FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

INTEGRATED MASTER IN INFORMATICS AND COMPUTING ENGINEERING

FORMAL METHODS IN SOFTWARE ENGINEERING

Practical work on VDM++ Stratego

Mariana Oliveira up201207835
Pedro Silva up201306032
Pedro Castro up201305337

January 8, 2017

Contents

1 Informal system description and list of requirements

1.1 Informal system description

Our goal with this project is to design a formal model of the Stratego board game in VDM++.

Stratego is a strategy board game for two players on a board of 10×10 squares. Each player controls 40 pieces representing individual soldier ranks in an army. The objective of the game is to find and capture the opponent's Flag, or to capture so many enemy pieces that the opponent cannot make any further moves.

Typically, one player uses red pieces, and the other uses blue pieces. The ranks are printed on one side only and placed so that the players cannot identify the opponent's pieces. Each player moves one piece per turn. Most pieces can move 1 square either forward or backward, left or right but not diagonally. Some pieces cannot move at all such as the Bomb and the Flag in the classic version, or move multiple squares like the Scout in the classic version. Some variants have additional moving rules associated with specific pieces.

When the player wants to attack, they move their piece onto a square occupied by an opposing piece. Both players then reveal their piece's rank; the weaker piece (there are exceptions; see below) is removed from the board. If the engaging pieces are of equal rank, both are removed. A piece may not move onto a square already occupied unless it attacks.

Two zones in the middle of the board, each 2×2, cannot be entered by either player's pieces at any time. They are shown as lakes on the battlefield and serve as choke points to make frontal assaults less direct.

1.2 List of Requirements

Table 1: Features

Name	**Priority**	**Description**
See Board	Mandatory	Player can see the current state of the game board.
Place Pieces	Mandatory	Player is able to place Pieces on the game board, in the respective area.
Start Game	Mandatory	The game starts when all the pieces are placed on the board game.
Move Pieces	Mandatory	Player is able to move pieces already in the game board.
Attack Pieces	Mandatory	Player is able to use his own pieces to attack the opponent's pieces.
Similar Moves	Mandatory	Player is blocked from making the same movement 3 times in a row.
Exit Game	Mandatory	The player is able to leave the game at any time.
Game Over	Mandatory	The game will end when a flag is captured.

2 Visual UML Model

2.1 Model Class Diagram

The Class Diagram UML of our model is the following:

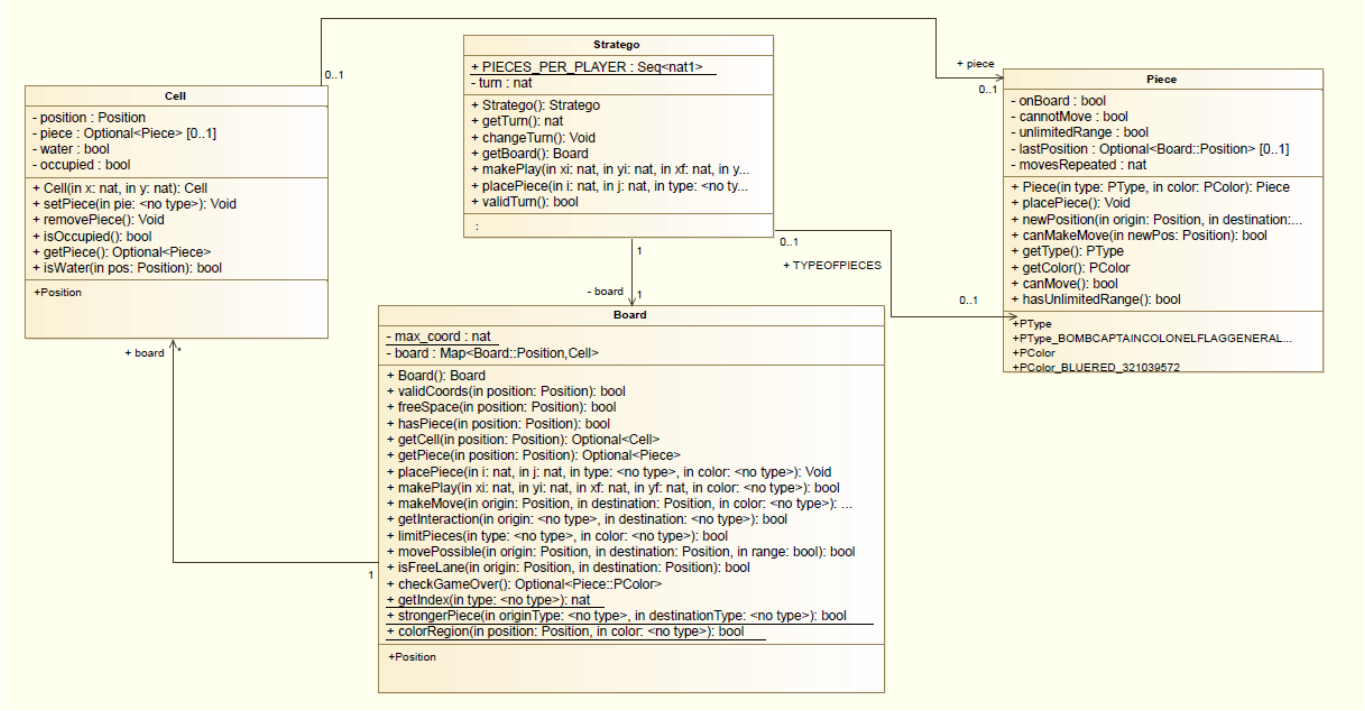


Table 2: Model UML Description

Class	Description
Stratego	Main Class contains the board of the game and is responsible for controlling the turns of playing.
Board	Contains all the information of the game board. It is also responsible for every operation that is made on the board.
Cell	Defines each block of the game board. It holds information about its characteristics and its current state of occupation.
Piece	Defines a Piece. Holds information about what type it is, how it should behave and to which player it belongs.

