

1 2 9 0



UNIVERSIDADE DE  
**COIMBRA**

## **Relatório do Assignment 2**

## **Qualidade e Confiabilidade de Software (QCS)**

Bruno Silva

2021232021

Diogo Honório

2021232043

# 1. FR3.16

A função `tyrePressureWarnings` simula a verificação das pressões dos pneus de um veículo em relação a um valor de referência (`target`). Ela recebe uma lista de pressões dos pneus (`pressure`) e um valor `target`. A função remove valores zero das pressões, calcula a média de cada sublistas e compara com o `target`, retornando os índices dos pneus com pressão abaixo do esperado, ordenados de forma crescente.

## Black-box testing

Black Box Testing é uma técnica de teste de software onde a avaliação do sistema é feita com base nos seus outputs, sem qualquer conhecimento sobre a estrutura ou código interno do programa. O objetivo é garantir que o comportamento do sistema esteja em conformidade com as especificações estabelecidas.

Realizámos os seguintes testes para verificar o comportamento do código 3\_16. Inicialmente, dividimos os testes em dois tipos: o primeiro tipo inclui testes em que os inputs (classes) são válidos, ou seja, os valores das pressões são fornecidos como uma lista contendo 4 sublistas com valores positivos e o `target` é um valor positivo. Em seguida, realizaremos testes em que os inputs não precisam, necessariamente, ser válidos.

Importante referir também que assumimos que os valores validos de pressões se encontram entre 0 e 100.

## Testes válidos

### 1 – Pressões iguais ao Target

Este teste tem como objetivo verificar se, no caso de a pressão medida ser exatamente igual ao valor da pressão desejada, o output se mantém vazio. O comportamento esperado é que o output permaneça vazio, uma vez que a pressão não está abaixo do valor de referência, não existe, por isso, motivo para gerar qualquer alerta.

**test\_BV1:** `tyrePressureWarnings([[35, 35], [35, 35], [35, 35], [35, 35]], 35)`

**Output Esperado:** []

**Output Obtido:** []

Dado que o output corresponde ao esperado (um array vazio), concluímos que o teste foi bem-sucedido para todos os pneus.

## **2 – Pressões inferiores ao Target**

Este teste tem como finalidade verificar se o código é capaz de detetar corretamente quando, em todos os índices, a pressão se encontra abaixo do valor de referência definido.

**test\_BV2:** `tyrePressureWarnings([[30, 30], [30, 30], [30, 30], [30, 30]], 35)`

**Output Esperado:** `[0,1,2,3]`

**Output Obtido:** `[0,1,2,3]`

O teste foi bem-sucedido, visto que todos os pneus com pressão inferior à desejada foram corretamente identificados no array de saída.

## **3 – Pressões superiores ao Target**

Este teste, ao contrário do anterior, tem como objetivo verificar se as pressões acima do valor de referência são corretamente identificadas como válidas, de modo a não gerar qualquer alerta.

**test\_BV3:** `tyrePressureWarnings([[40, 40], [40, 40], [40, 40], [40, 40]], 35)`

**Output Esperado:** `[]`

**Output Obtido:** `[]`

Como se pode observar, o sistema devolveu um array vazio, o que indica que nenhuma pressão foi considerada abaixo do valor de referência. Assim, o teste foi bem-sucedido.

## **4 a 7 - Pressão inferior ao Target no índice X**

Estes testes foram realizados com o objetivo de verificar se o programa consegue identificar corretamente qual é o pneu (ou seja, o índice correspondente) que apresenta uma pressão abaixo do valor do Target. Cada teste faz a verificação de um único pneu com pressão inferior de cada vez, permitindo assim validar a precisão do algoritmo na deteção de anomalias por cada pneu.

### **4 - Pressão inferior ao Target no índice 0**

**test\_BV4:** `tyrePressureWarnings([[30, 30], [40, 40], [40, 40], [40, 40]], 35)`

**Output Esperado:** `[0]`

**Output Obtido:** `[0]`

### **5 - Pressão inferior ao Target no índice 1**

**test\_BV5:** `tyrePressureWarnings([[40, 40], [30, 30], [40, 40], [40, 40]], 35)`

**Output Esperado:** `[1]`

**Output Obtido:** `[1]`

## **6 - Pressão inferior ao Target no índice 2**

**test\_BV6:** `tyrePressureWarnings([[40, 40], [40, 40], [30, 30], [40, 40]], 35)`

**Output Esperado:** [2]

**Output Obtido:** [2]

## **7 - Pressão inferior ao Target no índice 3**

**test\_BV7:** `tyrePressureWarnings([[40, 40], [40, 40], [40, 40], [30, 30]], 35)`

**Output Esperado:** [3]

**Output Obtido:** [3]

Verificamos que o programa identificou corretamente os pneus com pressão inferior em todos os casos, demonstrando que o algoritmo é eficaz na deteção individual.

## **8 – Pressões todas diferentes**

Este teste é realizado para verificar se, mesmo quando os valores das leituras das pressões são todos diferentes, as médias continuam precisas e os alertas são gerados apenas quando necessário.

**test\_BV8:** `tyrePressureWarnings([[38, 36], [33, 41], [37, 36], [39, 34]], 35)`

**Output Esperado:** []

**Output Obtido:** []

Como se pode observar, mesmo com leituras de pressões distintas, o sistema continua a funcionar corretamente, gerando alertas apenas quando necessário.

## **9 e 10 – Valores de vírgula flutuante no input X**

Estes testes foram realizados para verificar se o sistema continua funcional quando são utilizados valores com vírgula flutuante, seja nas pressões ou no valor do target. O objetivo é garantir que o sistema consiga gerar o alerta adequado mesmo quando a pressão for inferior ou superior ao target por apenas algumas décimas.

## **9 – Valores de vírgula flutuante no input Pressures**

**test\_BV9:** `tyrePressureWarnings([[34.2, 34.2], [40, 40], [40, 40], [40, 40]], 35)`

**Output Esperado:** [0]

**Output Obtido:** [0]

## **10 - Valores de vírgula flutuante no input Target**

**test\_BV10:** `tyrePressureWarnings([[33, 33], [38, 38], [38, 38], [38, 38]], 34.2)`

**Output Esperado:** [0]

### **Output Obtido: [0]**

Os resultados confirmam que os testes foram bem-sucedidos. O sistema lida corretamente com valores decimais e mantém o seu funcionamento adequado mesmo com diferenças mínimas nas leituras.

## **11 – Ordenar pressões**

Foi realizado este teste com o objetivo de verificar se a ordenação das pressões abaixo do valor do target seguia o comportamento esperado, ou seja, da menor para a maior.

**test\_BV11:** `tyrePressureWarnings([[38, 38], [32, 32], [31, 31], [33, 33]], 35))`

### **Output Esperado: [2,1,3]**

### **Output Obtido: [2,1,3]**

Os resultados confirmam que a ordenação está correta, sendo apresentados em primeiro lugar os pneus com menor pressão (maior nível de alerta), seguindo-se os restantes por ordem crescente da média de pressão.

## **12 – Diferentes números de leituras de pressões**

Este teste foi realizado com o objetivo de verificar se o número de leituras do sistema de pressões poderia influenciar o seu funcionamento.

**test\_BV12:** `tyrePressureWarnings([[31], [38, 38, 38], [33, 33, 33], [32, 32]], 35))`

### **Output Esperado: [0,3,2]**

### **Output Obtido: [0,3,2]**

O resultado obtido corresponde ao esperado, o que demonstra que o número de leituras efetuadas não afeta o funcionamento do sistema. Assim, conclui-se que o teste foi bem-sucedido."

## **13 – Ignorar os valores 0 nas pressões**

Este último teste foi realizado com o objetivo de verificar de que forma os valores de pressão iguais a 0 são interpretados e se são devidamente ignorados.

**test\_BV13:** `tyrePressureWarnings([[31, 31], [0, 38], [38, 38], [33, 33]], 35))`

### **Output Esperado: [0,3]**

### **Output Obtido: [0,3]**

Após a análise dos resultados, conclui-se que os valores de 0 são efetivamente ignorados, conforme requisitado. Assim, considera-se que o teste foi bem-sucedido.

## Testes inválidos

### 1 – Número de pneus (sublistas) inferior a 4

Este teste foi realizado para verificar o comportamento do sistema quando o número de sublistas fornecidas é diferente de quatro. Neste caso, o teste deverá apresentar um erro, visto que a ausência de quatro sublistas indica que o veículo não está a ler a pressão dos quatro pneus.

