Available online www.jocpr.com

Journal of Chemical and Pharmaceutical Research, 2014, 6(6):669-676

Research Article

ISSN : 0975-7384
CODEN(USA) : JCPRC5

An empirical study of collaboration methods for CEP based on algorithmic trading

Kong Xiangsheng

Department of Computer & Information, Xinxiang University, Xinxiang, China

---

ABSTRACT

Algorithmic trading has become more popular with large institutional investors these days. Complex event processing is a typical data processing technique which becomes the new spotlight of researches. In order to obtain semantic data, in this paper, we investigate the definition, detection, and management of events in the architecture of complex event processing based on algorithmic trading. Especially we propose a corresponding event model and develop an algorithm that can efficiently detect complex event over event stream.

Keywords: Algorithmic Trading, Complex Event Processing, Volume Weighted Average Price, Data Stream Management System

---

INTRODUCTION

Beginning in the late 1990s, the electronification of execution venues enabled market participants (banks, brokers and their institutional and retail clients) to remotely access electronic order books. Electronic trading refers to the ability to transmit orders electronically as opposed to via telephone, mail, or in person. Since most orders in today's financial markets are transmitted via computer networks, the term is rapidly becoming redundant. Over the past few years there has been a rapid increase in the volume of trading done by algorithms. Algorithmic trading (AT) for e-markets is more complex than electronic trading. AT for e-markets is the use of computer programs to enter trading orders with the computer algorithm deciding on characteristic of the order such as the timing, price, or quantity of the order and in many cases initiating the order without human intervention. In AT orders are placed with the algorithm which decides on various aspects of the order such as order price, size, timing of purchase etc [1].

Realizing AT for e-markets is a challenging task. One of the biggest challenges is to deal with a large amount of data produced in real-time. This is because e-markets involve a large number of users (possibly from all around the world) and have been growing in size rapidly over the last decade [2]. Other challenges include developing software components that interface effectively with the market feeds and handling the different types of data formats used to encode e-market transactions.

A large class of both well-established and emerging applications can best be described as event processing that need to process massive streams of events in (near) real-time. Event processing differs from general data stream management in two major aspects of the query workload. First, it has a distinct class of queries, which warrants special attention. In complex event processing (CEP), users are interested in finding matches to event patterns, which are usually sequences of correlated events. Second, in CEP, there is usually a large number of concurrent queries registered in the event processing systems. This is similar to the workload of publish/subscribe systems. In comparison, data stream management systems (DSMSs) are usually less scalable in the number of queries, capable of supporting only a small number of concurrent queries [3]. CEP provides flexibility in handling data in different formats without a pre-processing step and offers scalability in handling the increasing amount of data being produced in e-markets.

The conception of complex event that is typically expressed by means of patterns that declaratively specify the event sequences to be matched over a given data set origins from the research rule processing in active database. CEP Systems (shown in Figure 1) associate a precise semantics to the information items being processed: they are notifications of events happened in the external world and observed by sources. The CEP engine is responsible for filtering and combining such notifications to understand what is happening in terms of higher-level events (sometimes also called complex events or situations) to be notified to sinks which receive output events resulting from the queries running on CEP engines and act as event consumers [4].

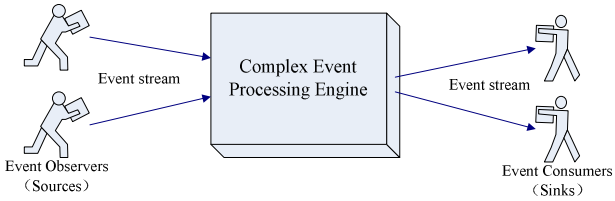

Figure 1. The High-level View of a CEP System

ARCHITECTURE OF CEP BASED ON ALGORITHMIC TRADING

The continuous analysis of streaming events is called complex event processing (CEP). CEP allows reacting immediately when specific events occur. An event is a pair (p, t) consisting of a payload p that is some information about a real or virtual event and a timestamp t that specifies the instant of time when the event occurred. Typically, CEP is used to combine events that occurred in a specific order or within a timeframe. For example, an application running on a touch screen device can wait for the user to put exactly three fingers into the upper right quarter of the screen within two seconds (correlation of events). Or a computer game can detect that a player has picked up some item at place A and brought it to place B before a predefined mission timer elapsed (pattern matching of events). Besides correlation and pattern matching, filtering and aggregating events are other important basic CEP operators. These four basic CEP operators can be combined arbitrarily to express more complex queries. Because every new input event can produce new results, all queries are performed continuously in an event-driven manner.

We consider a scenario from the financial computing domain, in which Web services provide live data about companies and stock prices. The aim is to combine the information in an XML document that is actively updated when the underlying data change. Figure 2 illustrates, on a high level, how data and events are received and processed.

