VDM++ Model and Model Validation

December 16, 2014

Contents

1 VDM++ Model

1.1 ExchangeSystem

```
class ExchangeSystem
/*
 Contains the core model of the exchange system.
 Defines the state variables and operations available to the users.
*/

types
 public String = seq of char;

instance variables
 -- products allowed to be bought/sold
 public products : set of Product := {};

 -- orders currently available for matches

 public orders : set of Order := {};

 -- sequence of transactions previously completed
 public history : seq of Transaction := [];


 -- no order available for matches should be fulfilled
 inv forall o in set orders & o.fulfilled = false;

 -- product ids should always be unique

 inv not exists p1, p2 in set products & p1 <> p2 and p1.id = p2.id;

operations
```

```
public ExchangeSystem: () ==> ExchangeSystem
ExchangeSystem() ==
 return self;

/* Admin Operations */

public insertProduct: Product ==> ()
insertProduct(product) ==
 products := products union {product}

post product in set products;

public removeProduct: Product ==> ()
removeProduct(product) ==
(
 products := products \ {product};


 -- after removing a product, all orders matching said product should also be removed
 for all order in set orders do
  if (order.product = product) then
   orders := orders \ {order};
)

pre product in set products
post not product in set products;

/* End-user Operations */

public insertOrder: Order ==> ()
insertOrder(order) ==
 orders := orders union {order}
pre order.product in set products
 and order.fulfilled = false
post order in set orders;

public cancelOrder: Order ==> ()
cancelOrder(order) ==
 orders := orders \ {order}
pre order in set orders;

-- Returns set of all available matches for a specific order.
public matchOrder: Order ==> set of Order
matchOrder(orderToMatch) ==
(
 dcl matches : set of Order := {};

 for all order in set orders do
 (
  -- orders should:
  -- be different and of different types (BUY/SELL)
  -- refer to the same product
  -- have intersecting ranges of prices
  if (orderToMatch <> order

    and orderToMatch.type <> order.type
    and orderToMatch.product = order.product
    and orderToMatch.minPrice <= order.maxPrice
    and order.minPrice <= orderToMatch.maxPrice)
   then
  (
   -- if buying, we're looking for an order that contains all the attributes we want
   -- if selling, we're looking for an order whose attribute are contained in ours
   if(orderToMatch.type = <BUY> and inMapSubset(orderToMatch.attributes, order.attributes)
    or orderToMatch.type = <SELL> and inMapSubset(order.attributes, orderToMatch.attributes))
    then
    (
     matches := matches union {order};
    );
   );
  );

  return matches;
 )
 pre orderToMatch in set orders;


 -- Matches two orders; removes both from orders available for matches and adds them to history.
 public pickMatch: Order * Order ==> ()
 pickMatch(order1, order2) ==
 (
  order1.setStatus(true);

  order2.setStatus(true);

  history := history ^ [new Transaction(order1, order2)];
  orders := orders \ {order1, order2};
 )
 -- orders may only be picked if they can actually be matched
 pre order1 <> order2
   and order1.type <> order2.type
   and order1.product = order2.product
   and order1.minPrice <= order2.maxPrice
   and order2.minPrice <= order1.maxPrice
   and (order1.type = <BUY> and inMapSubset(order1.attributes, order2.attributes)
   or order1.type = <SELL> and inMapSubset(order2.attributes, order1.attributes));

 public getHistory : () ==> seq of Transaction
 getHistory() ==
  return history;

 public getLastTransaction : () ==> Transaction
 getLastTransaction() ==
  return history(len history);

functions
 -- Checks if one map is subset of the other.
 -- Auxiliary when checking for matching attributes inside orders.
 public inMapSubset :  map String to token *  map String to token +> bool
 inMapSubset(map1, map2) ==
  -- map2 must have all of map1's keys
  -- map2's keys->values that match must be equal to map1's
  if(dom map1 subset dom map2
    and map1 ++ (dom map1 <: map2) = map1)
  then
   true
  else
   false;

end ExchangeSystem
```