2.2 Use Cases

There are two main use cases in our project:

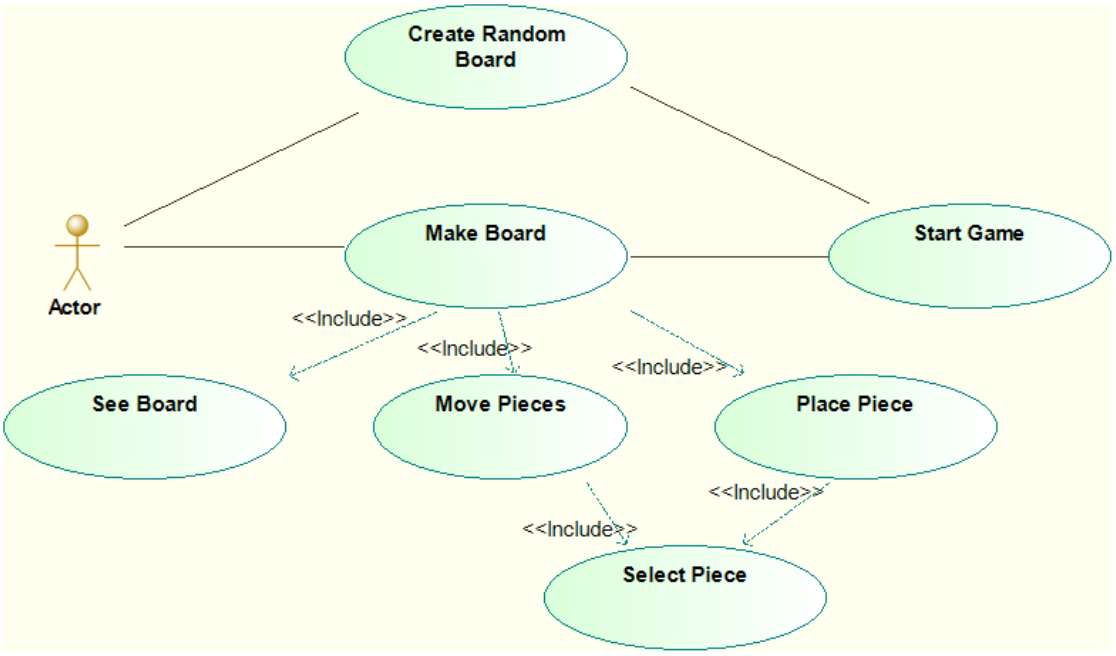


Figure 1: Make Board Use Case

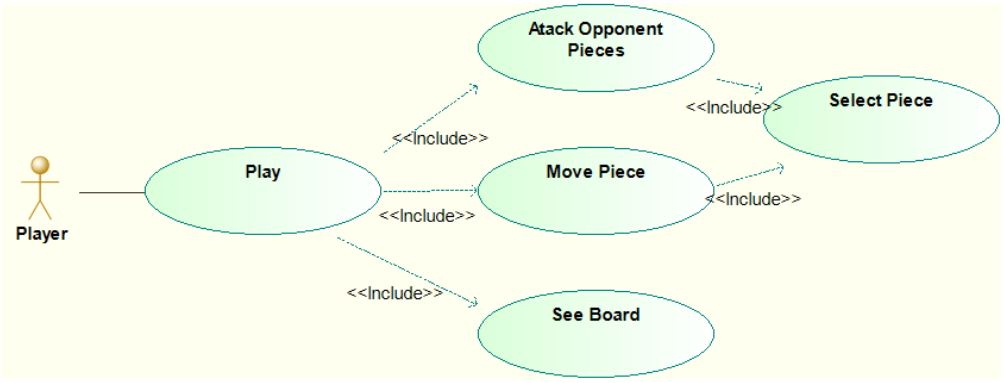


Figure 2: Play Game Use Case

The Use Cases are Described here :

Table 3: Use Case Make Board

Scenario	**Make Board**
Description	Allows the player to place his pieces on the game board, as well as moving them around. A player is only allowed to place his pieces and he must do it in his respective region.
Pre-conditions	N/A
Post-conditions	1. All the pieces must be placed. 2. The pieces must be placed in the respective players' region.
Steps	1. Place every piece in the desired place
Exceptions	N/A

Table 4: Use Case Play Game

Scenario	**Play Game**
Description	The player plays the game by moving his pieces. He can also attack the opponent pieces by moving his pieces towards the opponent's pieces.
Pre-conditions	1. Board placing complete 2. Game is started.
Post-conditions	1. Game is over. 2. One flag has been destroyed or no more possible moves left to make. 3. One of the players is victorious.
Steps	N/A
Exceptions	N/A