**test\_BI1:** `tyrePressureWarnings([[36, 36], [36, 36], [33, 33]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[2]`

A análise dos resultados mostra que o sistema não valida o número de pneus fornecidos. Em vez de gerar um erro, interpreta o input como válido e devolve um resultado assumindo que apenas três pneus estão a ser monitorizados. Isto indica uma falha na validação do input.

### 2 – Número de pneus (sublistas) superior a 4

Este teste tem exatamente o mesmo propósito do anterior, mas, neste caso, pretende verificar o comportamento do sistema quando é fornecido um número superior a quatro sublistas, ou seja, quando o veículo deteta mais pneus do que o suposto.

**test\_BI2:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [34, 34], [32, 32]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[4,3]`

Após verificarmos os resultados percebemos que a abordagem é exatamente a mesma usada no teste anterior, não é verificado o output e o sistema funciona de forma normal, mas com 5 sublistas. Isto indica mais uma vez uma falha na validação do input, algo que pode comprometer o sistema.

### 3 – Valores negativos nas pressões.

Este teste tem como objetivo verificar como o sistema interpreta valores de pressão inferiores aos limites estipulados, uma vez que um sistema não pode operar com pressões negativas nos pneus.

**test\_BI3:** `tyrePressureWarnings([-33, 36], [36, 36], [36, 36], [34, 34]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[0,3]`

Após analisarmos os resultados, verificámos que, tal como nos testes anteriores, o sistema não realiza qualquer validação do input. e funciona de forma normal utilizando os valores

negativos como valores normais. Em vez de gerar um erro ao receber valores negativos, o sistema processa os dados normalmente, tratando os valores negativos como pressões válidas e operando de forma errada comprometendo assim a fiabilidade do sistema.

#### **4 - Valores negativos no Target.**

Este teste tem o mesmo objetivo do anterior, mas neste caso utiliza uma variável distinta (target).

**test\_BI4:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [34, 34]], -35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

Tal como nos testes anteriores, não há validação do input e o sistema processa os dados como se fossem válidos, sem gerar o erro esperado.

#### **5 – Sublista sem valores detetados.**

Este teste tem como objetivo verificar se a ausência de dados numa das sublistas é corretamente interpretada como uma falha na leitura, devendo, nesse caso, ser apresentado um erro.

**test\_BI5:** `tyrePressureWarnings([], [36, 36], [36, 36], [34, 34]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `ZeroDivisionError: division by zero`

Neste teste, apesar de ter sido gerado um erro, este resulta de uma exceção não controlada no próprio código (divisão por zero) e não de uma validação explícita do input. Isto demonstra que o sistema não está preparado para lidar corretamente com sublistas vazias, comprometendo a robustez do mesmo.

#### **6 – Input incorreto (apenas uma lista).**

Este teste tem como objetivo verificar o comportamento do sistema quando o input não segue a estrutura esperada. Em vez de ser fornecida uma lista de listas com os valores das pressões, é apresentada apenas uma lista simples. Neste caso, deverá ser gerado um erro.

**test\_BI6:** `tyrePressureWarnings([34, 34], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `argument of type 'int' is not iterable`

Tal como no teste anterior, embora seja gerado um erro, trata-se de uma exceção lançada pelo próprio funcionamento do código e não de uma validação explícita do input.

## **7 – Classes incorretas.**

Estes últimos testes têm como objetivo verificar se o sistema é capaz de detetar quando os valores das pressões não são números inteiros nem valores decimais (números flutuantes), e se, nesses casos, é apresentada uma mensagem de erro apropriada.

**test\_BI7:** `tyrePressureWarnings([[None, 36], [36, 36], [36, 36], [34, 34]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `unsupported operand type(s) for +=: 'int' and 'NoneType'`

Mais uma vez, observa-se a ausência de validação do input, evidenciando assim a falta de eficácia do código a lidar com inputs inesperados.

## **8 – Valores acima dos limites nas pressões.**

Este teste tem como objetivo avaliar se o sistema é capaz de identificar valores de pressão claramente acima dos limites aceitáveis e, nesses casos, apresentar uma mensagem de erro em vez de executar normalmente.

**test\_BI8:** `tyrePressureWarnings([[110, 110], [120, 120], [200, 220], [150, 150]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

Após a análise dos resultados, conclui-se que o sistema não está preparado para lidar com limites máximos previamente definidos. Em vez de sinalizar os valores como inválidos, o código é executado normalmente, tratando pressões extremamente elevadas como se fossem válidas.

## **9 – Valores acima dos limites no target.**

Este teste, à semelhança do anterior, tem como objetivo verificar se o sistema é capaz de identificar valores anormalmente elevados no variável target e, nesse caso, apresentar uma mensagem de erro.

**test\_BI9:** `tyrePressureWarnings([[32, 32], [45, 45], [58, 58], [34, 34]], 150)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[0,3,1,2]`

Tal como observado no teste anterior, o sistema não realiza qualquer validação do valor do target em relação aos limites previamente definidos. O código é executado normalmente, assumindo um valor extremamente elevado como válido.

## **10 – Todas as leituras a 0.**

Dado que o sistema está concebido para ignorar leituras com valor 0, este teste visa verificar o seu comportamento quando todas as pressões lidas são 0. Numa situação real, isto indicaria uma falha na recolha de dados e, por isso, deveria ser apresentada uma mensagem de erro, uma vez que não existem leituras válidas.

**test\_BI10:** `tyrePressureWarnings([[0, 0], [0, 0], [0, 0], [0, 0]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[0,1,2,3]`

Os resultados mostram que os valores 0 não foram ignorados e que o sistema continuou a funcionar normalmente. Embora se possa argumentar que a decisão de gerar um erro depende da interpretação de cada caso, consideramos que o comportamento mais adequado seria rejeitar a entrada e apresentar uma mensagem de erro, dado que não existem dados válidos para análise.

## **BVA (Boundary Value Analysis)**

Estes testes foram realizados com base na premissa de que os erros ocorrem com maior frequência nos limites das classes de equivalência. Assim, os testes focam-se na avaliação do comportamento do sistema perante valores imediatamente abaixo e acima dos intervalos definidos. Desta forma, garantimos que o sistema é capaz de responder de forma adequada em todas as situações.

Para a implementação dos testes, utilizámos os casos já realizados para cada classe e, nas situações em que fazia sentido, aplicámos os princípios dos testes BVA.

Após a execução dos testes, verificámos que os resultados obtidos foram idênticos aos registados com os valores centrais. Por essa razão, optámos por não aprofundar a análise nesta secção, uma vez que os comportamentos observados não diferem significativamente dos anteriormente descritos.

