TiMOMI Mini Project Report
Automated Stock Broker

Anders Kaels Malmos

May 2011

1 System Description

The system is an automated stock broker, where you can specify a list of stocks which automaticly, can either be bought or sold. This is done by defining a prioritised list of stocks to observe, which each has defined a trigger that tells in which situation the system should react with either a buy or a sell action. The trigger is a rule defined upon the history and the current value of the stock.

A stock can be in two states, it can either be bought or be a potential buy. When a stock i added to the watch list a no-action-region is also defined relative to the current value. If the stock is in the no-action-region the system should do nothing, but if the stock value leaves this region, an action can be triggered. The system is only allowed to perform one sell action and one buy action per time step. If there is a choice between more than one stocks to sell at a time, the stock with the lowest cost should be picked and if the cost is eqaul the one with the highest priority should win. If there is a choice between more than one stocks to buy at a time, the stock with the highest gain should be picked and if the gain is eqaul the one with the highest priority should win.

Furthermore the system starts out with an amount of money which can be increased by selling a stock to a higher cost, but a stock can only be bought if the is enough money available at the time.

2 Purpose Of the Model

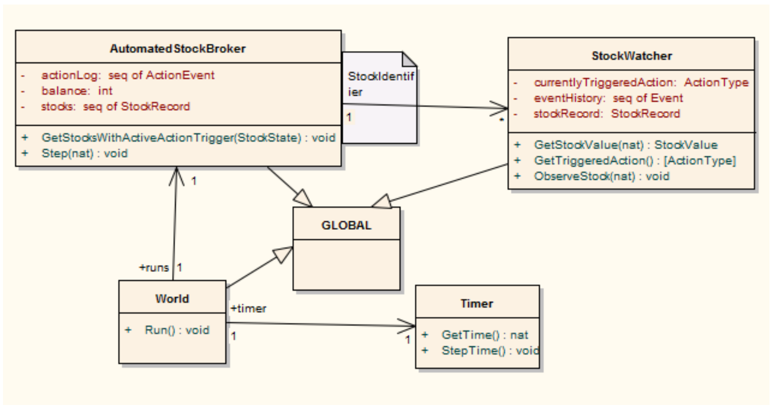
The purpose of the model is to make sure that the rules concerning how and when a buy or sell action is triggered for the currently available highest priority stock, is correctly understood and implemented.

3 Requirements

- **R1.** A computer based system is to be developed that can automaticly trigger a sell or buy action for a list of stocks.
- **R2** The system can only execute one sell and one buy action at a time.
- **R3** It should be possible to see the actions that was executed at specific time.
- **R4** It should be possible to see a list of the stocks with the currently active buy triggers or sell triggers.
- **R5** A trigger can only be active if is not in the no action region.
- **R6** A stock can only be bought if it is a potential buy, its buy trigger is active and the cost is lower than the current balance. If more than one fullfills this, the one with the highest value and highest priority should win.
- **R7** A stock can only be sold if it is bought and its sell trigger is active. If more than one fullfills this, the one with the highest gain where gain is the current value minus the buy cost and highest priority should win.
- **R8** A stock on the watchlist can never be bought two times in a row, a buy action for a specific stock can only be followed be a sell action and vice versa.
- **R9** The same Stock cannot be in the system more than once.

4 The Model

The main model consists of the following components, which will be explained in details below. There are some other parts of the system that concerns validation which will be described in section 5.


AutomatedStockBroker
- actionLog: seq of ActionEvent
- balance: int
- stocks: seq of StockRecord
+ GetStocksWithActiveActionTrigger(StockState) : void
+ Step(nat) : void
StockIdentifier
1
StockWatcher
- currentlyTriggeredAction: ActionType
- eventHistory: seq of Event
- stockRecord: StockRecord
+ GetStockValue(nat) : StockValue
+ GetTriggeredAction() : [ActionType]
+ ObserveStock(nat) : void
GLOBAL
+runs
1
World
+ Run() : void
+timer
Timer
+ GetTime() : nat
+ StepTime() : void

The UML diagram of the model shows som classes that will not be discussed, namely the World and the Timer which is only there to bootstrap and feed the model with a time. The Global Class defines som global types that can be used in all the subclasses so the naming is shorter.

