U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Tuesday, 03 January 2017

Pyramid Solitaire in VDM++

Mestrado Integrado em Engenharia Informática e Computação

Métodos Formais em Engenharia de Software

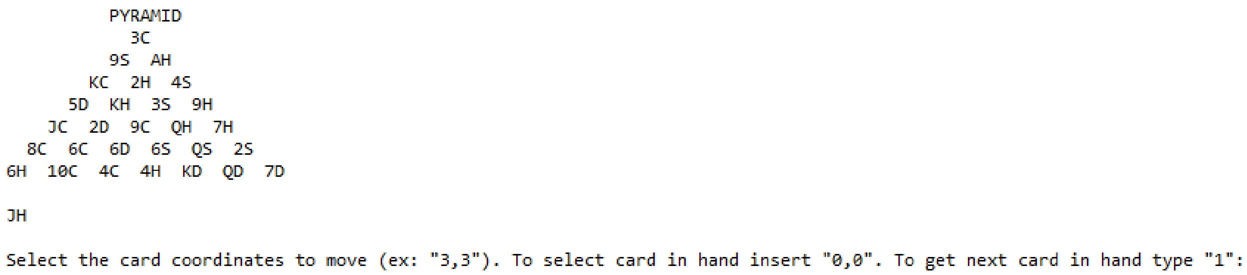
Fábio Amarante – ei10154

Luís Gonçalves – ei12080

Contents

1. Informal system description and list of requirements

1.1 Informal system description



```
          PYRAMID
            3C
          9S  AH
        KC  2H  4S
      5D  KH  3S  9H
    JC  2D  9C  QH  7H
  8C  6C  6D  6S  QS  2S
6H  10C  4C  4H  KD  QD  7D

JH

Select the card coordinates to move (ex: "3,3"). To select card in hand insert "0,0". To get next card in hand type "1":
```

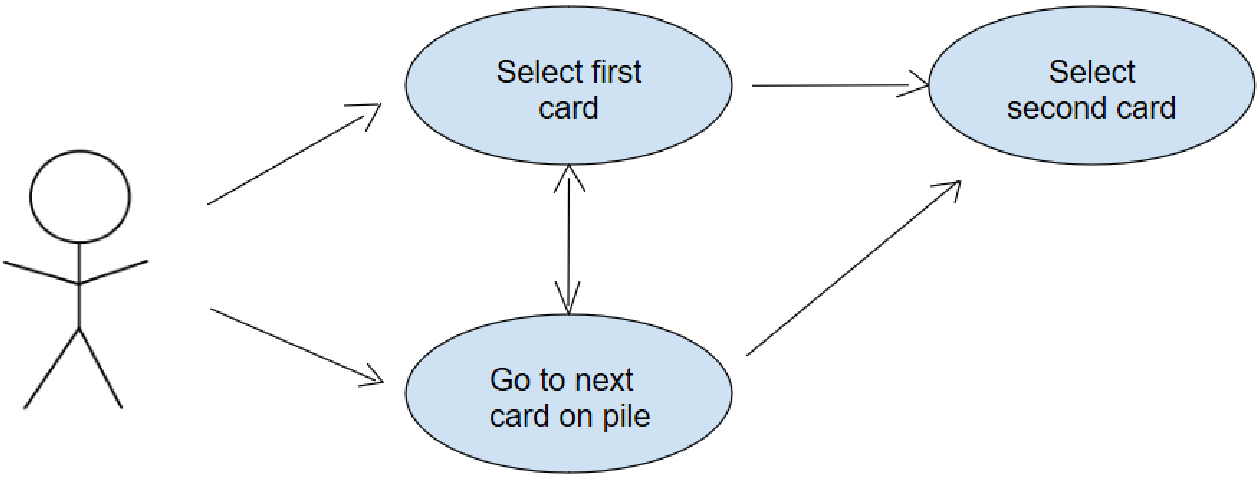
Displayed above, is the pyramid and bellow it the card in the users hand.

1.2 List of requirements

Id	Priority	Description
R1	Mandatory	The user can visualize the cards in the pyramid and in his card pile.
R2	Mandatory	The user can switch to the next card in the card pile.
R3	Mandatory	The user can combine two separate cards, which value amounts to 13, and remove them from the game.
R4	Mandatory	The user should be able to complete the game by combining all the cards from the table/pyramid.

