EasyMSXDoc Documentation

Release 2.1

Terrence C. Kim

Jul 10, 2019

Table of Contents:

The EasyMSX code samples are various projects to demonstrate how Bloomberg EMSX API along with other Bloomberg API and industry standard algorithms can be implemented for building and implementing an efficient trading environment.

Important: Please note that this is not a compiled binary. This is a CODE SAMPLE. There is no SLA (Service Level Agreement) or quality assessment or guarantees.

Warning: Please do not reach out to Bloomberg for support or help.

CHAPTER 1

Introduction

The EasyMSX wrappers consists of various code samples that can be compiled into a library. The EasyMSX are set of files to build a project that demonstrates one possible way to build caching data or a starting place to work with Bloomberg EMSX API and other Bloomberg APIs.

The EMSX API is just another service in Bloomberg API V3. The Bloomberg API follows an event-driven asynchronous API paradigm.

The Bloomberg API is lightweight, thread safe and maintains extensible service-oriented data model. The Bloomberg API generally understands the concept of subscription and request-response services.

CHAPTER 2

Projects

There are currently three active projects written in C Sharp and Python. Other programming languages and projects will follow shortly.

2.1 Prerequisites

- All C# projects requires .NET 4.0 but has no other external dependencies.
- The python projects runs in Python 3.

2.2 EasyMSX

The EasyMSX allows getting orders, routes, and static data from EMSX API service. The EasyMSX allows adding notification handler on the real-time events. There is an observer pattern that can throw exceptions.

2.2.1 Installing

- For C#, Download the source code and build the library from the source. Once the Bloomberg API SDK has been referenced in your project, create an instance of EasyMSX:-
- For python, create the new directory and extract easymsx-1.0.x into the new directory. Change the directory to easymsx-1.0.x and in the directory run the following command:-

```
C:\Users\Me\>cd easymsx-1.0.x

C:\Users\Me\easymsx-1.0.x>C:\Python34\python.exe setup.py install
```

- Run easymsxdemo.py

```
C:\Users\Me\>cd easymeasymsx-1.0.x

C:\Users\Me\easymsx-1.0.x>py -3 easymsxdemo.py
```

- The GitHub link to the EasyMSX sample in EasyMSX.

2.3 EasyMKT

The EasyMKT and EasyMKTSample are essentially an EasyMSX for market data where the project demonstrates one possible way to build caching data on Bloomberg real-time market data.

2.3.1 Installing

- For C#, Download the source code and build the library from the source. Once the Bloomberg API SDK has been referenced in your project, create an instance of EasyMKT:-
- For python, create the new directory and extract easymkt-1.0.x into the new directory. Change the directory to easymkt-1.0.x and in the directory run the following command:-

```
C:\Users\Me\>cd easymkt-1.0.x

C:\Users\Me\easymkt-1.0.x>C:\Python34\python.exe setup.py install
```

- Run easymktdemo.py

```
C:\Users\Me\>cd easymsx-1.0.x

C:\Users\Me\easymkt-1.0.x>py -3 easymktdemo.py
```

- The GitHub link to the EasyMKT sample in EasyMSX.

2.4 RuleMSX

Rather than making use of domain specific language (DSL) to define the rules, it exposes a series of abstract classes. It facilitates the creation of complex if-this-then-that behavior.

2.4.1 Installing

- For C#, Download the source code and build the library from the source.
- For python, create the new directory and extract easymsx-1.0.0 and rulemsx-1.0.0 into the new directory.

```
C:\Users\Me\_rulemsx>dir
Volume in drive C is Windows
Volume Serial Number is ABCD-1234

Directory of C:\Users\Me\_rulemsx

12/21/2017  09:08 AM    <DIR>          .
12/21/2017  09:08 AM    <DIR>          ..
12/21/2017  09:01 AM    <DIR>          easymsx-1.0.0
```

(continues on next page)

(continued from previous page)

```
12/21/2017  09:01 AM    <DIR>          rulemsx-1.0.0
12/21/2017  09:01 AM    <DIR>          RuleMSXDemo.py
               1 File(s)              0 bytes
               4 Dir(s)  11,538,878,464 bytes free
```

- Change the directory to rulemsx-1.0.0 and in the directory run the following command:-

```
C:\Users\Me\_rulemsx>cd rulemsx-1.0.0

C:\Users\Me\_rulemsx\rulemsx-1.0.0>C:\Python34\python.exe setup.py install
```

