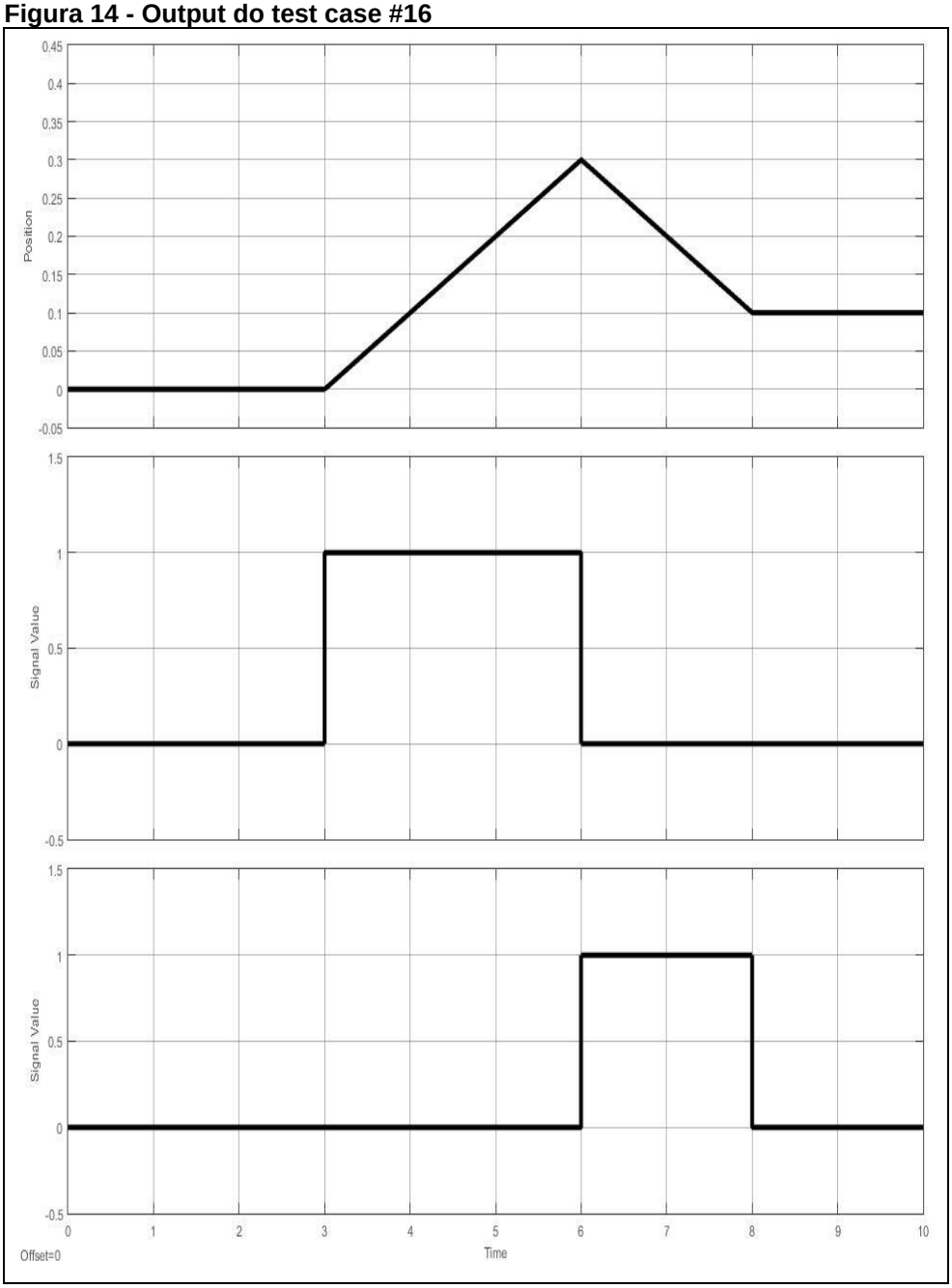


Fonte: Autoria própria

Com o ambiente de teste preparado, tem-se o necessário para executar a simulação do controlador. O teste é rodado e compara-se seu *output* com o que foi descrito na etapa *Então* do cenário. Se corresponder ao que é esperado que aconteça o teste está correto, senão o cenário deve ser reescrito até que atenda aos requisitos.

Após rodarmos o caso de teste #16, obtive-se o seguinte resultado representado nos três gráficos ilustrados na Figura 14. Cada gráfico representa a posição do vidro, o sinal de subida e o sinal de descida, todos em função do tempo.

Figura 14 - Output do test case #16



Fonte: Autoria própria

Ao analisar a saída do cenário implementado pode-se verificar que o cenário está correto, pois ao comparar o *output* gerado pela ferramenta com a etapa *Então* do cenário observa-se que no primeiro Gráfico da Figura 14, que a janela elétrica identifica um objeto no tempo de 6 segundos e se movimenta para baixo 10 cm, atendendo a condição do cenário e o requisito 6.

Portanto, este caso de teste deve receber o *status* de aprovado, dessa forma, deve-se voltar a etapa de Análise e Escolha de Caminhos, para que outro caminho possa ser testado, até que não haja mais caminhos para executar.

Nesta aplicação foram identificados 32 caminhos independentes, sendo que 30 caminhos foram considerados válidos.

Na etapa de construção de cenários, foram elaborados 14 cenários, os quais apresentavam tanto cenários com valores fixos quanto com valores tabulares. Ao realizar a geração automatizada dos cenários, foi obtido um total de 48 casos de teste, os quais foram testados e verificados se foram aprovados ou rejeitados pelo teste.

Na Quadro 12 é exibido os resultados de cada um dos casos de teste, seguindo de seu ID, a qual grupo pertence, requisitos que abrange e se o teste foi aprovado ou não. O “APROVADO” significa que o teste foi aprovado e atende as condições estabelecidas no cenário e “REPROVADO” representa que deve ser reportado a equipe de desenvolvimento, para que o erro seja corrigido e testado novamente. Outras implementações estão disponíveis no Apêndices deste Trabalho.

No Apêndice A, podemos observar os cenários escritos no formato de entrada para a ferramenta construída no desenvolvimento deste trabalho, os cenários que estão presentes nesta seção, cenários 8, 9, 22, 26, e 27, são alguns dos cenários que apresentaram algum erro em seu teste.

O Apêndice B, temos as ilustrações do ambiente de teste preparado referente a cada um dos cenários descritos no Apêndice A, gerados de forma automatizada através da ferramenta proposta. Por último, no Apêndice C, é possível verificar os resultados de cada simulação referente aos cenários 8, 9, 22, 26 e 27, neste apêndice podemos analisar e identificar os erros referentes a interrupção para baixo causada pelo motorista e a variável *Passenger Up,* que ao ser acionada por menos de 1 segundo, a variável não é desligada mesmo depois que a janela já está fechada.

