Zigzag Fractal DNA in Forex Time Series

Technical Documentation

Concepts:
gg53
crodzilla
smallcat
Kilian19
go4it
emonts
burnssss
candero
and many others

Author:
flx23

February 22, 2014

Abstract

This document summarizes the main concepts of a swing point prediction approach for forex time series using fractal patterns within zigzag legs which evolved from the "The Ultimate Truth" thread[1] on forexfacory.com. For further documentation and contributions please have a look at the "FractalPattern, ZZ & the Pissing Dog" thread [2].

This document is - just as the approach itself - a work in progress. Some sections may be erroneous or incomplete or may even be conflicting with the initial approach. However, the goal is to describe all preliminary findings as precise and unambiguous as possible. For this reason all advices and improvement suggestions are highly welcome. If you like to improve this document please feel free to contact me[3].

Particular thanks to all contributors!

[1] http://www.forexfactory.com/showthread.php?t=440372
[2] http://www.forexfactory.com/showthread.php?t=464948
[3] http://www.forexfactory.com/flx23

Contents

1 Introduction

"*Unlike gambling, lottery, Coin toss, etc. which are (hopefully) purely random - the Forex market is biased. It does have a "memory" and it's based on needs, speculation, and human behavior. [...] We know that the zigzag indicator (ZZ) best describe those swing points, but it does it in retrospect. For real-time trading it is lagging and repainting itself making it quite useless for accurate entries and exit in trading. We came out with an idea to analyze each ZZ leg by its fractals and regular bars in an attempt to find consistent, repeatable patterns. If we can find those patterns - we can predict where the ZZ leg will, or should be. [...]*
Why fractals? The whole idea of fractals is about patterns that repeat themselves on larger and smaller scales. If the fractals, or DNA, theories are valid - the DNA sequences should represent themselves within any shape and form. Since the smallest sequences that we can analyze are the 1 Min. bars - we need to compare their sequences to larger patterns and "filter out" the wrong sequences that doesn't exist in the larger patterns. Hence the choice of the ZZ leg - to represent the larger pattern that have smaller "sequences" - its fractals. [...]" - gg53

1.0.1 References

- Entire introduction: `http://www.forexfactory.com/showthread.php?p=7188904#post7188904`

2 Historical Data

Although the approach described in this article should work on different currency pairs and with different time frames we focus for reasons of comparability on M5 data of EURUSD within the two year period 2011 - 2012. Smallcat provided a zip file[4] containing detailed instructions on how to obtain historical data. This file also includes modified zigzag and fractal indicators which are required for this approach and will be presented in detail later on. Historical data can be exported using JForex[5] just as well as suggested by killerno.

[4]`http://www.forexfactory.com/attachment.php?attachmentid=1341201&d=1389043789`
[5]`https://www.dukascopy.com/europe/deutsch/forex/jforex/`

3 Fractals

FractalPattern in terms of this approach are reversal points indicated by candle patterns of lengths 5-9. A series of at least five candles is called an up fractal if it has the highest high in the middle and two lower highs on both sides. A series with the lowest low in the middle of two higher lows is called a down fractal respectively. Therefore a fractal appears two bars lagged. The following definition is acausal which means that the indexes $i-1$ and $i-2$ refer to future bars or - interpreted differently - an indicator using this definition is repainting. In order to obtain a causal - or non-repainting - implementation incoming bars must be delayed for two time steps.

Definition 1 Up fractals

```
// length 5
High[i]>High[i+1] && High[i]>High[i+2] &&
High[i]>High[i-1] && High[i]>High[i-2]
// length 6
High[i]==High[i+1] && High[i]>High[i+2] && High[i]>High[i+3] &&
High[i]>High[i-1] && High[i]>High[i-2]
// length 7
High[i]>=High[i+1] && High[i]==High[i+2] && High[i]>High[i+3] &&
High[i]>High[i+4] &&
High[i]>High[i-1] && High[i]>High[i-2]
// length 8
High[i]>=High[i+1] && High[i]==High[i+2] && High[i]==High[i+3] &&
High[i]>High[i+4] && High[i]>High[i+5] &&
High[i]>High[i-1] && High[i]>High[i-2]
// length 9
High[i]>=High[i+1] && High[i]==High[i+2] && High[i]>=High[i+3] &&
High[i]==High[i+4] && High[i]>High[i+5] && High[i]>High[i+6] &&
High[i]>High[i-1] && High[i]>High[i-2]
```

On the analogy of up fractals, down fractals are defined by the low prices and hence only specified for length 5.

