Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

Especificação e Teste do Jogo de Tabuleiro Quantum em VDM++

Carlos Frias ei04116@fe.up.pt

27 de novembro de 2011

Índice

Lista de requisitos e identificação das principais restrições

Regras de Jogo:

O jogo Quantum é jogado num tabuleiro rectangular e são possíveis diversas variantes do jogo. A variante aqui descrita usa um tabuleiro oito por oito. Em vez que de comer as peças do inimigo como no xadrez, os jogadores constroem pilhas de peças verticais movendo as suas peças para cima das peças do adversário.

Existem três tipos de peças:

- Quadrados, que se movem ao longo das linhas e das colunas, horizontalmente e verticalmente, como as torres no xadrez;
- Círculos, que se movem na diagonal, como os bispos no xadrez;
- Rainhas, com o símbolo mais, que podem mover-se em qualquer das oito direcções.

A distância que uma peça pode mover é determinada pelo número de peças na sua pilha. Todas as peças inicialmente são pilhas com um único elemento, e podem apenas mover um único quadrado. (Numa variação do jogo, uma peça pode também saltar por cima de uma única peça da mesma cor, assim viajando dois quadrados).

Pilhas com duas, três, quatro ou cinco peças podem mover qualquer distância até dois, três, quatro e cinco, respectivamente.

A cor de uma peça, e o tipo de movimento que pode fazer é determinado pela cor e tipo da peça que está no topo da pilha. Assim uma rainha mantém-se rainha, qualquer que seja a peça que apanhar. Pilhas com seis ou mais peças não podem mover-se. Não é permitido saltar por cima de nenhuma peça (excepto como descrito na variação acima referida) nem saltar para cima de uma peça da sua própria cor. Não é permitido mover-se para cima de pilhas com seis ou mais peças.

O objectivo do jogo é ser o primeiro jogador com três pilhas com seis ou mais peças.

Especificação dos requisitos em VDM++:

Para a especificação de cada peça utilizei o seguinte tipo:

```
19      public Peca::
20              tipo: TipoPeca
21              jogador: nat1
22              num_elem: nat1
23
24      inv jog == jog.jogador in set {1,2};
```

Figura 1: Definição do tipo Peca

Onde *tipo* define o tipo de peça que pode ser uma das três seguintes quotes: <*circle*>, <*square*> e <*queen*>. O *jogador* define a cor da peça, ou seja, de que jogador é essa peça (jogador 1 ou jogador 2) e *num_elem* indica o número de "peças" que a pilha dessa peça contém. Inicialmente é igual a um.
Cada peça está colocada numa casa do tabuleiro que especifiquei com o seguinte tipo:

```
9     private num_col: map char to nat1 = {'a'|->1, 'b'|->2, 'c'|->3, 'd'|->4, 'e'|->5, 'f'|->6, 'g'|->7, 'h'|->8};
10
11  types
12      public Casa ::
13              linha: nat1
14              coluna: char
15      inv casa == casa.linha in set rng num_col and casa.coluna in set dom num_col;
16
```

Figura 2: Definição do tipo Casa

Cada *Casa* não é mais que um tuplo de dois elementos representando a linha e a coluna do tabuleiro. O map *num_col* que se vê na figura 2 é também usado mais à frente na conversão entre os números inteiros de um a oito e as letras de *a* a *h*, e vice-versa.
Fazendo a ligação entre as casas do tabuleiro e as peças utilizei um map, designado *pecasNasCasas* que é actualizado a cada jogada. O domínio desse map é o conjunto das casas ocupadas do tabuleiro a cada momento e o contra domínio é o conjunto das peças existentes no tabuleiro.

```
26      --cada casa pode ter 0 ou 1a peça
27      public PecasNasCasas = map Casa to Peca;
28      public Estado = <inicial> | <vez_jog1> | <vez_jog2> | <fim>;
29      public Direcao = <horizontal> | <vertical> | <diagonal1> | <diagonal2> | <sem_ligacao>;
30
31  instance variables
32      public pecasNasCasas: PecasNasCasas := {|->};
```

Figura 3: Definição do mapa pecasNasCasas

Na figura 3 pode também observar-se a definição dos estados possíveis que o tabuleiro de jogo pode ter a cada momento: estado *inicial*, estado *vez_jog1* que indica que se está à espera de uma jogada do jogador um, estado *vez_jog2* que é idêntico ao anterior, mas para o jogador dois e o estado *fim* que representa o estado do jogo quando um dos dois jogadores ganha o jogo.
Ainda se pode observar na figura 3 a definição das direcções possíveis para deslocar as peças nas jogadas. Na figura seguinte pode ver-se a definição das direcções permitidas para cada tipo de peça.

```
34      private mapa_direcoes: map TipoPeca to set of Direcao := {<circle> |-> {<diagonal1>, <diagonal2>},
35                                                                <square> |-> {<horizontal>, <vertical>},
36                                                                <queen> |->{<horizontal>, <vertical>, <diagonal1>, <diagonal2>}};
37      private map_jog: map Estado to nat1 := {<vez_jog1> |-> 1, <vez_jog2> |-> 2};
```

Figura 4: Definição das direcções permitidas para cada peça

Esta ligação entre tipo de peça e direcções que pode seguir é feita por um map de tipo de peça para conjunto de direcções como se pode observar na figura 4. Ainda nessa figura está a estrutura que faz a correspondência entre os estados vez do jogador e o jogador.

