

# Primeiras Especificações em TLA+

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC  
Universidade do Estado de Santa Catarina - UDESC

28 de abril de 2025



# Conteúdo

Correção exercícios

Modelos em TLA+

Semáforos

Jogo da Velha



# Outline

Correção exercícios

Modelos em TLA+

Semáforos

Jogo da Velha



# Correção dos exercícios

Vamos corrigir juntos no VSCode.

- Mostrar em PDF



# Outline

Correção exercícios

Modelos em TLA+

Semáforos

Jogo da Velha

# Modelos em TLA+

Até agora, vimos principalmente a parte funcional da linguagem.

- Diferente de Quint, não temos operadores exclusivos para ações. Ações são bem parecidas com a parte funcional.
- Podemos identificar ações pelo operador *primed* ( ' ), e pelo **UNCHANGED**

## Estado inicial:

- Em Quint, o estado inicial é uma ação (como  $x' = 0$ )
- Em TLA+, o estado inicial é um predicado (como  $x = 0$ , que em Quint seria  $x==0$ ).
  - Lemos como “Um estado é um estado inicial sse satisfaz *Init*”



# Outline

Correção exercícios

Modelos em TLA+

**Semáforos**

Jogo da Velha

# Especificação para os semáforos

```
1  ----- MODULE Semaforos -----  
2  EXTENDS Integers, FiniteSets  
3  VARIABLE cores, proximo  
4  CONSTANT SEMAFOROS  
5  /* ...  
6  =====
```



# Especificação para os semáforos

```
1  ----- MODULE Semaforos -----  
2  EXTENDS Integers, FiniteSets  
3  VARIABLE cores, proximo  
4  CONSTANT SEMAFOROS  
5  \* ...  
6  =====
```

PS: Consultei a gramática de TLA+ e precisamos de pelo menos 4 hífens (----) antes e depois de **MODULE nome** e pelo menos 4 iguais (====) no final.

```
1  TLAPlusGrammar == \* ...  
2    G.Module ::= AtLeast4("-")  
3      & tok("MODULE") & Name & AtLeast4("-")  
4      & (Nil | (tok("EXTENDS") & CommaList(Name)))  
5      & (G.Unit)^*  
6      & AtLeast4("=") \* ...
```



# Estado inicial

```
1 Init ==  
2 /\ cores = [s \in SEMAFOROS |-> "vermelho"]  
3 /\ proximo = 0
```

## Próximo semáforo fica verde

```
1 FicaVerde(s) ==  
2   /\ proximo = s  
3   /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"  
4   /\ cores' = [cores EXCEPT ![s] = "verde"]  
5   /\ proximo' = (s + 1) % Cardinality(SEMAFOROS)
```

## Próximo semáforo fica verde

```
1 FicaVerde(s) ==  
2   /\ proximo = s  
3   /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"  
4   /\ cores' = [cores EXCEPT ![s] = "verde"]  
5   /\ proximo' = (s + 1) % Cardinality(SEMAFOROS)
```

Qual a pré-condição dessa ação?

## Próximo semáforo fica verde

```
1 FicaVerde(s) ==  
2   /\ proximo = s  
3   /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"  
4   /\ cores' = [cores EXCEPT ![s] = "verde"]  
5   /\ proximo' = (s + 1) % Cardinality(SEMAFOROS)
```

Qual a pré-condição dessa ação?

```
1 /\ proximo = s  
2 /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"
```

## Próximo semáforo fica verde: Exercício

```
1 FicaVerde(s) ==  
2   /\ proximo = s  
3   /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"  
4   /\ cores' = [cores EXCEPT ![s] = "verde"]  
5   /\ proximo' = (s + 1) % Cardinality(SEMAFOROS)
```

**Lendo essa ação:** tente preencher as lacunas

“FicaVerde para um semáforo  $s$  define uma transição onde, no estado atual, \_\_\_\_\_ deve ser \_\_\_\_\_ e \_\_\_\_\_ deve \_\_\_\_\_; e no próximo estado, \_\_\_\_\_ deve ser \_\_\_\_\_ e \_\_\_\_\_ deve ser \_\_\_\_\_”

## Próximo semáforo fica verde: Exercício

```
1 FicaVerde(s) ==  
2   /\ proximo = s  
3   /\ \A s2 \in SEMAFOROS : cores[s2] = "vermelho"  
4   /\ cores' = [cores EXCEPT ![s] = "verde"]  
5   /\ proximo' = (s + 1) % Cardinality(SEMAFOROS)
```

### Possível resolução:

“**FicaVerde** para um semáforo **s** define uma transição onde, no estado atual, o valor de proximo deve ser igual ao semáforo s e o valor de cores para cada semáforo em SEMAFOROS deve ser vermelho; e no próximo estado, o valor de cores deve ser igual a cores no estado atual, exceto que o valor de s deve ser verde e o valor de proximo deve ser o incremento do valor no estado atual módulo o tamanho do conjunto SEMAFOROS”.