Classes and Types

- **AutomatedStockBroker**: The entire system, which also contains a history of the triggered actions.
- **StockState**: (PotentialBuy , bought).
- **Region**: Defines an region which can be used to define the no action region.
- **ActionType**: (Buy,Sell)
- **ActionEvent**: Describes an action that has occured at a specific time.
- **StockRecord**: Data about a currently observed stock which contains the stock state and triggers.
- **ActionTrigger**: A sequence of events that should happens if the specified action is to be executed.
- **Event**: a stock value Event (UpperLimit,LowerLimit,Peak,Valley,EntersNoActionRegion,LeavesNoActionRegion
- **StockWatcher**: watches the stock value and captures the Events.

Of these components only AutomatedStockBroker and StockWatcher is classes, since theses are the only ones that needs nontrivial functionality. They will be described in detail below.

4.1 The AutomatedStockBroker Class

This is the main class in the model, which essentially models the entire system. We start by taking a look at the instance variables

Listing 1: Instance variables of the AutomatedStockBroker class

```
class AutomatedStockBroker is subclass of GLOBAL
...
 instance variables
  stocks : seq of StockRecord;
  stockWatchers : map StockIdentifier to StockWatcher;
  actionLog : seq of ActionEvent := [];
  balance : int;
...
```

The stocks variable is a sequence of StockRecord which models a priorities list of stocks currently being watched. This is needed because of requirement **R6** and **R7** that says that there is a priority on the observed stocks. The StockRecord represents a stock in the system and is modelled as a record type since it has no functionality. The definition of the StockRecord is shown below,and the types it consists of will be explained in detail along the way.

Listing 2: Instance variables of the AutomatedStockBroker class

```
StockRecord ::  Name : StockIdentifier
                Triggers : map StockState to ActionTrigger
                NoActionReg : Region
                Cost : StockValue
                State : StockState
```

The Name field is way to distinguis the stocks from one another, which we can deduce from the requirements is needed. It is modelled with the type StockIdentifier which i a token, since for the purpose of this model the way the stocks is identified does not matter.

Listing 3: Instance variables of the AutomatedStockBroker class

```
public StockIdentifier = token;
```

The Triggers field maps a StockState to a ActionTrigger, this defines the trigger to use in a certain state. This comes from requirement **R6** and **R7** that says that both a buy and a sell trigger should be specified, which determines when a stock can be bought or sold.

Listing 4: Instance variables of the AutomatedStockBroker class

```
public StockState = <PotentialBuy> | <Bought>;

public ActionTrigger :: Trigger : seq of EventType
                        Action : ActionType;
```

The StockState models the two states a stock can be in, which is a potential buy state and a bought state. It is modelled as a union type of a PotentialBuy quote and a Bought quote type. Next is the ActionTrigger which defines the condition for when the buy or sell action can be performed, it is a record type with two fields where the ActionType specifies what action that can be triggered.

Listing 5: Instance variables of the AutomatedStockBroker class

```
public EventType = <UpperLimit> | <LowerLimit>
                 | <Peak> | <Valley>
                 |<EntersNoActionRegion>
                 | <LeavesNoActionRegion>;

public ActionType = <Buy> | <Sell> ;
```

The EventType models the six different abstract states a stock value can be in, which is sufficient for the purpose of the model. We only need to consider a small simple subset of the these abstract states, since the system should react the same for more advanced patterns, because they essiently just tells the system if it is allowed to buy or sell. The purpose is not to model an advanced pattern recognizer and we therefore do not care about the in between states, a stock value can be in.