- Please make sure the path for python is set to where you currently have your python 3 installed. Change directory to easymsx-1.0.0 and in the diretory run the following command:-

```
C:\Users\Me\_rulemsx>cd easymsx-1.0.0

C:\Users\Me\_rulemsx\easymsx-1.0.0>C:\Python34\python.exe setup.py install
```

- Run RuleMSXDemo.py

```
C:\Users\Me\_rulemsx>py -3 RuleMSXDemo.py
Initialising RuleMSX...
RuleMSX initialised...
Initialising EasyMSX...
EasyMSX initialised...
Create RuleSet...
Building Rules...
Rules built.
RuleSet ready...
Press any to terminate
```

2.4.2 Getting Started

The following is the C# implementation of the RuleMSX sample. RuleMSX provides the core functionality of a rule engine. Once the library has been referenced in your project, create an instance of RuleMSX:-

```
RuleMSX rmsx = new RuleMSX();
```

RuleMSX is divided into 'Rules'_, 'DataPoints'_ and 'Actions'_. Rules are organized into 'RuleSets'_:-

```
RuleSet myRuleSet = this.rmsx.CreateRuleSet("MyRuleSet");
```

A RuleSet contains one or more Rules, and each Rule is made up of one or more RuleConditions. Each RuleCondition has a RuleEvaluator which is the code written by the developer. Each rule also has one or more RuleAction associated with it. When all the RuleConditions are met, the RuleAction is excuted.

To create a Rule:-

```
Rule myNewRule = myRuleSet.AddRule("NewRule");
```

To create a RuleCondition:-

```
RuleCondition myRuleCondition = new RuleCondition("MyConditoin", new␣
↪MyCondtionCode());
```

The 'MyConditionCode' class extends the RuleEvaluator abstract class, guarenteeing the presence of an Evalute() method. This method must return a boolean value.

For example:-

```
class MyConditionCode : RuleEvaluator
{
    public MyConditionCode()
    {
        // constructor code
    }

    public override bool Evalute(DataSet dataSet)
    {
        if(<sometest>) {
            return True;
        }
        else
        {
            return False;
        }
    }
}
```

Add the RuleCondition to the Rule:-

```
myNewRule.AddRuleCondtion(myRuleCondition);
```

Alternatively:-

```
myRuleCondition.AddRuleConditino(new RuleCondition("MyCondition", new
↪MyConditionCode()));
```

When the RuleEvaluator of each of the RuleConditions Associated with a Rule return True, then any Actions associated with the Rule will be fired.

Actions are created independently of a Rule, so that a single action can be reused across multipel Rules, An action consists of a Rule object, and an associated RuleEvaluator which is extended by the developer.

To create an Action:-

```
Action myAction = rmsx.CreateAction("MyAction", new MyActionCode());
```

The 'MyActionCode' class extends the ActionExecutor abstract class, guarenteeing the presence of an Execute() method.

For example:-

```
class MyActionCode: ActionExecutor
{

    public MyActionCode()
    {
        // constructor code
    }

    public void Execute(DataSet, dataset)
    {
        // do something here
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Add the Action to the Rule:-

```
myNewRule.AddAction(myAction);
```

Alternatively:-

```
myNewRule.AddAction(rmsx.CreateAction("MyAction", new MyActionCode()));
```

The data to be processed is a RuleSet is defined as 'DataPoints'_, which are organized into 'DataSets'_.

A DataPoint is a single named item of data that has an assocated DataPointSource. The DataPointSource is an abstract class that the developer extends, which guarentees teh presense of a GetValue() method. Think of the DataSet as an object with properties. Think of the DataSet as a collection of DataPoints, each of which is a key-value pair.

You submit a DataSet for execution by a RuleSet's execution agent, as follows:-

```
myRuleSet.execute(myDataSet);
```

To create a DataSet:-

```
DataSet myDataSet = rmsx.CreateDataSet("<some unique name>");
```

To create a DataPoint, you first need to create a DataPointSource. This is done by creating a class that extends DataPointSource:-

```
private class ConstantDataPointSource : DataPointSource
{
    string retValue;

    public TestDataPointSource(string retValule)
    {
        this.retValue = retValue;
    }
    public override object GetValue()
    {
        return retValue;
    }
}
```

An instance of thi sclass will return the value that was passesd to the constructor whenever the GetValue() method is called.

Create the DataPoint as follows:-