Quadro 12 - Análise do estudo de caso

Test Case ID	Grupo	Requisitos Analisados	Test Report
0	Motorista	1, 2, 3, 10	APROVADO
1	Motorista	1, 2, 3, 5, 10	APROVADO
2	Motorista	2, 6, 7, 10	APROVADO
3	Motorista	2, 6, 7, 10	APROVADO
4	Motorista	1, 2, 3, 10	APROVADO
5	Motorista	1, 2, 3, 5, 10	APROVADO
6	Motorista	2, 5, 10	APROVADO
7	Motorista	2, 5, 10	APROVADO
8	Motorista	2, 5, 10	REPROVADO
9	Motorista	2, 5, 10	REPROVADO
10	Motorista	2, 5, 10	APROVADO
11	Motorista	2, 5, 10	APROVADO
12	Motorista	2, 5, 10	APROVADO
13	Motorista	2, 5, 10	APROVADO
14	Motorista	2, 5, 10	APROVADO
15	Motorista	2, 5, 10	APROVADO
16	Motorista	2, 5, 10	APROVADO
17	Motorista	2, 5, 10	APROVADO
18	Motorista	2, 5, 10	APROVADO
19	Motorista	2, 5, 10	REPROVADO
20	Motorista	2, 5, 10	APROVADO
21	Motorista	2, 5, 10	APROVADO
22	Passageiro	1, 2, 3, 10	REPROVADO
23	Passageiro	1, 2, 3, 5, 10	APROVADO
24	Passageiro	2, 6, 7, 10	APROVADO
25	Passageiro	2, 6, 7, 10	APROVADO
26	Passageiro	1, 2, 3, 10	REPROVADO
27	Passageiro	1, 2, 3, 5, 10	REPROVADO
28	Passageiro	2, 5, 10	REPROVADO
29	Passageiro	2, 5, 10	REPROVADO
30	Passageiro	2, 5, 10	APROVADO
31	Passageiro	2, 5, 10	APROVADO
32	Passageiro	2, 5, 8, 10	APROVADO
33	Passageiro	2, 5, 8, 10	APROVADO
34	Passageiro	2, 5, 8, 10	APROVADO
35	Passageiro	2, 5, 8, 10	APROVADO
36	Passageiro	2, 5, 10	APROVADO
37	Passageiro	2, 5, 10	APROVADO
38	Passageiro	2, 5, 10	REPROVADO
39	Passageiro	2, 5, 10	APROVADO
40	Passageiro	2, 5, 10	APROVADO
41	Passageiro	2, 5, 10	APROVADO
42	Passageiro	2, 5, 10	APROVADO
43	Passageiro	2, 5, 10	APROVADO
44	Neutro	9, 10	APROVADO
45	Neutro	9, 10	APROVADO
46	Neutro	10	APROVADO
47	Neutro	10	APROVADO

Fonte: Autoria própria

Ao analisar Quadro 12, observa-se que a maioria dos cenários implementados foram aprovados em seu teste, encontrando 9 casos em que foram

reprovados. O número que representa a quantidade de testes implementados e aprovado é de 81% e 19% dos testes foram reprovados.

Ao observar os casos de teste que apresentaram falhas, é possível indicar que o erro está presente no sistema. Este erro foi observado na variável do *passenger_up*, está além de apresentar erros em testes o qual era o principal atuadora no cenário, também interferia em outros casos de teste que foram reprovados.

Tendo este conhecimento, é dever da equipe de teste documentar o erro encontrado e retornar o *feedback* para a equipe de desenvolvimento, para que eles possam corrigir o controlador e retornar para a equipe verificar se o sistema está conforme o solicitado pelo cliente.

5.4 ANÁLISE DA ABORDAGEM PROPOSTA COM A LITERATURA

Após aplicar a abordagem proposta em uma situação problema é possível realizar uma comparação com um trabalho relacionado. O resultado da comparação é apresentado na Tabela 6.

Tabela 2 - Comparação da abordagem proposta com o trabalho da literatura

	Abordagem Proposta	**Santos et al (2015)**
Número de casos de teste identificados	48	4
Número de *test cases* aprovados	39	4
Número de *test cases* reprovados	9	0
Número de requisitos atendidos	9	10

Fonte: Autoria própria

Comparando os resultados dos trabalhos, é observável que a abordagem proposta identificou um número maior de casos de teste comparado com a utilizada por Santos *et al* (2015) em seu trabalho. Foram identificados 48 caminhos que deveriam ser executados, enquanto o artigo que foi utilizando para comparação encontrou apenas 4 casos.

Outro fator importante é a quantidade de testes reprovados, enquanto este trabalho identificou 9 casos de teste com erros, o trabalho relacionado não apresentou nenhum teste reprovado, o que pode resultar problemas futuros para o sistema em desenvolvimento, visto que foram encontrados erros no controlador. É

importante ressaltar que mesmo 9 casos sendo reprovados, foram verificados e validados 38 casos de teste. Conclui-se que com a quantidade de casos encontrados pode-se aumentar a confiabilidade do sistema projetado.

Os testes realizados por Santos *et al* (2015) abrangeram todos os requisitos encontrados, enquanto a abordagem deste trabalho cobriu 9 dos 10 requisitos identificados, o requisito número 4, "*O sistema de controle deve operar entre uma voltagem de 12,5V e 14,5V",* não foi possível ser atendido, isto devido estar realizando uma simulação não conseguimos, neste trabalho, verificar a voltagem em que o sistema estava atuando. Por este motivo nesta etapa de testes não conseguimos validar todos os requisitos.

5.5 CONSIDERAÇÕES DO CAPITULO

Este capítulo mostrou a aplicação da abordagem proposta em uma situação problema na área automotiva.

Com isto foi possível observar que a abordagem proposta é capaz de identificar um grande número de casos de testes quando comparado com o trabalho relacionado, dessa forma abrangendo diversas situações do sistema que podem vir a ocorrer.

Aplicando a abordagem, também foi possível identificar erros no funcionamento do sistema que havia sido implementado, e que antes não tinham sido descobertos por Santos et al (2015).

Encontrar erros no início do projeto, é importante, pois problemas ainda no começo do desenvolvimento do sistema, podem acarretar grandes prejuízos ao projeto, visto que o erro poderia ser encontrado em etapas futuras do projeto, a dificuldade para a manutenção do erro seria maior, assim como a sua identificação. Sendo capaz de apontar onde está o erro ainda no início do projeto como consequência economizamos tempo e recursos.

Mesmo que a abordagem trabalhada não tenha validado todos os requisitos do problema proposto, se apresentou útil na identificação de casos de teste, assim como na capacidade encontrar falhas no sistema, além de apresentar uma documentação abrangente, que apresenta de forma clara e concisa o que está

sendo testado, com a possibilidade de ser reutilizada para a geração de dados de entrada de forma automatizada para realização de testes em software automotivo.

Portanto, com os resultados obtidos, é possível dizer que a aplicação da abordagem proposta para o desenvolvimento de testes em projetos de software automotivos, pode vir apresentar vantagens ao projeto.

6 CONCLUSÃO

Com o estudo realizado no referencial teórico sobre teste de software e desenvolvimento automotivo, foi possível aplicar os conceitos da área da engenharia de software na produção de testes para sistemas automotivos.

Desse modo, este trabalho apresentou uma nova abordagem para testes de software automotivo na etapa MIL, afim de melhorar este processo, constituída por seis etapas, sendo elas: *Construção do grafo de fluxo do controlador, Aplicação do teste de caminhos, Análise escolha dos caminhos encontrados, Escrever cenário em formato BDD, Implementar os cenários, Testar o cenário.* Vale ressaltar que para aplicar esta abordagem é necessário como entrada o controlador que deve estar testado e os requisitos solicitado pelo cliente.

A abordagem proposta apresenta uma documentação do que está sendo realizado dentro do projeto, que vir a ser automatizado para a geração de dados de entrada para a realização de testes. Por isto, a utilização do BDD e do teste de caminhos foram essenciais na concepção da abordagem.

A abordagem proposta foi aplicada em uma situação problema de um sistema de um vidro elétrico referente ao lado do passageiro de um automóvel que estava descrito no trabalho de Santos *el al.* (2015).Com os resultados obtidos é possível dizer que a abordagem pode trazer benefícios ao projeto, devido a quantidade de casos de teste encontrados e por apresentar um número maior do que atingido por Santos *et al* (2015), chegando a encontrar erros no controlador que não estava previsto anteriormente.

Quanto aos resultados pode-se dizer que foram positivos visto que, identificaram-se 48 casos de teste, sendo que 9 apresentaram resultados inesperados, devido a variável *passenger_up*, que possuia erros em seu desenvolvimento. Mesmo não podendo afirmar que todos os casos de teste do sistema foram encontrados e testados, parte deles foram validados. Esses resultados podem trazer um retorno positivo ao projeto, pois descobrindo erros ainda no início do desenvolvimento do projeto tem-se uma redução de custo e tempo em sua construção e aumento da confiabilidade do sistema, considerando que o sistema foi mais verificado antes de ir para o cliente final.