Function or operation	Line	Coverage	Calls
ExchangeSystem	15	100.0%	12
cancelOrder	43	100.0%	2
getHistory	94	100.0%	2

getLastTransaction	98	100.0%	1
inMapSubset	103	100.0%	11
insertOrder	36	100.0%	24
insertProduct	19	100.0%	17
matchOrder	48	92.5%	8
pickMatch	78	85.2%	3
removeProduct	24	100.0%	2
ExchangeSystem.vdmpp		93.7%	82

1.2 Product

```
class Product
/*
 Defines a product that may be bought/sold in the exchange system.
*/

types
 -- identifier can be anything, as long as it is unique to the exchange system (verified inside respective class)
 public Identifier = token;


instance variables
 public id: Identifier;

operations
 public Product: Identifier ==> Product
 Product(newId) == (
  id := newId;
 );

end Product
```

Function or operation	Line	Coverage	Calls
Product	10	100.0%	17
Product.vdmpp		100.0%	17

1.3 Order

```
class Order
/*
 Defines an order that may be placed in the exchange system.
*/

types
 public OrderType = <BUY> | <SELL>;
 public String = seq of char;

instance variables
 public type : OrderType;
 public product : Product;

 -- order may have several filters in regards to attributes
 -- for example, for product "Car", user may request "color"->"red" and/or "year"->"1990"
 public attributes: map String to token := { |-> };


 -- range of acceptable prices when buying or selling
 public minPrice : real;
 public maxPrice : real;

 -- by default, an order should not be fulfilled
 public fulfilled: bool := false;

 inv minPrice <= maxPrice;


operations
 public Order: OrderType * Product * map String to token * real * real ==> Order
 Order(ty, prod, attr, min, max) == (
  type := ty;
  product := prod;
  attributes := attr;
  minPrice := min;
  maxPrice := max
 );

 public setStatus: bool ==> ()
 setStatus(status) ==
  fulfilled := status
 post fulfilled = status;

end Order
```

Function or operation	Line	Coverage	Calls
Order	18	100.0%	22
setStatus	27	100.0%	6
Order.vdmpp		100.0%	28

1.4 Transaction

```
class Transaction
/*
 Defines a transaction that previously took place in the exchange system.
 A transaction is composed of a pair of orders.
*/

types

instance variables
 -- different names for easy access to specific type of order
 public buyOrder : Order;
 public sellOrder : Order;


 -- orders should be different and of different types
 inv buyOrder <> sellOrder
  and buyOrder.type = <BUY>
  and sellOrder.type = <SELL>;
 -- orders in a transaction should always have their status set to fulfilled
 inv buyOrder.fulfilled = true
  and sellOrder.fulfilled = true;


operations
 public Transaction: Order * Order ==> Transaction
 Transaction(order1, order2) ==
 (
  -- checking for type here so that we know specific orders are assigned to the correct variables
  if(order1.type = <BUY>) then
  (
   buyOrder := order1;
   sellOrder := order2;
  )
  else
  (
   buyOrder := order2;
   sellOrder := order1;
  )
 )
 post buyOrder.type = <BUY>
   and buyOrder.fulfilled = true
   and sellOrder.type = <SELL>
   and sellOrder.fulfilled = true;

end Transaction
```

Function or operation	Line	Coverage	Calls
Transaction	14	85.7%	2
Transaction.vdmpp		91.2%	2

2 Model Validation

2.1 MyTestCase

```
class MyTestCase
/*
  Superclass for test classes, simpler but more practical than VDMUnit`TestCase.
  Provided by Professor Jo o Carlos Pascoal Faria.
*/

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 :  (");
    IO`print(actual);
    IO`println(")");
    IO`print("Expected:  (");
    IO`print(expected);
    IO`println(")\n")
  )
 post expected = actual

end MyTestCase
```

Function or operation	Line	Coverage	Calls
assertEqual	20	35.0%	26
assertTrue	12	0.0%	0
MyTestCase.vdmpp		31.8%	26