2. Visual UML model [1]

2.1 Use case model [2]

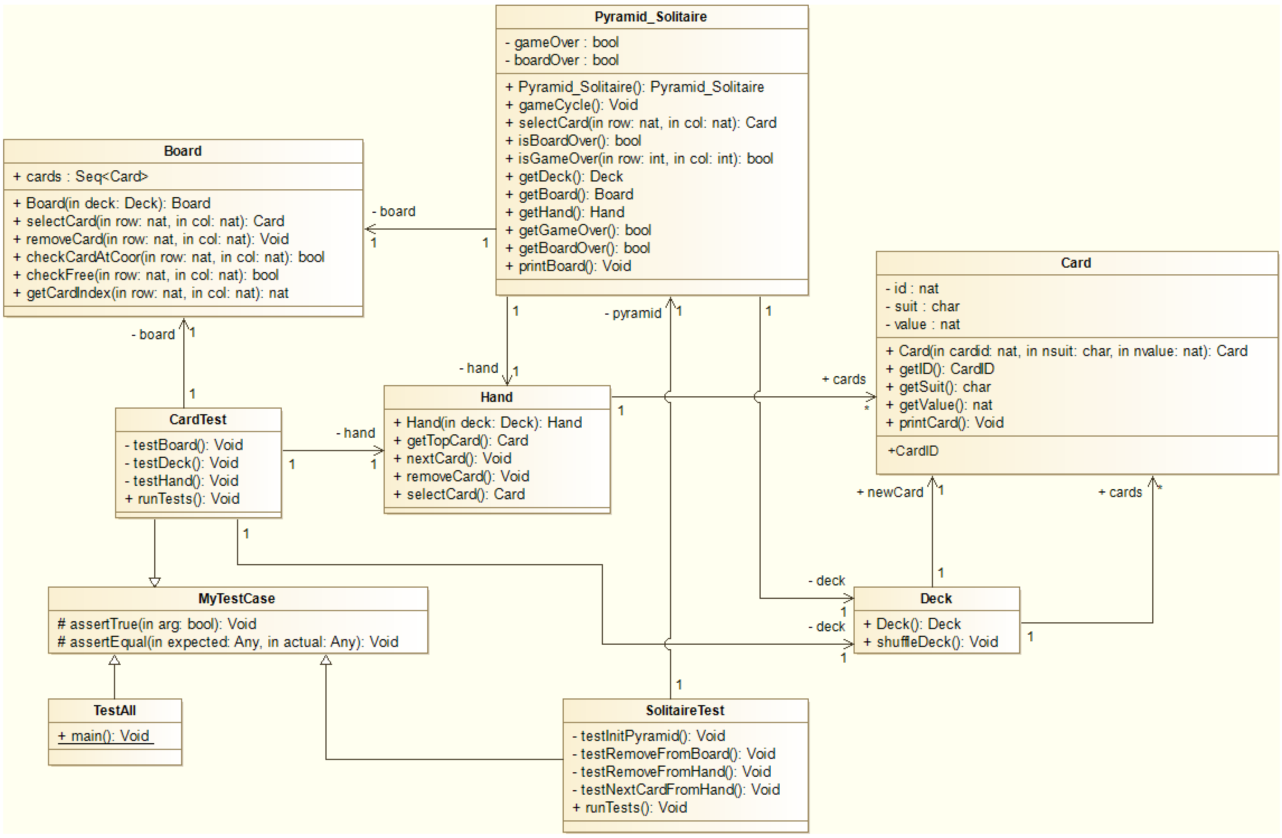


Main Use cases:

Scenario	**Select first card**
Description	Normal scenario for user to pick his first card to pair.
Pre-conditions	1. Row must be greater than 0 and smaller than 8. 2. Column must be greater than 0 and smaller than 8.
Post-conditions	(none)
Steps	1. The user is prompted with a text to insert the column and row. 2. The user inserts the desired values.
Exceptions	(none)

Scenario	**Go to next card on pile**
Description	Normal scenario where user cycles throw the cards on his pile.
Pre-conditions	1. The pile mustn't be empty.
Post-conditions	1. The pile size must continue the same after this action.
Steps	1. The user is prompted with a text to insert the column and row. 2. The user selects the coordinates (0,0).
Exceptions	(none)