Uma das dificuldades encontradas no desenvolvimento deste trabalho foi o processo de automatização de cenários, pois é necessário o entendimento de como as ferramentas para o software automotivo funcionam e como aceitam dados de entrada para execução de testes.

6.1 TRABALHOS FUTUROS

Como trabalhos futuros, pode-se trabalhar aplicando a abordagem proposta em situações mais complexas, visto que mesmo identificando um número maior de casos de teste alguns requisitos não foram cobertos.

Outra área de pesquisa que pode ser explorada no mesmo tema, é a validação do caso de teste de forma automatizada, além de explorar o campo de identificação de caso de teste e produção dos mesmos de forma automatizada. Por meio disto, é possível validar ou reprovar o caso de teste ao mesmo tempo.

REFERÊNCIAS

AGILE ALIANCE. **Manifesto for Agile Software Development**. 2001. Disponível em: < http://agilemanifesto.org/>. .

BANIK, T. M. **Processo de Desenvolvimento Baseado em Modelo para software automotivo**: Migração para o padrão AUTOSAR. 2017. 68 f. Trabalho de Conclusão de Curso (Curso Superior em Bacharelado em Engenharia Eletrônica) – Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2017.

CARDOSO, M. J. S. M. **Modelo de Processo de Teste Para Sistemas de Software Críticos.** 2010. 171 f. Dissertação (Programa de Pós-Graduação em Engenharia Elétrica) – Pontifícia Universidade Católica de Minas Gerais, Belho Horizonte, 2010.

DEVELOPERWORKS. **Criação e Geração de Planos de Teste de Software**. 2012. Disponível em: <https://www.ibm.com/developerworks/br/local/rational/criacao_geracao_planos_testes_software/index.html >. Acesso em: 16 jan. 2018

FELCHAR, C.; ROQUETTE, J. H. **dag_all_paths.** 2018. Disponível em: <https://gist.github.com/htmk/2946a51eaefa3ac6083b90730679cc7b >. Acesso em: 25 jun. 2018

FERNANDES, G. F. D. **Geração Automática de Casos de Teste a partir dos Requisitos.** 2014. 88 f. Dissertação (Engenharia e Gestão de Sistemas de Informação) - Universidade do Minho, Guimarães, 2014.

GOMES, E. W. C.; SILVA, R. A. **Verificação do Subframework de Análise Semântica de Fórmulas Utilizando Testes de Software na Fase de Unidade.** 2009. 159 f. Trabalho de Conclusão de Curso (Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas) – Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2009.

HERMANS, T., RAMAEKERS, P., DENIL, J., DE MEULENAERE, P., ANTHONIS, Incorporation of AUTOSAR in an Embedded Systems Development Process: a Case Study. In: Conference on Software Engineering and Advanced Applications. 37. 2011, Oulu. **Anais...** IEEE: Oulu, Finland, 2011. pp. 247-250

MATHWORKS. **Why Use Model-Based Design?** Disponível em: <http://www.mathworks.com/model-based-design/>. Acesso em: 04 jun. 2018

MATHWORKS. **Simulation and Model-Based-Design.** Disponível em: < https://www.mathworks.com/products/simulink.html > Acesso em: 04 jun. 2018

SANTOS, M. M. D., NEME, J. H., FRANCO, F. R. Rapid Control Prototyping Automotive Software in Power Windowns Systems**. International Journal of Inovate Computing Information and Control**. v.11, n. 4, p.1341-1356, 2015.

NUNES, L. R. **Projeto e Validação de Software Automotivo com o Método de Desenvolvimento Baseado em Modelos**. 2017. 156 f. Dissertação (Mestrado em Engenharia Elétrica) – Universidade Tecnológica Federal do Paraná, Ponta Grossa, 2017.

PRESSMAN, R. S. **Engenharia de Software**: Uma Abordagem Profissional. 7. Ed. São Paulo: AMGH. 2011.

SHOKRY, H. HINCHEY, M. Model-Based Verification of Embedded Software. **Computer**. IEEE, v. 42, Issue 4, p. 53-59, abr. 2009.

SMART, J. F. **BDD in Action**: Behaviour-Driven Develompment for the Whole Software Lifecycle. 1 ed. Shelter Island: Manning Publications, 2014.

STELLA, G. **Aplicando a metodologia de desenvolvimento baseado em Modelos para funções de software automotivo**. 2015. 123 f. Dissertação – Universidade Tecnológica Federal do Paraná – PG, Ponta Grossa, 2015.

VIDANAPATHIRANA, A., DEWASURENDRA, S. D., Model in the loop of Complex Reactive Systems**.** In: International Conference on Industrial and Information Systems. 8. 2013, Peradeniya. **Anais...** IEEE: Peradeniya, Sri Lanka. 2013. p. 18-20.

APÊNDICE A - Cenários Reprovados da Aplicação

Quadro 13 - Cenários 8 e 9 Reprovados da Aplicação

time = 10;
alarm = 0;
acessory = 0;
run = 0;
crank = 0;
battery = 1;
obstacle = 0;

Título: Motorista enviou sinal para cima e teve uma interrupção
Dado que a janela está aberta
Quando $variavel1 for acionado no intervalo %tempo1 até %tempo2 segundos
E aconteceu uma interrupção através da $varivel2 do %tempo3 até %tempo4 segundos
Então o vidro automático deve reconhecer a interrupção e mudar sua direção
E se a interrupção for causada pelo passageiro não deve ocorrer

variavel1	variavel2	tempo1	tempo2	tempo3	tempo4
driver_up	driver_up	1	2	4	5
driver_up	driver_up	1	2	4	6
driver_up	driver_down	1	4	3	4
driver_up	driver_down	1	4	3	6
driver_up	passenger_up	1	2	4	5
driver_up	passenger_down	1	2	4	6
driver_up	passenger_up	1	4	3	4
driver_up	passenger_down	1	4	3	6

Fonte: Autoria própria

Quadro 14 - Cenário 22 Reprovado da Aplicação

time = 10;
passenger_down = 0;
driver_up = 0;
driver_down = 0;
alarm = 0;
acessory = 0;
run = 0;
crank = 0;
battery = 1;
obstacle = 0;

Título: Passageiro enviou sinal para cima por menos de 1 segundo
Dado que a janela do passageiro está aberta
Quando $passenger_up for acionado no intervalo %3 até %4 segundos
Então a janela deve fechar totalmente em menos de 4 segundos

Fonte: Autoria própria

Quadro 15 - Cenário 26 Reprovado da Aplicação

time = 10;
driver_up = 0;
driver_down = 0;
alarm = 0;
acessory = 0;
run = 0;
crank = 0;
battery = 1;
obstacle = 0;

Titulo: Passageiro enviou sinal para baixo por menos de 1 segundo
Dado que $passenger_up foi acionado no intervalo %0 até %1 segundos
E a janela está totalmente fechada
Quando $passenger_down for acionado no intervalo %5 até %6 segundos
Então a janela deve abrir totalmente em menos de 4 segundos