```
DataPoint myDataPoint = new ConstantDataPointSource("Return this!");
```

Add the DataPoint to the DataSet:-

```
myDataSet.AddDataPoint("DataPoint1", myDataPoint);
```

Alternatively:-

```
myDataSet.AddDataPoint("DataPoint1", new ConstantDataPointSource("Return this!"));
```

2.4.3 Operation

The execution agent that underlies a RuleSet operates in its own thread. When a RuleSets Execute() method is first invoked, the execution agent is created. Thereafter, any further calls to Execute() will result in the DataSet simply being passed to the already running agent.

When a DataSet is ingested by the execution agent, all the Rules will be tested. Once a rule is tested, it will not be tested again, unless it is re-introduced. This happesn when a RuleCondition whin the rule has a delcared dependency on a DataPoint whos DataPointSource has been marked as stale. This is done on the client side, by calling SetStale() on a DataPointSource object. Any Rule that has a dependency on that DataPoint will be re-introduced into the queue of Rules to be tested.

This means that RuleCondition can be created that depends on the value of a variable or field that will change over time. When the rule is fist tested, perhpas the value is in a state that means that the Evaluate() method will return False. However, it may change later. The rule will not be tested again under normal circumstances. But if the variable or field changes values, simply call the SetStale() method on the DataPointSource object, and any and all Rules which have a RuleCondition that depends on its value will be re-tested. This means that the RuleCondition may now return True, and the associated ActionExecutor of Rule will be fired.

2.4.4 Tests

NUnit unit tests, as well as integration tests, are included in the project.

2.4.5 Deployment

Simply distribute the library with any application distribution.

2.4.6 License

This project is under the MIT License - see the License file for details.

2.5 EMSX API in Julia language

EMSX API in Julia Programming Language coming soon!

CHAPTER 3

Resource

3.1 EMSX Route Status

Field	Previous Value	New Value	Definition
EMSX_STATUS	null	SENT	New route (placement) created.
EMSX_STATUS	SENT	SENT	Field update on sent.
EMSX_STATUS	SENT	WORKING	ACK received from the broker.
EMSX_STATUS	WORKING	PARTFILL	First fill or multiple fills. (<100%)
EMSX_WORKING	n	<n and >0	
EMSX_STATUS	PARTFILL	PARTFILL	Middle fill or multiple fills. (<100%)
EMSX_WORKING	n	<n and >0	
EMSX_STATUS	PARTFILL	FILLED	Final fill or multiple fills. (100%)
EMSX_WORKING	>0	0	
EMSX_STATUS	WORKING	FILLED	Full single fill.
EMSX_WORKING	>0	0	
EMSX_STATUS	null	FILLED	Historic 100% fill on INIT_PAINT.
EMSX_STATUS	null	WORKING	Working route (placement) on INIT_PAINT.
EMSX_STATUS	null	PARTFILL	Part filled route (placement) on INIT_PAINT.
EMSX_STATUS	null	CXLREQ	Cancel requested on route in INIT_PAINT.
EMSX_STATUS	WORKING	CXLREQ	Cancel route request sent.
EMSX_STATUS	CXLREQ	WORKING	Broker rejected cancel request.
EMSX_STATUS	CXLREQ	CXLPEN	Broker sent ACK for cancel request.
EMSX_STATUS	CXLPEN	WORKING	Broker rejected cancel request.
EMSX_STATUS	CXLREQ	CANCEL	Broker cancelled route from request.
EMSX_STATUS	CXLPEN	CANCEL	Broker cancelled route from request.
EMSX_STATUS	PARTFILL	CXLREQ	Cancel requested on part filled route.
EMSX_STATUS	CXLREQ	PARTFILL	Broker rejected cancel request.
EMSX_STATUS	CXLPEN	PARTFILL	Broker rejected cancel request.
EMSX_STATUS	WORKING	CXLRPRQ	Modify (cancel/replace) request sent to broker.
EMSX_STATUS	CXLRPRQ	REPPEN	Broker sent ACK for modify request.

Continued on next page

Table 1 – continued from previous page

Field	Previous Value	New Value	Definition
EMSX_STATUS	REPPEN	WORKING	Broker rejected modify request on working route.
EMSX_BROKER_STATUS	n/a	CXRPRJ	
EMSX_STATUS	REPPEN	WORKING	Broker accepted and applied the modify request on working route. (placement)
EMSX_BROKER_STATUS	n/a	MODIFIED	
EMSX_STATUS	PARTFILL	CXLRPRQ	Modify (cancel/replace) request sent to broker.
EMSX_STATUS	REPPEN	PARTFILL	Broker rejected modify request on part filled route. (placement)
EMSX_BROKER_STATUS	n/a	CXRPRJ	
EMSX_STATUS	REPPEN	PARTFILL	Broker accepted and applied the modify request on part filled route. (placement)
EMSX_BROKER_STATUS	n/a	MODIFIED	
EMSX_STATUS	SENT	REJECTED	Broker rejected the order from sent status.
EMSX_STATUS	null	REJECTED	INIT_PAINT shows route (placement) rejected.
EMSX_STATUS	null	CANCEL	INIT_PAINT shows route (placement) cancelled.
EMSX_STATUS	CXLRPRQ	WORKING	Modify rejected from request.
EMSX_STATUS	PARTFILL	CANCEL	Part filled route cancelled by broker.
EMSX_STATUS	WORKING	CANCEL	Working route cancelled by broker.
EMSX_STATUS	WORKING	REJECTED	Route rejected from working.

3.2 RuleMSX

The RuleMSX is a business rule management project using Bloomberg EMSX API for trading, Bloomberg Market Data for market data and rete algorithm for efficient business rule management.

RuleMSX provides the core functionality of a rule engine. It is designed to inter-operate with the `EasyMSX` and `EasyMKT` which use the Bloomberg API to access Bloomberg EMSX and market data. The RuleMSX is designed to use the rete algorithm to create an efficient business rule management system to work with Bloomberg EMSX API for automated trade execution for equities, futures, and options.

This functionality is provided in the shape of `RuleSets`, `DataSets` and `Actions`. By defining Rules and the conditions that must exist for these Rules to be triggered, the user can build complex reasoning based on the content of a DataSet, and how that DataSet changes over time. The Actions are the tasks performed as a result of a Rule being triggered.

3.2.1 .NET Reference for RuleMSX Project