Definition 2 Down fractals

```
// length 5
Low[i]<Low[i+1] && Low[i]<Low[i+2] &&
Low[i]<Low[i-1] && Low[i]<Low[i-2]

...
```

3.0.2 References

- Why using fractals? `http://www.forexfactory.com/showthread.php?p=7188904#post7188904`
- MT4 fractals implementation: `http://ta.mql4.com/indicators/bills/fractal`

Note: Does anyone use other / modified / additional fractal definitions?

4 Zigzag Indicator

The zigzag indicator is a series of trend lines connecting significant peaks and foundations. Minimum price change parameter determines the percentage for the price to move in order to form a new "Zig" or "Zag" leg. This indicator eliminates those changes that are less than the given value. Therefore the Zigzag reflects "significant" changes only. It is important to note that this indicator is repainting undetermined in terms of lag. I.e. the last section of the indicator may vary depending on the changes of data. In our approach we use the zigzag indicator for the decomposition of our historical data retrospectively. A zigzag leg connects by definition two major swing points and can thus be a characteristic, higher level unit of candle data. Since we intend to analyze fractal patterns (which will be introduced in the next section) within zigzag legs the standard implementation must be synchronized with fractal bars. Therefore we delay an arising leg endpoint until the next fractal is emerging. A different method has been introduced by Kilian19. It has been considered appropriate to choose $depht = 13$, $deviation = 8$ and $backstep = 5$ as zigzag parameter set.

4.0.3 References

- Background - Zigzag `http://www.forexfactory.com/showthread.php?p=7188904#post7188904`
- MT4 zigzag implementation: `http://codebase.mql4.com/265`
- Zigzag indicator by Kilian19 `http://www.forexfactory.com/attachment.php?attachmentid=1341201&d=1389043789`

5 Data sets

Initially, the historical bar series must be sequenced in a long bit string, for instance {0101011...}. Each bar is stored as "0" or "1" depending on if it is a down bar, a up bar or a neutral doji[6] bar. In the latter case the current bar is treated as the previous one. The following code specifies the procedure for a bar at position i.

Algorithm 3 Bar sequencing

```
if (Close[i] < Open[i])
  bitString.Add(1); // Up bar
else if (Close[i] > Open[i])
  bitString.Add(0); // Down bar
else
{
  if (Abs(Open[i] - High[i]) > Abs(Open[i] - Low[i]))
    bitString.Add(1); // Up doji bar
  else if (Abs(Open[i] - High[i]) < Abs(Open[i] - Low[i]))
    bitString.Add(0); // Down doji bar
  else
    bitString.Add(bitString.Last); // Neutral doji bar
}
```

[6] `http://en.wikipedia.org/wiki/Doji`

5.1 Set 1

Set 1 consists of all continuous sub-strings of lengths 4-13 occurring in the original bit string, henceforth referred to as (4-13)-sub-strings.
These sub-strings can be found by iterating over each index i of the original bit string with length of N and storing all sub-strings (i, j) where $j = i + length - 1 < N$ for each $length \in [4; 13]$.

Given an original bit string $\{011001\}$. All possible (4-13)-sub-strings are then:
0110, 1100, 1001 - length 4
01100, 11001 - length 5
011001 - length 6

If an encountered sub-string is already stored in Set 1 its occurrence counter must be increased. Therefore the structure of Set 1 can be described as a dictionary mapping (4-13)-sub-strings keys to their frequency of occurrence within the original bit string:

Definition 4 Set 1 data structure

```
Set1 := Dictionary<Key:BitString, Value:Integer>
```

The following code illustrates the procedure.

Algorithm 5 Generating (4-13)-sub-strings

```
for (var i = 0; i <= bitString.Count - 4; i++)
{
  for (var length = 4; length <= 13; length++)
  {
    if (i + length - 1 >= bitString.Count)
      break;

    var subBitString = new BitString(length);

    for (var j = 0; j < length; j++)
      subBitString[j] = bitString[i + j];

    if (Set1.ContainsKey(subBitString))
      Set1[subBitString]++; // Increment occurrence counter
    else
      Set1.Add(subBitString, 1); // Add new sub-string with value = 1
  }
}
```

Since a bit string with length of k can represent 2^k distinct patterns there are $\sum_{k=4}^{13} 2^k = 16368$ (4-13)-sub-strings at the maximum.