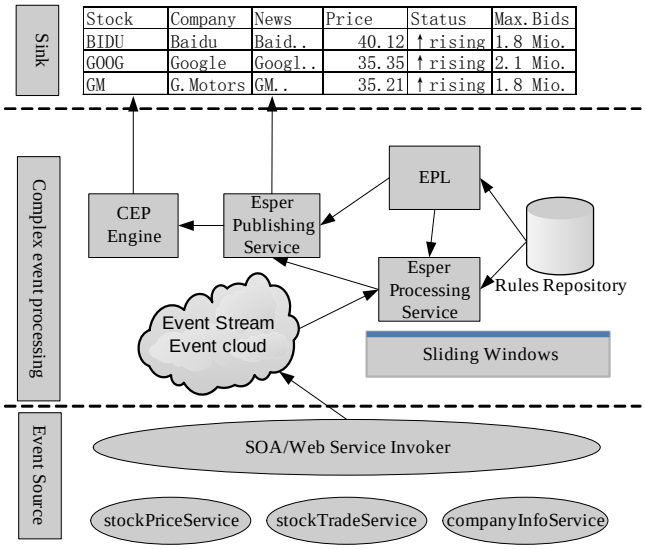

Figure 2. data and events processed

AT & VWAP

There are various powerful algorithms being used by various organizations like Volume Weighted Average Price (VWAP), Time Weighted Average Price (TWAP), Market on Close (MOC) and Information shortfall [5]. Of all these VWAP has been the most popular model over the years. When a bulk order is placed the algorithm breaks it into several smaller orders according to the historical data as computed to be optimum. Volume curves are generated which describe the objective buying. In case of intra-day buying if large fluctuations are observed then the curve is interpolated accordingly to reflect the current scenario. Orders are executed according to the volume curve as long as they are within the maximum order size supported by the algorithms. Once the limit is reached, trading stops. Hence

brokers try to optimize the algorithm to increase the maximum trading limit. It is particularly useful when the trader is not able to gauge the current market trends.

For instance the VWAP of a stock can simply be explained as the average price paid per share during a specified time, usually a day. This means that the price of each transaction in the market is weighted by its volume. In VWAP-trading the goal is to buy or sell a fixed number of shares at price that closely tracks the VWAP. VWAP is especially common in automatic trading algorithms, especially in optimal trading execution strategies [6]. The formula for calculating VWAP is as follows (1).

$$P_{VMAP} = \frac{\sum_j P_j * Q_j}{\sum_j Q_j} \quad (1)$$

where:
PVWAP = Volume-Weighted Average Price
Pj = price of trade j
Qj = quantity of trade j
j = each individual trade that takes place over the defined period of time, excluding cross trades and basket cross trades.

Here is an AT & VWAP example.
IF(
MSFT's price moves outside 1% of MSFT-15-minute-VWAP
FOLLOWED-BY{
CSCO' price moves up or down by 0.5%
AND
IBM' price moves up by 3%
OR
MSFT' price moves down by 1%
}
) ALL WITHIN any 120 seconds time period
THEN {
BUY MSFT;
SELL IBM;
}

Complex Events and Event Operators
An event is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest at a point in time. It is the smallest, atomic occurrence in a system that may require a response. By atomic, we mean that either the event happens completely or it does not happen at all. A set of attributes can be associated with each primitive event. These attributes can carry information which can be used when a complex event occurs (at a later time) about the action that caused the event to occur.

Similar events can be grouped into an event type that gives the metadata for events that belong to the same class and includes the attributes of these events, and an event type is expressed by an event expression. An event instance is a single occurrence of an event of a particular type. We consider E1, E2, …, En as being primitive event types and e1, e2, …, en some of their respective instances.

Although an event is assumed to instantaneously occur at a time point, the event might be initiated at a prior time point, thus yielding a closed time interval between the start and end points. That is each event instance, whether primitive or complex, has both a start and end timestamp. Two special event types—START and END—are added internally by Synoptic to keep track of initial and terminal events in the traces [7]. A complex event is defined by applying an event operator to constituent events that are primitive or other complex events. In the absence of event operators, several rules are required to specify a complex event. Furthermore, some control information needs to be made a part of a rule specification (shown in Figure 3).

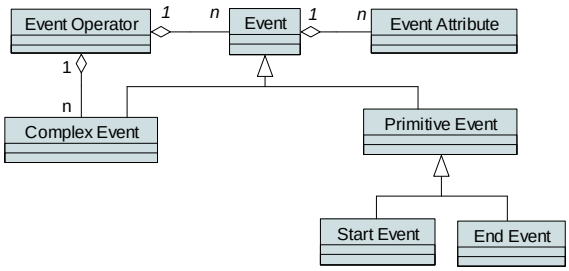

Figure 3. The Class Diagram of Events

An event E (either primitive or complex) is a function from the time domain onto the boolean values, True and False. E : T → {True, False} given by (2).

$$E(t)=\begin{cases} T(rue)\cdots \text{if an event of type E occurs} \\ \qquad\qquad \text{at time point t} \\ F(alse)\cdots \text{otherwise} \end{cases} \tag{2}$$