```
Bloomberg API SDK in CSharp
(e.g. c:\blp\DAPI\APIv3\DotnetAPI\v3.8.9.2\lib\Bloomberglp.Blpapi.dll)

EasyMKT.dll
(e.g. c:\... \cs_EasyMKT-master\EasyMKT\bin\Debug\EasyMKT.dll)

EasyMSX.dll
(e.g. c:\... \cs_EasyMSX-master\EasyMSX\bin\Debug\EasyMSX.dll)

RuleMSX.dll
(e.g. c:\... \cs_RuleMSX-master\RuleMSX\bin\Debug\RuleMSX.dll)
```

3.2.2 ActionExecutors

An `ActionExecutor` is the client-side code that is run when an Action is executed. It is an abstract class that contains an Execute method that must be overridden.

When the `RouteOrdertoBB` action is executed, the Execute method of the instance of the abstract class would be called. This is the code that would create and send the route to the broker. Just as with the `RuleCondition` evaluators, the executors are passed the current dataset as a parameter when they are called.

3.2.3 DataPoints

A `DataPoint` is an object that represents a single piece of data. Fundamentally, it is a simple key-value pair. A `DataPoint` doesn't have value itself, but rather has an underlying `DataPointSource` which is used to provide the value.

Examples of DataPoints would be OrderNumber, OrderStatus, OrderExchange, etc.

3.2.4 DataPointSource

A `DataPointSource` is a client-side code that provides a value for a named `DataPoint`. It is an abstract class with a `GetValue` method that must be overridden. It also provides a `SetStale` method that is used to indicate to the `ExecutionAgent` that the value must be re-examined. This will cause any `WorkingRules` for Rule that has a dependency on this `DataPoint` to be the queue for re-evaluation on the next cycle.

The `DataPointSources` for the above example `DataPoints` would access the `EMSX` data to return the correct `EMSX_SEQUENCE` and `EMSX_STATUS`, and perhaps use the reference data service to get the exchange code for the ticker on the order.

3.2.5 DataSets

`DataSets` are named entities that represent a collection of `DataPoint` objects. They are only used to organize `DataPoints` into logical groupings.

In our current example, we would create a `DataSet` object for each order. Once the `DataSet` object is defined, we can add it to the list of `DataSets` being run through a `RuleSet` by the `ExecutionAgent`.

3.2.6 ExecutionAgent

When the application has completed the configuration of all the main elements (`Rules`, `RuleConditions`, `Evaluators`, `Action`, `Executors`, and etc.), one or more `RuleSets` can be executed.

This involves taking a `DataSet` and asking the `RuleSet` to be executed against that DataSet: -