5.1.1 References

- Set 1 `http://www.forexfactory.com/showthread.php?p=7188904#post7188904`

5.2 Set 2

Set 2 will store bit string keys as well but unlike Set 1 these keys are now the fractal patterns of each zigzag leg in the historical bar series. Fractal patterns are defined as a ternary-encoded sequence of up and down fractals encountered within each zigzag leg. Down fractals are marked as "0", up fractals as "1" and up and down fractals both occurring at the same bar are marked as "2". Suppose a leg contains two down fractals followed by one up fractal and a twin fractal. The fractal pattern is then encoded as $\{0012\}$. Since we like to use a binary encoding, equally to sub-string patterns, twin fractals must be unfold to a sequence $\{01\}$ or $\{10\}$ depending on which fractal occurred earlier. Crodzilla suggested the following method:

Algorithm 6 Unfolding twin fractals

```
if (fractalSequence[i] == 2) // Twin fractal?
{
  if (bitString[i] == 0) // Down bar in original bar bit string?
    fractalPattern.Add({1, 0});
  else
    fractalPattern.Add({0, 1});
}
```

For each zigzag leg we store a data record addressed by the its fractal pattern. This record contains both the length of the leg measured in original bars and the information if it is an up or down leg. Hence the structure of Set 2 is defined as follows:

Definition 7 Set 2 data structure

```
Set2 := Dictionary<Key:BitString, Value:List<Tuple<Integer, Bool>>>
```

Algorithm 8 Generating Set 2

```
// Get fractal and zigzag sequences
var fractalsSequence = FractalSequencer.ProcessAll(OHLCBars);
var zigzagSequence = ZzSequencer.ProcessAll(OHLCBars, depth,
    deviation, backstep);

var lastLegEndIdx = 0;
for (var i = 0; i < bitString.Count; i++)
{
  if (zigzagSequence[i] == -1)
    continue; // Skip bars that are not starting point of a new leg

  var fractalPattern = new BitString();
  for (var j = lastLegEndIdx; j < i; j++)
  {
    if (fractalSequence[j] == -1)
      continue; // Skip non-fractal bars

    if (fractalSequence[j] == 2) // Twin fractal? -> {01} or {10}
    {
      if (bitString[j] == 0) // Down bar in original bar bit string?
        fractalPattern.Add({1, 0});
      else
```

```
          fractalPattern.Add({0, 1});
        }
        else
          fractalPattern.Add(fractalSequence[j]); // Add up or down fractal
      }

      if (fractalPattern.Count > 0)
      {
        var barCount = i - lastLegEndIdx; // Calculate leg length
        var upOrDown = zigzagSequence[i]; // Up or down leg?

        if (Set2.ContainsKey(fractalPattern))
          Set1[fractalPattern].Add(new Tuple(barCount, upOrDown));
        else
          Set1.Add(fractalPattern, new List(new Tuple(barCount,
             upOrDown)));
      }

      lastLegEndIdx = i;
    }
```

5.2.1 References

- Set 2 `http://www.forexfactory.com/showthread.php?p=7189894#post7189894`

5.3 Set 3

Set 3 is the key-wise set-theoretic intersection of Set 1 and the (4-13)-sub-strings of all keys within Set 2. I.e. we create all (4-13)-sub-strings of each fractal pattern key contained in Set 2. For fractal patterns longer than 13 only the 13 right-most bits are relevant. Suppose a fractal pattern {110010101101010} with length of 15. The first two bits are ignored meaning that only {0010101101010} is used for (4-13)-sub-string generation. These sub-strings are then stored in an auxiliary filtering set. All record stored in Set 1 with keys matching those in the auxiliary filtering set are copied to Set 3. Or alternatively Set 3 can be seen as a copy of Set 1 where all records with keys not contained in the filtering set must be removed. Hence the structure of Set 3 is identical to that of Set 1. The following code illustrates the second method.

Algorithm 9 Generating Set 3

```
var auxFilteringSet = new Set1();
foreach (var fractal in Set2.GetAllKeys())
{
  // Take right-most fractal bits only
  var rightMostFractal = fractal.TakeLast(13);

  // Add all new sub-strings within rightMostFractal to auxFilteringSet
  foreach(var subString in CreateSubStrings(rightMostFractal, 4, 13))
    if (!auxFilteringSet.ContainsKey(subString))
      auxFilteringSet.Add(subString);
}

Set3 = Set1.Clone(); // Copy Set 1

// Filtering: Set3 = key-wise intersection(Set1, auxFilteringSet)
foreach (var key in Set1.GetAllKeys())
  if (!auxFilteringSet.Contains(key))
    Set3.Remove(key);
```

It should be noted that this method is also a filtering in terms of the theory of fractals. Set 1 consists of all sub-strings encountered in the historical raw data whereas Set 3 only contains those patterns occurring both on a bar level within the raw data and additionally on a higher fractal level within zigzag legs. Therefore we expect Set 3 to contain relevant patterns filtered from noises, overshooting and other mismatches occurring in the raw data.

