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

16 de setembro de 2024

Conteúdo

Outline

Correção dos exercícios

Vamos corrigir juntos no VSCode.

- Mostrar em PDF

Outline

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

Especificação para os semáforos

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

Especificação para os semáforos

```
----- MODULE Semaforos ------
EXTENDS Integers, FiniteSets
VARIABLE cores, proximo
CONSTANT SEMAFOROS
\* ...
=============================
```

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.

```
TLAPlusGrammar == \* ...
  G.Module ::= AtLeast4("-")
     & tok("MODULE") & Name & AtLeast4("-")
     & (Nil | (tok("EXTENDS") & CommaList(Name)))
     & (G.Unit)^*
     & 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 prencher 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:

```
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.

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

Outline

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 tictactoexwin ---------
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

```
BoardIs(coordinate, player) ==
    board[coordinate[1]][coordinate[2]] = player

BoardFilled ==
    \* There does not exist
    ~\E i \in 1..3, j \in 1..3:
        \* an empty space
        LET space == board[i][j] IN
        space = "_"

BoardEmpty ==
    \* There does not exist
    \A i \in 1..3, j \in 1..3:
        \* an empty space
        LET space == board[i][j] IN
        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.

```
WinningPositions == {
    \* Horizonal wins
    <<<<1,1>>, <<1,2>>, <<1,3>>>>,
    <<<<2,1>>, <<2,2>>, <<2,3>>>>,
    <<<<3,1>>, <<3,2>>, <<3,3>>>>,
    \* Vertical wins
    <<<<1,1>>, <<2,1>>, <<3,1>>>>,
    <<<<1,2>>, <<2,2>>, <<3,2>>>>,
    <<<<1,3>>, <<2,3>>, <<3,3>>>>,
    \* Diagonal wins
    <<<<1,1>>, <<2,2>>, <<3,3>>>>,
    <<<<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

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

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

- Qual é a pré-condição pra essa ação?

Ações - MoveO

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

- Qual é a pré-condição pra essa ação?

```
    /\ nextTurn = "O" \* Only enabled on O's turn
    /\ ~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

```
Corners == {
    <<1,1>>,
    <<3,1>>,
    <<1,3>>,
    <<3,3>>
}

StartInCorner ==
    \E corner \in Corners:
        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

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

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

Condições para as jogadas III

```
CanTakeCenter == board[2][2] = "_"

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

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

- Qual é a pré-condição pra essa ação?

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

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

Ações - TakeCenter e SetupWin

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

Ações - MoveX

```
MoveX ==
    /\ nextTurn = "X" \* Only enabled on X's turn
    /\ ~Won("O") \* And X has not won
    \* This specifies the spots X will move on X's
   turn
    /\ \/ /\ BoardEmpty
          /\ StartInCorner
       \/ /\ ~BoardEmpty \* If its not the start
          /\ \/ /\ CanWin
                /\ Win
             \/ /\ ~CanWin
                /\  \/ /\ CanBlockWin
                       /\ BlockWin
                    \/ /\ ~CanBlockWin
                       /\ \/ /\ CanTakeCenter
                             /\ TakeCenter
                          \/ /\ ~CanTakeCenter
                             /\ \/ /\ CanSetupWin
                                   /\ 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

```
GameOver == Won("X") /\ Won("O") /\ BoardFilled

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

Transições

```
GameOver == Won("X") /\ Won("O") /\ BoardFilled

\* Every state, X will move if X's turn, O will move on O's turn
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`.

```
\* Every state, X will move if X's turn, O will move on O's turn
Next == MoveX \/ MoveO

\* A description of every possible game of tic-tac-toe
\* will play until the board fills up, even if someone won
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

16 de setembro de 2024