Fonte: Autoria própria

Quadro 16 -Cenário 27 Reprovado da Aplicação

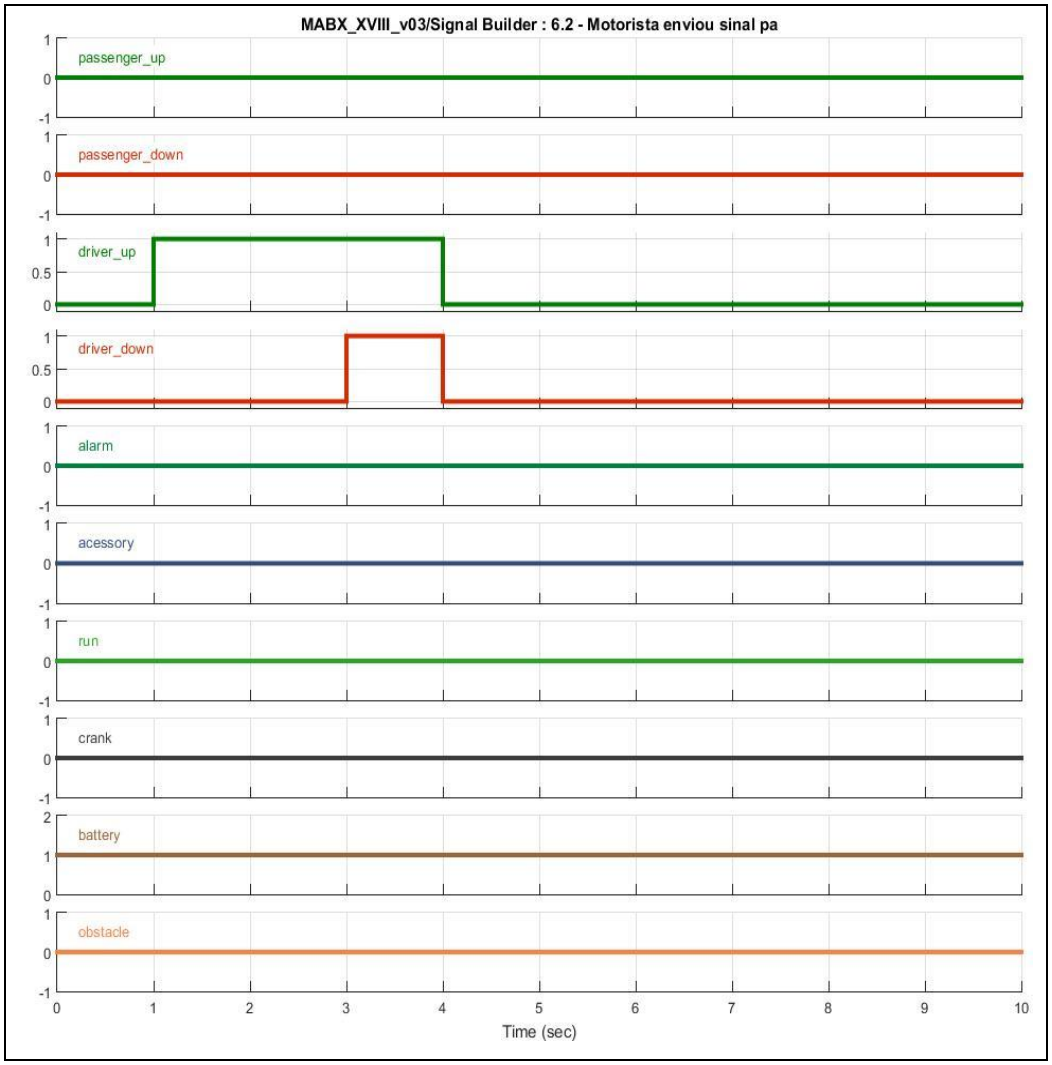
time = 10;
driver_up = 0;
driver_down = 0;
alarm = 0;
acessory = 0;
run = 0;
crank = 0;
battery = 1;
obstacle = 0;

Título: Passageiro enviou sinal para baixo por menos de 1 segundo
Dado que $passenger_up foi acionado no intervalo %0 até %1 segundos
E a janela está totalmente fechada
Quando $passenger_down for acionado no intervalo %5 até %7 segundos
Então a janela deve fechar até o instante que o sinal foi acionado

Fonte: Autoria própria

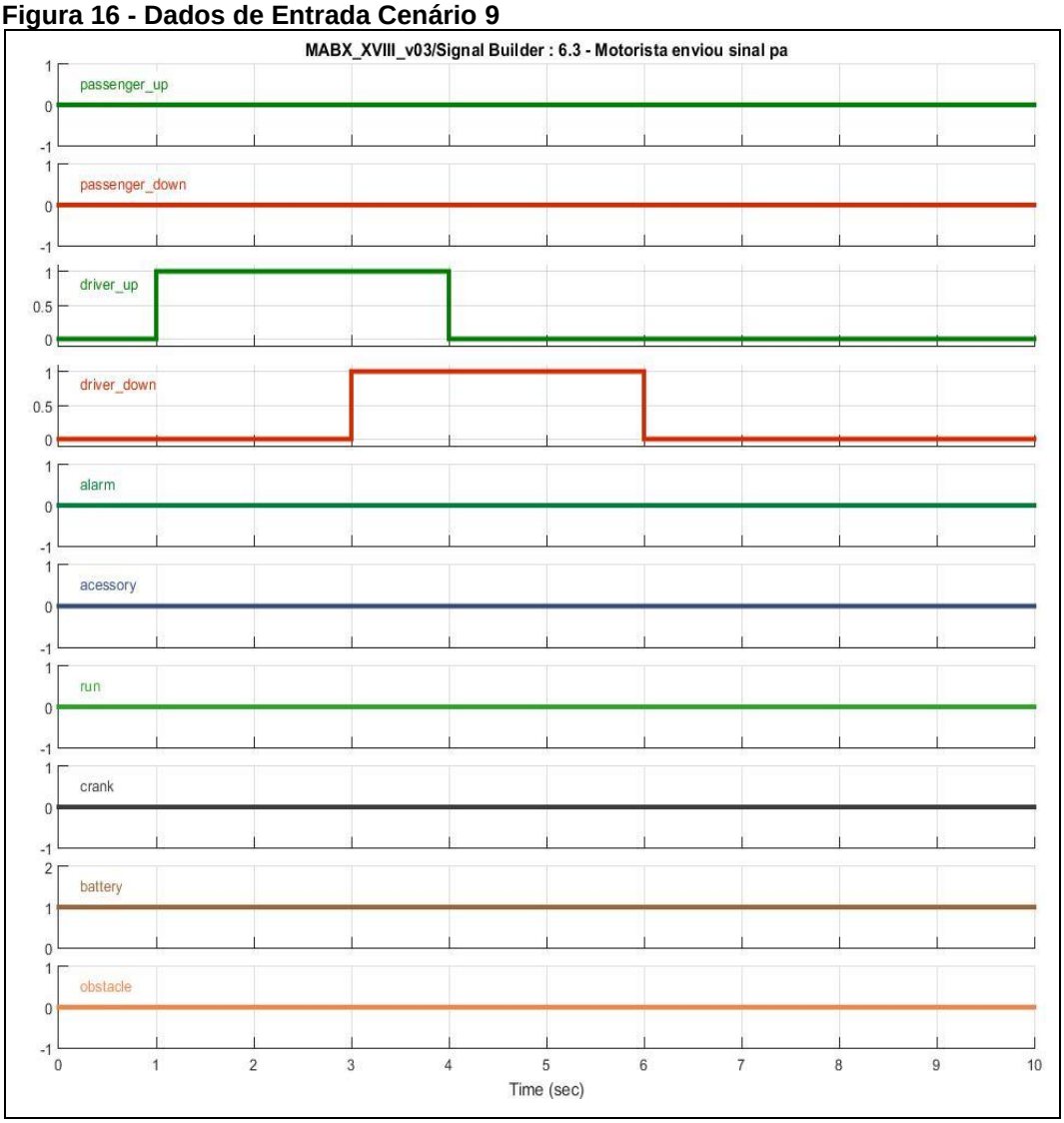
APÊNDICE B - Dados de Entrada Para Simulação da Aplicação dos Casos de Teste Reprovados