# Um semáforo que está verde fica amarelo

```
1 FicaAmarelo(s) ==  
2   /\ cores[s] = "verde"  
3   /\ cores' = [cores EXCEPT ![s] = "amarelo"]  
4   /\ UNCHANGED << proximo >>
```



# Um semáforo que está verde fica amarelo

```
1 FicaAmarelo(s) ==  
2   /\ cores[s] = "verde"  
3   /\ cores' = [cores EXCEPT ![s] = "amarelo"]  
4   /\ UNCHANGED << proximo >>
```

Qual a pré-condição dessa ação?

## Um semáforo que está verde fica amarelo

```
1 FicaAmarelo(s) ==  
2   /\ cores[s] = "verde"  
3   /\ cores' = [cores EXCEPT ![s] = "amarelo"]  
4   /\ UNCHANGED << proximo >>
```

Qual a pré-condição dessa ação?

```
1 cores[s] = "verde"
```

# Um semáforo que está amarelo fica vermelho

```
1 FicaVermelho(s) ==  
2   /\ cores[s] = "amarelo"  
3   /\ cores' = [cores EXCEPT ![s] = "vermelho"]  
4   /\ UNCHANGED << proximo >>
```

# Um semáforo que está amarelo fica vermelho

```
1 FicaVermelho(s) ==  
2   /\ cores[s] = "amarelo"  
3   /\ cores' = [cores EXCEPT ![s] = "vermelho"]  
4   /\ UNCHANGED << proximo >>
```

Qual a pré-condição dessa ação?

# Um semáforo que está amarelo fica vermelho

```
1 FicaVermelho(s) ==  
2   /\ cores[s] = "amarelo"  
3   /\ cores' = [cores EXCEPT ![s] = "vermelho"]  
4   /\ UNCHANGED << proximo >>
```

Qual a pré-condição dessa ação?

```
1 cores[s] = "amarelo"
```



# Função de próximo estado

```
1 Next == \E s \in SEMAFOROS : FicaVerde(s) \/
      FicaAmarelo(s) \/ FicaVermelho(s)
```

# Função de próximo estado

```
1 Next == \E s \in SEMAFOROS : FicaVerde(s) \/
      FicaAmarelo(s) \/ FicaVermelho(s)
```

Lembrando que TLA+ permite usarmos ações dentro de um *exists*, diferentemente de Quint onde é necessário usar o *nondet* e *oneOf*.



# Propriedades

Uma invariante para check de sanidade:

```
1 Inv == cores[2] /= "amarelo"
```



# Propriedades

Uma invariante para check de sanidade:

```
1 Inv == cores[2] /= "amarelo"
```

Uma propriedade do sistema: para que os veículos não colidam, não podemos ter mais de um semáforo aberto ao mesmo tempo.

```
1 SemColisao == Cardinality({s \in SEMAFOROS : cores[s]  
    = "verde"}) <= 1
```



# Outline

Correção exercícios

Modelos em TLA+

Semáforos

Jogo da Velha

# Jogo da Velha em TLA+

Vamos ver a mesma especificação do jogo da velha, agora em TLA+

- Vamos direto pra versão onde o jogador X usa estratégia
- Vou usar a versão original do autor (SWART, 2022)
  - A especificação em Quint é baseada nesta, mas não é uma tradução direta. Poderíamos escrever a tradução mais próxima possível, mas não é esse o caso. Eu escrevi como achei que seria melhor, dado os recursos da linguagem.