2.2 Class model [3] [4]



Class	Description
Board	Core model; defines the board and all its elements and functions.
Pyramid_Solitaire	Responsible for calling all the major game functions present in other classes.
Card	Core model; defines element Card and all its elements and functions.
Hand	Core model; defines the Cards in user hand and all its elements and functions.
MyTestCase	Superclass for test classes; defines `assertEquals` and `assertTrue`.
CardTest	Defines the test/usage scenarios and test cases for the card element.
BoardTest	Defines the test/usage scenarios and test cases for the board.
SolitaireTest	Defines the test/usage scenarios and test cases for the game and its initialization.
TestAll	Calls all the test functions inside all the other Test Classes

3. Formal VDM++ model

3.1 Class Pyramid_Solitaire

```
class Pyramid_Solitaire

types

values
```

```
instance variables
    private deck: Deck;
    private board: Board;
    private hand: Hand;
    private gameOver: bool := false;
    private boardOver: bool := false;

operations

    public Pyramid_Solitaire : () ==> Pyramid_Solitaire
        Pyramid_Solitaire() ==
            (
                deck := new Deck();
                deck.shuffleDeck();
                board := new Board(deck);
                hand := new Hand(deck);
            );

    public gameCycle : () ==> ()
        gameCycle() ==
        (
        printBoard();
        while (gameOver = false) do
            (


                boardOver := isBoardOver();
                --gameOver := isGameOver();
            )
        );

    public selectCard : nat * nat ==> Card
        selectCard(row, col) == (
            if row = 0 then (
                return hand.selectCard();
            );
            if row > 0 then (
                return board.selectCard(col, row);
            );
            return new Card(0,'N',0)
        );

    public isBoardOver : () ==> bool
        isBoardOver() == (
            if board.cards(1) = new Card(0,'N',0) then return true
            else return false
        )
    pre len board.cards = 28;

    public isGameOver : int * int ==> bool
        isGameOver(row, col) == (
            if row = -1 and col = -1 then return true
            else return false
        );

    public pure getDeck : () ==> Deck
        getDeck() == return deck;

    public pure getBoard : () ==> Board
```

```
      getBoard() == return board;

public pure getHand : () ==> Hand
      getHand() == return hand;

public pure getGameOver : () ==> bool
getGameOver() == return gameOver;

public pure getBoardOver : () ==> bool
getBoardOver() == return boardOver;

public printBoard : () ==> ()
printBoard() == (
      IO`println("               PYRAMID               ");
      IO`print("            ");
      IO`print(board.cards(1).getValue());
      IO`println(board.cards(1).getSuit());
      IO`print("          ");
      IO`print(board.cards(2).getValue());
      IO`print(board.cards(2).getSuit());
      IO`print("  ");
      IO`print(board.cards(3).getValue());
      IO`println(board.cards(3).getSuit());
      IO`print("        ");
      IO`print(board.cards(4).getValue());
      IO`print(board.cards(4).getSuit());
      IO`print("  ");
      IO`print(board.cards(5).getValue());
      IO`print(board.cards(5).getSuit());
      IO`print("  ");
      IO`print(board.cards(6).getValue());
      IO`println(board.cards(6).getSuit());
      IO`print("      ");
      IO`print(board.cards(7).getValue());
      IO`print(board.cards(7).getSuit());
      IO`print("  ");
      IO`print(board.cards(8).getValue());
      IO`print(board.cards(8).getSuit());
      IO`print("  ");
      IO`print(board.cards(9).getValue());
      IO`print(board.cards(9).getSuit());
      IO`print("  ");
      IO`print(board.cards(10).getValue());
      IO`println(board.cards(10).getSuit());
      IO`print("    ");
      IO`print(board.cards(11).getValue());
      IO`print(board.cards(11).getSuit());
      IO`print("  ");
      IO`print(board.cards(12).getValue());
      IO`print(board.cards(12).getSuit());
      IO`print("  ");
      IO`print(board.cards(13).getValue());
      IO`print(board.cards(13).getSuit());
      IO`print("  ");
      IO`print(board.cards(14).getValue());
      IO`print(board.cards(14).getSuit());
      IO`print("  ");
      IO`print(board.cards(15).getValue());
      IO`println(board.cards(15).getSuit());
```

```
             IO`print("  ");
             IO`print(board.cards(16).getValue());
             IO`print(board.cards(16).getSuit());
             IO`print("  ");
             IO`print(board.cards(17).getValue());
             IO`print(board.cards(17).getSuit());
             IO`print("  ");
             IO`print(board.cards(18).getValue());
             IO`print(board.cards(18).getSuit());
             IO`print("  ");
             IO`print(board.cards(19).getValue());
             IO`print(board.cards(19).getSuit());
             IO`print("  ");
             IO`print(board.cards(20).getValue());
             IO`print(board.cards(20).getSuit());
             IO`print("  ");
             IO`print(board.cards(21).getValue());
             IO`println(board.cards(21).getSuit());
             IO`print(board.cards(22).getValue());
             IO`print(board.cards(22).getSuit());
             IO`print("  ");
             IO`print(board.cards(23).getValue());
             IO`print(board.cards(23).getSuit());
             IO`print("  ");
             IO`print(board.cards(24).getValue());
             IO`print(board.cards(24).getSuit());
             IO`print("  ");
             IO`print(board.cards(25).getValue());
             IO`print(board.cards(25).getSuit());
             IO`print("  ");
             IO`print(board.cards(26).getValue());
             IO`print(board.cards(26).getSuit());
             IO`print("  ");
             IO`print(board.cards(27).getValue());
             IO`print(board.cards(27).getSuit());
             IO`print("  ");
             IO`print(board.cards(28).getValue());
             IO`println(board.cards(28).getSuit());

             IO`println("");
             IO`print(hand.getTopCard().getValue());
             IO`println(hand.getTopCard().getSuit());
      );

functions

traces

end Pyramid_Solitaire
```