Figura 15 - Dados de Entrada Cenário 8



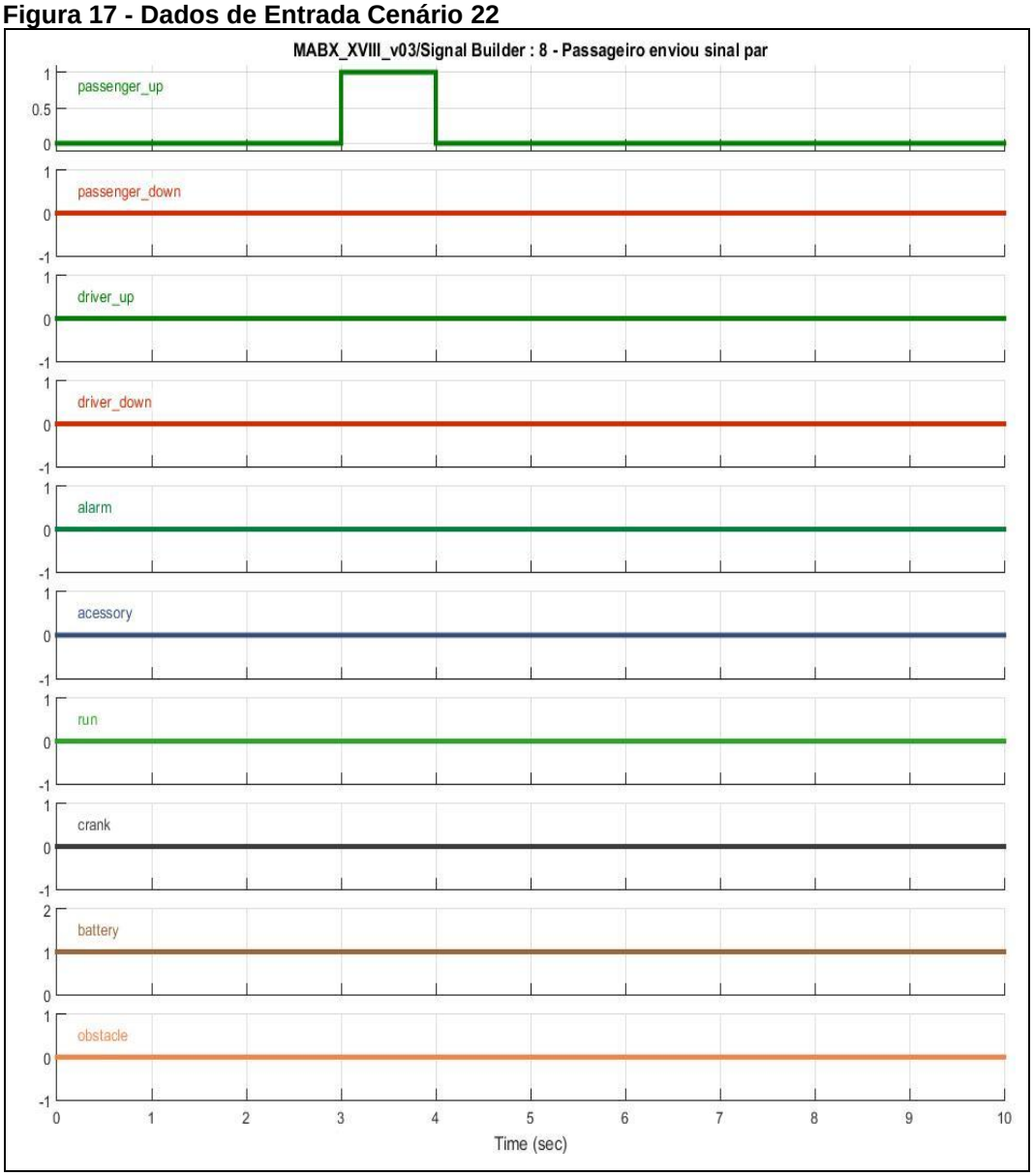
Fonte: Autoria própria

Figura 16 - Dados de Entrada Cenário 9



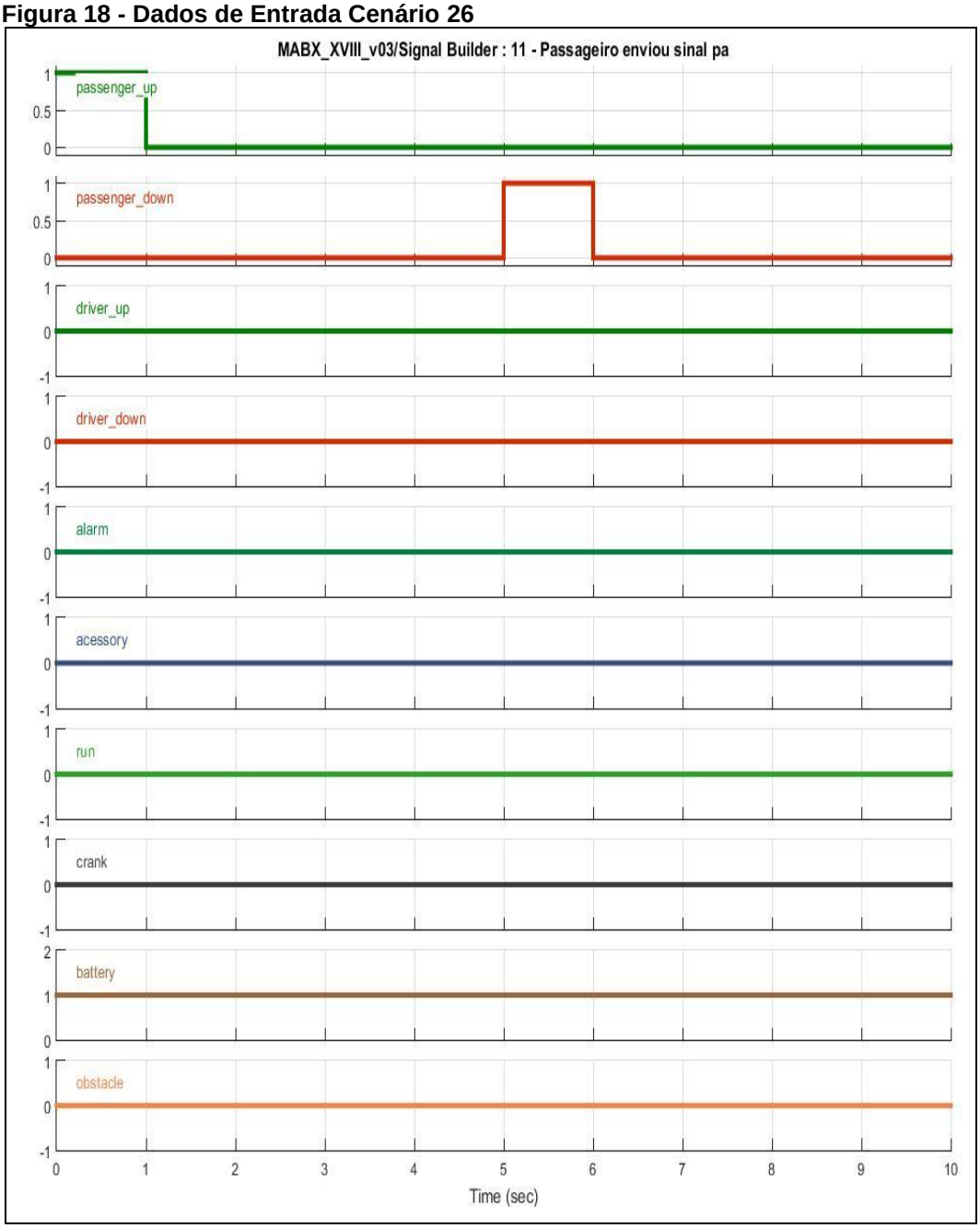
Fonte: Autoria própria

Figura 17 - Dados de Entrada Cenário 22



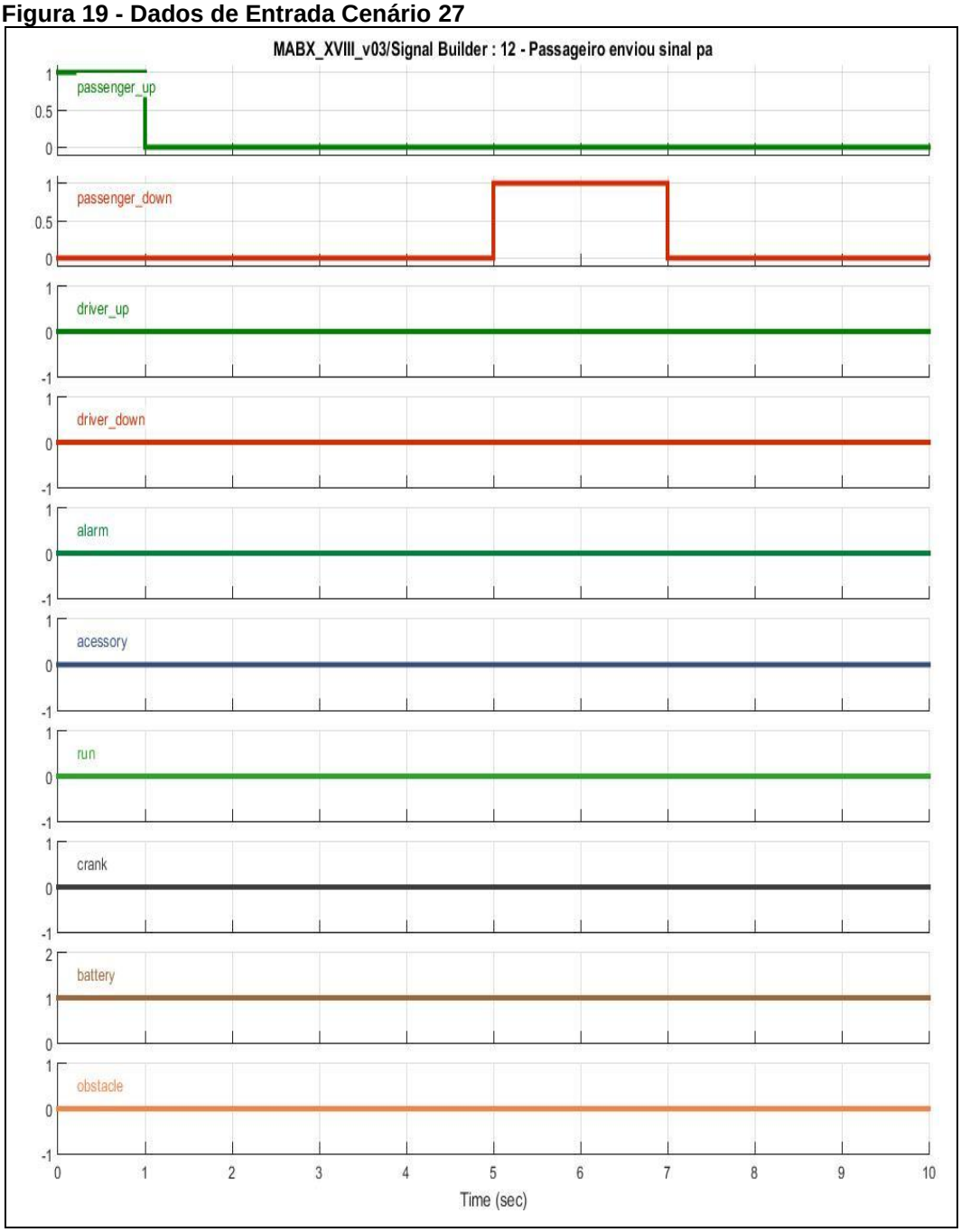