3 Formal VDM++ model

3.1 Class Board

```
class Board

-- Position is a pair of coordenates
types
  --Type of Position used for Coordinates
  public Position:: x:nat y:nat;
values
  --Max Size of map Square
  private max_coord : nat = 9;

instance variables
 --Board of the game, maps Position to a Cell
 private board : map Position to Cell := {|->};
operations

  --Constructer for main board
```

```
 public Board : () ==> Board
  Board() ==
  (
   for i = 0 to max_coord by 1 do
   (
    for j = 0 to max_coord by 1 do
    (
     board := board ++ {mk_Position(i,j)|-> new Cell(i,j)};
    )
   )
 );


--Valid coordinates verifier
 public pure validCoords : Position ==> bool
 validCoords(position) ==
 (
  return position.x <= max_coord and position.y <= max_coord;
 );


 public pure freeSpace : Position ==> bool
 freeSpace(position) ==
 (
  return not getCell(position).isOccupied();
 )
 pre validCoords(position);


 --Checks if Position has Piece

 public pure hasPiece : Position ==> bool

 hasPiece(position) ==
 (
 return getPiece(position) <> nil

 )
pre validCoords(position);

 --Returns Cell of Position
 public pure getCell : Position ==> [Cell]
 getCell(position) ==
 (

 return board(position)
 )
 pre validCoords(position);

 --Returns Piece from a Position
 public pure getPiece : Position ==> [Piece]
 getPiece(position) ==
 (
 return getCell(position).getPiece()
 )
 pre validCoords(position);

 --Places a piece in the given coordinates
 public placePiece : nat*nat*Piece`PType*Piece`PColor ==> ()
 placePiece(i,j,type,color) ==
 (
```

```
 dcl piece : Piece := new Piece(type,color);
 board(mk_Position(i,j)).setPiece(piece);
)
pre freeSpace(mk_Position(i,j)) and colorRegion(mk_Position(i,j),color) and limitPieces
    (type,color)
post hasPiece(mk_Position(i,j));

public makePlay : nat*nat*nat*nat*Piece`PColor ==> bool
makePlay(xi,yi,xf,yf,color) ==
(
 return makeMove(mk_Position(xi,yi),mk_Position(xf,yf),color)
);

public makeMove : Position*Position*Piece`PColor ==> bool
makeMove(origin,destination,color) ==
(
 dcl pieceOrigin : [Piece] := getPiece(origin);
 dcl pieceDestination : [Piece] := getPiece(destination);


 if movePossible(origin,destination,pieceOrigin.hasUnlimitedRange())
 then (
  if pieceOrigin.canMakeMove(destination)
   then (
    if pieceDestination = nil --move Piece

    then (
      board(origin).removePiece();
      board(destination).setPiece(pieceOrigin);
      pieceOrigin.newPosition(origin,destination);
      return true;
     )
     else (
      if pieceOrigin.getColor() = pieceDestination.getColor()
       then return false
      else (
       if pieceOrigin.getType() = pieceDestination.getType()

        then (

         board(origin).removePiece();
         board(destination).removePiece();
         return true;
        )
        else (
         if getInteraction(pieceOrigin,pieceDestination)
         then (
          board(origin).removePiece();
          board(destination).removePiece();
          board(destination).setPiece(pieceOrigin);
          pieceOrigin.newPosition(origin,destination);
          return true;

         )
         else (
          board(origin).removePiece();
          return true;
         )
        )
      )
    )
```

```
   )
    else return false;
  )
  else return false;
 )pre validCoords(origin) and validCoords(destination) and hasPiece(origin) and getPiece
     (origin).getColor() = color and getPiece(origin).canMove();

--true if it eats the second piece, false if the second piece its stronger
 public pure getInteraction : Piece * Piece ==> bool
getInteraction(origin,destination) ==

(
 dcl originType : Piece`PType := origin.getType();
 dcl destinationType : Piece`PType := destination.getType();
 cases originType :
 <MINER> -> return destinationType = <BOMB>,
 <SPY> -> return destinationType = <MARSHALL>,

 others -> strongerPiece(originType,destinationType)
 end
);

--Limits number of Pieces of each type, by player

public pure limitPieces : Piece`PType*Piece`PColor ==> bool
limitPieces(type,color) ==
(
 dcl pieces : nat := 0;

 for all cell in set rng board do
  (if cell.getPiece() <> nil
   then
    (if (cell.getPiece().getType() = type and cell.getPiece().getColor() = color)
     then pieces := pieces + 1));

 return pieces < Stratego`PIECES_PER_PLAYER(getIndex(type));
) pre type in set elems Stratego`TYPEOFPIECES;

--Checks if a move is possible to be done
public pure movePossible : Position*Position*bool ==> bool
movePossible(origin,destination,range) ==
(
 if range
  then return isFreeLane(origin,destination)
  else if origin.x = destination.x

   then return abs(origin.y - destination.y) = 1
   else return abs(origin.x - destination.x) = 1;
)
pre validCoords(origin) and validCoords(destination) and origin.x = destination.x or
     origin.y =destination.y;

--Checks if there is a straight line free between 2 spaces
public pure isFreeLane : Position*Position ==> bool
isFreeLane(origin,destination) ==
(
 if origin.x = destination.x
 then
  for i = origin.y to destination.y by (if origin.y >= destination.y then -1 else 1) do
   (if (i <> destination.y and i <> origin.y) then
    if getCell(mk_Position(destination.x,i)).isOccupied()
    then return false;)
```

```
  else if origin.y = destination.y then
   for i = origin.x to destination.x by (if origin.x >= destination.x then -1 else 1) do
    (if (i <> destination.x and i <> origin.x) then
     if getCell(mk_Position(i,destination.y)).isOccupied()
      then return false;)
  else return false;

  return true;
)pre validCoords(origin) and validCoords(destination) and not getCell(destination).
    getWater();

--Check if Game is over and returns the winner
public pure checkGameOver : () ==> [Piece`PColor]
checkGameOver() ==
(
 dcl flagBlue : [Piece] := nil;
 dcl flagRed : [Piece] := nil;
 for all cell in set rng board do
  if cell.getPiece() <> nil
   then
    if cell.getPiece().getType() = <FLAG>
     then
      if cell.getPiece().getColor() = <BLUE>
       then flagBlue := cell.getPiece()
      else
       if cell.getPiece().getColor() = <RED>
        then flagRed := cell.getPiece();

 if flagBlue <> nil and flagRed <> nil then return nil;
 if flagBlue = nil and flagRed = nil then return nil;
 if flagBlue = nil then return flagRed.getColor() else return flagBlue.getColor();
);

functions

 --get index of type of piece
 public getIndex : Piece`PType -> nat
 getIndex(type) ==
 (
  [i | i in set inds Stratego`TYPEOFPIECES & Stratego`TYPEOFPIECES(i) = type](1)
 ) pre type in set elems Stratego`TYPEOFPIECES;

 --Checks if a piece is stronger then onether
 public strongerPiece : Piece`PType*Piece`PType -> bool
 strongerPiece(originType,destinationType) ==
 (
   getIndex(originType) < getIndex(destinationType)
 ) pre originType in set elems Stratego`TYPEOFPIECES and destinationType in set elems
     Stratego`TYPEOFPIECES;

 public colorRegion : Position*Piece`PColor -> bool
 colorRegion(position, color) ==
  (
  if color = <BLUE> then position.y < 4 else position.y > 5
  );
end Board
```

Function or operation	Line	Coverage	Calls
Board	17	100.0%	2
checkGameOver	168	100.0%	4
colorRegion	147	100.0%	23
freeSpace	35	100.0%	29
getCell	48	100.0%	218
getIndex	136	100.0%	54
getInteraction	95	100.0%	4
getPiece	44	100.0%	157
hasPiece	42	100.0%	53
isFreeLane	119	100.0%	13
limitPieces	106	100.0%	23
makeMove	55	100.0%	25
makePlay	71	100.0%	4
movePossible	107	100.0%	37
placePiece	45	100.0%	23
strongerPiece	142	100.0%	4
validCoords	29	100.0%	610
Board.vdmpp		100.0%	1283