5.3.1 References

- Set 3 `http://www.forexfactory.com/showthread.php?p=7189951#post7189951`

5.4 Set Data Statistics

For reasons of comparability the sizes of Sets 1-3 generated from historical EURUDS data (M5, 2011/01/01 - 1012/12/31) using the zigzag standard parameters (13, 8, 5) are listed below:

- Set 1: 16368 distinct bar patterns / keys
- Set 2: ca. 990 distinct fractal patterns / keys, ca. 8000 zigzag legs total
- Set 3: ca. 2000 distinct filtered bar patterns / keys

5.5 Implementation Aspects

Please note that the foregoing data structures and algorithms are not necessarily much sophisticated or efficient. However, this subsection is primary concerned with aspects of implementation and efficiency.
Go4it provided documents, tools[7] and interfaces[8] for accessing a SQL database from MT4. It should be mentioned that this method is - although it might be convenient - extremely inefficient and thus consumptive of time. Whenever possible, data structures should be kept internally within the storage area of a single application. In the case of an external SQL database the total delay is caused by the query synthesis, the communication delay between the application and the data base server, the search request itself and the decoding of the results. Since databases are internally organized as tree structures a search request taken by itself must consume at least time $O(log(n))$ where n is the number of stored records. We will later see that we can achieve search times within $O(1)$ meaning a constant delay independent of the number of stored records. However, in this particular case, substantial delays caused by the communication with the database might even outweigh this fact.
With respect to efficiency the implementation of the complete algorithm in an autonomous programming environment is clearly the optimal solution. But if we store data internally how can we overcome the disadvantage that the storage area of an application obviously is non-persistent[9]? Programming languages such as C# and Java provide convenient serialization methods that allow us to export objects (thus data structures as well) and store them as binary files persistently on hard-disks. With the next initialization they can be read in again.

5.5.1 Characteristics of Sets 1-3

Before a specific data structure is used or implemented requirements should be identified. Set 1, Set 2 and Set 3 share the property of bit string keys addressing potentially several data records. Hence, a data structure that will suffice all sets can be generalized as follows.

Definition 10 Generic set data structure

```
Set<T> := Dictionary<Key:BitString, Value:T>
// where T can be any data type, e.g. integer, list, array, ...
```

As the definition indicates data structures mapping a unique key to a value (or any abstract data type) are called dictionaries. Since our data structure is basically generated from historical data and only changed or extended due to prediction errors the majority of accesses will be non-modifying search requests. Therefore we declare the dictionary as quasi-static and should draw our attention to the efficiency of those requests. Efficiency means in that case to find at least better search routines than comparing each data record's key with the requested one which would take time $O(n)$.

5.5.2 Hash Tables and the C# / Java Dictionary

Hash tables are a highly efficient implementation of dictionaries and also used for the standard C# / Java dictionaries. A hash table is an array mapping indexes to (lists of) values. Suppose we have 128 key-value-pairs with consecutive keys from $\{0\}$ to $\{111111\}$. We can now allocate an array of that size and store each record at a different position. But which position to choose for a given key and how to find them again? We can interpret the key's bit string as an integer.

[7] http://www.forexfactory.com/showthread.php?p=7197052#post7197052
[8] http://www.tradingapi.net/mt4-mssql-api
[9] I.e. changes of the database f.i. due to prediction errors are lost after the application's running time.

The position of key $k = \{b_0, b_1, ..., b_{n-1}\}$ with $b \in \{0; 1\}$ of length n is then $pos(k) = \sum_{k=0}^{n-1} b_k 2^k$. Whenever we search for this specific key we can find it by accessing the array at $pos(k)$. Note that this is independent of the number of records, so search time is $O(1)$.
$pos(k)$ is a very simple hash function mapping a key to an index. Obviously, if the keys to be stored are not consecutive a hash function cannot be as simple as $pos(k)$. Furthermore if we intend to store "a few" (say 1000) keys of a potentially large range (say $[0; 10000000]$) it is impracticable to allocate an array of the size of the range. Therefore a hash function must map keys of a potentially large range to a much smaller table and ensure a good distribution in order to avoid conflicts, i.e. two different keys map to the same position. However, there are hash functions that guarantee an averaged constant search time even under these circumstances.
Back to the basics: dictionaries as provided by C# or Java are very efficient. A single search request on a dictionary based Set 1 should take about hundreds to a few thousands of nanoseconds. The keys of Sets 1-3 should not be stored as strings but rather as bit vectors.

5.5.3 Binary Tries / Prefix Trees

[coming soon]

6 Model Logic without Error Prediction & Adaptability

Note: @gg53, crodzilla, killerno, emonts, burnssss, candero & everyone who implemented the logic:
Please look closely at the transition conditions mapping one state to another and the invoked procedures. Please let me know if you find errors or ambiguous descriptions or if you use alternative methods (and like to share them).