```
myRuleSet.Execute(dataSet_1);
```

If this is the first time this `RuleSet` has been executed, a new `ExecutionAgent` will be created for the `RuleSet`. If the `RuleSet` already has an `ExecutionAgent`, it will be reused. The specified `DataSet` is then passed to the `RuleSet`'s `ExecutionAgent`: -

```
executionAgent = new ExecutionAgent(myRuleSet, dataSet_1);
```

or

```
executionAgent.AddDataSet(dataSet_1);
```

Each ExecutionAgent has a DataSetQueue. Adding a DataSet to an ExecutionAgent simply adds the DataSet reference into the DataSetQueue. This is used to ensure that new DataSets are only ingested at the correct time, and not at the mid-point of a cycle.

A new ExecutionAgent will create a new internal thread that will operate a WorkingSetAgent. This WorkingSetAgent is the main loop that controls the execution of the rules and actions for a RuleSet, and it continues to run until stopped by an external request (a call to the stop() method).

Each cycle of the WorkingSetAgent begins with ingesting any DataSets in the ExecutionAgent's DataSetQueue. This is the process of creating a WorkingRule for each Rule in the RuleSet and the specified DataSet.

To create a WorkingRule, a Rule and a DataSet are required. A process known as dereferencing takes place, which has two steps. The first step is to take each Action associated with the Rule and add the ActionExecutor references to the WorkingRule's Executors collection.

The second part of the dereferencing process is to iterate each RuleCondition of the Rule, and add it's RuleEvaluator to the Evaluators collection of the WorkingRule. Each RuleEvaluator has a collection of DataPoint names that it depends on. For each of these dependant data point names, we find the actual DataPoint in the DataSet that matches the name. The WorkingRule is then added to the AssociatedWorkingRules collection of the DataPoint's DataPointSource object.

The reason for doing this is that when a DataPointSource's value changes, its SetStale() method is (should be) fired. This forces each WorkingRule dependency of the DataPointSource to be added to the OpenSetQueue in the WorkingSetAgent for execution in the next cycle, unless the WorkingRule is already in the OpenSetQueue.

Following the ingestion process, the current OpenSetQueue becomes the OpenSet, and the OpenSetQueue is then reset to empty. The OpenSet is now iterated, and each WorkingRule in the queue is processed. Each Evaluator in the WorkingRule is fired, passing it the WorkingRule's DataSet. If all Evaluators in the WorkingRule return true, then the action process begins. Each action associated with the WorkingRule is executed.

3.2.7 RETE Algorithm

The word rete is Latin for net or network. The rete algorithm is essentially a pattern matching algorithm.

The main objective behind rete algorithm for RuleMSX is to decouple the various trading or business rules from rule execution or executing sequences on a particular data set.

The data set here can be both trading data obtained from EMSX API, market data, or non-trading based proprietary data set.

The RuleMSX views each rule exists as a stand-alone rule that is either true or false at any given moment.

A pattern contains one or more rules. All the rules in a pattern must evaluate to true for the action attached to the pattern to be executed. In this case, the action itself is responsible for introducing the new rules to be checked and/or new patterns or patterns to be removed from the set.

3.2.8 Rules

Each Rule in a RuleSet is a named collection of RuleConditions and RuleActions. When all conditions in a Rule evaluate to True, the associated actions are executed.

Following the above example, and single rule within the AutoRoute ruleset would be `RouteUStoBB`, which would route any orders on the `US` exchange code to the broker known as `BB`. The other rule example could be `RouteLNtoBMTB`.

3.2.9 RuleActions

A Rule can have many `RuleActions`. Each `RuleAction` has a client-side component called an''ActionExecutor''. When a Rule evaluates to True, all associated `RuleActions` are executed.

For example, we would have a `RuleAction` called `RouteOrdertoBB`. which would be called as a consequence of the `RouteUStoBB` rules all evaluating to True.

3.2.10 RuleConditions

A `RuleCondition` is a named item within a Rule, which evaluates to either True or False. It does this through client-side code using a `RuleEvaluator`. A single Rule can have multiple `RuleConditions`, and they must all evaluate to True for the associated `RuleActions` to be executed.

For our `RouteUStoBB` example, we would have a condition called `MustBeUSExchange` that checked the order to ensure that it was for the `US` exchange. Another condition would be that the order must be in a `NEW` state, perhaps called `CheckNEWState`, to ensure that this rule is only triggered once.

3.2.11 RuleEvaluator

A `RuleEvaluator` is an abstract class that must be implemented in the client-side code. This abstract class has an `Evaluate` method that must be overridden. This method must return `True` or `False`. When the Evaluate method is called, it is passed the current DataSet as a parameter, to support the determination of the return value.

The abstract class also provides a mechanism for creating a dependency between a Rule and named DataPoints. To do this, we call the `AddDependantDataPointName` method of the class, as follows :-