2.2 TestExchangeSystem

```
class TestExchangeSystem is subclass of MyTestCase
/*
  Contains the test cases for the exchange system.

  Illustrates a scenario-based testing approach.
  The test cases aim to cover all usage scenarios as well as all states and transitions.
*/

operations
 -- Auxiliary. Inserts all products from a set of products into the system.

 private insertProducts: ExchangeSystem * set of Product ==> ()

 insertProducts(ex, products) ==
 (
  for all product in set products do
   ex.insertProduct(product);
 );

 -- Auxiliary. Inserts all orders from a set of orders into the system.

 private insertOrders: ExchangeSystem * set of Order ==> ()
 insertOrders(ex, orders) ==
 (
  for all order in set orders do
   ex.insertOrder(order);
 );

 /***** TEST CASES WITH VALID INPUTS ******/



 -- Scenario 1:
 -- The system administrator should be able configure the products that are allowed to be bought/sold.
 public testInsertProducts: () ==> ()
 testInsertProducts() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod1 : Product := new Product(mk_token("car"));
  dcl prod2 : Product := new Product(mk_token("phone"));


  insertProducts(ex, {prod1, prod2});
  assertEqual(ex.products, {prod1, prod2});
 );

 -- Scenario 2:
 -- The system administrator should be able to remove a product and all orders associated to it.

 public testRemoveProducts: () ==> ()
 testRemoveProducts() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod1 : Product := new Product(mk_token("car"));
  dcl prod2 : Product := new Product(mk_token("phone"));
  dcl order1 : Order := new Order(<SELL>, prod1, {|->}, 0, 100);
  dcl order2 : Order := new Order(<SELL>, prod2, {|->}, 0, 100);

  insertProducts(ex, {prod1, prod2});

  insertOrders(ex, {order1, order2});

  ex.removeProduct(prod1);

  assertEqual(ex.products, {prod2});
  assertEqual(ex.orders, {order2});

  ex.removeProduct(prod2);
  assertEqual(ex.products, {});
  assertEqual(ex.orders, {});
 );

 -- Scenario 3:
 -- The end user (buyer or seller) should be to place a new order on the system (either buying or selling).
 public testInsertOrder: () ==> ()

 testInsertOrder() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod : Product := new Product(mk_token("phone"));
  dcl order : Order := new Order(<SELL>, prod, {"color" |-> mk_token("red"), "brand" |-> mk_token("Samsung")}, 0,

  insertProducts(ex, {prod});
  insertOrders(ex, {order});

  assertEqual(ex.orders, {order});
  assertEqual(ex.products, {prod});
 );

 -- Scenario 4:
 -- The end user (buyer or seller) should be able to cancel an order previously placed if it has not yet been mat
 public testCancelOrder: () ==> ()
 testCancelOrder() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod : Product := new Product(mk_token("phone"));
  dcl order : Order := new Order(<SELL>, prod, {"color" |-> mk_token("red"), "brand" |-> mk_token("Samsung")}, 0,

  insertProducts(ex, {prod});
  insertOrders(ex, {order});
  ex.cancelOrder(order);
  assertEqual(ex.orders, {});
 );


 -- Scenario 5:
 -- The end user (buyer or seller) should be able to check all orders that match a given order.
 public testCheckMatches: () ==> ()
 testCheckMatches() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();

  dcl prod1 : Product := new Product(mk_token("car"));
  dcl prod2 : Product := new Product(mk_token("phone"));

  dcl order1: Order := new Order(<BUY>,  prod1, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 0,
  dcl order2 : Order := new Order(<SELL>, prod1, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 1
  dcl order3 : Order := new Order(<SELL>, prod1, {|->}, 30, 50);


  dcl order4 : Order := new Order(<SELL>,  prod2, {"color" |-> mk_token("red"), "brand" |-> mk_token("Samsung")},
  dcl order5 : Order := new Order(<SELL>, prod1, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 3

  insertProducts(ex, {prod1, prod2});

  -- should return no matches because order2's price range does not intersect