3.2 Class Board

```
class Board

types

values

instance variables
```

```
    public cards: seq of Card;

operations
    public Board: Deck ==> Board
        Board(deck) == (
            cards := [];
            for i = 1 to 28 do
            (
                cards := cards ^ [deck.cards(i)];
            );
            return self;
        )
    post len cards = 28;

    public selectCard : nat * nat ==> Card
        selectCard(row, col) ==
        (
                if checkCardAtCoor(row, col) = true then (
                    if checkFree(row, col) = true then (
                        dcl cardIndex : nat := getCardIndex(row, col);
                        return cards(cardIndex);
                    );
                );
                return new Card(0,'N',0);
        )
    pre row > 0 and row < 8 and col < 8 and col > 0;

    public removeCard : nat * nat ==> ()
        removeCard(row, col) == (
            dcl index : nat := getCardIndex(row, col);
            cards(index) := new Card(0,'N',0);
        )
    post len cards = len cards~;

    public checkCardAtCoor : nat * nat ==> bool
        checkCardAtCoor(row, col) ==
        (
            dcl index : nat := getCardIndex(row, col);
            if cards(index).getValue() > 0 then return true
            else return false;
        )
    pre row > 0 and row < 8 and col < 8 and col > 0;

    public checkFree : nat * nat ==> bool
        checkFree(row, col) ==
        (
            if row = 7 then return true;
            if checkCardAtCoor(row + 1, col) = false then (
                if checkCardAtCoor(row+1, col+1) = false then return true;
            );
            return false
        )
    pre row > 0 and row < 8 and col < 8 and col > 0;

    public getCardIndex : nat * nat ==> nat
        getCardIndex(row, col) ==
        (
            dcl index : nat := 0;
            for i = 1 to row do
            (
                for j = 1 to i do
                (
                    index := index + 1;
                    if i = row and j = col then return index;
                )
            );
            return index;
        )
    pre row > 0 and row < 8 and col < 8 and col > 0;

functions

traces

end Board
```