```
this.AddDependantDataPointName("OrderStatus")
```

In this case, we are saying that this particular Rule uses the value of the `OrderStatus` `DataPoint`. The purpose of using this mechanism is to ensure that if the value of `OrderStatus` in any `DataSet` changes, any `WorkingRules` add queue to be re-tested in the next cycle. The change to the value of a `DataPoint` is indicated by calling the `SetStale` method (see `DataPointSource`).

3.2.12 RuleSets

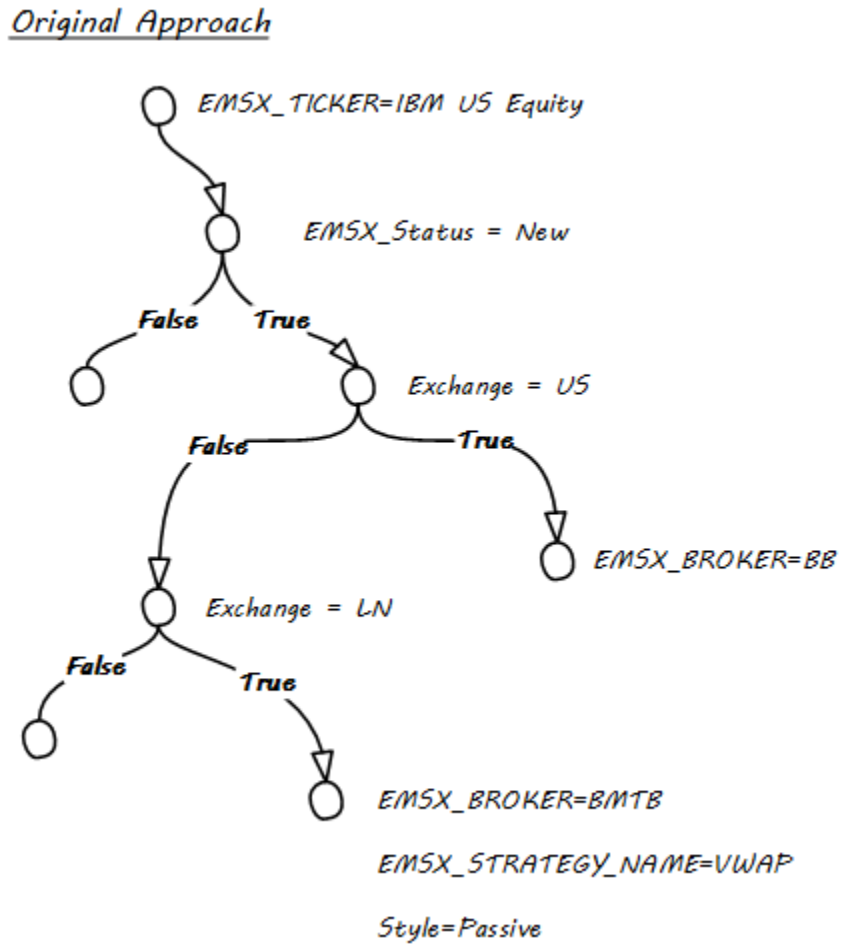
RuleSets are named entities that represent a collection of RuleSet objects. This is only used to organize rules into logical groupings. A RuleSet is a named collection of Rules.

An example of a RuleSet would be to route new orders to a particular broker code, based on certain criteria, such as the exchange. We will call this the “AutoRoute” ruleset.