In this section we will describe the actual business logic or the model of our approach which is based on the Set 1-3 data structures and three consecutive parts plus an initial state. These are:

- InitialState (S0)
- FractalPrediction (S1)
- TotalLengthPrediction (S2)
- NextBarOppositePrediction (S3)

We treat these parts as processing states of the model. The model itself handles incoming real-time candles provided by MT4 or directly by a broker and executes a processing step for each newly formed candle. A method called $Process(OHLCCandle)$ receives the candle, adds it to an internal list $OHLCCandles$ and extends the $OHCLBitString$ pattern depending on if it is an up or down candle. Likewise, the $FractalSequence$ list is extended depending on if the new candle forms a twin, up or down fractal (2, 1, 0) or no fractal at all (-1). Note that this requires a causal (delayed), thus non-repainting, fractal indicator as introduced before. In contrast, the $FractalPattern$ list contains only true fractals (ignoring -1 values) and represents the current fractal pattern. Twin fractals (2) must be unfold as described within the fractals section before. All these internal variables - or model properties - are initially empty. In their entirety the model properties represent an internal model state in terms of a certain time step. After that the $Process(OHLCCandle)$ method will delegate the additional processing to the $Process()$ method of the model's intitial processing state. **Figure 1** illustrates the concept.

6.1 Processing States - Overview

Below we describe the processing states and their transition conditions. Before going into detail we give a short overview of the individual state's objectives:
(S0): We start with the initial state waiting for a new incoming candle. If so, i.e $Process()$ has been invoked, we move to FractalPrediction.
(S1): In the FractalPrediction state we wait for new fractals, i.e an updating of $FractalPattern$. If a true new fractal has formed, we inspect the probabilities of more incoming fractals, i.e if Set 2 contains a more probable fractal pattern longer than the current $FractalPattern$ otherwise back to the initial state. If the current fractal pattern is the most probable the processing state is changed to NextTotalLengthPrediction.
(S2): Within this state we test the probability if the current $FractalPattern$ has already evolved into a complete zigzag pattern, i.e if it's length, allowing a [-3;+3]-tolerance, matches the lengths of identical historical Set 2 patterns. If so we move to the NextBarOppositePrediction state and if not we go back to the initial state.
(S3): This last consecutive state now queries Set 3 whether the next candle probably will be opposite to the current zigzag's direction or not. That means, if we expect a bearish candle, we see for each length $l \in [4; 13]$ if there are more patterns beginning with the last $l - 1$

bits of *CandleBitString* extended by 0 than patterns beginning with the last $l - 1$ bits of *CandleBitString* extended by 1. If there is consensus on all length levels our entire signal conditions are met and we draw a trading signal opposite to the current zigzag leg's direction. Otherwise we go back to the initial state. Therefore a signal requires successful consecutive transitions through all processing states.
Figure 2 shows this concept by use of a state transition graph annotated with condition action tuples {C, A}.

6.2 Set Queries

There are two basic Set queries implicitly mentioned above: We need a query method called $P(pattern)$ for both Set 2 and Set 3 to get the probability (the frequency of occurrence) of a specific pattern. Also we need a second method called $Q(pattern)$ for Set 2 to count data records addressed by a specific pattern key and a given candle length which is necessary for TotalLengthPrediction.

Algorithm 11 P(pattern) Query for Set 2

```
int Set2.P(BitString pattern)
{
  if (pattern == null)
    return 0;

  if (!Set2.ContainsKey(pattern))
    return 0;

  // Return number of records stored in the given pattern's list
  return Set2[pattern].Count;
}
```

Algorithm 12 P(pattern) Query for Set 3

```
int Set3.P(BitString pattern)
{
  if (pattern == null)
    return 0;

  if (!Set3.ContainsKey(pattern))
    return 0;

  // Return frequency of occurrence for given pattern
  return Set3[pattern];
}
```

Algorithm 13 Q(pattern) Query for Set 2

```
int Set2.Q(BitString pattern, int lengthFilter)
{
  if (pattern == null)
    return 0;

  if (!Set3.ContainsKey(pattern))
    return 0;

  // Return number of records with zz-leg length == lengthFilter
  // stored in the given pattern's list
  return Set2[pattern]
    .Where(record.cLength => record.cLength == lengthFilter)
    .Count();
}
```

Additionally, the FractalPrediction state requires a method to estimate a probable fractal pattern's length with respect to the historical data within Set 2. Crodzilla introduced this *ProjectCandleLen* method in the following way:

Algorithm 14 ProjectCandleLen - crodzilla's version

```
int ProjectCandleLen(BitString fractalPattern)
{
  const int threshold = 0.50;
  // Get maximum length of all records (legs) adressed by
     fractalPattern
  var maxCandleLength = Set2.Get(fractalPattern).Max();

  for (var i = 1; i <= maxCandleLength; i++)
  {
    var inLengthCount = 0;   // # records with lengths \in [i-3;i]
    var outLengthCount = 0;  // # records with lengths \in [i;i+3]

    for (var tolerance = -3; tolerance <= 0; tolerance++)
      inLengthCount += Set2.Q(fractalPattern, i + tolerance);

    for (var tolerance = 0; tolerance <= 3; tolerance++)
      outLengthCount += Set2.Q(fractalPattern, i + tolerance);

    // Accept i if i is quasi-gauss peak of local cluster
    if (inLengthCount / (inLengthCount + outLengthCount) > threshold)
      return i;
  }

  return 0;
}
```

Note: @crodzilla: Is this equivalent to your current projected length method?

In contrast to this a second version similar to the length estimation used within the Total-LengthPrediction state introduced by gg53 is also conceivable:

Algorithm 15 ProjectCandleLen - based on gg53's total length estimation

```
int ProjectCandleLen(BitString fractalPattern)
{
  // Get maximum length of all records (legs) adressed by
     fractalPattern
  var maxCandleLength = Set2.Get(fractalPattern).Max();

  for (var i = 1; i <= maxCandleLength; i++)
  {
    var maxInLengthCount = 0;
    for (var tolerance = -3; tolerance <= 3; tolerance++)
    {
      var tolLengthCount = Set2.Q(FractalPattern, i + tolerance);
      if (tolLengthCount > maxInLengthCount)
      maxInLengthCount = tolLengthCount;
    }

    var outLenghtCount = Set2.Q(fractalPattern, i + 4);

                  // Accept i if maximum probability within tolerances
                     [i-3;i+3] > probability i+4
                  if (maxInLengthCount > outLenghtCount)
      return i;
  }

  return 0;
}
```

Note: @gg53: Do you use such an additional criterion $ProjectCandleLen(longerFractalPattern) > currentFractalPattern.Length$ within the FractalPrediction state at all? If so which method do you use?

6.3 State Transitions

In this section we describe the transitions between consecutive states. A transition is a mapping from one state S to the next state $S*$ subject to a certain condition. F.i in Figure 2 condition C3 changes the $NextBarOppositePrediction$ state to the $InitialState$ while drawing a $Signal$. The conditions transferring one state to another will be discussed precisely in two ways. Once by use of pseudo code and later on formally using an abstract but compact notation.

6.3.1 Initial State S0

For the initial state there is only one trivial transition, namely $S0 \mapsto S1$ triggered by the $Process()$ method and invoked in case of a new incoming candle.

6.3.2 FractalPrediction S1

In the FractalPrediction state S0 there are two transitions: $S1 \mapsto S2$ subject to C1 and $S1 \mapsto S0$ subject to the logical opposite of C1. C1 is true if the probability - or frequency of occurrence - of the current fractal pattern regarding Set 2 is greater than zero and neither a longer fractal pattern extended by 0 nor a pattern extended by 1 is more likely then the current one. A longer fractal pattern is more probable than the current one if it occurs more often in Set 2 and if it's projected (expected) length is also longer than the current fractals length. Therefore we can write C as follows:

Algorithm 16 C1: $S1 \mapsto S2$ and C6: $S1 \mapsto S0$

```
var expLonger0 = Set2.P(FractalPattern.Add(0)) > Set2.P(FractalPattern)
                 && ProjectCandleLen(FractalPattern.Add(0)) >
                    FractalPattern.Length;

var expLonger1 = Set2.P(FractalPattern.Add(1)) > Set2.P(FractalPattern)
                 && ProjectCandleLen(FractalPattern.Add(1)) >
                    FractalPattern.Length;

var c1 = Set2.P(FractalPattern) > 0 && !(expLonger0 || expLonger1);
var c6 = !c1;

if(c1) Model.ChangeState(TotalLengthPrediction);
else if(c6) Model.ChangeState(InitialState);
```

6.3.3 TotalLengthPrediction S2

TotalLengthPrediction branches to the NextBarOppositePrediction via C2 or to the initial state via C5. C2 is true if the current fractal's length is more probable regarding Set 2 than a longer fractal's length with a [-3;+3]-tolerance in mind. Gg53 suggested the following method:

Algorithm 17 C2: $S2 \mapsto S3$ and C5: $S2 \mapsto S0$

```
var pCurrFracLen = 0;
for (var k = -3; k <= 3; k++)
{
  var kFracLen = Set2.Q(FractalPattern, FractalPattern.Length + k);
  if (kFracLen > pCurrFracLen)
    pCurrFracLen = kFracLen;
}

var pLongerFracLen = Set2.Q(FractalPattern, FractalPattern.Length + 4);

var c2 = pCurrFracLen > pLongerFracLen;
var c5 = !c2;

if(c2) Model.ChangeState(NextBarOppositePrediction);
else if(c5) Model.ChangeState(InitialState);
```

6.3.4 NextBarOppositePrediction S3

The final state NextBarOppositePrediction leads back to the initial state either via C3 triggering a signal or directly via C4. In case of an expected bearish signal C3 is true if for all lengths $k \in [4; 13]$ the candle bit strings built from the last $k-1$ bits of the current *CandleBitString* in addition to "0" are more probable than the particular ones extended by "1". Also all the actual *CandleBitString* sub-strings built from the last k bits must be more probable than those that are changed at their last position (i.e flipping the last bit). C3 is defined analogously for expected bullish signals.

Algorithm 18 C3: $S3 \mapsto S0$ and C4: $S3 \mapsto S0$

```
var signal = null;
var nextBarConsensus = true;
var currentBarConsensus = true;

if(expectBearish)
{
  signal = Signal.Bearish;
  for (var k = 4; k <= 13; k++)
  {
    nextBarConsensus = nextBarConsensus &&
        Set3.P(CandleBitString.TakeLast(k - 1).Add(0) >
        Set3.P(CandleBitString.TakeLast(k - 1).Add(1);
  }
}
else if(expectBullish)
{
  signal = Signal.Bullish;
  for (var k = 4; k <= 13; k++)
  {
    nextBarConsensus = nextBarConsensus &&
        Set3.P(CandleBitString.TakeLast(k - 1).Add(1) >
        Set3.P(CandleBitString.TakeLast(k - 1).Add(0);
  }
}

for (var k = 4; k <= 13; k++)
{
  currentBarConsensus = currentBarConsensus &&
      Set3.P(CandleBitString.TakeLast(k) >
      Set3.P(CandleBitString.TakeLast(k).FlipLast();
}

var c3 = nextBarConsensus && currentbarConsensus;
var c4 = !c3;

if(c3)
{
  Model.ChangeState(InitialState);
  Model.Signal(signal);
}
else if(c4) Model.ChangeState(InitialState);
```

The equivalent formal definitions are consolidated in a state transition table, see **Table 1**.