O tabuleiro inicial do jogo é construído no construtor da classe Tabuleiro da seguinte forma:

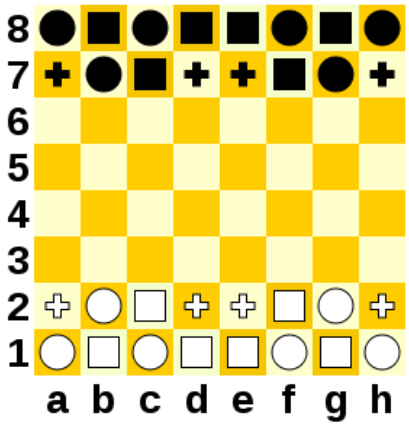

Figura 5: Tabuleiro inicial do jogo

```
40      --construtor - para inicializar o tabuleiro
41      public Tabuleiro: () ==> Tabuleiro
42      Tabuleiro() ==
43      (
44          --mapear as peças para as casas.
45          pecasNasCasas :=
46          {mk_Casa((jog-1)*7 + 1, coluna)|-> mk_Peca(<circle>, jog, 1) | coluna in set{'a','c','f','h'}, jog in set {1,2}}
47          munion
48          {mk_Casa((jog-1)*5 + 2, coluna)|-> mk_Peca(<circle>, jog, 1) | coluna in set{'b','g'}, jog in set {1,2}}
49          munion
50          {mk_Casa((jog-1)*7 + 1, coluna)|-> mk_Peca(<square>, jog, 1) | coluna in set{'b','d','e','g'}, jog in set {1,2}}
51          munion
52          {mk_Casa((jog-1)*5 + 2, coluna)|-> mk_Peca(<square>, jog, 1) | coluna in set{'c','f'}, jog in set {1,2}}
53          munion
54          {mk_Casa((jog-1)*5 + 2, coluna)|-> mk_Peca(<queen>, jog, 1) | coluna in set{'a','d','e','h'}, jog in set {1,2}};
55
56          tabEstado := <vez_jog1>;
57      );
```

Figura 6: Método construtor da classe tabuleiro

A colocação das peças nas casas no tabuleiro inicial segue a indicação da figura 5 e no método construtor é feita a alocação das peças de uma forma relativamente inteligente tentando minimizar o código e não colocando peça por peça.
A execução das jogadas é feita de forma alternada entre os dois jogadores. Nesta implementação coloquei o jogador um como primeiro a jogar, como se pode ver na última linha da figura 6.
Para executar jogadas existe a operação *jogada()* que recebe duas casas como argumentos. Esta operação analisa se a jogada é apenas a movimentação de uma peça para outra casa ou se é a movimentação de uma peça para cima de outra peça adversária.

```
65      public jogada: Casa * Casa ==> ()
66      jogada(casa1, casa2) ==
67      (
68          if casa2 in set dom pecasNasCasas then come(casa1, casa2)
69          else move_para(casa1, casa2);
70      )
71      pre pode_mover(casa1, casa2);
```

Figura7: Operação jogada()

Se existir uma peça na segunda casa então trata-se de uma movimentação de uma peça para cima de outra peça adversária, deste modo chama-se uma operação denominada *come()*, senão trata-se apenas da deslocação de uma peça de uma casa para outra e é chamada a operação *move_para()*.
A operação jogada tem uma pre-condição *pode_mover* que, basicamente, averigua se a distância entre as duas casas é menor ou igual ao número de elementos (*num_elem*) da peça que está prestes a mover-se, se o número de elementos dessa peça é inferior a seis, se a direcção entre as duas casas é

uma das direcções que a peça pode efectuar, se não existe peças entre essas duas casas e se a peça que se está a tentar movimentar pertence ao jogador actual.

```
92      --operação para verificar se a peça pode mover de casa1 para casa2
93      private pode_mover: Casa * Casa ==> bool
94      pode_mover(casa1, casa2) ==
95      (
96          return
97              distancia(casa1, casa2) <= getPeca(casa1).num_elem
98              and getPeca(casa1).num_elem <6
99              and qual_direcao(casa1, casa2) in set mapa_direcoes(getPeca(casa1).tipo)
100             and not existe_pecas(casa1, casa2) and
101             map_jog(tabEstado) = getPeca(casa1).jogador;
102     )
103     pre casa1 in set dom pecasNasCasas;
```

Figura 8: Pré condição pode_mover()

Repare-se que são chamadas várias operações para testar se a peça pode mover-se de casa1 para casa2, designadamente:

- *distancia()* – que retorna a distância (valor natural) entre as duas casas que recebe como argumento, se direcção entre as duas casas for <*sem_direcao*> então retorna 100;
- *getPeca()* – que retorna a peça que está na casa que é dada como argumento;
- *qual_direcao()* – que devolve a direcção entre das duas casas passadas como argumento, que pode ser: <*horizontal*>, <*vertical*>, <*diagonal1*>, <*diagonal2*> e <*sem_direcao*>, se não for nenhuma das anteriores;
- *existe_pecas()* – que informa se entre as duas casas existem peças no tabuleiro.