Fonte: Autoria própria

Figura 18 - Dados de Entrada Cenário 26



Fonte: Autoria própria

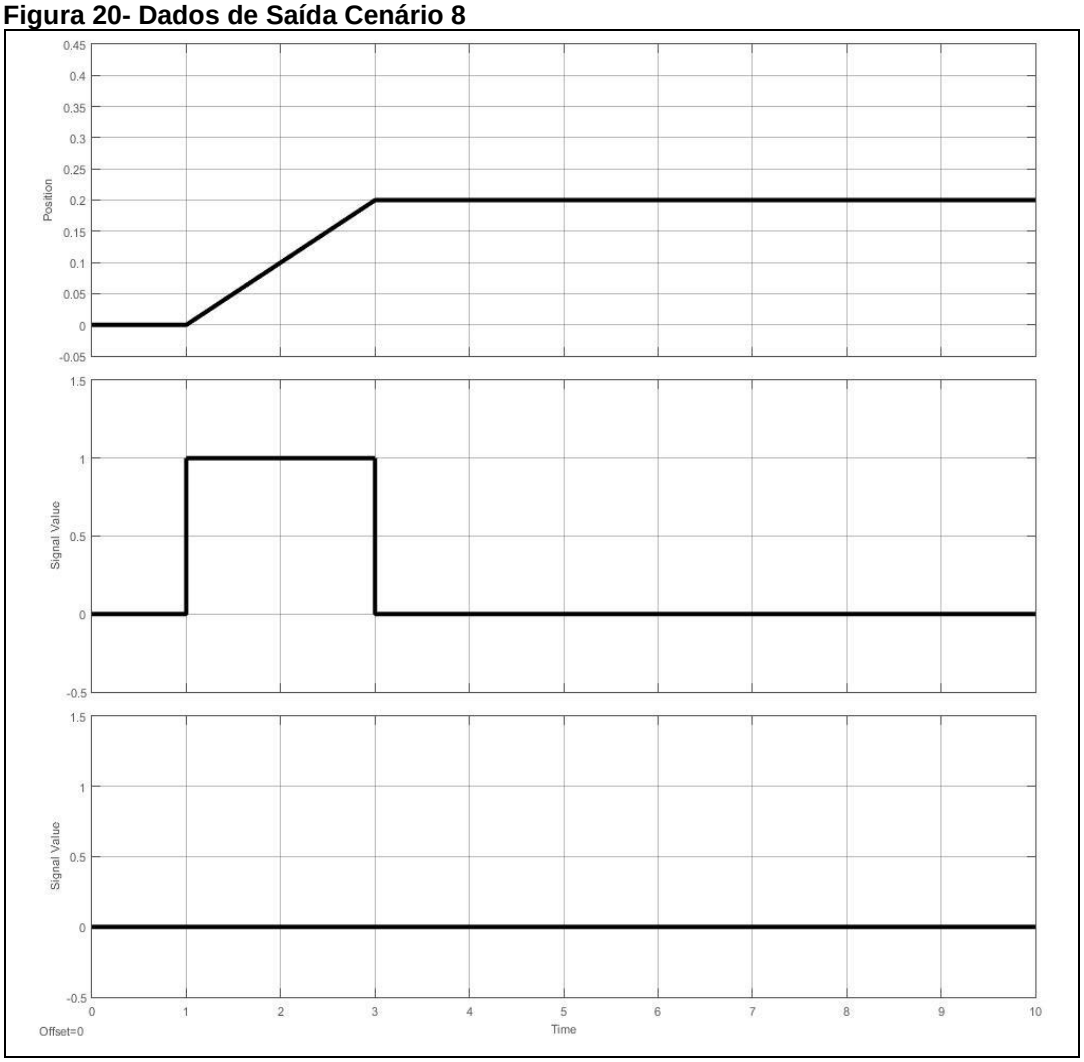
Figura 19 - Dados de Entrada Cenário 27



Fonte: Autoria própria

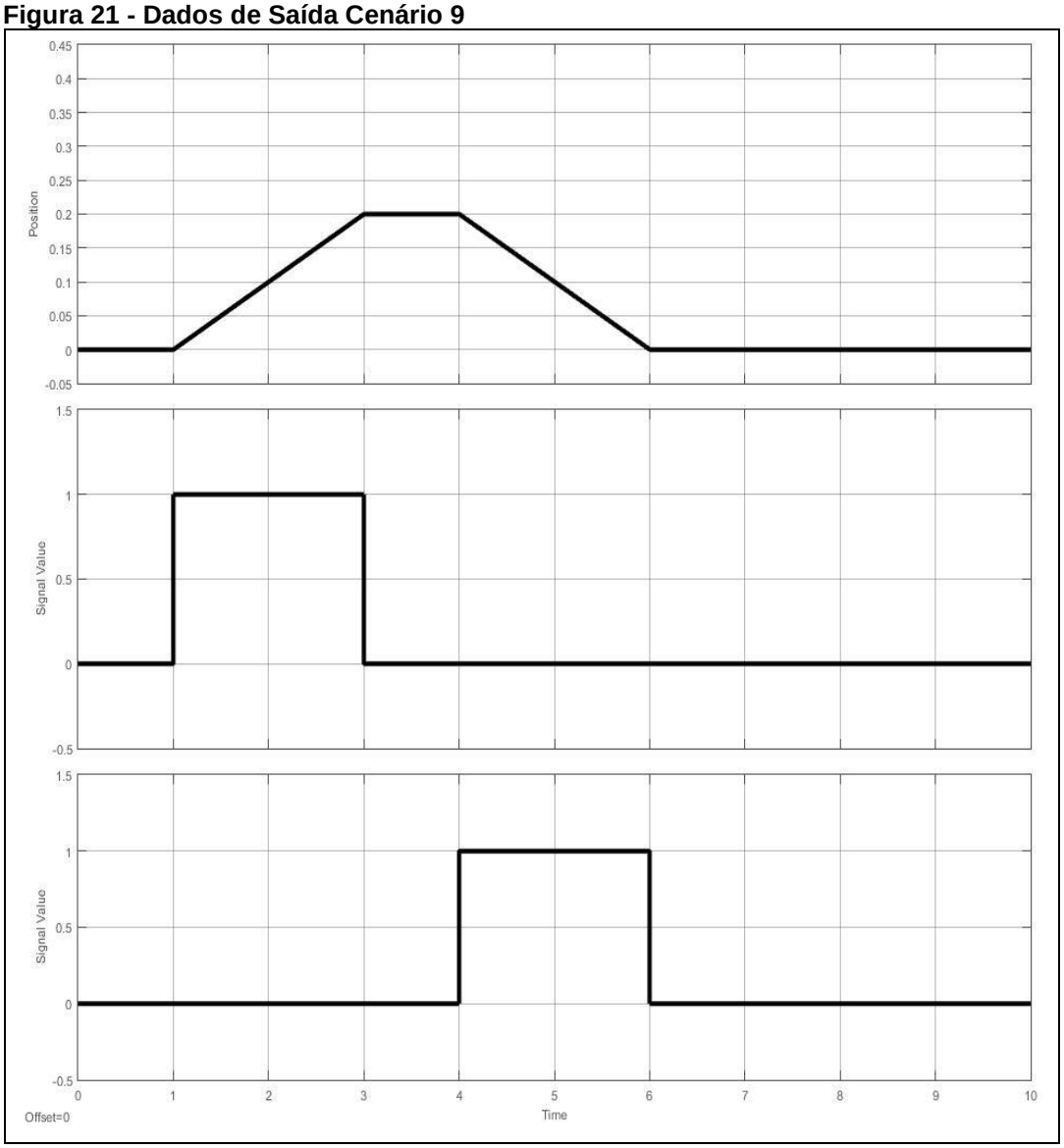
APÊNDICE C - Dados de Saída Gerados da Aplicação dos Casos de Teste Reprovados