### **Testes válidos:**

#### **2b - Pressões inferiores ao Target**

**test\_BV2b\_1:** `tyrePressureWarnings([[34, 34], [34, 34], [34, 34], [34, 34]], 35)`

**Output Esperado:** `[0,1,2,3]`

**Output Obtido:** `[0,1,2,3]`

**test\_BV2b\_2:** `tyrePressureWarnings([[1, 1], [1, 1], [1, 1], [1, 1]], 35)`

**Output Esperado:** `[0,1,2,3]`

**Output Obtido:** `[0,1,2,3]`

### **3b – Pressões superiores ao Target**

**test\_BV3b\_1:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [36, 36]], 35))`

**Output Esperado:** `[]`

**Output Obtido:** `[]`

**test\_BV3b\_2:** `tyrePressureWarnings([[99, 99], [99, 99], [99, 99], [99, 99]], 35))`

**Output Esperado:** `[]`

**Output Obtido:** `[]`

### **4b – Pressão inferior ao Target no índice 0**

**test\_BV4b\_1:** `tyrePressureWarnings([[34, 34], [36, 36], [36, 36], [36, 36]], 35))`

**Output Esperado:** `[0]`

**Output Obtido:** `[0]`

**test\_BV4b\_2:** `tyrePressureWarnings([[1, 1], [99, 99], [99, 99], [99, 99]], 35))`

**Output Esperado:** `[0]`

**Output Obtido:** `[0]`

### **5b – Pressão inferior ao Target no índice 1**

**test\_BV5b\_1:** `tyrePressureWarnings([[36, 36], [34, 34], [36, 36], [36, 36]], 35))`

**Output Esperado:** `[1]`

**Output Obtido:** `[1]`

**test\_BV5b\_2:** `tyrePressureWarnings([[99, 99], [1, 1], [99, 99], [99, 99]], 35))`

**Output Esperado:** `[1]`

**Output Obtido:** `[1]`

### **6b – Pressão inferior ao Target no índice 2**

**test\_BV6b\_1:** `tyrePressureWarnings([[36, 36], [36, 36], [34, 34], [36, 36]], 35))`

**Output Esperado:** `[2]`

**Output Obtido:** `[2]`

**test\_BV6b\_2:** `tyrePressureWarnings([[99, 99], [99, 99], [1, 1], [99, 99]], 35))`

**Output Esperado:** `[2]`

**Output Obtido:** `[2]`

## **7b – Pressão inferior ao Target no índice 3**

**test\_BV7b\_1:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [34, 34]], 35))`

**Output Esperado:** [3]

**Output Obtido:** [3]

**test\_BV7b\_2:** `tyrePressureWarnings([[99, 99], [99, 99], [99, 99], [1, 1]], 35))`

**Output Esperado:** [3]

**Output Obtido:** [3]

## **9b – Valores de vírgula flutuante no input Pressures**

**test\_BV9b\_1:** `tyrePressureWarnings([[34.9, 34.9], [35.1, 35.1], [35.1, 35.1], [35.1, 35.1]], 35))`

**Output Esperado:** [0]

**Output Obtido:** [0]

**test\_BV9b\_2:** `tyrePressureWarnings([[0.1, 0.1], [99.9, 99.9], [99.9, 99.9], [99.9, 99.9]], 35))`

**Output Esperado:** [0]

**Output Obtido:** [0]

## **10b – Valores de vírgula flutuante no input Target**

**test\_BV10b\_1:** `tyrePressureWarnings([[34.1, 34.1], [34.3, 34.3], [34.3, 34.3], [34.3, 34.3]], 34.2))`

**Output Esperado:** [0]

**Output Obtido:** [0]

**test\_BV10b\_2:** `tyrePressureWarnings([[0.1, 0.1], [99.9, 99.9], [99.9, 99.9], [99.9, 99.9]], 34.2))`

**Output Esperado:** [0]

**Output Obtido:** [0]

## **Testes inválidos:**

### **3b – Valores negativos nas pressões.**

**test\_BI3b\_1:** `tyrePressureWarnings([[-0.1, 36], [36, 36], [36, 36], [34, 34]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[0,3]`

**test\_BI3b\_2:** `tyrePressureWarnings([-999, 36], [36, 36], [36, 36], [34, 34]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[0,3]`

### **4b - Valores negativos no Target.**

**test\_BI4b\_1:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [34, 34]], -0.1)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

**test\_BI4b\_2:** `tyrePressureWarnings([[36, 36], [36, 36], [36, 36], [34, 34]], -999)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

### **8b – Valores acima dos limites nas pressões.**

**test\_BI8b\_1:** `tyrePressureWarnings([[100.1, 100.1], [100.1, 100.1], [100.1, 100.1], [100.1, 100.1]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

**test\_BI8b\_2:** `tyrePressureWarnings([[999, 999], [999, 999], [999, 999], [999, 999]], 35)`

**Output Esperado:** `ValueError`

**Output Obtido:** `[]`

# White-box testing

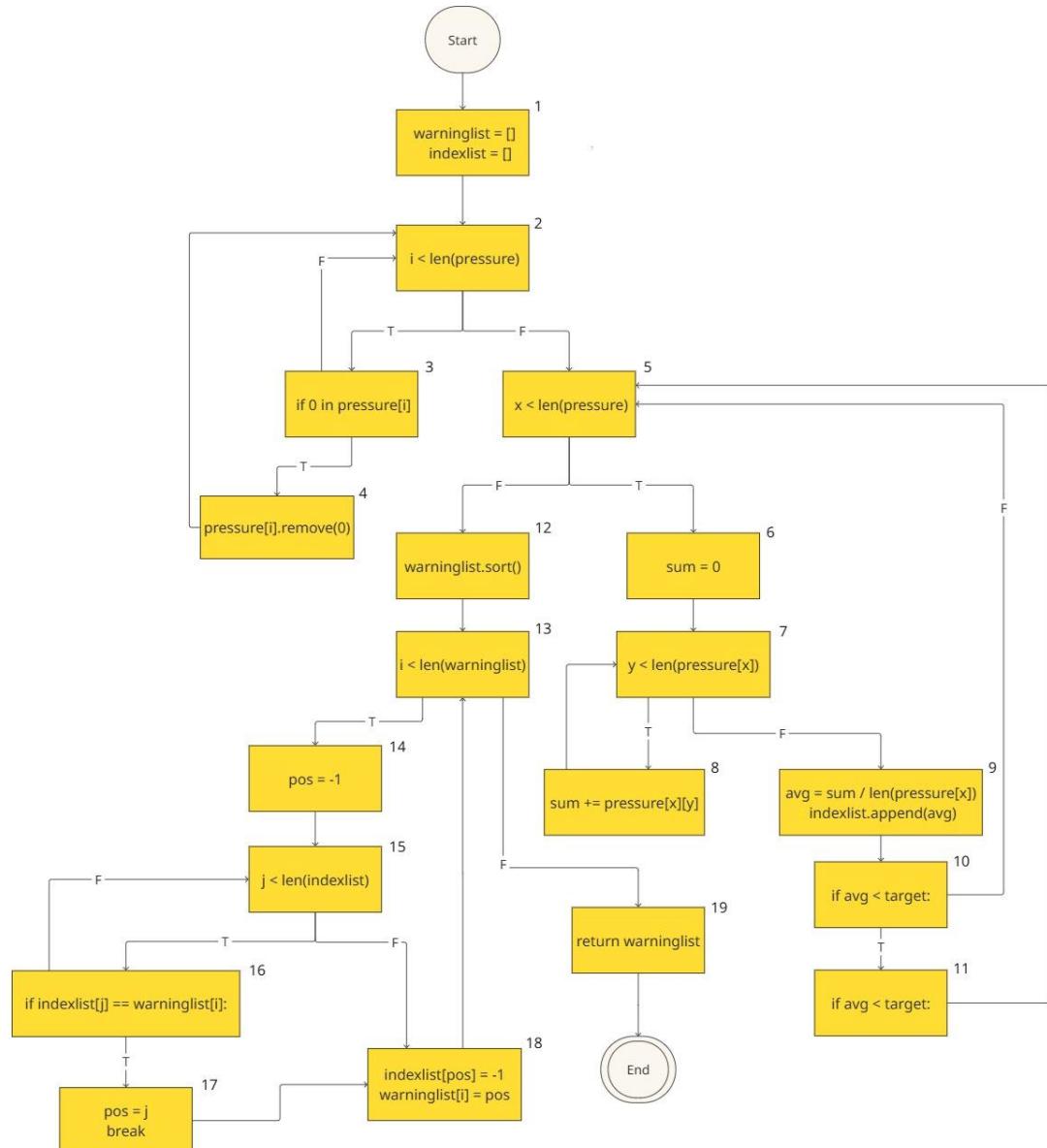
White Box Testing é uma técnica de teste de software em que a avaliação do sistema é feita com base no conhecimento completo da sua estrutura interna e código fonte. O objetivo é verificar a funcionalidade do programa, analisando o fluxo de controle, as estruturas de dados e os caminhos lógicos, garantindo que todas as partes do código sejam testadas e funcionem conforme esperado.

## Control Flow Graph

O Control Flow Graph (CFG) é uma técnica de *white-box testing*, tendo como principal objetivo identificar erros relacionados com a lógica e as estruturas de controlo do programa, como por exemplo condições, ciclos e chamadas de funções.

O CFG representa o fluxo de execução de uma função do código. É composto por nós, que representam blocos de instruções executadas sequencialmente, sem desvios, e por arestas, que representam as transições entre esses blocos, podendo corresponder a decisões ou saltos no fluxo de execução.

De seguida, apresentamos o nosso CFG para o excerto de código 3\_16:



Após a construção do CFG, definimos as coverages que foram tidas em consideração:

- **Node Coverage:** garante que todos os nós do grafo são executados pelo menos uma vez.
- **Edge Coverage:** garante que todas as arestas entre os nós são percorridas.
- **Path Coverage:** garante que todos os caminhos independentes do grafo são percorridos.