A operação *come()* que recebe como argumentos duas casas do tabuleiro é usada para a movimentação de uma peça para cima de uma peça do adversário.

```
82  --operação para comer a peça de um adversário
83  private come: Casa * Casa ==> ()
84  come(casa1, casa2) ==
85  (
86      pecasNasCasas := {casa1} <-: pecasNasCasas ++
87                       {casa2|-> mk_Peca(getPeca(casa1).tipo, getPeca(casa1).jogador, getPeca(casa1).num_elem + getPeca(casa2).num_elem)};
88      muda_jogador();
89  )
90  pre pode_comer(casa1, casa2);
```

Figura 9: Operação come()

Basicamente o que se faz é um override do map *pecasNasCasas* que contém a relação entre as casas e as peças com a relação nova casa, nova peça e retirar a relação casa anterior, peça anterior. Após esta execução é feita a mudança de jogador pela operação *muda_jogador()*.
Esta operação têm a pré condição *pode_comer()* que é uma operação que retorna *true* se existe uma peça na segunda casa, se pode mover (*pode_mover*) entre as duas casas, se o dono da peça na segunda casa não é o jogador actual, isto é, se se trata de uma peça adversária e se o número de elementos da segunda peça é inferior a seis.

```
104     private pode_comer: Casa * Casa ==> bool
105     pode_comer(casa1, casa2) ==
106     (
107         return
108             casa2 in set dom pecasNasCasas and
109             pode_mover(casa1, casa2) and
110             map_jog(tabEstado) <> getPeca(casa2).jogador and
111             getPeca(casa2).num_elem < 6;
112     );
```

Figura 10: Pré condição pode_comer()

A operação *mover_para()* :

```
73      --operação para mover a peça P para a casa C
74      private move_para: Casa * Casa ==> ()
75      move_para(casa1, casa2) ==
76      (
77          if casa2 not in set dom pecasNasCasas then pecasNasCasas := {casa1} <-: pecasNasCasas ++ {casa2 |-> getPeca(casa1)};
78          muda_jogador();
79      )
80      pre pode_mover(casa1, casa2);
```

Figura 11: Operação mover_para

É utilizada apenas para deslocar uma peça para uma casa vazia e fá-lo com um override do map *pecasNasCasas* com a nova relação peça, casa nova e elimina a relação peça casa anterior. No final é executada a mudança de jogador. Esta operação tem também a pré condição *pode_mover()*.

Para determinar o fim do jogo é utilizada a operação fim_jogo() .

```
58      --operação que verifica o fim do jogo
59      public fim_jogo: ()==> bool
60      fim_jogo() ==
61          return (card (dom (pecasNasCasas :> {mk_Peca(tipo,1, n)| tipo in set {<circle> , <square> , <queen>}, n in set {6,...,10}})) >= 3)
62          or
63          (card (dom (pecasNasCasas :> {mk_Peca(tipo, 2, n)| tipo in set {<circle> , <square> , <queen>}, n in set {6,...,10}}))) >= 3);
```

Figura 12: Operação fim_jogo().

Esta operação procura para cada jogador as casas com peças cujo número de elementos é um valor entre seis a dez (uma vez que não pode ser superior) e se esse número de casas for superior ou igual a três é retornado true (fim do jogo).
Esta operação é chamada na operação *muda_jogador()* que por sua vez é executada sempre que se move uma peça ou se come uma peça.