# Módulo

```
1 ----- MODULE tictactoeexwin -----  
2  
3 EXTENDS Naturals  
4  
5 /* ...  
6  
7 =====
```

# Tipos e variáveis

Definimos as seguintes variáveis

```
1 VARIABLES
2   board, /* board[1..3][1..3] A 3x3 tic-tac-toe board
3   nextTurn /* who goes next
```

# Definições sobre coordenadas e tabuleiro

```
1 BoardIs(coordinate, player) ==
2     board[coordinate[1]][coordinate[2]] = player
3
4 BoardFilled ==
5     \* There does not exist
6     ~\E i \in 1..3, j \in 1..3:
7         \* an empty space
8         LET space == board[i][j] IN
9         space = "_"
10
11 BoardEmpty ==
12     \* There does not exist
13     \A i \in 1..3, j \in 1..3:
14         \* an empty space
15         LET space == board[i][j] IN
16         space = "_"
```

## Definindo “ganhar” - coordenadas

- Como o tabuleiro é sempre 3x3, é mais fácil listar todas as combinações de coordenadas que levam a uma vitória do que implementar os cálculos.
- Usamos tuplas! Na implementação em Quint, usamos `filter` e `size` para ver se um *pattern* tinha dois X e um branco, um X e dois brancos, etc. Aqui, o autor usa listas de permutação.

```
1 WinningPositions == {  
2   \* Horizontal wins  
3   <<<<1,1>>, <<1,2>>, <<1,3>>>>,  
4   <<<<2,1>>, <<2,2>>, <<2,3>>>>,  
5   <<<<3,1>>, <<3,2>>, <<3,3>>>>,  
6   \* Vertical wins  
7   <<<<1,1>>, <<2,1>>, <<3,1>>>>,  
8   <<<<1,2>>, <<2,2>>, <<3,2>>>>,  
9   <<<<1,3>>, <<2,3>>, <<3,3>>>>,  
10  \* Diagonal wins  
11  <<<<1,1>>, <<2,2>>, <<3,3>>>>,  
12  <<<<3,1>>, <<2,2>>, <<1,3>>>>
```

## Definindo “ganhar” - operador won

Usamos a definição `winningPositions` para determinar se um jogador venceu.

```
1 Won(player) ==  
2   \* A player has won if there exists a winning  
   position  
3   \E winningPosition \in WinningPositions:  
4     \* Where all the needed spaces  
5     \A i \in 1..3:  
6       \* are occupied by one player  
7       board[winningPosition[i][1]][  
winningPosition[i][2]] = player
```



## Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
1 Move(player, coordinate) ==  
2   /\ board[coordinate[1]][coordinate[2]] = "_"  
3   /\ board' = [board EXCEPT  
4                        ![coordinate[1]][coordinate  
                        [2]] = player]
```

- Qual é a pré-condição pra essa ação?

## Ações - Move

Um dado jogador faz uma jogada (um *move*) em uma dada coordenada

- Determinístico

```
1 Move(player, coordinate) ==  
2   /\ board[coordinate[1]][coordinate[2]] = "_"  
3   /\ board' = [board EXCEPT  
4                        ![coordinate[1]][coordinate  
                        [2]] = player]
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que a coordenada esteja vazia

## Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 MoveToEmpty(player) ==  
2   /\ \E i \in 1..3: \E j \in 1..3: \* There exists a  
   position on the board  
3   /\ board[i][j] = "_" \* Where the board is  
   currently empty  
4   /\ Move(player, <<i,j>>)
```

- Qual é a pré-condição pra essa ação?

## Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 MoveToEmpty(player) ==  
2   /\ \E i \in 1..3: \E j \in 1..3: \* There exists a  
   position on the board  
3   /\ board[i][j] = "_" \* Where the board is  
   currently empty  
4   /\ Move(player, <<i,j>>)
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que o jogo ainda não tenha acabado

## Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 MoveToEmpty(player) ==  
2   /\ \E i \in 1..3: \E j \in 1..3: \* There exists a  
   position on the board  
3   /\ board[i][j] = "_" \* Where the board is  
   currently empty  
4   /\ Move(player, <<i,j>>)
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que o jogo ainda não tenha acabado
- Aonde temos não determinismo aqui?

## Ações - MoveToEmpty

Um dado jogador faz uma jogada em **alguma** coordenada

- Não-determinístico

```
1 MoveToEmpty(player) ==  
2   /\ \E i \in 1..3: \E j \in 1..3: \* There exists a  
   position on the board  
3   /\ board[i][j] = "_" \* Where the board is  
   currently empty  
4   /\ Move(player, <<i,j>>)
```

- Qual é a pré-condição pra essa ação?
  - A pré-condição para essa ação é que o jogo ainda não tenha acabado
- Aonde temos não determinismo aqui?
  - No uso da ação **Move**, que atualiza a variável **board** dentro de um *exists* (**\E**).

## Ações - MoveO