The NoActionReg field of the StockRecord is of type Region and it specifies the values for which the system should not perform any actions, as defined in the requirements. The Region record type specifies and UpperValue and LowerValue the invariant says the the UpperValue allways must be greater than or equal to the LowerValue.

Listing 6: Instance variables of the AutomatedStockBroker class

```
public Region :: UpperValue : StockValue
                 LowerValue : StockValue
                 inv mk_Region(p1,p2) == p1 >= p2;

public StockValue = nat;
```

The last two fields are Cost and State. State is the current state of the stock and have allready been mentioned. Cost is the StockValue that was paid for the stock, this is needed because we need to evaluate which of the potential buys that has the highest gain.

The second instans variable of maps a StockIdentifier to a StockWatcher, which is a class that will be decribed in more details in section 4.2. But in short this class is responsible for detecting if the current trigger is active by analysing the EventType sequences.

Next is the actionLog which is required because we need to be able to get the see the actions performed at any given time in that past of the system. It is a sequence of type ActionEvent which is a record type defined as

Listing 7: Instance variables of the AutomatedStockBroker class

```
public ActionEvent :: Type : ActionType
                      Time : nat
                      StockName : StockIdentifier
                      Value : StockValue;
```

and contains The ActionType, time of event, name of the stock and the value at the time of executing the action. The last intance variable balance is there to model that we have limit funds and we cannot buy any stocks that has a higher cost than this.

To be able to make sure that the requirements are met at all times, several invariant are defined. First of all we need to make sure that the balance is never negative, since this means that we have bought a stock we couldnt afford. This is expressed in the following invariant:

Listing 8: Instance variables of the AutomatedStockBroker class

```
  inv balance >= 0;
```

The requirement that says that it is not allowed to have more than one stock with the same StockIdentifier in the system at a time, is expressed in the following invariant:

Listing 9: Instance variables of the AutomatedStockBroker class

```
  inv forall x,y in set inds stocks &
  x <> y => stocks(x).Name <> stocks(y).Name;
```

The requirement that says that for the same stock a buy action should allways be followed by a sell action and vice verca, is expressed in the following invariant

Listing 10: Instance variables of the AutomatedStockBroker class

```

inv let stockIdentifiers = {si.Name | si in set elems stocks}
  in
  forall stockIdentifier in set stockIdentifiers &
  let allEventsStock = [ actionLog(i) | i in set inds actionLog &
                        actionLog(i).StockName = stockIdentifier]
   in
   (forall e in set inds allEventsStock &
   (e <> len allEventsStock) =>
   (allEventsStock(e).Type <> allEventsStock(e+1).Type));
```

Informally explained this makes a set of all the stock identifiers and makes a sequence of all its log messages and checks to see if they changes from buy to sell and from sell to buy.

The last invariant in the class is for the **R2** requirement, which says that if event has the same time it must have af different action type:

Listing 11: Instance variables of the AutomatedStockBroker class

```
inv forall x,y in set inds actionLog &
 (x <> y and actionLog(x).Time = actionLog(y).Time) =>
 (actionLog(x).Type <> actionLog(y).Type);
```

4.1.1 Operations

From the requirements we can deduce that we at least needs the following operations

Operations

- **PerformBuy**: Performs a buy action on a stock if the conditions are as describe in requirement **R6**.
- **PerformSell**: Performs a sell action on a stock if the conditions are as describe in requirement **R7**.
- **GetStocksWithActiveActionTrigger**: Returns all the stocks with active ActionTriggers.
- **Step**: Move time one step forward.