Classe Tabuleiro

Definição em VDM++ da classe Tabuleiro, conforme slides da aula teórica (ficheiro VDM1.ppt).

```
--                Projecto Quantum - Métodos Formais de Engenharia de Software
--                            Carlos Eduardo Mesquita Frias - ei04116
class Tabuleiro

values
      --mapeamento para o número da coluna
      private num_col: map char to nat1 = {'a'|->1, 'b'|->2, 'c'|->3, 'd'|->4,
'e'|->5, 'f'|->6, 'g'|->7, 'h'|->8};

types
      public Casa ::
                  linha: nat1
                  coluna: char
      inv casa == casa.linha in set rng num_col and casa.coluna in set dom
num_col;

      public TipoPeca = <circle> | <square> | <queen>;

      public Peca::
                  tipo: TipoPeca
                  jogador: nat1
                  num_elem: nat1

      inv jog == jog.jogador in set {1,2};
```

```
      --cada casa pode ter 0 ou 1a peça
      public PecasNasCasas = map Casa to Peca;
      public Estado = <inicial> | <vez_jog1> | <vez_jog2> | <fim>;
      public Direcao = <horizontal> | <vertical> | <diagonal1> | <diagonal2> |
<sem_ligacao>;

instance variables
      public pecasNasCasas: PecasNasCasas := {|->};
      public tabEstado: Estado := <inicial>;
      private mapa_direcoes: map TipoPeca to set of Direcao := {<circle> |->
{<diagonal1>, <diagonal2>},

            <square> |-> {<horizontal>, <vertical>},

            <queen> |->{<horizontal>, <vertical>, <diagonal1>, <diagonal2>}};
      private map_jog: map Estado to nat1 := {<vez_jog1> |-> 1, <vez_jog2> |->
2};

operations
      --construtor – para inicializar o tabuleiro
      public Tabuleiro: () ==> Tabuleiro
      Tabuleiro() ==
      (
            --mapear as peças para as casas.
            pecasNasCasas :=
            {mk_Casa((jog-1)*7 + 1, coluna)|-> mk_Peca(<circle>, jog, 1) |
coluna in set{'a','c','f','h'}, jog in set {1,2}}
            munion
            {mk_Casa((jog-1)*5 + 2, coluna)|-> mk_Peca(<circle>, jog, 1) |
coluna in set{'b','g'}, jog in set {1,2}}
            munion
            {mk_Casa((jog-1)*7 + 1, coluna)|-> mk_Peca(<square>, jog, 1) |
coluna in set{'b','d','e','g'}, jog in set {1,2}}
            munion
            {mk Casa((jog-1)*5 + 2, coluna)|-> mk Peca(<square>, jog, 1) |
coluna in set{'c','f'}, jog in set {1,2}}
            munion
            {mk_Casa((jog-1)*5 + 2, coluna)|-> mk_Peca(<queen>, jog, 1) | coluna
in set{'a','d','e','h'}, jog in set {1,2}};

            tabEstado := <vez_jog1>;
      );
      --operação que verifica o fim do jogo
      public fim_jogo: ()==> bool
      fim_jogo() ==
            return (card (dom (pecasNasCasas :> {mk_Peca(tipo,1, n)| tipo in set
{<circle> , <square> , <queen>}, n in set {6,...,10}})) >= 3)
            or
            (card (dom (pecasNasCasas :> {mk_Peca(tipo, 2, n)| tipo in set
{<circle> , <square> , <queen>}, n in set {6,...,10}})) >= 3);

      public jogada: Casa * Casa ==> ()
      jogada(casa1, casa2) ==
      (
            if casa2 in set dom pecasNasCasas then come(casa1, casa2)
            else move_para(casa1, casa2);
      )
      pre pode_mover(casa1, casa2);
```

```
      --ver isto melhor(porque está a repetir)

      --operação para mover a peça P para a casa C
      private move_para: Casa * Casa ==> ()
      move_para(casa1, casa2) ==
      (
            if casa2 not in set dom pecasNasCasas then pecasNasCasas := {casa1}
<-: pecasNasCasas ++ {casa2 |-> getPeca(casa1)};
            muda_jogador();
      )
      pre pode_mover(casa1, casa2);

      --operação para comer a peça de um adversário
      private come: Casa * Casa ==> ()
      come(casa1, casa2) ==
      (
            pecasNasCasas := {casa1} <-: pecasNasCasas ++
                                          {casa2|-> mk_Peca(getPeca(casa1).tipo,
getPeca(casa1).jogador, getPeca(casa1).num_elem + getPeca(casa2).num_elem)};
            muda_jogador();
      )
      pre pode_comer(casa1, casa2);

      --operação para verificar se a peça pode mover de casa1 para casa2
      private pode_mover: Casa * Casa ==> bool
      pode_mover(casa1, casa2) ==
      (
            return
                  distancia(casa1, casa2) <= getPeca(casa1).num_elem
                  and getPeca(casa1).num_elem <6
                  and qual direcao(casa1, casa2) in set
mapa_direcoes(getPeca(casa1).tipo)
                  and not existe_pecas(casa1, casa2) and
                  map_jog(tabEstado) = getPeca(casa1).jogador;
      )
      pre casa1 in set dom pecasNasCasas;
      private pode_comer: Casa * Casa ==> bool
      pode_comer(casa1, casa2) ==
      (
            return
                  casa2 in set dom pecasNasCasas and
                  pode_mover(casa1, casa2) and
                  map_jog(tabEstado) <> getPeca(casa2).jogador and
                  getPeca(casa2).num_elem < 6;
      );

      --operação para calcular a distância entre duas casas
      private distancia: Casa * Casa ==> nat
      distancia(casa1, casa2)==
      (
            if casa1.linha = casa2.linha then return abs(num_col(casa1.coluna) -
num_col(casa2.coluna))
            elseif casa1.coluna = casa2.coluna then return abs(casa1.linha -
casa2.linha)
            elseif abs(casa1.linha - casa2.linha) = abs(num_col(casa1.coluna) -
num_col(casa2.coluna)) then return abs(casa1.linha - casa2.linha)
            else return 100;--apenas por agora...
      );
```

```
      --operação para determinar qual a direcao no movimento
      private qual_direcao: Casa * Casa ==> Direcao
      qual_direcao(casa1, casa2) ==
      (
            if casa1.linha = casa2.linha then return <horizontal>
            elseif casa1.coluna = casa2.coluna then return <vertical>
            elseif casa1.linha - casa2.linha = num_col(casa1.coluna) -
num_col(casa2.coluna) then return <diagonal1>
            elseif casa1.linha - casa2.linha = -(num_col(casa1.coluna) -
num_col(casa2.coluna)) then return <diagonal2>
            else return <sem_ligacao>
      );
      --operação que averigua se existem peças entre duas casas
      private existe_pecas: Casa * Casa ==> bool
      existe_pecas(casa1, casa2) ==
      (