## Contagem dos Caminhos independentes

Para a realização do path testing, foi necessário calcular o número máximo de caminhos independentes presentes no grafo. Para isso, recorremos ao cálculo do número de McCabe, que indica exatamente este valor. Existem duas fórmulas que permitem obter este número:

$$\text{Fórmula 1: } V(G) = \text{Número arestas} - \text{Número de nós} + 2$$

$$= 28 - 21 + 2 = 9$$

$$\text{Fórmula 2: } V(G) = \text{Número de nós de decisão} + 1$$

$$= 8 + 1 = 9$$

Podemos concluir, portanto, que ambas as fórmulas coincidem no resultado, indicando que o número de caminhos independentes no grafo é 9.

## Criação dos Caminhos independentes

Durante a criação dos caminhos independentes, verificámos que grande parte dos caminhos eram *unfeasible*, o que significa que não existe qualquer *input* capaz de os percorrer. Em alguns casos, estes caminhos são logicamente impossíveis de tornar *feasible*, mas noutras é possível fazê-lo através de pequenas alterações na sequência do caminho.

Dessa forma, optámos por corrigir os caminhos *unfeasible* sempre que possível, de modo a identificar o maior número possível de caminhos diferentes *feasible*. Estes caminhos corrigidos foram assinalados com o número do caminho seguido da letra (b).

Nº do Caminho	Percorso	Feasible	pressure	target	Esperado	Obtido
P1	S, 1, 2, 5, 12, 13, 19, E	✓	[]	10	ValueError	[]

<b>P2</b>	S, 1, 2, 5, 12, 13, 14, 15, 18, 13, 19, E	X	-	-	-	-
<b>P3</b>	S, 1, 2, 5, 12, 13, 14, 15, 16, 15, 18, 13, 19, E	X	-	-	-	-
<b>P4</b>	S, 1, 2, 5, 12, 13, 14, 15, 16, 17, 18, 13, 19, E	X	-	-	-	-
<b>P5</b>	S, 1, 2, 5, 6, 7, 9, 10, 5, 12, 13, 19, E	X	-	-	-	-
<b>P6</b>	S, 1, 2, 5, 6, 7, 9, 10, 11, 5, 12, 13, 19, E	X	-	-	-	-
<b>P7</b>	S, 1, 2, 5, 6, 7, 8, 7, 9, 10, 5, 12, 13, 19, E	X	-	-	-	-
<b>P8</b>	S, 1, 2, 3, 2, 5, 12, 13, 19, E	X	-	-	-	-
<b>P9</b>	S, 1, 2, 3, 4, 2, 5, 12, 13, 19, E	X	-	-	-	-

**Corrigidos:**

Nº do Caminho	Percurso	Feasible	pressure	target	Esperado	Obtido
<b>P4b</b>	S, 1, 2, <b>3</b> , 2, 5, <b>6</b> , <b>7</b> , <b>8</b> , 7, <b>9</b> , <b>10</b> , 5, 12, 13, 14, 15, 16, 17, 18, 13, 19, E	✓	[[15]]	10	[]	[]
<b>P5b</b>	S, 1, 2, <b>3</b> , <b>2</b> , 5, 6, 7, 9, 10, 5, 12, 13, 19, E	✓	[[[]]]	0	ValueError	division by zero

P6b	1, 2, <b>3, 2, 5, 6, 7, 8, 7, 9,</b> 10, 11, 5, 12, 13, <b>14, 15,</b> <b>16, 17, 18, 13,</b> 19, E	✓	[[5]]	10	[0]	[0]
P7b	S, 1, <b>2, 3, 2, 5, 6, 7, 8, 7,</b> 9, 10, 5, 12, 13, 19, E	✓	[[20]]	10	[]	[]
P8b	S, 1, 2, 3, <b>4, 2, 3, 2, 5, 6,</b> <b>7, 8, 7, 9, 10, 5,</b> 12, 13, 19, E	✓	[[0,5]]	10	[0]	[0]
P9b	S, 1, 2, 3, 4, 2, 5, <b>6, 7, 9,</b> <b>10, 5,</b> 12, 13, 19, E	✓	[[0]]	0	ValueError	division by zero

Como podemos verificar, os caminhos P2 e P3 não foram possíveis de tornar *feasible*.

No caso do P2, isto deve-se ao facto de o caminho exigir que a condição ( $i < \text{len}(\text{warninglist})$ ) seja verdadeira em simultâneo com a condição  $\sim(i < \text{len}(\text{indexlist}))$ . No entanto, este caso nunca pode ocorrer, pois, assim que um valor é adicionado à `warninglist`, implica automaticamente que também tenha sido adicionado um valor à `indexlist`.

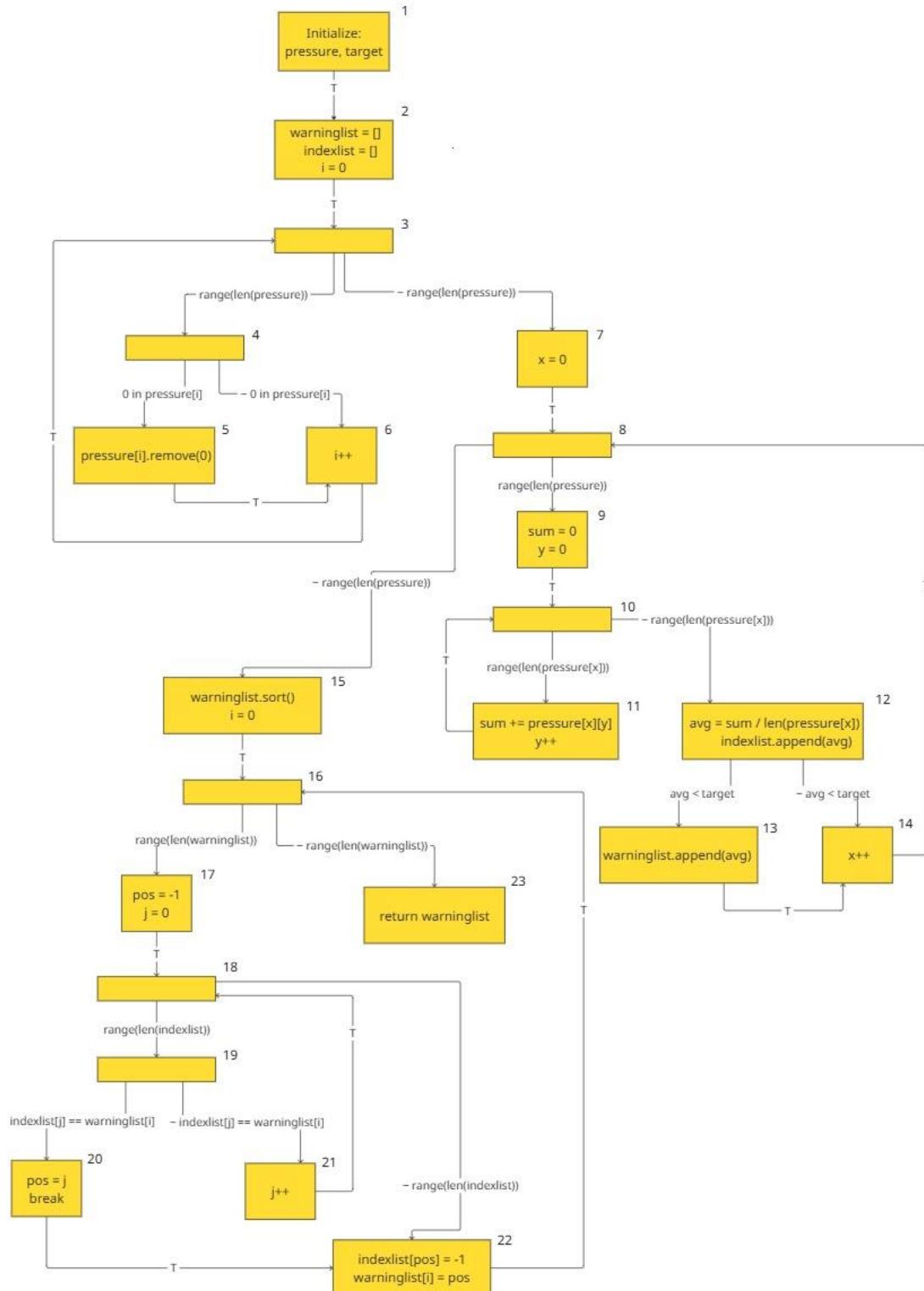
Por outro lado, o caminho P3 não pode ser feasible porque não é possível percorrer o array `indexlist` sem que, pelo menos uma vez, a condição `if indexlist[j] == warninglist[i]` seja verdadeira. Assim, tornando esse caminho logicamente impossível.