6.3.5 References

- Model Logic `http://www.forexfactory.com/showthread.php?p=7190111#post7190111`

Figure 1: Model

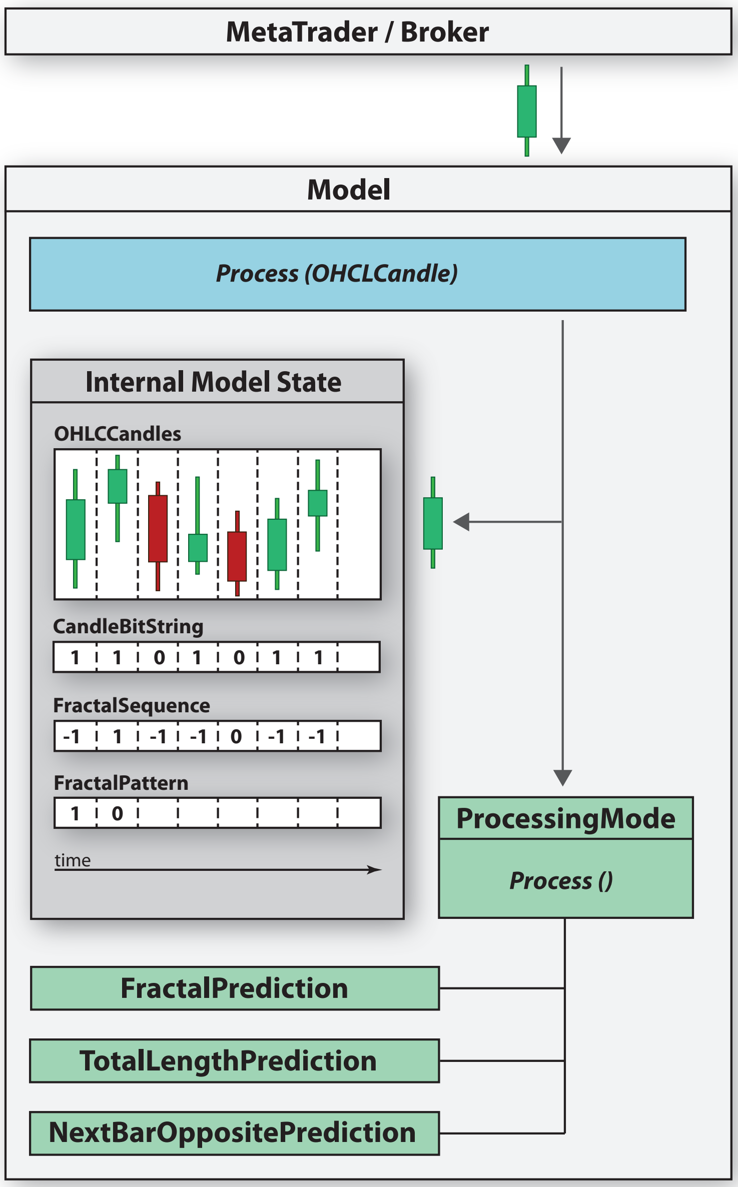

Figure 2: Processing States

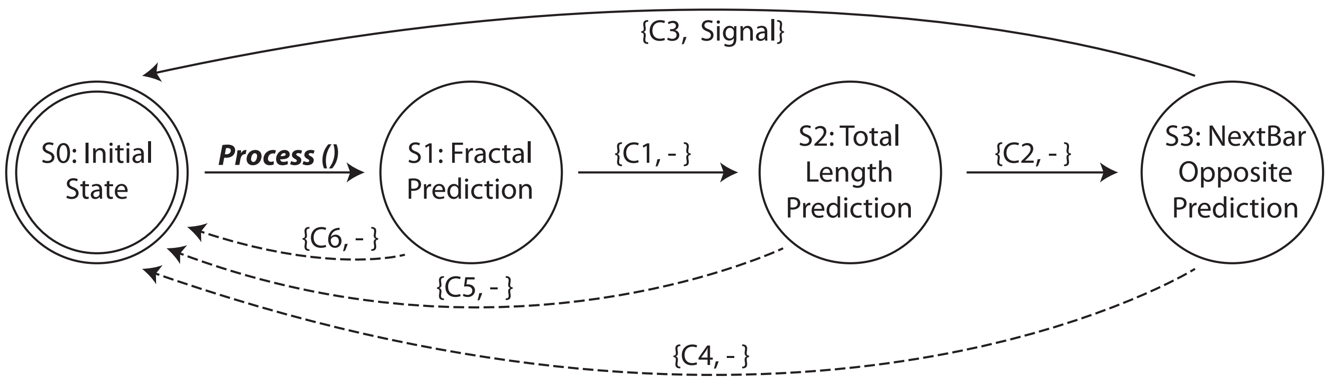

Table 1: State Transition Table

S	**Condition**	**Action**	**S***
S0	$Process()$	-	S1
S1	$C1 := Set2.P(FractalPattern) > 0 \wedge \neg\, (expLonger0 \vee expLonger1)$ where $expLonger0$ $:= Set2.P(FractalPattern.Add(0)) > Set2.P(FractalPattern)$ $\wedge\, ProjectCandleLen(FractalPattern.Add(0)) > FractalPattern.Len$ and $expLonger1$ $:= Set2.P(FractalPattern.Add(1)) > Set2.P(FractalPattern)$ $\wedge\, ProjectCandleLen(FractalPattern.Add(1)) > FractalPattern.Len$	-	S2
	$C6 := \neg\, C1$	-	S0
S2	$C2 := pCurrFracLen > pLongerFracLen$ where $pCurrFracLen$ $:= \max_{k\in[-3;3]}\{Set2.Q(FractalPattern, FractalPattern.Len + k)\}$ and $pLongerFracLen$ $:= Set2.Q(FractalPattern, FractalPattern.Len + 4)$	-	S3
	$C5 := \neg\, C2$	-	S0
S3	$C3 := \begin{cases} nextBarConsensus0 \wedge currBarConsensus, & \text{if } expectBearish \\ nextBarConsensus1 \wedge currBarConsensus, & \text{if } expectBullish \end{cases}$ where $nextBarConsensus0$ $:= \bigwedge_{\forall k\in[4;13]}\{\ Set3.P(CandleBitString.TakeLast(k-1).Add(0)$ $> Set3.P(CandleBitString.TakeLast(k-1).Add(1)\ \}$ and $nextBarConsensus1$ $:= \bigwedge_{\forall k\in[4;13]}\{\ Set3.P(CandleBitString.TakeLast(k-1).Add(1)$ $> Set3.P(CandleBitString.TakeLast(k-1).Add(0)\ \}$ and $currBarConsensus$ $:= \bigwedge_{\forall k\in[4;13]}\{\ Set3.P(CandleBitString.TakeLast(k)$ $> Set3.P(CandleBitString.TakeLast(k).FlipLast()\ \}$	Signal	S0
	$C4 := \neg\, C3$	-	S0