            if casas_entre(casa1, casa2) inter dom pecasNasCasas = {} then
return false
            else return true
      );
      --operação que devolve as casas entre duas peças dadas
      private casas_entre: Casa * Casa ==> set of Casa
      casas_entre(casa1, casa2) ==
      (
            --se a direcao entre a casa1 e casa2 for vertical verifica se no
mapa de casas para peças existe algum mapeamento que ligue uma peça a uma
            --casa que esteje na mesma coluna entre as casas 1 e 2
            if qual_direcao(casa1, casa2) = <vertical> then return
                  {mk_Casa(i, casa1.coluna)| i in set{min(casa1.linha,
casa2.linha) + 1,..., max(casa1.linha, casa2.linha) - 1}}
            --se a direcao entre a casa1 e casa2 for horizontal verifica se no
mapa de casas para peças existe algum mapeamento que ligue uma peça a uma
            --casa que esteje na mesma linha entre as casas 1 e 2
            elseif qual_direcao(casa1, casa2) = <horizontal> then return
                  {mk_Casa(casa1.linha, (inverse num_col)(i))| i in set
{min(num col(casa1.coluna), num col(casa2.coluna)) + 1,...

      , max(num_col(casa1.coluna), num_col(casa2.coluna)) - 1}}
            --se a direcao entre a casa1 e casa2 for diagonal verifica se no
mapa de casas para peças existe algum mapeamento que ligue uma peça a uma
            --casa que esteje na mesma linha diagonal entre as casas 1 e 2
            elseif qual_direcao(casa1, casa2) = <diagonal1> then return
                  {mk_Casa(maiorCasa(casa1, casa2).linha - i, (inverse
num_col)(num_col(maiorCasa(casa1, casa2).coluna) + i)) |
                              i in set {1 ,..., max(casa1.linha, casa2.linha) -
min(casa1.linha, casa2.linha) - 1}}
            elseif qual_direcao(casa1, casa2) = <diagonal2> then return
                  {mk_Casa(maiorCasa(casa1, casa2).linha - i, (inverse
num_col)(num_col(maiorCasa(casa1, casa2).coluna) + i)) |
                              i in set {1 ,..., max(casa1.linha, casa2.linha) -
min(casa1.linha, casa2.linha) - 1}}
            else return {}
      );
      --operação que devolve a casa com a maior linha de entre duas casas que
recebe como argumento
      private maiorCasa: Casa * Casa ==> Casa
      maiorCasa(casa1, casa2) ==
      (
            if casa1.linha > casa2.linha then return casa1
```

```
            else return casa2
      );
      --operação para mudar de jogador
      private muda_jogador: () ==> ()
      muda_jogador() ==
      (
            if not fim_jogo() then
            (
                  if tabEstado = <vez_jog1> then tabEstado := <vez_jog2>
                  else tabEstado := <vez_jog1>;
            )
            else tabEstado := <fim>;
      );
      --operação para retornar o mínimo entre dois números naturais
      private min: nat * nat ==> nat
      min(n1, n2) ==
      (
            if n1 > n2 then return n2
            else return n1
      );
      --operação para retornar o máximo entre dois números naturais
      private max: nat * nat ==> nat
      max(n1, n2) ==
      (
            if n1 > n2 then return n1
            else return n2
      );

      --operação para retornar a peca que está numa casa específica
      private getPeca: Casa ==> Peca
      getPeca(casa) ==
            return pecasNasCasas(casa)
      pre casa in set dom pecasNasCasas;

      --Vamos executar uma sucessão de jogadas para garantir que todas as
funções estão a executar correctamente..
      public teste: () ==> ()
      teste () ==
      (
            --sequencia de quinze jogadas consecutivas em que são comidas 3
pecas e fica uma torre queen na casa D5 com quatro peças
            jogada(mk_Casa(2, 'a'), mk_Casa(3, 'b')); --rainha branca de A2 para
B3
            jogada(mk_Casa(7, 'b'), mk_Casa(6, 'c')); --círculo preto de B7 para
C6
            jogada(mk_Casa(2, 'c'), mk_Casa(3, 'c')); --quadrado branco de C2
para C3
            jogada(mk_Casa(6, 'c'), mk_Casa(5, 'b')); --círculo preto de C6 para
B5
            jogada(mk_Casa(3, 'b'), mk_Casa(4, 'b')); --rainha branca de B3 para
B4
            jogada(mk_Casa(7, 'f'), mk_Casa(6, 'f')); --quadrado preto de F7
para F6
            jogada(mk_Casa(4, 'b'), mk_Casa(5, 'b')); --rainha branca de B4 para
B5 (come círculo preto) - torre de altura 2
            jogada(mk_Casa(7, 'h'), mk_Casa(6, 'h')); --rainha preta de H7 para
H6
            jogada(mk_Casa(3, 'c'), mk_Casa(4, 'c')); --quadrado branco de C3
para C4
```

```
            jogada(mk_Casa(6, 'f'), mk_Casa(6, 'e')); --quadrado preto de F6
para E6
            jogada(mk_Casa(4, 'c'), mk_Casa(5, 'c')); --quadrado branco de C4
para C5
            jogada(mk_Casa(6, 'e'), mk_Casa(5, 'e')); --quadrado preto de E6
para E5
            jogada(mk_Casa(5, 'c'), mk_Casa(5, 'd')); --quadrado branco de C5
para D5
            jogada(mk_Casa(5, 'e'), mk_Casa(5, 'd')); --quadrado preto de E5
para D5 (come quadrado branco) torre de altura 2
            jogada(mk_Casa(5, 'b'), mk_Casa(5, 'd')); --rainha branca de B5 para
D5 (come torre quadrado preto) torre de altura 4
      );
      public teste_FimJogo: () ==> Estado
      teste_FimJogo() ==
      (
            pecasNasCasas := {mk_Casa(1,'a') |-> mk_Peca(<queen>, 1, 6),
                              mk_Casa(1,'b') |-> mk_Peca(<queen>, 1, 6),
                              mk_Casa(1,'c') |-> mk_Peca(<queen>, 1, 5),
                              mk_Casa(6,'c') |-> mk_Peca(<queen>, 2,5)};
            jogada(mk_Casa(1,'c'), mk_Casa(6,'c'));
            return tabEstado;
      );
      --public debug: () ==> set of Casa
      --debug() ==
      --(
            --pecasNasCasas := {mk_Casa(1,'h') |-> mk_Peca(<queen>, 1, 5),
            --                  mk_Casa(8,'a') |-> mk_Peca(<square>, 2, 1)};
            --jogada(mk_Casa(2,'e'), mk_Casa(4,'c'));
            --return
            --    {mk Casa(maiorCasa(mk Casa(8,'a'), mk Casa(1,'h')).linha - i,
(inverse num_col)(num_col(maiorCasa(mk_Casa(8,'a'), mk_Casa(1,'h')).coluna) +
i)) |
            --                i in set {1 ,..., max(mk_Casa(8,'a').linha,
mk_Casa(1,'h').linha) - min(mk_Casa(8,'a').linha, mk_Casa(1,'h').linha) - 1}}
      --);
end Tabuleiro
```