## Data Flow Graph

O Data Flow Graph (DFG) também é uma técnica de *white-box testing*, assim como o CFG, mas tem como objetivo detetar erros relacionados com a forma como as variáveis são definidas, utilizadas e eventualmente descartadas. Dessa forma, é possível identificar problemas como variáveis não inicializadas ou valores atribuídos, mas nunca utilizados.

O DFG é composto por **nós**, onde as variáveis são definidas e utilizadas (Global defs e Global c-uses), e por arestas, que ilustram o percurso dos dados entre essas definições e utilizações.

De seguida, apresentamos o nosso Data Flow Graph correspondente ao excerto de código 3\_16:



A coverage que escolhemos para o nosso DFG foi a All-du-paths (ADUP). Esta é uma das coverages mais robustas, focando-se na seleção de caminhos completos que incluem todos os du-paths a partir de um nó onde uma variável global é definida, assegurando que todas as utilizações subsequentes dessa mesma variável sejam testadas

## Construção dos Du-paths

Para a implementação dos du-paths, consideramos essencial, inicialmente, identificar os Global defs, Global c-uses e p-uses de cada variável. Este passo foi crucial para facilitar a construção e o mapeamento dos du-paths correspondentes.

De seguida, apresentamos as tabelas que documentam as anotações realizadas para cada variável, seguidas de uma tabela detalhada com todos os du-paths associados a cada uma das variáveis analisadas.

Variável	Global defs	Global c-uses	p-uses
<b>pressure</b>	1	5, 11, 12	(3,4), (3,7), (4,5), (4,6), (8,9), (8,15), (10,11), (10,12)
<b>target</b>	1	No use	(12,13), (12,14)
<b>warninglist</b>	2, 22	13, 15, 23	(16,17), (16,23), (19,20), (19,21)
<b>indexlist</b>	2, 22	12	(19,20), (19,21), (18,19), (18,22)
<b>i</b>	2, 15	6, 5, 22	(4,5), (4,6), (19,20), (19,21)
<b>x</b>	7	11, 12, 14	(10,11), (10,12)
<b>y</b>	9	11	No use
<b>sum</b>	9, 11	11, 12	No use
<b>avg</b>	12	12, 13	(12,13), (12,14)
<b>pos</b>	17, 20	22	No use

<b>j</b>	17	20	(19,20), (19,21)
----------	----	----	------------------

Variável	Du-paths
<b>pressure</b>	(1,2,3,4,5) / (1,2,3,4) / (1,2,3,7) / (1,2,3,4,6) / (1,2,3,7,8,9) / (1,2,3,7,8,15) / (1,2,3,7,8,9,10,11) / (1,2,3,7,8,9,10,12)
<b>target</b>	(1,2,3,7,8,9,10,12,13) / (1,2,3,7,8,9,10,12,14)
<b>warninglist</b>	(2,3,7,8,9,10,12,13) / (2,3,7,8,15) / (2,3,7,8,15,16,17) / (2,3,7,8,15,16,23) / (2,3,7,8,15,16,17,18,19,20) / (2,3,7,8,15,16,17,18,19,21) / (22, 16, 23)
<b>indexlist</b>	(2,3,7,8,9,10,12) / (2,3,7,8,15,16,17,18,19) / (2,3,7,8,15,16,17,18,22) / (2,3,7,8,15,16,17,18,19,20) / (2,3,7,8,15,16,17,18,19,21)
<b>i</b>	(2,3,4,6) / (2,3,4,5) / (15,16,17,18,22) / (15,16,17,18,19,20) / (15,16,17,18,19,21)
<b>x</b>	(7,8,9,10,11) / (7,8,9,10,12) / (7,8,9,10,12,14)
<b>y</b>	(9,10,11)
<b>sum</b>	(9,10,11) / (9,10,12)
<b>avg</b>	(12,13) / (12,14)
<b>pos</b>	(17,18,22) / (20,22)
<b>j</b>	(17,18,19,20) / (17,18,19,21)

## Construção dos Complete paths

Após implementarmos todos os du-paths, gerámos todos os complete paths de modo a garantir que todos os du-paths fossem percorridos. Tal como no CFG, verificámos que a maior parte dos testes eram unfeasible. Assim, procedemos à sua correção, visando obter o maior número possível de caminhos feasible diferentes, da mesma forma que fizemos anteriormente.

Nº do Caminho	Percorso	Feasible	pressure	target	Esperado	Obtido
P1	1, 2, 3, 4, 5, 6, 3, 7, 8, 15, 16, 23	X	-	-	-	-
P2	1, 2, 3, 7, 8, 15, 16, 23	✓	[]	10	ValueError	[]
P3	1, 2, 3, 4, 6, 3, 7, 8, 15, 16, 23	X	-	-	-	-
P4	1, 2, 3, 7, 8, 9, 10, 12, 14, 8, 15, 16, 23	X	-	-	-	-
P5	1, 2, 3, 7, 8, 9, 10, 11, 10, 12, 14, 8, 15, 16, 23	X	-	-	-	-
P6	1, 2, 3, 7, 8, 9, 10, 12, 13, 14, 8, 15, 16, 23	X	-	-	-	-
P7	1, 2, 3, 7, 8, 15, 16, 17, 18, 22, 16, 23	X	-	-	-	-
P8	1, 2, 3, 7, 8, 15, 16, 17, 18, 19, 20, 22, 16, 23	X	-	-	-	-
P9	1, 2, 3, 7, 8, 15, 16, 17, 18, 19, 21, 18, 22, 16, 23	X	-	-	-	-

**Corrigidos:**

Nº do Caminho	Percorso	Feasible	pressure	target	Esperado	Obtido
P1b	1, 2, 3, 4, 5, 6, 3, 7, 8, <b>9, 10, 12, 14, 8,</b> 15, 16, 23	✓	[[0]]	0	ValueError	division by zero
P3b	1, 2, 3, 4, 6, 3, 7, 8, <b>9, 10, 11, 10, 12, 13, 14, 8, 15, 16, 17, 18, 19, 20, 22, 16,</b> 23	✓	[[10]]	20	[0]	[0]
P4b	1, 2, 3, <b>4, 6, 3,</b> 7, 8, 9, 10, 12, 14, 8, 15, 16, 23	✓	[[[]]]	0	ValueError	division by zero
P5b	1, 2, 3, <b>4, 6, 3,</b> 7, 8, 9, 10, 11, 10, 12, 14, 8, 15, 16, 23	✓	[[20]]	10	[]	[]
P6b	1, 2, 3, <b>4, 6, 3,</b> 7, 8, 9, 10, 12, 13, 14, 8, 15, 16, <b>17, 18, 19, 20, 22, 16,</b> 23	✓	[[[]]]	10	ValueError	division by zero
P8b	1, 2, 3, <b>4, 5, 6, 3, 4, 6, 3, 7, 8, 9, 10, 11, 10, 12, 13, 14, 8,</b> 15, 16, 17, 18, 19, 20, 22, 16, 23	✓	[[0,10]]	20	[0]	[0]

Como podemos observar os P7 e P9 não foram possíveis tornar feasible pelos mesmos motivos mencionados acima no CFG.

## 2. FR2.1

O seguinte problema FR 2 consiste numa simulação do comportamento de um sistema de cruise control de um carro, isto é, ajustar a velocidade do carro através de um programa. O código em análise é o FR2\_1.py que demonstra uma possível solução sem *Fault Tolerance* ao problema. Este tem como parâmetros de entrada: velocidade atual (`current_speed`), histórico de velocidades anteriores (`prev_speed`), velocidade pretendida (`set_speed`), aceleração máxima possível (`max_accel`) e erro (`error`); retornando então a aceleração ou desaceleração aplicada ao veículo.

### Black-box Testing

#### Testes válidos

##### 1. Teste de velocidade atual é igual à velocidade pretendida:

Este teste verifica o comportamento do sistema quando a velocidade atual já é igual à desejada. Espera-se que, nenhuma aceleração ou desaceleração seja, necessária.

**Test:** `cruise_control(100, [100,100], 100, 10, 10)`

**Output esperado:** 0

**Output recebido:** 0

O programa retorna 0, o que confirma que nenhum ajuste foi aplicado à velocidade. Isso está correto e esperado.