```
1 MoveO ==  
2   /\ nextTurn = "O"  \* Only enabled on O's turn  
3   /\ ~Won("X") \* And X has not won  
4   /\ MoveToEmpty("O") \* O still tries every empty  
   space  
5   /\ nextTurn' = "X" \* The future state of next turn  

```

- Qual é a pré-condição pra essa ação?

## Ações - MoveO

```
1 MoveO ==  
2   /\ nextTurn = "O"  \* Only enabled on O's turn  
3   /\ ~Won("X") \* And X has not won  
4   /\ MoveToEmpty("O") \* O still tries every empty  
   space  
5   /\ nextTurn' = "X" \* The future state of next turn  

```

- Qual é a pré-condição pra essa ação?

```
1   /\ nextTurn = "O"  \* Only enabled on O's turn  
2   /\ ~Won("X") \* And X has not won
```

- Implicitamente, também temos a pré-condição de **MoveToEmpty** empregada nessa ação



# Estratégia para o jogador X

## Estratégia:

- A primeira jogada é sempre nos cantos
- As outras jogadas fazem a primeira jogada possível nessa lista de prioridade:
  - Ganhar
  - Bloquear
  - Jogar no centro
  - Preparar uma vitória (preenchendo 2 de 3 quadrados numa fila/coluna/diagonal)
  - Jogada qualquer

# Começando com os cantos

```
1  Corners == {  
2      <<1,1>>,  
3      <<3,1>>,  
4      <<1,3>>,  
5      <<3,3>>  
6  }  
7  
8  StartInCorner ==  
9      \E corner \in Corners:  
10         Move("X", corner)
```



## Condições para as jogadas

Precisamos definir as condições que determinam se cada uma das jogadas na lista de prioridade pode ser feita.

## Condições para as jogadas

Precisamos definir as condições que determinam se cada uma das jogadas na lista de prioridade pode ser feita.

Para isso, nessa especificação, definimos as permutações, e fazemos um nível a mais de interação (com *exists*), verificando, para cada **winningPosition** e para cada permutação, se aquela permutação da **winningPosition** é uma ordem específica do que queremos (X, X, e vazio).

```
1 PartialWins == {  
2   <<1,2,3>>,  
3   <<2,3,1>>,  
4   <<3,1,2>>  
5 }
```

## Condições para as jogadas II

```
1 CanWin == \E winningPostion \in WinningPositions,
    partialWin \in PartialWins:
2   /\ BoardIs(winningPostion[partialWin[1]], "X")
3   /\ BoardIs(winningPostion[partialWin[2]], "X")
4   /\ BoardIs(winningPostion[partialWin[3]], "_")
5
6 CanBlockWin == \E winningPostion \in WinningPositions,
    partialWin \in PartialWins:
7   /\ BoardIs(winningPostion[partialWin[1]], "O")
8   /\ BoardIs(winningPostion[partialWin[2]], "O")
9   /\ BoardIs(winningPostion[partialWin[3]], "_")
```

## Condições para as jogadas III

```
1 CanTakeCenter == board[2][2] = "_"
2
3 CanSetupWin == \E winningPostion \in WinningPositions,
   partialWin \in PartialWins:
4   /\ BoardIs(winningPostion[partialWin[1]], "X")
5   /\ BoardIs(winningPostion[partialWin[2]], "_")
6   /\ BoardIs(winningPostion[partialWin[3]], "_")
```

## Ações - Win

```
1 Win == \E winningPostion \in WinningPositions ,  
    partialWin \in PartialWins:  
2   /\ BoardIs(winningPostion[partialWin[1]], "X")  
3   /\ BoardIs(winningPostion[partialWin[2]], "X")  
4   /\ BoardIs(winningPostion[partialWin[3]], "_")  
5   /\ Move("X", winningPostion[partialWin[3]])
```

- Qual é a pré-condição pra essa ação?

## Ações - Win

```
1 Win == \E winningPostion \in WinningPositions ,  
    partialWin \in PartialWins:  
2   /\ BoardIs(winningPostion[partialWin[1]],"X")  
3   /\ BoardIs(winningPostion[partialWin[2]],"X")  
4   /\ BoardIs(winningPostion[partialWin[3]],"_")  
5   /\ Move("X", winningPostion[partialWin[3]])
```

- Qual é a pré-condição pra essa ação?