3.2 Class Cell

```
class Cell
types
 public Position :: x:nat y:nat;
instance variables
 private position : Position;
 private piece : [Piece] := nil;
 private water : bool := false;
 private occupied : bool := false;

 --Invariants
 inv occupied= true <=> piece <> nil;
 inv water = true => piece = nil and occupied = false;
operations
  public Cell : nat * nat ==> Cell
   Cell(x,y) ==
   (

    position := mk_Position(x,y);
    water := isWater(position);
  );

 public setPiece : Piece ==> ()
 setPiece(pie) ==
 (

  atomic(
   piece := pie;
   occupied := true;
   )
 )
```

```
 pre occupied = false and piece = nil and water = false
 post occupied = true and piece <> nil;

 public removePiece : () ==> ()

 removePiece() ==
 (
  atomic(
   piece := nil;
   occupied := false;
  )
 )
 pre occupied = true and piece <> nil and water = false
 post occupied = false and piece = nil;


 public pure isOccupied : () ==> bool
 isOccupied() == return occupied or water;


 public pure getPiece : () ==> [Piece]
 getPiece() == return piece;


 public pure getWater : () ==> bool
 getWater() == return water;

 public pure isWater : Position ==> bool
 isWater(pos) ==

 (
  return (pos.x = 2 or pos.x = 3 or pos.x = 6 or pos.x = 7) and (pos.y = 4 or pos.y = 5);
 );
end Cell
```

Function or operation	Line	Coverage	Calls
Cell	17	100.0%	200
getPiece	45	100.0%	3190
getWater	54	100.0%	14
isOccupied	42	100.0%	46
isWater	48	100.0%	200
removePiece	33	100.0%	24
setPiece	24	100.0%	40
Cell.vdmpp		100.0%	3714

3.3 Class Piece

```
class Piece
types
   public PType = <BOMB> | <MARSHALL> | <GENERAL> | <COLONEL> | <MAJOR> | <CAPTAIN> | <
       LIEUTENANT> | <SERGEANT> | <MINER> | <SCOUT> | <SPY> | <FLAG>;
  public PColor = <RED> | <BLUE>;
instance variables
  private PieceType : PType;
  private PieceColor : PColor;

  private onBoard : bool := false;
  private cannotMove : bool;
  private unlimitedRange : bool;

   private lastPosition : [Board`Position] := nil;
   private movesRepeated : nat := 0;

   --If it has unlimited range then the piece can move

   inv unlimitedRange => not cannotMove;
   --If the pice cannot move then it wont have a last position or moves repeated
   inv cannotMove => lastPosition = nil and movesRepeated = 0;
   --If there are moves repeated then there must be a last position
   inv movesRepeated <> 0 => lastPosition <> nil;
operations
  public Piece : PType*PColor ==> Piece
    Piece(type,color) ==
    (
     PieceType := type;
     PieceColor := color;

     cannotMove := PieceType = <BOMB> or PieceType = <FLAG>;

     unlimitedRange := PieceType = <SCOUT>;

     placePiece();
    )
    pre type <> nil and color <> nil;


    public placePiece : () ==> ()

    placePiece() ==
    (
     onBoard := true;

    )
    post self.onBoard = true;


    --Updates the move in order with the previous
    public newPosition : Board`Position*Board`Position ==> ()
    newPosition(origin,destination) ==

    (
     if destination <> lastPosition then
     (
       movesRepeated := 0;
       lastPosition := origin;
```

```
    )
    else
    (

     movesRepeated := movesRepeated + 1;
     lastPosition := origin;
    )
   );

   --Checks if the new move can be made
   public canMakeMove : Board`Position ==> bool
   canMakeMove(newPos) ==
   (
    return newPos <> lastPosition or movesRepeated < 3;
   );

   public pure getType : () ==> PType
   getType() == return PieceType;

   public pure getColor : () ==> PColor
   getColor() == return PieceColor;

   public pure canMove : () ==> bool
   canMove() == return not cannotMove;

   public pure hasUnlimitedRange : () ==> bool
   hasUnlimitedRange() == return unlimitedRange;
end Piece
```

Function or operation	Line	Coverage	Calls
Piece	18	100.0%	23
canMakeMove	55	100.0%	23
canMove	44	100.0%	25
getColor	41	100.0%	91
getType	38	100.0%	303
hasUnlimitedRange	47	100.0%	25
newPosition	37	100.0%	17
placePiece	31	100.0%	23
Piece.vdmpp		100.0%	530

3.4 Class Stratego

```
class Stratego

values
  public PIECES_PER_PLAYER : seq of nat1 = [6,1,1,2,3,4,4,4,5,8,1,1];
  public TYPEOFPIECES : seq of Piece`PType = [ <BOMB> , <MARSHALL> , <GENERAL> , <COLONEL
      > , <MAJOR> , <CAPTAIN> , <LIEUTENANT> , <SERGEANT> , <MINER> , <SCOUT> , <SPY> , <
      FLAG>];
instance variables
  private board : Board;
  private turn : nat := 0;

  -- Turn is either Red or Blue
  inv turn = 1 or turn = 0;
operations
  public Stratego : () ==> Stratego
  Stratego() ==
  (
   board := new Board();
  )
  post validTurn();


  public pure getTurn : () ==> nat
  getTurn() == return turn;

  public changeTurn : () ==> ()
  changeTurn() ==
  (

  if turn = 0 then turn := 1
   else turn := 0;
  )

  pre validTurn()
  post validTurn();

  public pure getBoard : () ==> Board
  getBoard() == return board;

  public makePlay : nat*nat*nat*nat ==> bool
  makePlay(xi,yi,xf,yf) == (



   if turn = 0 then if board.makePlay(xi,yi,xf,yf,<BLUE>) then
   (
    changeTurn();
    return true
   )
    else return false
   else if board.makePlay(xi,yi,xf,yf,<RED>) then
   (
    changeTurn();

    return true
   )
    else return false;
  );
```

```
  public placePiece : nat*nat*Piece`PType*Piece`PColor ==> ()
  placePiece(i,j,type,color) == return board.placePiece(i,j,type,color);

  public pure validTurn : () ==> bool
  validTurn() == return turn = 0 or turn = 1;
end Stratego
```

Function or operation	Line	Coverage	Calls
Stratego	19	100.0%	1
changeTurn	29	100.0%	2
getBoard	37	100.0%	51
getTurn	26	100.0%	3
makePlay	37	100.0%	4
placePiece	46	100.0%	23
validTurn	37	100.0%	5
Stratego.vdmpp		100.0%	89

4 Model Validation

For model validation purposes we created a class, StrategoTest, to test and validate our model.