##### 2. Teste de velocidade atual abaixo à velocidade pretendida sem erro:

Aqui testa-se se o sistema acelera corretamente quando a velocidade atual está abaixo da desejada, sem tolerância de erro.

**Test:** `cruise_control(45, [100,100], 50, 5, 0)`

**Output esperado:** 5

**Output recebido:** 5

O valor retornado foi 5, que corresponde exatamente ao valor de `max_accel`, indicando que o sistema atua corretamente para acelerar até atingir a meta.

### **3. Teste de velocidade atual acima à velocidade pretendida sem erro:**

Verifica-se se o programa desacelera corretamente quando a velocidade atual está acima da meta, com erro igual a 0.

**Test:** cruise\_control(55, [100,100], 50, 5, 0)

**Output esperado:** -5

**Output recebido:** -5

O sistema retorna -5, demonstrando desaceleração total dentro dos limites definidos.

### **4. Teste de velocidade atual abaixo à velocidade pretendida com erro:**

O objetivo deste teste corresponde à velocidade atual está abaixo da meta, mas dentro de uma margem de erro. O sistema deve aplicar correção proporcional.

**Test:** cruise\_control(90, [100, 110], 100, 10, 5)

**Output esperado:** 5

**Output recebido:** 5

O valor 5 retornado corresponde à diferença dentro do intervalo permitido pelo erro. O sistema responde corretamente, aplicando aceleração leve.

### **5. Teste de velocidade atual acima à velocidade pretendida com erro:**

Similar ao teste anterior, mas com velocidade acima da desejada. O teste verifica se o sistema aplica desaceleração proporcional dentro da margem de erro.

**Test:** cruise\_control(110, [100, 110], 100, 10, 5)

**Output esperado:** -5

**Output recebido:** -5

O valor -5 indica que o sistema está a reduzir a velocidade adequadamente. Comportamento conforme esperado.

### **6. Teste de aceleração máxima igual a zero:**

Verifica se o sistema respeita a limitação imposta por max\_accel = 0, ou seja, se evita acelerar ou desacelerar mesmo que haja diferença de velocidade.

**Test:** cruise\_control(90, [100, 110], 100, 0, 5)

**Output esperado:** 0

**Output recebido:** 0

O retorno 0 confirma que o sistema não atua, respeitando o limite imposto. Resultado correto.

## 7. Teste de velocidade pretendida fora de alcance:

O sistema é testado com uma meta de velocidade muito elevada para verificar se lida com metas incomuns sem quebrar.

**Test:** cruise\_control(50, [50, 50], 250, 10, 5)

**Output esperado:** 10

**Output recebido:** 10

O programa retorna 10, ou seja, aceleração máxima permitida. Mostra que o sistema opera bem mesmo com valores atípicos.

## Testes inválidos

### 1. Teste de valores de entrada não numéricos:

Testa-se se o sistema valida corretamente os tipos de entrada, fornecendo valores como strings ao invés de números.

**Test:** cruise\_control("90", ["100", "110"], "100", "10", "5")

**Output esperado:** 5 ou Erro

**Output recebido:** 0

O retorno foi 0 ao invés de erro, mostrando que não há validação de tipo, o que pode gerar falhas graves durante a execução.

### 2. Teste de histórico de velocidades vazio:

Verifica-se se o sistema consegue operar sem histórico de velocidades, algo que pode ocorrer na inicialização.

**Test:** cruise\_control(90, [], 100, 10, 5)

**Output esperado:** 5

**Output recebido:** 5

O sistema retorna 5 e executa normalmente. Embora o resultado esteja correto, isso revela ausência de validação do histórico.

### **3. Teste de velocidade atual com valores negativos:**

Fornece-se um valor negativo de velocidade para verificar se o sistema rejeita dados fisicamente inválidos.

**Test:** cruise\_control(-50, [0, 0], 100, 10, 5)

**Output esperado:** Erro

**Output recebido:** 10

O sistema retorna 10, tratando o valor como válido. Isso indica uma falha séria de validação, pois velocidade negativa é irrealista.

### **4. Teste de aceleração máxima com valores negativos:**

O objetivo é verificar se o sistema recusa um valor negativo de aceleração máxima, o que não faz sentido.

**Test:** cruise\_control(45, [40,45], 50, -5, 2)

**Output esperado:** Erro

**Output recebido:** 0

O retorno foi 0, em vez de erro. O sistema não valida o campo, o que pode levar a comportamento incoerente.

### **5. Teste de erro com valores negativos:**

Testa-se se o campo erro aceita valores negativos, o que violaria o conceito de margem.

**Test:** cruise\_control(45, [40,45], 50, 5, -2)

**Output esperado:** Erro

**Output recebido:** 0

O sistema não gera erro. Ausência de validação para um campo que não deveria aceitar valores negativos.

### **6. Teste de valores de entrada nulos:**

O sistema é testado com valores *None*, para ver se possui proteção contra ausência de dados.

**Test:** cruise\_control(None, [], None, None, None)

**Output esperado:** Erro

**Output recebido:** 0

O programa retorna 0 sem lançar erro. Isso demonstra falta de robustez contra dados indefinidos ou incompletos, podendo levar a falhas inesperadas.

## 7. Teste de valores de entrada nulos:

Testa-se se o campo set\_speed aceita valores negativos.

**Test:** cruise\_control(45, [40,45], -50, 5, 2)

**Output esperado:** Erro

O sistema não gera erro. Ausência de validação para um campo que não deveria aceitar valores negativos.

## BVA (Boundary Value Analysis)

Considerando o limite máximo de velocidade de um carro convencional seria 200km/h, definimos um intervalo aceitável de [0, 200] km/h.

Com isto, foram realizados testes com base nesse intervalo verificando se perante valores imediatamente abaixo e acima resulta em erro. Os testes foram escolhidos os mesmos que previamente foram citados, porém apenas os casos ao qual se demonstra relevante ao problema são implementados testes BVA.