We start out with the Step operation which fullfills the need of some notion of time that moves forward. It takes a time parameter, modelled as a basic nat type, which defines the current time in the entire system. A nat is sufficient in this model as no exact time is needed, a token could not be used here as it does not define the < and > operations which is needed. The Step Operation is shown below

Listing 12: The Step Operation

```
public Step: nat ==> ()
Step(time) == (

ObserveAllStocks(time);

let potBuys = GetStocksWithActiveActionTrigger(<PotentialBuy>) ,
    potSells = GetStocksWithActiveActionTrigger(<Bought>),
    validBuy = FindValidBuy(potBuys,time)
      in
      (
      if(len potSells > 0)
      then (PerformSell(FindValidSell(potSells,time), time);)
      else skip;
      if(validBuy <> nil)
      then (PerformBuy(validBuy, time);)
      else skip;
      );
      )
post MaxOneOfEachActionTypePerTime(actionLog);
```

The step operation calls the ObserveAllStocks which makes all StockWatchers update the all the actionTriggers for the defined time. The GetStocksWithActiveActionTrigger finds all the active by making a list comprehension that returns all the stocks that the StockWatcher has marked as active.

Listing 13: The GetStocksWithActiveActionTrigger Operation

```
GetStocksWithActiveActionTrigger:
StockState ==> seq of StockRecord
GetStocksWithActiveActionTrigger(ss) == (

return [stocks(s) | s in set inds stocks &
  (stockWatchers(stocks(s).Name).GetTriggeredAction() <> nil)
   and (stocks(s).State = ss)];
   )
post let res = RESULT in forall i in set inds res & res(i).State = ss;
```

This ensures that all the stocks that are delivered to FindValidSell and FindValidBuy are only stocks with an active ActionType of the right kind. The FindValid opertions take the last step of requirement **R6** and **R7**.

The findValidBuy starts by making a list comprehension with all the stock that has a cost lower than the current balance. From this list i finds the one with the highest cost. If nothing mets the requirements nil is returned.

Listing 14: The FindValidBuy Operation

```
FindValidBuy: seq of StockRecord * nat ==> [StockRecord]
FindValidBuy(potBuys,time) == (

return let affordableStocks =
[potBuys(x) | x in set inds potBuys & CanAfford(potBuys(x),balance)]
in
(
 if(len affordableStocks > 0)
 then let x in set inds affordableStocks be st
  (forall y in set inds affordableStocks &
(stockWatchers(affordableStocks(x).Name).
GetStockValue(time)) >=
 (stockWatchers(affordableStocks(y).Name).
GetStockValue(time)))
 in affordableStocks(x)
 else nil
);
);
```

The findValidSell finds the stock with the highest gain.

Listing 15: The FindValidSell Operation

```
FindValidSell: seq of StockRecord * nat==> StockRecord
FindValidSell(potSells,time) == (
 return
 (let x in set inds potSells be st
 (forall y in set inds potSells &
  (stockWatchers(potSells(x).Name).GetStockValue(time)
   - potSells(x).Cost) >=
  (stockWatchers(potSells(y).Name).GetStockValue(time)
   - potSells(y).Cost))
   in potSells(x));
   )
  pre len potSells > 0
  post
IsGTAll(stockWatchers(RESULT.Name).GetStockValue(time)
- RESULT.Cost,
  {stockWatchers(x.Name).GetStockValue(time)
- x.Cost | x in set elems potSells});
```

4.2 The StockWatcher Class

The StockWatcher class models the part that finds out if the stock has an active Action-Trigger or not.

Listing 16: Instance variables of the StockWatcher class

```
class AutomatedStockBroker is subclass of GLOBAL
...
 instance variables
  eventHistory : seq of Event;
  stockRecord : StockRecord;
  sm : [StockMarket];
  currentlyTriggeredAction : [ActionType];
...
```

As seen there some of the types which have not been discused, which is used in the validation by simulating stock values which will be mentioned breifly in section 5. The main purpose of this class is to model whether a stock has an active trigger or not. So one could say that it is not needed at all given the purpose of the model, since we could just pass a configuration of Stocks. But this choise is made because the wish to make more realistic simulations of the model

5 Validation Of The Model

The model is both validated through syntax check and by simulation of the model, where the Stock class and a StockMarket class provides the system with randomly generated Events. Is is also validated through predefined Events were an expected outcome anticipated. An example an be seen here where three stocks are added to the system.