4.1 StrategoTest

```
class StrategoTest is subclass of TestCase
      instance variables
            s : Stratego := new Stratego();
      operations

         -- Tests with valid Inputs

            private testValidCoords: () ==> ()
            testValidCoords() ==
            (
                  assertEqual(true,s.getBoard().validCoords(mk_Board`Position(0,0)));

                  assertEqual(true,s.getBoard().validCoords(mk_Board`Position(9,5)));

                  assertEqual(false,s.getBoard().validCoords(mk_Board`Position(15,5)));
            );


            private testFreeSpace: () ==> ()
            testFreeSpace() ==
            (
                  assertEqual(true,s.getBoard().freeSpace(mk_Board`Position(0,0)));

                  assertEqual(false,s.getBoard().freeSpace(mk_Board`Position(2,5)));
```

```
      assertEqual(false,s.getBoard().freeSpace(mk_Board`Position(3,4)));

      assertEqual(true,s.getBoard().freeSpace(mk_Board`Position(9,9)));
);


private placePiece: () ==> ()
placePiece() ==
(
   s.placePiece(0,0,<CAPTAIN>,<BLUE>);

   -- six bombs, maximum
   s.placePiece(0,2,<BOMB>,<BLUE>);
   s.placePiece(1,2,<BOMB>,<BLUE>);
   s.placePiece(2,2,<BOMB>,<BLUE>);
   s.placePiece(3,2,<BOMB>,<BLUE>);
   s.placePiece(4,2,<BOMB>,<BLUE>);
   s.placePiece(1,3,<BOMB>,<BLUE>);

   -- Works because is the other player
   s.placePiece(5,9,<BOMB>,<RED>);


   s.placePiece(6,1,<SCOUT>,<BLUE>);
   s.placePiece(6,2,<SCOUT>,<BLUE>);
   s.placePiece(4,3,<SCOUT>,<BLUE>);
   s.placePiece(4,6,<SCOUT>,<RED>);


   s.placePiece(1,6,<MINER>,<RED>);


   s.placePiece(0,6,<SCOUT>,<RED>);
   s.placePiece(0,7,<SCOUT>,<RED>);

  --check freeSpace

      assertEqual(false,s.getBoard().freeSpace(mk_Board`Position(0,0)));
      assertEqual(false,s.getBoard().freeSpace(mk_Board`Position(0,2)));
);


private testHasPiece: () ==> ()
testHasPiece() ==
(
 assertEqual(true,s.getBoard().hasPiece(mk_Board`Position(0,0)));
 assertEqual(true,s.getBoard().hasPiece(mk_Board`Position(0,2)));

 --water
 assertEqual(false,s.getBoard().hasPiece(mk_Board`Position(3,4)));
);


 private possibleMove: () ==> ()
possibleMove() ==
(
   --Test on Brand Board

   dcl board : Board := new Board();

   assertEqual(true,board.movePossible(mk_Board`Position(0,0),mk_Board`
       Position(0,1),false));
```

```
    --fails precondition, you cant make moves sideways
    --assertEqual(false,board.movePossible(mk_Board`Position(0,0),mk_Board`
        Position(1,1),false));

    assertEqual(false,board.isFreeLane(mk_Board`Position(0,0),mk_Board`
        Position(1,1)));

    --with(out) range
    assertEqual(false,board.movePossible(mk_Board`Position(0,0),mk_Board`
        Position(0,5),false));
    assertEqual(true,board.movePossible(mk_Board`Position(0,0),mk_Board`
        Position(0,5),true));

    --walking to and over water

    assertEqual(true,board.movePossible(mk_Board`Position(2,2),mk_Board`
        Position(2,3),true));
    --Fails because precondition wont allow for moves to water
    --assertEqual(false,board.movePossible(mk_Board`Position(2,2),mk_Board`
        Position(2,4),true));
    assertEqual(true,board.getCell(mk_Board`Position(2,4)).getWater());
    assertEqual(false,board.movePossible(mk_Board`Position(2,2),mk_Board`
        Position(2,7),true));


  assertEqual(true,s.getBoard().movePossible(mk_Board`Position(0,0),mk_Board`
      Position(0,1),false));
  assertEqual(true,s.getBoard().movePossible(mk_Board`Position(1,3),mk_Board`
      Position(2,3),false));
  assertEqual(true,s.getBoard().movePossible(mk_Board`Position(1,3),mk_Board`
      Position(2,3),true));

  --Pre condition fails
  --assertEqual(false,s.getBoard().movePossible(mk_Board`Position(50,0),
      mk_Board`Position(50,50),false));
  --assertEqual(false,s.getBoard().movePossible(mk_Board`Position(50,0),
      mk_Board`Position(50,50),true));

  assertEqual(true,s.getBoard().movePossible(mk_Board`Position(1,0),mk_Board`
      Position(9,0),true));

  assertEqual(false,s.getBoard().movePossible(mk_Board`Position(1,0),mk_Board
      `Position(9,0),false));

    -- with range
    assertEqual(false,s.getBoard().movePossible(mk_Board`Position(0,6),
        mk_Board`Position(0,9),true));
    assertEqual(false,s.getBoard().movePossible(mk_Board`Position(0,6),
        mk_Board`Position(2,6),true));



);

private makeMove: () ==> ()
 makeMove() ==
(

  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(0,0),mk_Board`
      Position(1,0),<BLUE>));
  assertEqual(false,s.getBoard().makeMove(mk_Board`Position(0,6),mk_Board`
      Position(0,7),<RED>));
```

```
  assertEqual(false,s.getBoard().makeMove(mk_Board`Position(6,1),mk_Board`
      Position(6,2),<BLUE>));


  --test scout range
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,6),mk_Board`
      Position(4,3),<RED>));
  assertEqual(false,s.getBoard().hasPiece(mk_Board`Position(4,6)));