### 2b - Teste de velocidade atual abaixo à velocidade pretendida sem erro

**Test:** cruise\_control(49.9, [100,100], 50, 1, 0)

**Output Esperado:** 0.1

**Output Obtido:** 0.10000000000000142

### 2c - Teste de velocidade atual abaixo à velocidade pretendida sem erro

**Test:** cruise\_control(0.1, [100,100], 50, 49, 0)

**Output Esperado:** 49

**Output Obtido:** 49

### 3b - Teste de velocidade atual acima à velocidade pretendida sem erro

**Test:** cruise\_control(50.1, [100,100], 50, 1, 0)

**Output Esperado:** -0.1

**Output Obtido:** -0.10000000000000142

### **3c - Teste de velocidade atual acima à velocidade pretendida sem erro**

**Test:** cruise\_control(199.9, [100,100], 50, 50, 0)

**Output Esperado:** -50.0

**Output Obtido:** -50.0

### **4b - Teste de velocidade atual abaixo à velocidade pretendida com erro**

**Test:** cruise\_control(49.9, [100,100], 50, 1, 0.01)

**Output Esperado:** 0.09

**Output Obtido:** 0.090000000000000341

### **4c - Teste de velocidade atual abaixo à velocidade pretendida com erro**

**Test:** cruise\_control(0.1, [100,100], 50, 49, 49.8)

**Output Esperado:** 0.1

**Output Obtido:** 0.10000000000000284

### **5b - Teste de velocidade atual acima à velocidade pretendida com erro**

**Test:** cruise\_control(50.1, [100,100], 50, 1, 0.01)

**Output Esperado:** -0.09

**Output Obtido:** -0.09000000000000341

### **5c - Teste de velocidade atual acima à velocidade pretendida com erro**

**Test:** cruise\_control(199.9, [100,100], 50, 50, 149.8)

**Output Esperado:** -0.1

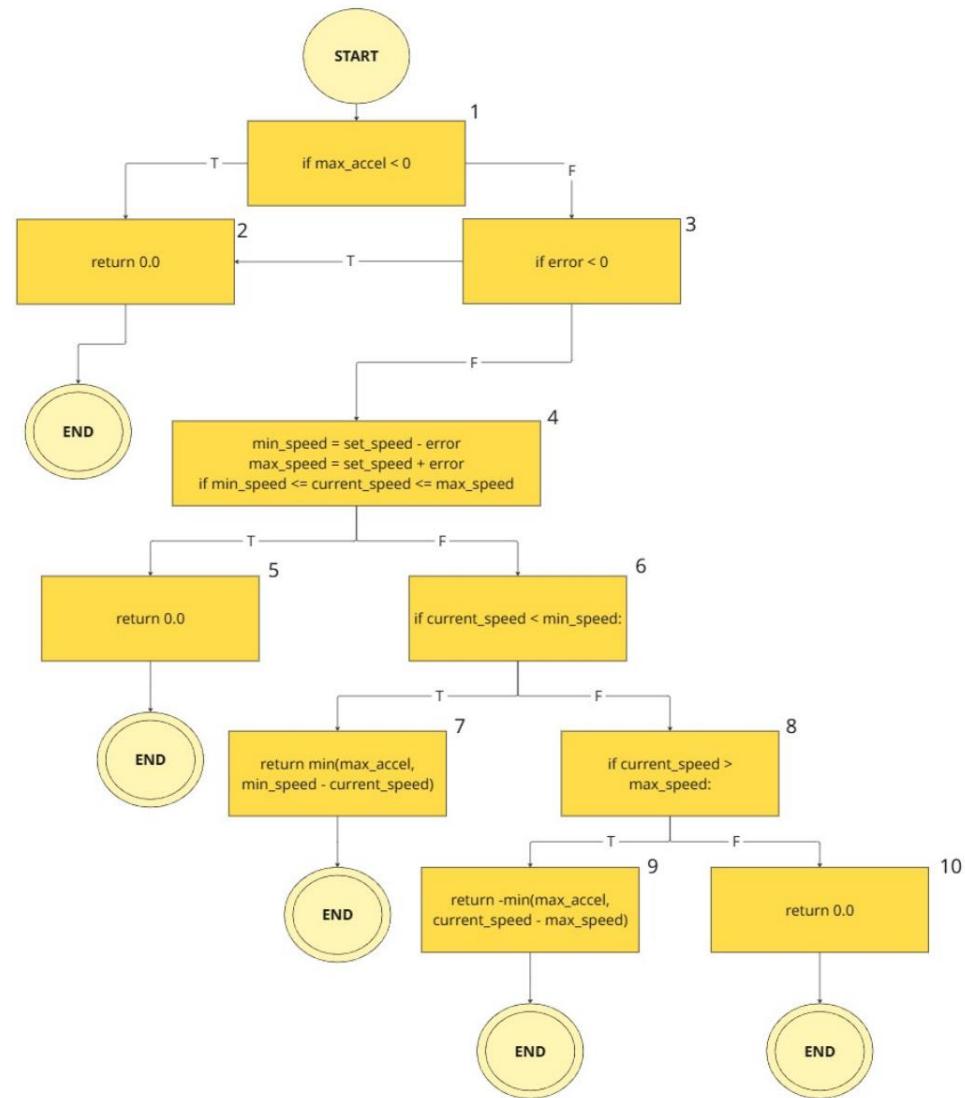
**Output Obtido:** -0.09000000000000341

Após a execução dos testes, verificámos que os resultados obtidos não foram idênticos aos registados nos testes anteriores. Os outputs aos quais não foram idênticos aos esperados devesse ao fenómeno de imprecisão de ponto flutuante (floating point precision error). Assim, como os valores são armazenados em float usando uma representação binária com o padrão IEE 754, reparamos que nem todos os números decimais podem ser representados exatamente em binário, ou, também pode ser devido à acumulação de erro em operações com floats.

# White-box testing

## Control Flow Graph

Encontra-se demonstrado abaixo o nosso CFG para o código FR2\_1:



## Contagem do número de caminhos prioritários:

Como o número de nós de saída do grafo é superior a um apenas é possível calcular usando a seguinte fórmula:

$$\text{Fórmula: } V(G) = \text{Número de nós de decisão} + 1$$

$$= 5 + 1 = 6$$

## Criação dos Caminhos independentes

Ao desenvolver os percursos para os caminhos independentes denotamos que todos os caminhos são feasible, isto é, existe uma combinação de entradas que faz com que esse caminho seja seguido pelo programa em análise.

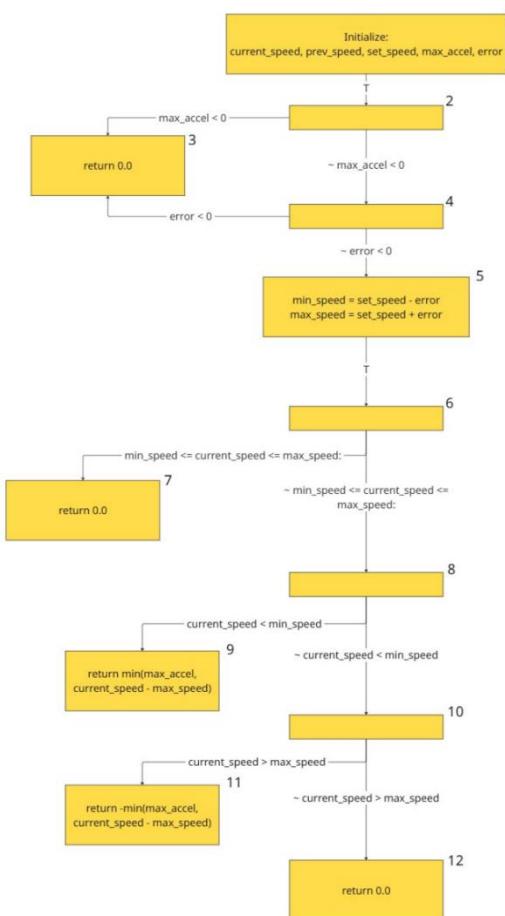
Nº	Percorso	Feasible	Current_Speed	Prev_Speed	Set_Speed	Max_accel	Error	Esperado / Obtido
P1	S, 1, 2, E	✓	NI	NI	NI	<0	>0	Erro / 0
P2	S, 1, 3, 2, E	✓	NI	NI	NI	>0	<0	Erro / 0
P3	S, 1, 3, 4, 5, E	✓	100	NI	100	NI	0	0 / 0
P4	S, 1, 3, 4, 6, 7, E	✓	80	NI	100	10	15	5 / 5
P5	S, 1, 3, 4, 6, 8, 9, E	✓	100	NI	80	10	10	-10 / -10
P6	S, 1, 3, 4, 6, 8, 10, E	✓	100	NI	120	10	10	10 / 10

Legenda:

NI – Não influencia o caminho independente

# Data Flow Graph

De seguida encontra-se o grafo DFG correspondente ao código em análise, FR2\_1.



## Construção dos Du-paths

Para a construção dos du-paths, foi criada primeiramente a tabela que identifica as definições globais, c-uses globais e p-uses de cada variável. O resultado dessa construção encontra-se abaixo:

Variável	Global defs	Global c-uses	p-uses
<b>current_speed</b>	1	9, 11	(6,7), (6,8), (8,9), (8,10), (10,11), (10,12)

<b>prev_speed</b>	1	No use	No use
<b>set_speed</b>	1	5	No use
<b>max_accel</b>	1	9, 11	(2,3), (2,4)
<b>error</b>	1	5	(4,3), (4,5)
<b>min_speed</b>	5	No use	(6,7), (6,8), (8,9), (8, 10)
<b>max_speed</b>	5	9, 11	(6,7), (6,8), (10,11), (10,12)

Com essa tabela conseguimos, então, fazer os du-paths para cada uma das variáveis, sendo o resultado demonstrado abaixo:

Variável	Du-paths
<b>current_speed</b>	(1,2,4,5,6,7), (1,2,4,5,6,8), (1,2,4,5,6,8,9), (1,2,4,5,6,8,10), (1,2,4,5,6,8,10,11), (1,2,4,5,6,8,10,12)
<b>prev_speed</b>	No path
<b>set_speed</b>	(1,2,4,5)
<b>max_accel</b>	(1,2,3), (1,2,4), (1,2,4,5,6,8,9), (1,2,4,5,6,8,10,11)
<b>error</b>	(1,2,4,3), (1,2,4,5)
<b>min_speed</b>	(5,6,7), (5,6,8), (5,6,8,9), (5,6,8,10)
<b>max_speed</b>	(5,6,7), (5,6,8), (5,6,8,9), (5,6,8,10,11), (5,6,8,10,12)

## Construção dos Complete paths

Com tudo isto realizado, encontra-se de seguida os complete paths possíveis:

Nº	Percorso	Feasible	Current_Speed	Prev_Speed	Set_Speed	Max_accel	Error	Esperado / Obtido
<b>P1</b>	1,2,3	✓	NI	NI	NI	<0	>0	Erro / 0
<b>P2</b>	1,2,4,3	✓	NI	NI	NI	>0	<0	Erro / 0
<b>P3</b>	1,2,4,5,6,7	✓	100	NI	100	NI	0	0 / 0
<b>P4</b>	1,2,4,5,6,8,9	✓	80	NI	100	10	15	5 / 5
<b>P5</b>	1,2,4,5,6,8,10,11	✓	100	NI	80	10	10	-10 / -10
<b>P6</b>	1,2,4,5,6,8,10,12	✓	100	NI	120	10	10	10 / 10

É de notar que todos os caminhos criados são feasible.