```
1 \E winningPostion \in WinningPositions , partialWin \in  
    PartialWins:  
2   /\ BoardIs(winningPostion[partialWin[1]],"X")  
3   /\ BoardIs(winningPostion[partialWin[2]],"X")  
4   /\ BoardIs(winningPostion[partialWin[3]],"_")
```

- Percebam que essa pré condição é exatamente **CanWin**
- Porém, não conseguimos usar **CanWin** aqui porque precisamos saber em qual posição jogar.



## Ações - BlockWin

De forma semelhante, **BlockWin**:

```
1 BlockWin == \E winningPostion \in WinningPositions ,  
    partialWin \in PartialWins:  
2   /\ BoardIs(winningPostion[partialWin[1]], "0")  
3   /\ BoardIs(winningPostion[partialWin[2]], "0")  
4   /\ BoardIs(winningPostion[partialWin[3]], "_")  
5   /\ Move("X", winningPostion[partialWin[3]])
```

## Ações - TakeCenter e SetupWin

```
1 TakeCenter ==
2   /\ Move("X", <<2,2>>)
3
4 SetupWin == \E winningPostion \in WinningPositions,
5   partialWin \in PartialWins:
6   /\ BoardIs(winningPostion[partialWin[1]], "X")
7   /\ BoardIs(winningPostion[partialWin[2]], "_")
8   /\ BoardIs(winningPostion[partialWin[3]], "_")
9   /\ \E i \in 2..3:
10    Move("X", winningPostion[partialWin[i]])
```

# Ações - MoveX

```
1 MoveX ==
2   /\ nextTurn = "X" /* Only enabled on X's turn
3   /\ ~Won("O") /* And X has not won
4   /* This specifies the spots X will move on X's
   turn
5   /\ \/ /\ BoardEmpty
6       /\ StartInCorner
7       \/ /\ ~BoardEmpty /* If its not the start
8       /\ \/ /\ CanWin
9           /\ Win
10          \/ /\ ~CanWin
11          /\  \/ /\ CanBlockWin
12              /\ BlockWin
13              \/ /\ ~CanBlockWin
14              /\  \/ /\ CanTakeCenter
15                  /\ TakeCenter
16                  \/ /\ ~CanTakeCenter
17                  /\  \/ /\ CanSetupWin
18                      /\ SetupWin
```

# Estado inicial

```
1 Init ==  
2 /\ nextTurn = "X" /* X always goes first  
3 /* Every space in the board states blank  
4 /\ board = [i \in 1..3 |-> [j \in 1..3 |-> "_"]]
```

# Transições

```
1 GameOver == Won("X") /\ Won("O") /\ BoardFilled
2
3 /* Every state, X will move if X's turn, O will move
   on O's turn
4 Next == MoveX \/ MoveO \/ (GameOver /\ UNCHANGED <<
   board, nextTurn >>)
```

# Transições

```
1 GameOver == Won("X") /\ Won("O") /\ BoardFilled
2
3 /* Every state, X will move if X's turn, O will move
   on O's turn
4 Next == MoveX \/ MoveO \/ (GameOver /\ UNCHANGED <<
   board, nextTurn >>)
```

- Nota: isso está um pouco diferente na especificação original, porque o autor usa a definição de **Spec** ao invés de somente **Init** e **Next**.

```
1 /* Every state, X will move if X's turn, O will move
   on O's turn
2 Next == MoveX \/ MoveO
3
4 /* A description of every possible game of tic-tac-toe
5 /* will play until the board fills up, even if someone
   won
6 Spec == Init /\ [][Next]_<<board,nextTurn>>
```

# Invariantes

```
1 XHasNotWon == ~Won("X")
2 OHasNotWon == ~Won("O")
3
4 /* It's not a stalemate if one player has won or the
   board is not filled
5 NotStalemate ==
6     \/ Won("X")
7     \/ Won("O")
8     \/ ~BoardFilled
```



# Referências

SWART, E. **Introduction to pragmatic formal modeling**. Disponível em:  
<<https://elliotswart.github.io/pragmaticformalmodeling/>>.



# Primeiras Especificações em TLA+

Aula para disciplina de Métodos Formais

Gabriela Moreira

Departamento de Ciência da Computação - DCC  
Universidade do Estado de Santa Catarina - UDESC

28 de abril de 2025