3.3 Class Deck

```
class Deck
types

values

instance variables
    public cards : seq of Card;
    public newCard : Card;

operations

    public Deck: () ==> Deck
        Deck() == (
            cards := [];

            for nvalue = 1 to 13 do
            (
                newCard := new Card(nvalue, 'S', nvalue);
                cards := cards ^ [newCard];
                newCard := new Card(nvalue + 13, 'C', nvalue);
                cards := cards ^ [newCard];
                newCard := new Card(nvalue + 26, 'H', nvalue);
                cards := cards ^ [newCard];
                newCard := new Card(nvalue + 39, 'D', nvalue);
                cards := cards ^ [newCard];
            );
            return self;
        )
    post len cards = 52;

    public shuffleDeck : () ==> ()
        shuffleDeck() ==
        (
                dcl newIndex : nat1;
        dcl tempCard : Card;

        for index = 1 to 52 do
        (
            tempCard := cards(index);
            newIndex := MATH`rand(52) + 1;
```

```
                    cards(index) := cards(newIndex);
                    cards(newIndex) := tempCard;
              );
              )
       post len cards = 52;

functions

traces

end Deck
```

3.4 Class Card

```
class Card

types
       public CardID = nat;

values

instance variables
       id : nat;
       suit : char;
       value : nat;

operations
       public Card : nat * char * nat ==> Card
              Card(cardid, nsuit, nvalue) ==
                 (
                            id := cardid;
                     suit := nsuit;
                     value := nvalue;

                     return self
                 )
       pre (nvalue >= 0 and nvalue <= 13);

       public pure getID : () ==> CardID
              getID() == return id;

       public pure getSuit : () ==> char
              getSuit() == return suit;

       public pure getValue : () ==> nat
              getValue() == return value;

       public printCard : () ==> ()
              printCard() ==
              (
                     if value < 11 and value > 1 then
                            (
                                   IO`print(value);
                                   IO`printf("%s", [suit]);
                                   return;
                            );
                     if value = 1 then
                     (
                            IO`printf("%s", ['A']);
```

```
                IO`printf("%s", [suit]);
                return;
            );
            if value = 11 then
            (
                IO`printf("%s", ['J']);
                IO`printf("%s", [suit]);
                return;
            );
            if value = 12 then
            (
                IO`printf("%s", ['Q']);
                IO`printf("%s", [suit]);
                return;
            );
            if value = 13 then
            (
                IO`printf("%s", ['K']);
                IO`printf("%s", [suit]);
                return;
            );
        );

functions

traces

end Card
```

3.5 Class Hand

```
class Hand

types

values

instance variables
    public cards: seq of Card := [];

operations
    public Hand: Deck ==> Hand
        Hand(deck) == (
            cards := [];
            for i = 29 to 52 do
            (
                cards := cards ^ [deck.cards(i)];
            );
            return self;
        )
    post len cards = 24;

    pure public getTopCard : () ==> Card
        getTopCard() ==     return hd cards
    pre len cards > 0;

    public nextCard : () ==> ()
        nextCard() == cards := tl cards ^ [hd cards]
    pre len cards > 1
```

```
        post len cards = len cards~;

        public removeCard : () ==> ()
               removeCard() == cards := tl cards
        pre len cards > 0
        post len cards = (len cards~) - 1;

        public selectCard : () ==> Card
               selectCard() == return hd cards
        pre len cards > 0;

functions

traces

end Hand
```

4. Model validation

4.1 Class MyTestCase

```
class MyTestCase
/*
  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 MyTestCase
```

4.2 Class CardTest

```
class CardTest is subclass of MyTestCase
```

```
types

values

instance variables
    private deck: Deck := new Deck();
    private board: Board := new Board(deck);
    private hand: Hand := new Hand(deck);

operations
    private testBoard: () ==> ()
    testBoard() ==
    (
        assertEqual(28, len board.cards);
        for index = 1 to len board.cards do
        (
            dcl tempCard: Card := deck.cards(index);
            assertTrue(tempCard.getValue() > 0);
            assertTrue(tempCard.getSuit() = 'S' or tempCard.getSuit() = 'C' or
tempCard.getSuit() = 'D' or tempCard.getSuit() = 'H');
        )
    );

    private testDeck: () ==> ()
    testDeck() ==
    (
        assertEqual(52, len deck.cards);
    );

    private testHand: () ==> ()
    testHand() ==
    (
        assertEqual(24, len hand.cards);
        for index = 1 to len hand.cards do
        (
            dcl tempCard: Card := hand.cards(index);
            assertTrue(tempCard.getValue() > 0);
            assertTrue(tempCard.getSuit() = 'S' or tempCard.getSuit() = 'C' or
tempCard.getSuit() = 'D' or tempCard.getSuit() = 'H');
        )
    );

    public runTests: () ==> ()
    runTests() ==
    (
        testBoard();
        testDeck();
        testHand();
    );

functions

traces

end CardTest
```