  assertEqual(false,s.getBoard().hasPiece(mk_Board`Position(4,3)));


);

private testInteraction: () ==> ()
 testInteraction() ==
(
  --test interaction
  s.placePiece(9,6,<LIEUTENANT>,<RED>);
  s.placePiece(9,3,<MAJOR>,<BLUE>);


  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(9,6),mk_Board`
      Position(9,5),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(9,5),mk_Board`
      Position(9,4),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(9,4),mk_Board`
      Position(9,3),<RED>));

  assertEqual(<BLUE>, s.getBoard().getPiece(mk_Board`Position(9,3)).getColor
      ());

  --test another interaction

  s.placePiece(8,6,<MAJOR>,<RED>);
  s.placePiece(8,3,<LIEUTENANT>,<BLUE>);


  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(8,6),mk_Board`
      Position(8,5),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(8,5),mk_Board`
      Position(8,4),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(8,4),mk_Board`
      Position(8,3),<RED>));

  assertEqual(<RED>, s.getBoard().getPiece(mk_Board`Position(8,3)).getColor()
      );

  --test miner


  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(1,6),mk_Board`
      Position(1,5),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(1,5),mk_Board`
      Position(1,4),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(1,4),mk_Board`
      Position(1,3),<RED>));

  assertEqual(<RED>, s.getBoard().getPiece(mk_Board`Position(1,3)).getColor()
      );

  --test spy
```

```
  s.placePiece(4,6,<SPY>,<RED>);
  s.placePiece(4,3,<MARSHALL>,<BLUE>);

  --impossible move

  assertEqual(false,s.getBoard().makeMove(mk_Board`Position(4,6),mk_Board`
      Position(4,8),<RED>));

  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,6),mk_Board`
      Position(4,5),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,5),mk_Board`
      Position(4,4),<RED>));
  assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,4),mk_Board`
      Position(4,3),<RED>));

  assertEqual(<RED>, s.getBoard().getPiece(mk_Board`Position(4,3)).getColor()
      );

);

private testIfGameOver : () ==> ()
testIfGameOver() == (
  --game not started
  assertEqual(nil,s.getBoard().checkGameOver());
  s.placePiece(3,7,<FLAG>,<RED>);
  --red won
  assertEqual(<RED>,s.getBoard().checkGameOver());
  s.placePiece(3,3,<FLAG>,<BLUE>);
  --during game
  assertEqual(nil,s.getBoard().checkGameOver());
  s.getBoard().getCell(mk_Board`Position(3,7)).removePiece();
  --blue won
  assertEqual(<BLUE>,s.getBoard().checkGameOver());

);

private StrategoTurns : () ==> ()
StrategoTurns() == (
 --Play returning false
 assertEqual(false,s.makePlay(6,1,6,2));

 --Blue Turn
 assertEqual(0,s.getTurn());
 --Blue piece
 assertEqual(true,s.makePlay(6,1,5,1));

 --Now red Turn
 assertEqual(1,s.getTurn());

 --Play returning false
 assertEqual(false,s.makePlay(1,3,3,3));

 --Blue Piece, must fails precondition
 --s.makePlay(5,1,6,1);
 --Red Piece, Play made
 assertEqual(true,s.makePlay(1,3,1,4));
 assertEqual(0,s.getTurn());

);

private repeatPlays : () ==> ()
repeatPlays() == (
```

```
        assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,3),mk_Board`
            Position(4,4),<RED>));
        assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,4),mk_Board`
            Position(4,3),<RED>));
        assertEqual(true,s.getBoard().makeMove(mk_Board`Position(4,3),mk_Board`
            Position(4,4),<RED>));

        --After 3 plays to the same position it returnes false
        assertEqual(false,s.getBoard().makeMove(mk_Board`Position(4,4),mk_Board`
            Position(4,3),<RED>));
      );



      -- Invalid Inputs, Pre Condition Failing, one at a time

      private placePieceOnWater : () ==> ()

      placePieceOnWater() == s.placePiece(4,5,<BOMB>,<BLUE>);

      private placePieceOutOfRegion : () ==> ()

      placePieceOutOfRegion() == s.placePiece(9,9,<BOMB>,<BLUE>);

      private excessiveTypePieces : () ==> ()
      excessiveTypePieces() == (
      -- six bombs, maximum
      s.placePiece(0,2,<BOMB>,<BLUE>);
      s.placePiece(1,2,<BOMB>,<BLUE>);
      s.placePiece(2,2,<BOMB>,<BLUE>);
      s.placePiece(3,2,<BOMB>,<BLUE>);
      s.placePiece(4,2,<BOMB>,<BLUE>);
    s.placePiece(1,3,<BOMB>,<BLUE>);

      s.placePiece(6,2,<BOMB>,<BLUE>);

      );

      public static main: () ==> ()
      main() ==
      (
            dcl test : StrategoTest := new StrategoTest();

            --Test Spaces
            test.testValidCoords();
            test.testFreeSpace();

            --Test Pieces
            test.placePiece();
            test.testHasPiece();

            --Tests Moves
            test.possibleMove();
            test.makeMove();
            test.testInteraction();

            --Test Game
            test.testIfGameOver();
            test.StrategoTurns();
```

```
                test.repeatPlays();
            );
end StrategoTest
```