Cobertura de testes

```
Tabuleiro
```

Aqui deveria aparecer a matriz de rastreabilidade dos testes realizados, no entanto, por qualquer razão que desconheço não consegui fazer com que nos ficheiros criados pelo pritty print do VDM++ tools de extensão .rtf.rtf contivessem essa tabela, ao invés contém uma série de caracteres sem significado aparente, como o professor terá oportunidade de observar nos ficheiros que seguem anexos a este relatório.

Classe TesteTabuleiro

Definição em VDM++ da classe TesteTabuleiro, para testar a classe Tabuleiro.

```
class TesteTabuleiro
      instance variables
            public tab: Tabuleiro;

      operations
            public Teste1: () ==> ()
            Teste1()==
            (
                  tab := new Tabuleiro();
                  tab.teste();
            );
            public Teste2: () ==> Tabuleiro`Estado
            Teste2() ==
            (
                  tab:= new Tabuleiro();
                  return tab.teste_FimJogo();
            );
end TesteTabuleiro
```

Cobertura de testes

```
TesteTabuleiro
```

Diagrama conceptual de classes do sistema

No código VDM++ implementado apenas foram implementadas duas classes, pois achei que o problema ficava perfeitamente resolvido definindo tipos na classe tabuleiro, nomeadamente, o tipo Peca, o tipo Casa e o tipo PecasNasCasas. Assim, sendo o conceito do sistema é algo do género:

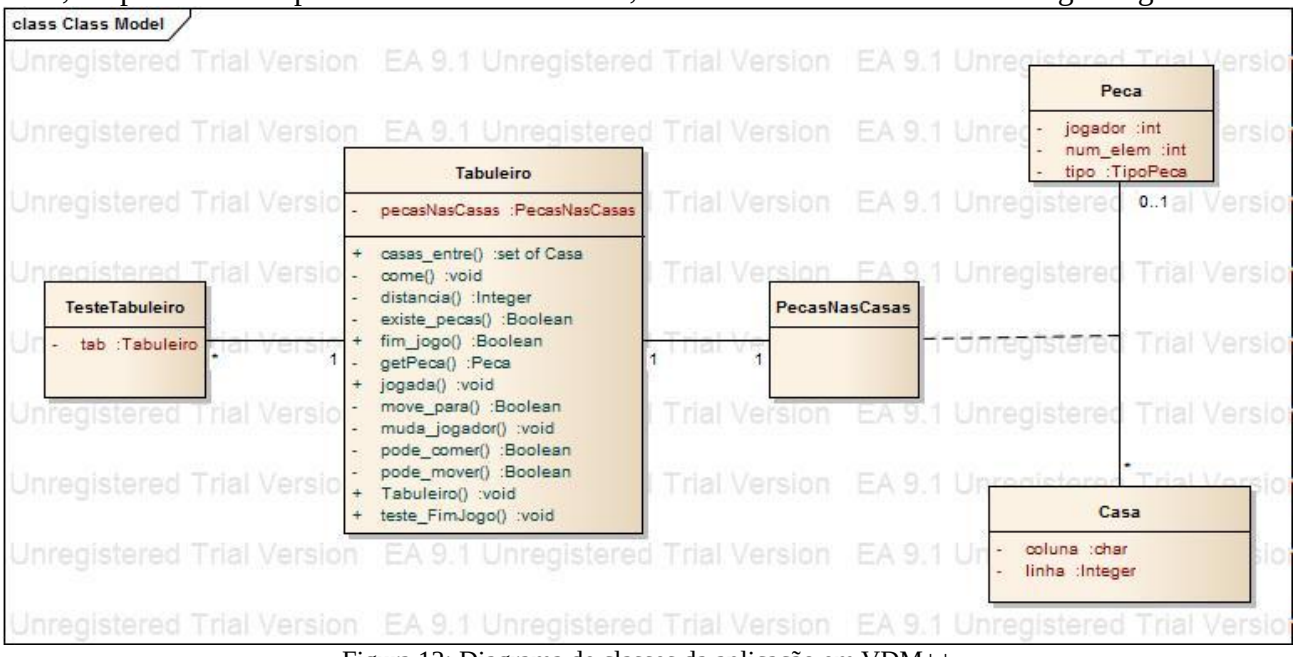

Figura 12: Diagrama de classes da aplicação em VDM++

Para ilustrar os diversos estados por que o jogo atravessa pode ver-se o seguinte diagrama de estados:

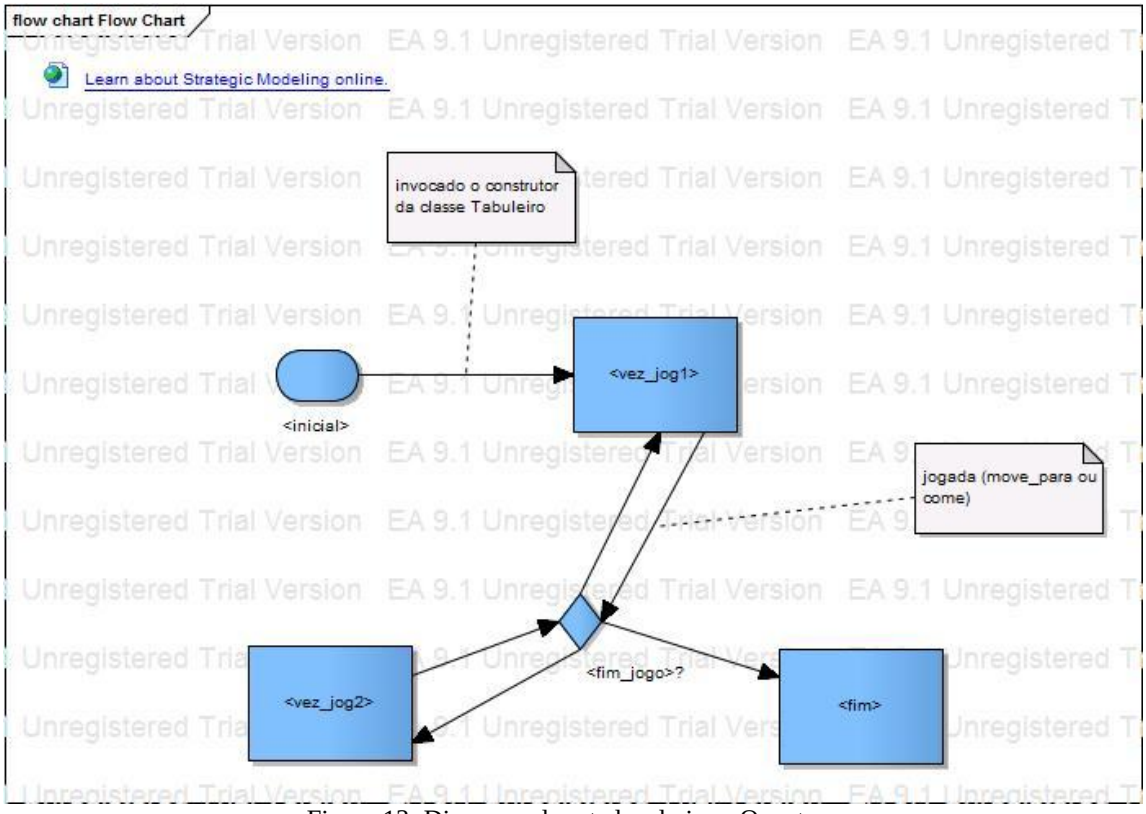

Figura 13: Diagrama de estados do jogo Quantum

Ficheiros de Teste

Teste1.arg

```
new TesteTabuleiro().Teste1()
```

Teste1.arg.exp

```
(no return value)
```

Teste2.arg

```
new TesteTabuleiro().Teste2()
```

Teste2.arg.exp

```
<fim>
```

vdmloop.bat

```
@echo off
rem Runs a collection of VDM++ testExamples
rem Assumes specification in Word RTF files

set T1 = Tabuleiro.rtf
set T2 = TesteTabuleiro.rtf

"C:\Program Files (x86)\The VDM++ Toolbox Academic v8.0\bin\vppde" -p -R
vdm.tc %T1% %T2%
```

```
for /R %%f in (*.arg) do call vdmtest "%%f"
```

vdmtest.bat

```
@echo off
rem Tests the date book specification for one test case (argument)
rem -- Output the argument to stdout (for redirect) and "con" (for user
feedback)
echo VDM Test: '%1'>con
echo VDM Test: '%1'

rem short names for specification files in Word RTF Format
set T1=Tabuleiro.rtf
set T2=TesteTabuleiro.rtf

rem -- Calls the interpreter for this test case
"C:\Program Files (x86)\The VDM++ Toolbox Academic v8.0\bin\vppde" -i -D -I
-P -Q -R vdm.tc -O %1.res %1 %T1% %T2%

rem -- Check for difference between result of execution and expected result.
if EXIST %1.exp fc /w %1.res %1.exp

:end
```