Figura 20- Dados de Saída Cenário 8



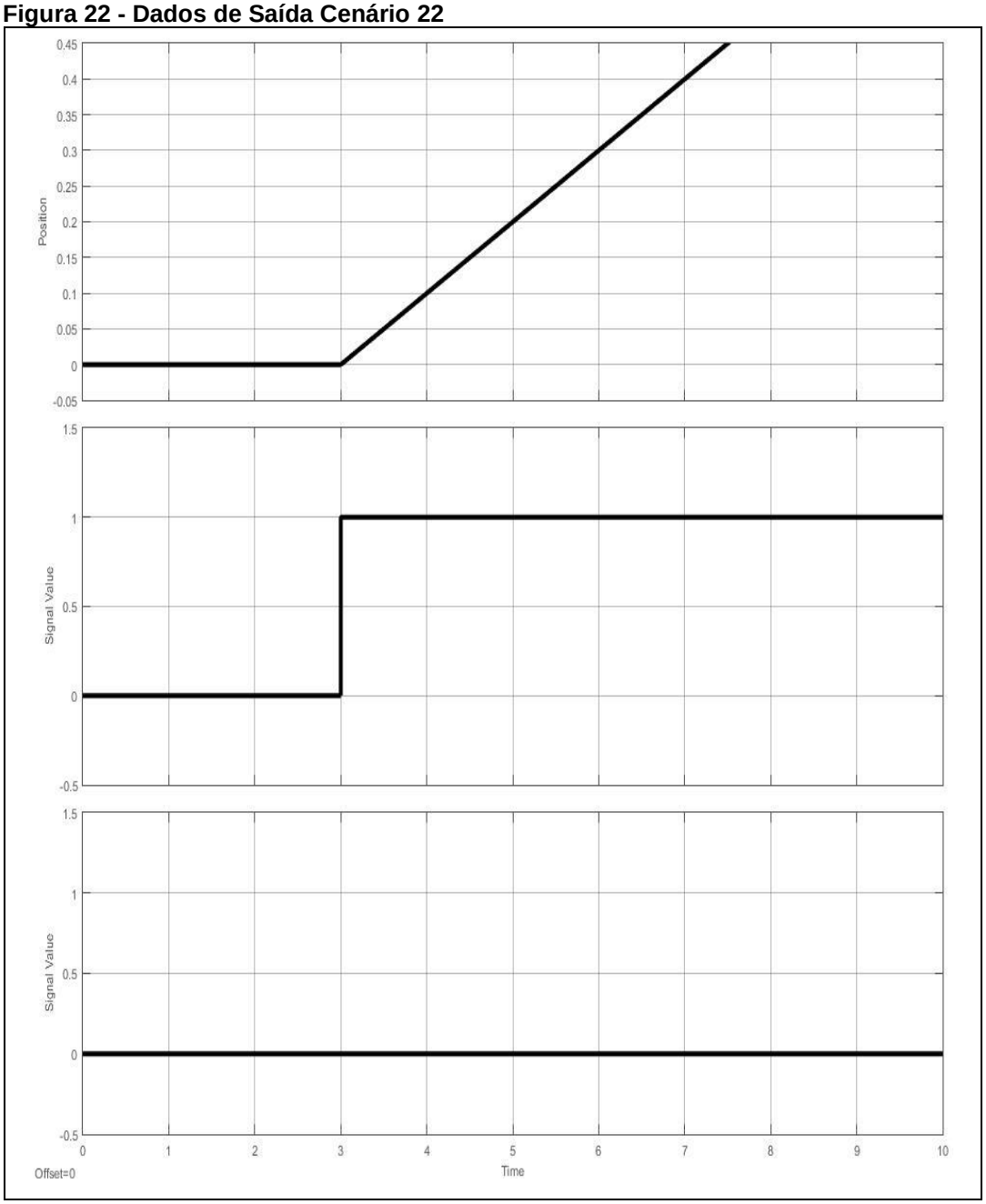
Fonte: Autoria própria

Figura 21 - Dados de Saída Cenário 9



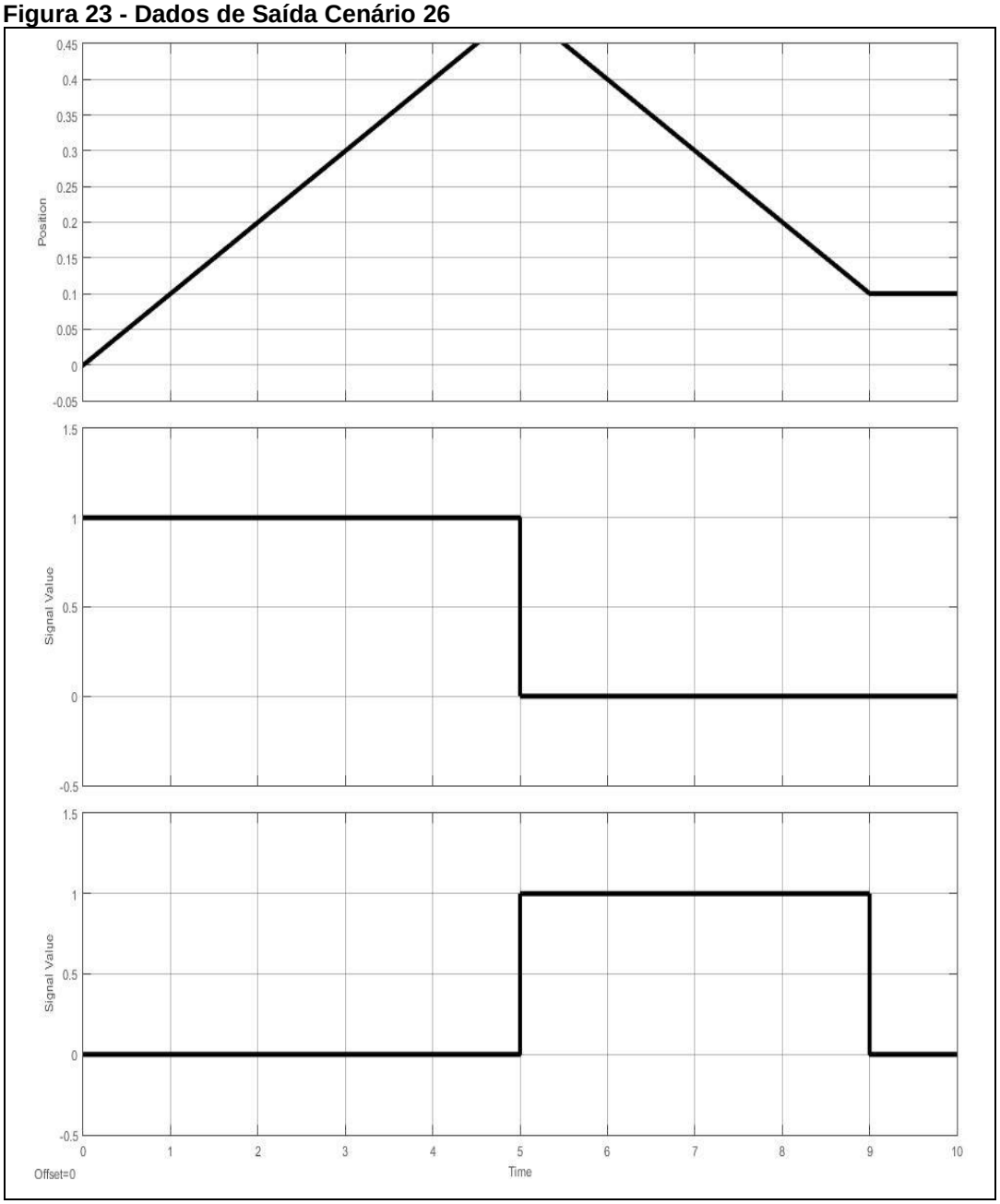
Fonte: Autoria própria