  insertOrders(ex, {order1, order2});
  assertEqual(ex.matchOrder(order1), {});

  -- should return no matches because order3 has none of the required attributes
  insertOrders(ex, {order3});
  assertEqual(ex.matchOrder(order1), {});


  -- should return no matches because order4 has a diff key->value on a required key
  insertOrders(ex, {order4});
  assertEqual(ex.matchOrder(order1), {});

  -- should return, since the new order matches

  insertOrders(ex, {order5});
  assertEqual(ex.matchOrder(order1), {order5});
 );

 -- Scenario 6:
 -- The end user (buyer or seller) should be able to make a trade by matching two given orders.

 public testPickOrder: () ==> ()
 testPickOrder() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod : Product := new Product(mk_token("car"));
  dcl order1: Order := new Order(<BUY>,  prod, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 0,
  dcl order2 : Order := new Order(<SELL>, prod, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 30

  insertProducts(ex, {prod});
  insertOrders(ex, {order1, order2});
  ex.pickMatch(order1, order2);
 );

 -- Scenario 7:
 -- The end user (buyer or seller) should be able to check all transactions previously completed.
 public testCheckHistory: () ==> ()
 testCheckHistory() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod : Product := new Product(mk_token("car"));


  dcl buyOrder: Order := new Order(<BUY>,  prod, {"cor" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 0,
  dcl sellOrder : Order := new Order(<SELL>, prod, {"cor" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 3

  dcl lastTransaction : Transaction;

  insertProducts(ex, {prod});
  insertOrders(ex, {buyOrder, sellOrder});
  ex.pickMatch(buyOrder, sellOrder);

  assertEqual(len ex.history, 1);
  assertEqual(ex.getHistory(), ex.history);

  lastTransaction := ex.getLastTransaction();
  assertEqual(ex.history, [lastTransaction]);
  assertEqual(ex.getHistory(), ex.history);

  assertEqual(lastTransaction.buyOrder, buyOrder);
  assertEqual(lastTransaction.sellOrder, sellOrder);
 );

 /***** TEST CASES WITH INVALID INPUTS ******/

 public testFailPickOrder: () ==> ()
 testFailPickOrder() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();

  dcl prod : Product := new Product(mk_token("car"));
  dcl order1: Order := new Order(<BUY>,  prod, {"color" |-> mk_token("blue"), "brand" |-> mk_token("Nissan")}, 0,
  dcl order2 : Order := new Order(<SELL>, prod, {"color" |-> mk_token("red"), "brand" |-> mk_token("Nissan")}, 30,

  insertProducts(ex, {prod});
  insertOrders(ex, {order1, order2});
  ex.pickMatch(order1, order2);
 );

 public testFailInsertProductsSameID: () ==> ()
 testFailInsertProductsSameID() ==
 (
  dcl ex : ExchangeSystem := new ExchangeSystem();
  dcl prod1 : Product := new Product(mk_token("id"));
  dcl prod2 : Product := new Product(mk_token("id"));

  insertProducts(ex, {prod1, prod2});
 );


 /***** Entry point that runs all tests with valid inputs ******/
  public testAll: () ==> ()
  testAll() ==
  (
   testInsertProducts();
   testRemoveProducts();
   testInsertOrder();
   testCancelOrder();
   testCheckMatches();
   testPickOrder();
   testCheckHistory();
  );
```

```
end TestExchangeSystem
```

Function or operation	Line	Coverage	Calls
insertOrders	11	100.0%	16
insertProducts	10	100.0%	12
loadProducts	4	100.0%	12
testAll	179	100.0%	1
testCancelOrder	52	100.0%	1
testCheckHistory	121	100.0%	1
testCheckMatches	66	100.0%	1
testFailInsertProductsSameID	132	0.0%	0
testFailLoadProductSameID	153	100.0%	1
testFailPickOrder	108	0.0%	0
testInsertOrder	27	100.0%	1
testInsertProducts	27	100.0%	1
testLoadAndCancelOrder	36	100.0%	1
testLoadAndCheckHistory	126	100.0%	1
testLoadAndCheckMatches	55	100.0%	1
testLoadAndFailPickOrder	108	100.0%	1
testLoadAndInsertOrder	18	100.0%	1
testLoadAndPickOrder	90	100.0%	1
testPickOrder	95	100.0%	1
testRemoveProducts	42	100.0%	1
TestExchangeSystem.vdmpp		86.7%	55