4.3 Class SolitaireTest

```
class SolitaireTest is subclass of MyTestCase

types

values

instance variables
	private pyramid: Pyramid_Solitaire := new Pyramid_Solitaire();

operations
	private testInitPyramid: () ==> ()
	testInitPyramid() ==
	(
		assertEqual(false, pyramid.getGameOver());
		assertEqual(false, pyramid.getBoardOver());
	);

	private testRemoveFromBoard: () ==> ()
	testRemoveFromBoard() ==
	(
		dcl index: nat := pyramid.getBoard().getCardIndex(1, 1);
		assertTrue(pyramid.getBoard().cards(index).getValue() > 0);
		pyramid.getBoard().removeCard(1, 1);
		assertTrue(pyramid.getBoard().cards(index).getValue() = 0);
	);

	private testRemoveFromHand: () ==> ()
	testRemoveFromHand() ==
	(
		dcl tempCard: Card := pyramid.getHand().getTopCard();
		assertTrue(tempCard.getValue() > 0);
		pyramid.getHand().removeCard();
		assertTrue(pyramid.getHand().getTopCard().getValue() <> tempCard.getValue()
or pyramid.getHand().getTopCard().getSuit() <> tempCard.getSuit());
	);

	private testNextCardFromHand: () ==> ()
	testNextCardFromHand() ==
	(
		dcl tempCard: Card := pyramid.getHand().getTopCard();
		assertTrue(tempCard.getValue() > 0);
		pyramid.getHand().nextCard();
		assertTrue(pyramid.getHand().getTopCard().getValue() <> tempCard.getValue()
or pyramid.getHand().getTopCard().getSuit() <> tempCard.getSuit());
	);

	public runTests: () ==> ()
	runTests() ==
	(
		testInitPyramid();
		testRemoveFromBoard();
		testRemoveFromHand();
		testNextCardFromHand();
	);

functions
```

```
traces

end SolitaireTest
```

4.4 TestAll

```
class TestAll is subclass of MyTestCase

types

values

instance variables

operations

  public static main: () ==> ()
  main() ==
  (
    dcl boardTest : CardTest := new CardTest();
    dcl solitaireTest : SolitaireTest := new SolitaireTest();
    boardTest.runTests();
    solitaireTest.runTests();
  );

functions

traces

end TestAll
```

5. Model verification

5.1 Example of domain verification

One of the proof obligations generated by Overture is:

No.	PO Name	Type
3	Board`selectCard(nat,nat)	legal sequence application

The code under analysis (with the relevant map application underlined) is:

```
        public selectCard : nat * nat ==> Card
                selectCard(row, col) ==
                (
                        if checkCardAtCoor(row, col) = true then (
                                if checkFree(row, col) = true then (
                                        dcl cardIndex : nat := getCardIndex(row, col);
                                        return cards(cardIndex);
                                );
                        );
                return new Card(0,'N',0);
        )
   pre row > 0 and row < 8 and col < 8 and col > 0;
```

In this case the proof is trivial because the preconditions'row > 0 and row < 8 and col < 8 and col > 0' assures that the sequence is accessed only inside its domain.

5.2 Example of invariant verification

Another proof obligation generated by Overture is:

No.	PO Name	Type
13	Deck`shuffleDeck()	state invariant holds

The code under analysis (with the relevant state changes underlined) is:

```
public shuffleDeck : () ==> ()
        shuffleDeck() ==
        (
                dcl newIndex : nat1;
                dcl tempCard : Card;

                for index = 1 to 52 do
                (
                        tempCard := cards(index);
                        newIndex := MATH`rand(52) + 1;
                        cards(index) := cards(newIndex);
                        cards(newIndex) := tempCard;
                );
        ) post len cards = 52;
```

The relevant invariant under analysis is:

```
inv len cards = 52;
```

The function only randomly selects an index (within bounds) from the sequence and swaps the card with a different one.

6. Conclusions

The model that was developed covers all the requirements.

If time permitted, as future work, it would be interesting to develop a more appealing interface for the user to play the game, a graphical one for example. This project took approximately 14 hours to develop, with an equal contribution to the final product from both elements.

7. References

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