Figura 22 - Dados de Saída Cenário 22



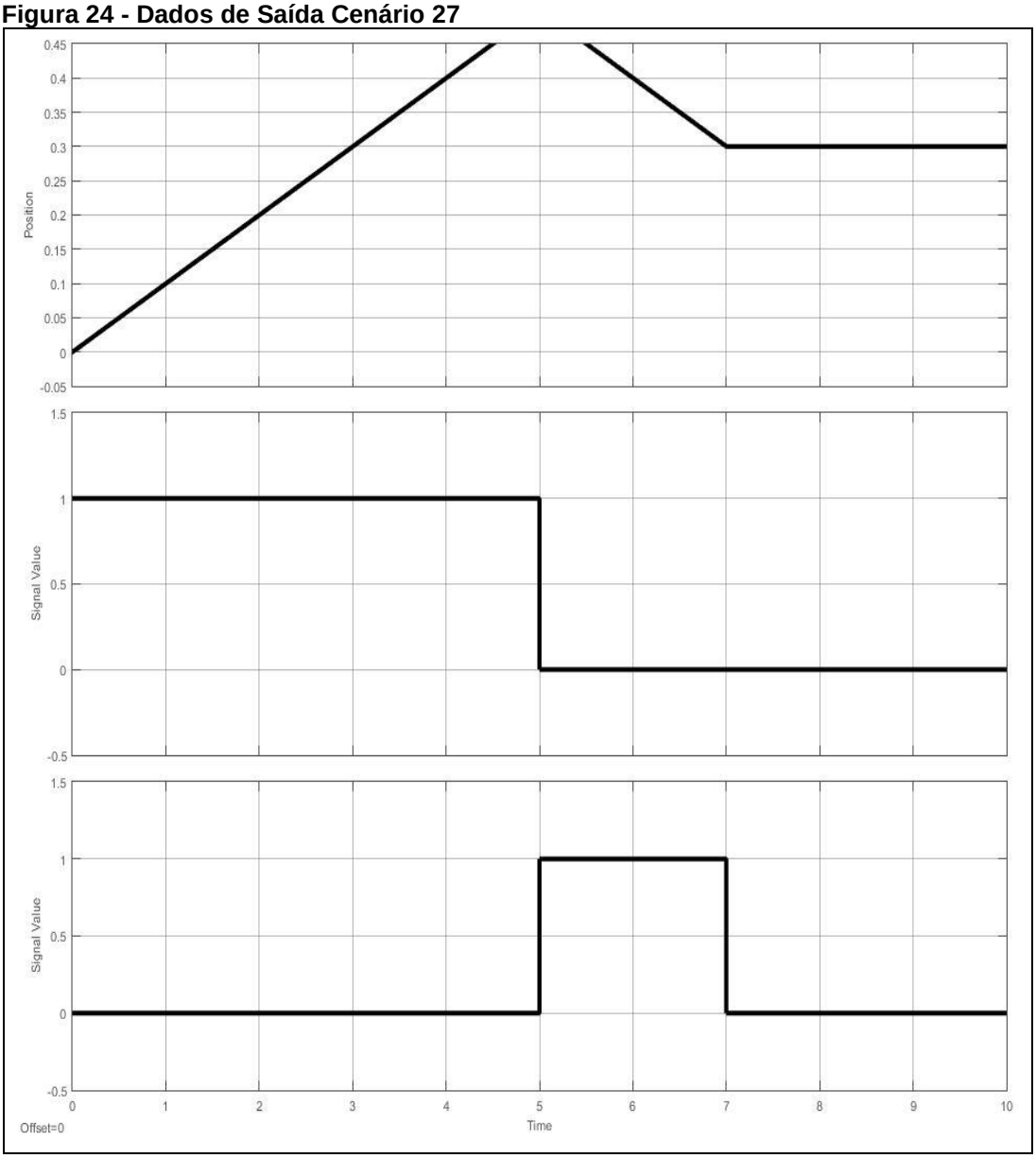
Fonte: Autoria própria

Figura 23 - Dados de Saída Cenário 26



Fonte: Autoria própria

Figura 24 - Dados de Saída Cenário 27



Fonte: Autoria própria