In red (as well as some documented) are operations that cannot be called because they break their preconditions. We thought it was best to keep them there but not run them on the main function as it would halt the execution.

4.2 TestCase

```
class TestCase
/*
 Superclass for test classes, simpler but more practical than VDMUnit`TestCase.
 For proper use, you have to do: New -> Add VDM Library -> IO.
 JPF, FEUP, MFES, 2014/15.
*/

operations

 -- Simulates assertion checking by reducing it to pre-condition checking.
 -- If 'arg' does not hold, a pre-condition violation will be signaled.

 protected assertTrue: bool ==> ()
 assertTrue(arg) ==
  return
 pre arg;

 -- Simulates assertion checking by reducing it to post-condition checking.
 -- If values are not equal, prints a message in the console and generates
 -- a post-conditions violation.

 protected assertEqual: ? * ? ==> ()
 assertEqual(expected, actual) ==
  if expected <> actual then (
    IO`print("Actual value (");
    IO`print(actual);
    IO`print(") different from expected (");
    IO`print(expected);
    IO`println(")\n")
 )
 post expected = actual

end TestCase
```

5 Model Verification

In this section we address the proof obligations that are generated by the the Proof Obligation tool available in the Overture Tools.

5.1 Domain Verification

This domain verification proof obligation generated by the Overture Tools is a set of 3 proof obligations that can be proved with the same answer. These are the the proof obligations:

No.	PO Name	Type
01	Board‘limitPieces(PType,PColor)	Legal Sequence Application
02	Board‘getIndex(PType)	Legal Sequence Application
03	Board‘getIndex(PType)	Legal Sequence Application

The code for 01 is (the operation is too big and unnecessary to put here as we shall see further ahead):

```
public pure limitPieces : Piece‘PType*Piece‘PColor ==> bool
limitPieces(type,color) ==
(
        (...)
        return pieces < Stratego‘PIECES_PER_PLAYER(getIndex(type));
) pre type in set elems Stratego‘TYPEOFPIECES;
```

The code for 02 and 03 is in the same function of the source code:

```
public getIndex : Piece‘PType -> nat
getIndex(type) ==
(
      [i | i in set inds Stratego‘TYPESOFPIECES &
          Stratego‘TYPESOFPIECES(i) = type](1)
)pre type in set elems Stratego‘TYPEOFPIECES;
```

The proof for this last function is in the expression *i in set inds Stratego‘TYPESOFPIECES*. This means that the *i* variable will contain an index that is in the domain of Stratego‘TYPESOFPIECES. The fact that that index exists is confirmed by the precondition that assures that the type does exist. This proves the fact that i is a not nil nat1 type of variable.

The proof for 01 is the same, as it is a call to the previously proved function and has the same precondition, therefore the access is made to the domain of the sequence, as proven before.

5.2 Invariant Preservation

One example of proof obligation concerning Invariant Preservation is:

No.	PO Name	Type
04	Cell‘Cell(nat,nat)	State Invariant Holds

The code analyzed for 04 is the Cell constructor:

```
public setPiece : Piece ==> ()
setPiece(pie) ==
(
        atomic(
                piece := pie;
                occupied := true;
                )
)
pre occupied = false and piece = nil and water = false
post occupied = true and piece <> nil;
```

The relevant invariant under analysis is:

```
inv occupied <=> piece <> nil;
```

This means that after the atomic block, the Piece is set in the Cell and:

```
(if occupied then piece <> nil) and (if piece <> nil then occupied = true)
```

The invariant holds before this operation is called because it is assured in the pre-conditions.

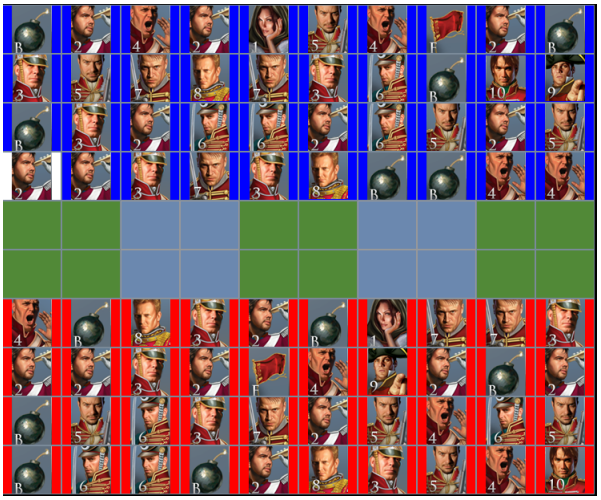
If each of the code lines in the atomic block ran out of an atomic block it would have failed the invariant. If a piece was set as not nil, then the occupied was still false, so it would break the second part. If occupied was set as true, then the piece was still set as nil, so it would break the first part. When both are ran, piece is not nil and occupied is true so it holds the invariant in both parts of it.

6 Code Generation - Graphical User Interface

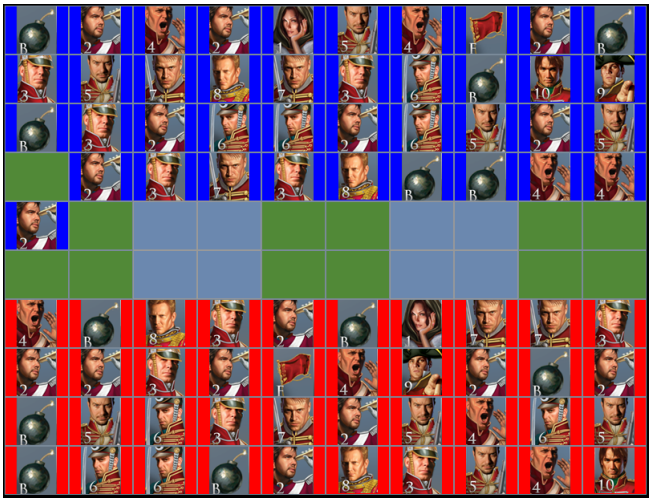
Using the Overture Tools we generated this model in Java language. The Overture Java Code Generator Tool was not properly configured by default so we had to tweak some of the settings such as defining an output package. We then developed a Graphical User Interface to enhance the user experience.

We hope that with this GUI we can present a better representation of the model developed. The correspondent code can be found as an attachment with the rest of the VDM++ project.

6.1 Move Piece

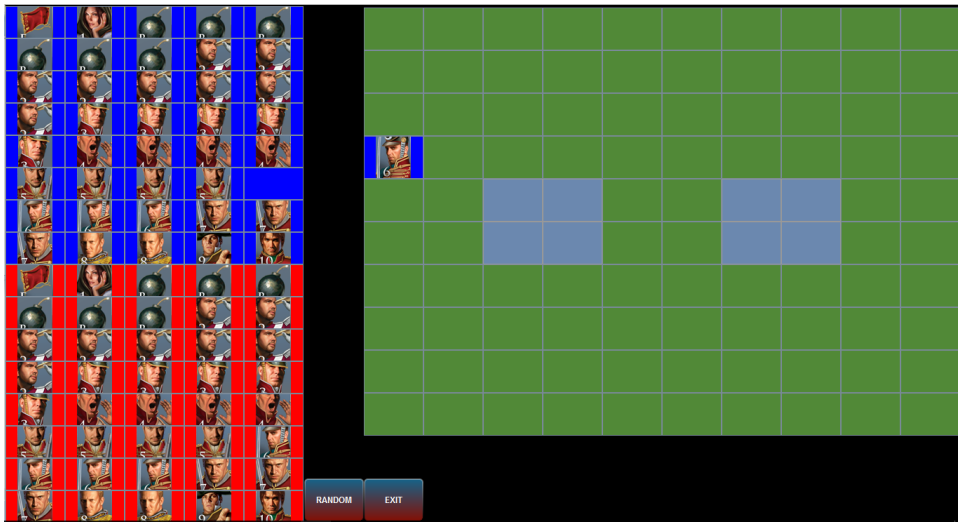


In this image we can see a piece being selected to start the moving action. When a piece is picked its background changes to white for the player to be aware of the piece he picked.