Análise da consistência do modelo

Ao longo da implementação do código VDM++, na fase de especificação, tive sempre a preocupação de testar cada método desenvolvido até ter a certeza que está livre de bugs, fi-lo criando métodos de teste e analisando os resultados no interpretador do VDM Tools, só depois é que arrancava para a elaboração do método seguinte. Deste modo, a fase de especificação foi lenta, mas os resultados obtidos foram consistentes.
Finda a fase de especificação, seguiu-se a fase de gerar o código java e implementar uma interface simples para verificação do jogo e das jogadas. Após esta fase, e mesmo com a exaustiva bateria de testes que fui fazendo ao longo da fase de especificação, ainda foram detectados dois bugs, nomeadamente na determinação do fim do jogo e na deslocação de peças numa das diagonais. Para resolver estes problemas voltei ao código e com pequenos reajustes eliminei todos os bugs detectados, o código java foi gerado novamente. Após esta rectificação não detectei mais irregularidades e executei o jogo do início ao fim diversas vezes.
Deixei para final a construção da classe TesteTabuleiro que basicamente tem dois testes a aplicar ao jogo, o teste1 é uma sequência de quinze jogadas consecutivas e o teste2 é realizado para num dado momento do tabuleiro para testar o final do jogo. Correndo no interpretador esses testes são validados, assim como, quando são executados na consola os scripts de teste atrás descritos. Infelizmente, pelas razões que já indiquei neste relatório não consegui obter as matrizes de restreabilidade.

Implementação do jogo em java a partir do código gerado pelo VDM Tools

Após a fase de especificação e solucionado todos os bugs, foi gerado o código java do projecto. Esse consistiu em diversos ficheiros com extensão.java: o ficheiro Tabuleiro.java e uma série de ficheiros para definir as quotes que ficaram dentro da pasta "quotes".
Para além desses criei um outro com o nome JanelaQuantum, subclasse de JFrame para visualizar o tabuleiro e para melhorar a jogabilidade. Apesar disso, pela escassez de tempo a visualização da movimentação das peças tem de ser feita em modo de texto no output do IDE utilizado (no caso o Netbeans). Assim pode-se ver um printscrean do jogo a correr:

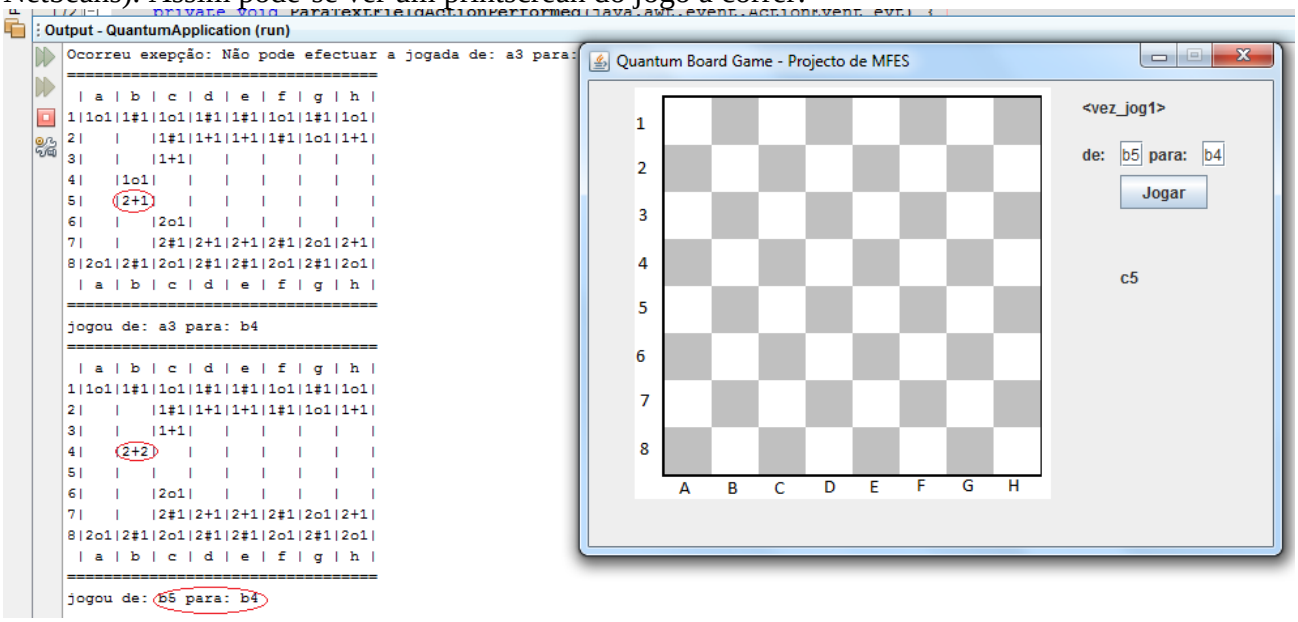

Figura 14: Interface gráfica e execução em java do jogo

A interface gráfica utiliza a posição e clique do rato no tabuleiro ou também permite introduzir as jogadas nos campos de texto "de" e "para".
O código java gerado, bem como as classes, criadas para a construção da interface seguem anexas ao relatório.