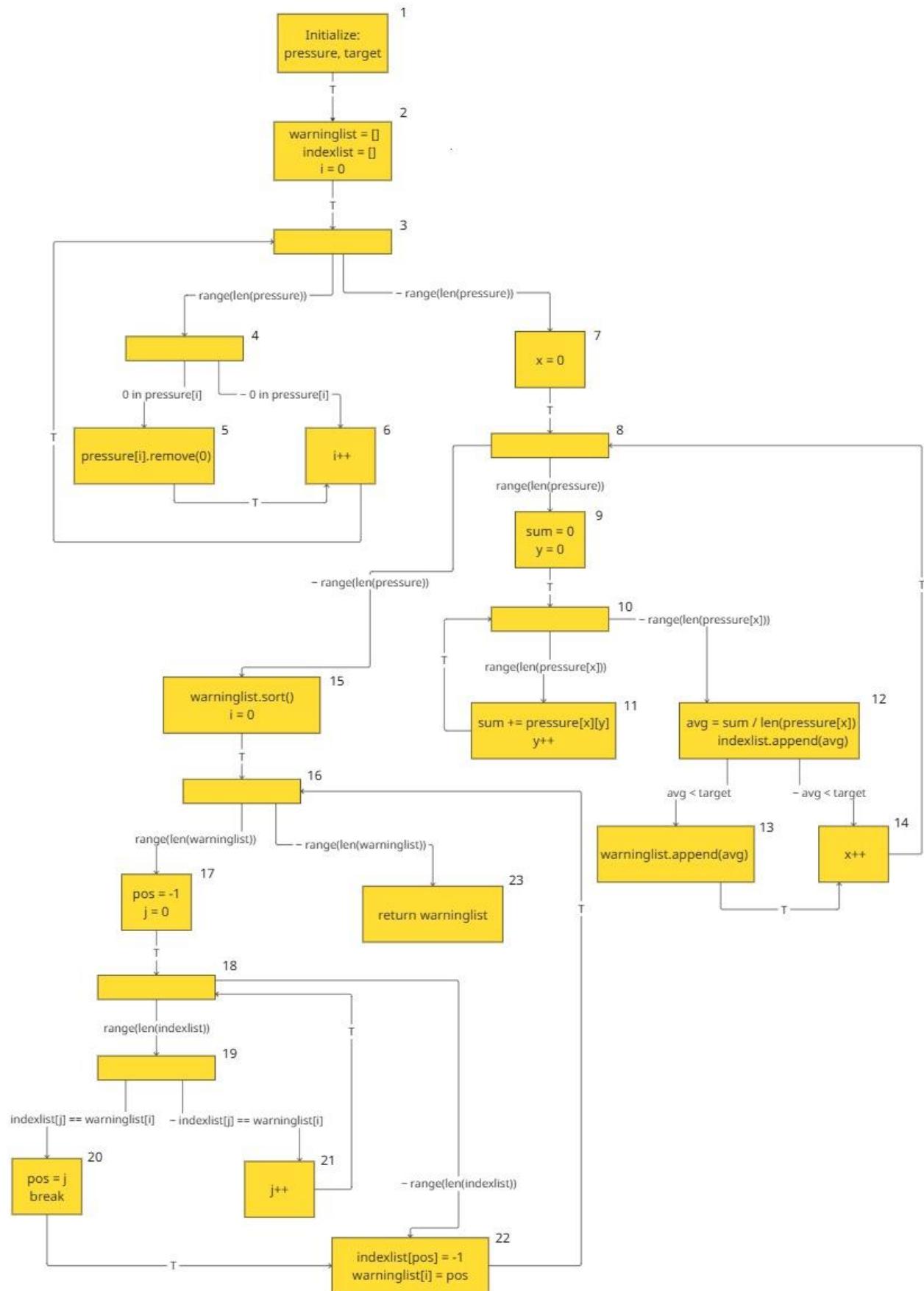
## Anexo A:

### Gráficos com Resolução Aumentada

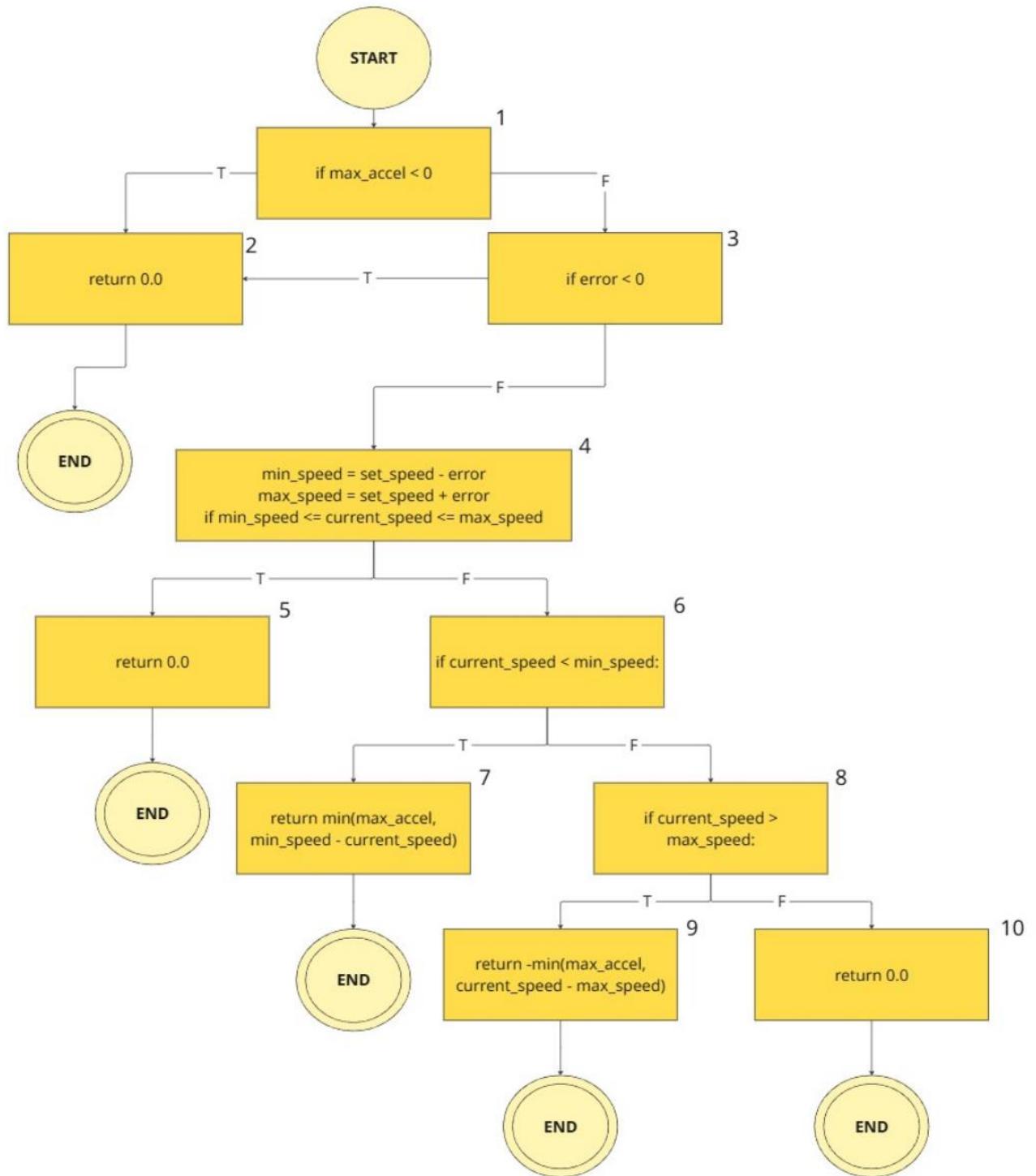
CFG FR3\_16



## DFG FR3\_16



## CFG FR2\_1



## DFG FR2\_1