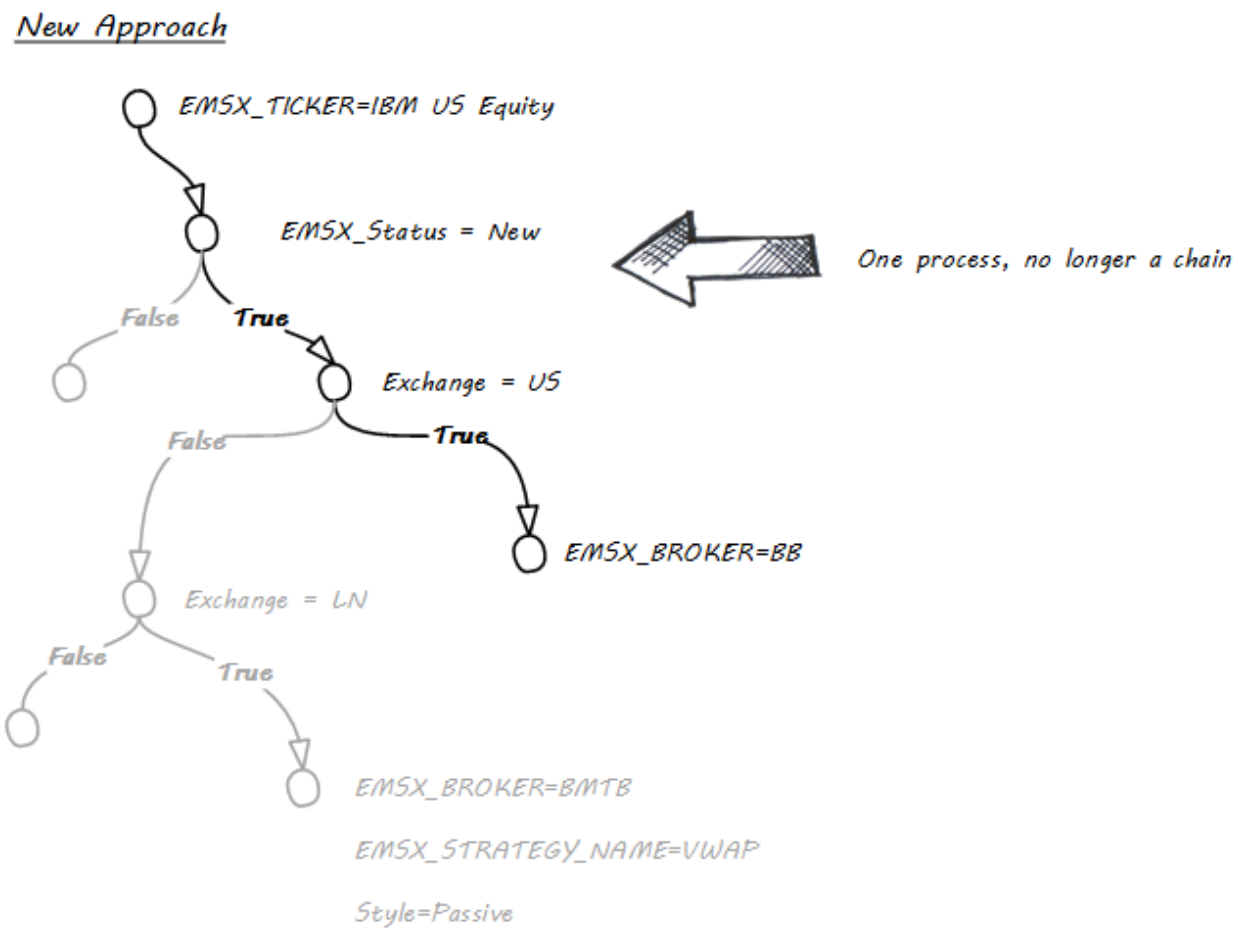
Once we have a RuleSet and a DataSet object, we can execute the RuleSet. RuleSets need one or more supporting DataSets to operate against.

3.3 Earlier Version

The initial approach to RuleMSX handled the rete in the following structure where each RuleSet consists of a single rule. Each rule consisted of child rules and rule evaluator.



As part of the reiteration of RuleMSX, we have made the changes to reflect the rete algorithm in the following structure:



3.4 Full Code Samples

3.4.1 Full EasyMSX Code Samples

The link to the main EasyMSX Code Sample.

3.4.2 Full EMSX API Documentation

The link to the main EMSX API Documentation.

3.4.3 Full EMSX API Code Samples

The github link to the EMSX API Code Sample.

3.4.4 Full Bloomberg API Developer Guide

The link to the Open API Core Developer Guide.

CHAPTER 4

License