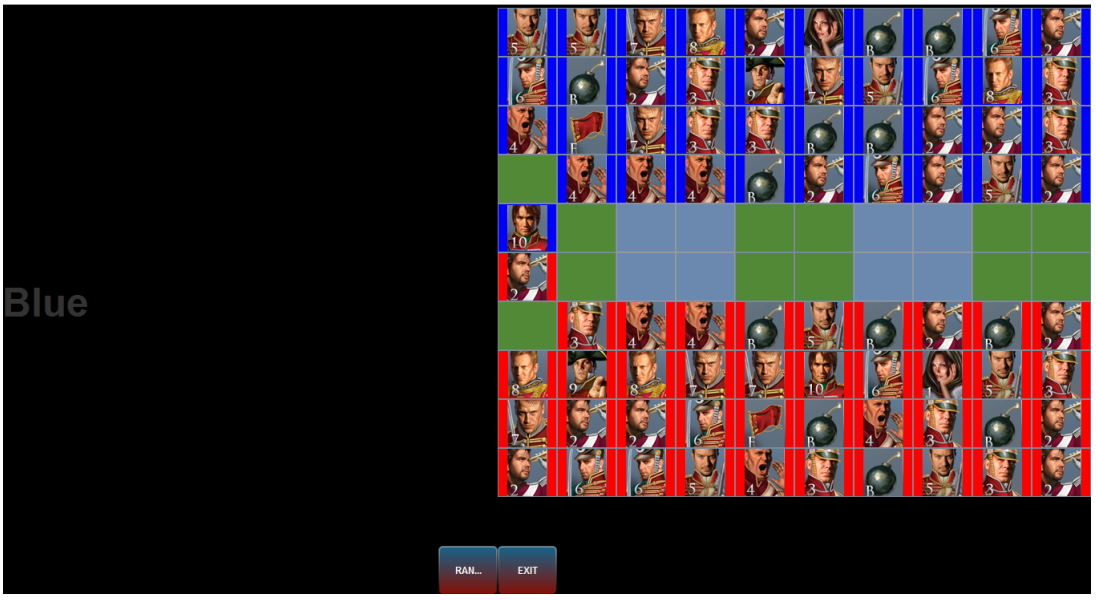
Now we see that the piece was successfully moved.

6.2 Place Piece Action

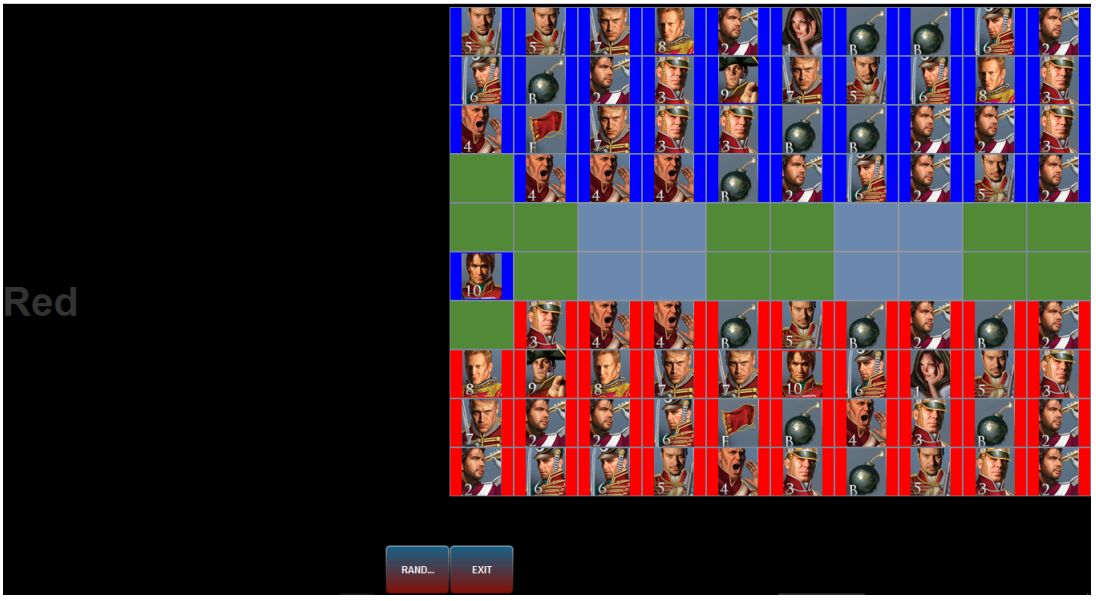


In this image it is possible to see that a Blue Piece was placed in the Board. To withdraw the piece from the Initial Board Piece you just need to pick the piece you want to move with your mouse and place it in the position of the Board you desire (if the position is valid, otherwise it cannot be placed).

6.3 Attack Opponent Pieces Action



In this image we have a situation where it is time for the Blue Player to play. He has some possibilities of plays but if he wants he can attack the red piece that is placed in front of his piece. To execute that same action the blue player just has to pick the blue piece and then click on the red piece to make the blue piece attack the red one. If the blue piece has a higher rank it will eliminate the red piece.



As stated in the previous image description, the blue player chose to attack the red piece and since the blue piece had a higher rank than the red piece, the blue prevailed above the red one as shown on the image below.

7 Conclusions

In the end we are pleased with the work we have achieved. We did everything we promised ourselves to do and we have implemented an optimized model in VDM++. As bonus we also implemented a good Graphical User Interface.

If time permitted, as future work, we would like to perfect our user interface as well as adding some additional tournament rules.

This project took approximately 30 hours to develop. The contribution of each element was the same.

8 References

1. VDM-10 Language Manual, Peter Gorm Larsen et al, Overture Technical Report Series No. TR-001, March 2014
2. Overture tool web documentation, http://overturetool.org