We denote the negation of the boolean function E as ~E, and ~E expresses the non-occurrence of E at a given point in time. The other event operators and the semantics of complex events formed by these event operators are as follows: Definition 1.OR (∨): Disjunction of two events E1 and E2, denoted E1 V E2, occurs when E1 occurs or E2 occurs. This captures those points in time, where at least one of E1 and E2 occurs. Formally (3),

$$(E_1 \vee E_2)(t) = E_1(t) \vee E_2(t) \tag{3}$$

Definition 2. AND (∧): Conjunction of two events E1 and E2, denoted E1 ∧ E2, occurs when both E1 and E2 occur, irrespective of their order of occurrence. This captures those points in time where an instance of E1 occurs, E2 having occurred earlier (or at the same instant in time), or vice versa. Formally (4),

$$\begin{aligned}(E_1 \wedge E_2)(t) = ((\exists t_1)(E_1(t_1) \wedge E_2(t)) \vee \\ (E_2(t_1) \wedge E_1(t)) \wedge t_1 \le t)\end{aligned} \tag{4}$$

Definition 3.Sequence (;): Sequence of two events E1 and E2, denoted E1;E2, occurs when E2 occurs provided E1 has already occurred. This implies that the time of occurrence of E1 is guaranteed to be less than the time of occurrence of E2. This sequencing operator looks for the occurrence of E1 followed in time by the occurrence of E2, i.e. an occurrence of a is followed by an occurrence of b; thereby a must end before b starts [8]. Formally (5),

$$(E_1;E_2)(t) = ((\exists t_1)(E_1(t_1) \wedge E_2(t)) \wedge t_1 \le t) \tag{5}$$

For example if the pattern being detected is C∧(A;B) if an instance of B follows an instance of A, the sequence operator will be fired as long as an instance of C does not occur between them.

Definition 4.WITHIN operator: WITHIN operator allows the user to specify not only the order of events participating in an operator but also how far they are allowed to be from each other. For example WITHIN (A;B)1h specifies that an instance of B needs to happen after an instance of A but within 1 hour from it [9].We can express a wide variety of complex events by these operators. here is an example (shown in Figure 4):

WITHIN (((StockQuote (symbol = IBM, price>40)) ∧(StockQuote (symbol = CSCO, price < 36))) ; StockQuote (symbol = MSFT, price > 30)) 120s .

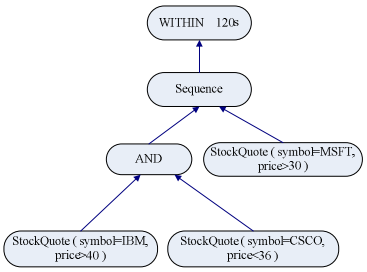

Figure 4. The Example of a Complex Event

Architecture of CEP Based on AT

Figure 5 shows the architecture of CEP Based on AT. The AT rule definitions is done by the Analyzer and the Constructor, well separated from the runtime tasks，represented by the Complex Event Detector, the Event Manager and the Executor. The following briefly describes these components.

The Analyzer principally analyzes a rule definition and produces—an intermediate representation of the rule which is sent to the constructor, and code corresponding to the condition and action of the rule.

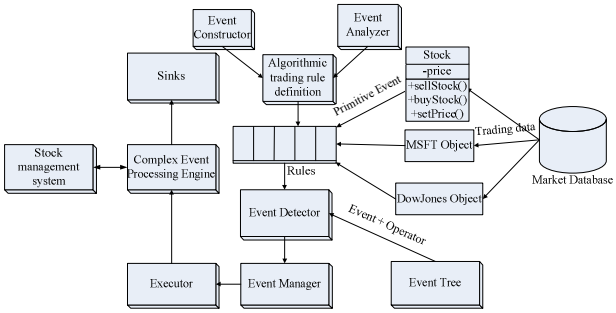

Figure 5. Architecture of CEP Based on AT

The Constructor creates a persistent representation of rules and offers a low level interface well adapted for software integrators and developers who need basic reactive capabilities for supporting some functions of the system they want to implement.

The Event Executor is responsible for processing rules taking into account coupling modes, rule priorities. It realizes quite complex execution semantics and this combined with the need for runtime efficiency represents the main reasons for having implemented the part.

The Event Detector is responsible for detecting primitive events and for signaling them to the event manager. The latter recognizes complex events using a detection graph and signals both primitive and complex events to the Event Executor.

The event manager has to represent the information gained from the analysis of event definitions, i.e., it is responsible for managing the event base which consists of all defined events patterns. If the event type is primitive the Event Manger subscribes it to the CEP Engine. If the event type is complex, the Event Manager subscribes the primitive event types composing the event type to the event detector and builds an event tree representing the complex event type.

COMPLEX EVENT DETECTION

Development of algorithms involves a high level of collaboration with the client as algorithms are meant to meet the trading strategy objectives of the trader. Algorithms are meaningless if the strategies don't perform. The basic processes involved are: closely interacting with the users to understand their strategies, creating an algorithm based on the inputs, presenting the client with results of back tests and analysis using historical tick-level data. The algorithm is then released to one or two beta clients, who begin to use it on small volumes of live trades. From that point the vendor and the client will engage in a period of iterative feedbacks during which they conduct post-trade analysis to ensure that the desired results are being achieved. The final product is moved up and down the development chain with constant feedback from the end user. Once the required results are obtained the product is finalized. The basic fact to remember is that the client is just interested in results and they demand good performance, speed of execution.

So the manner in which an algorithm is tested or the manner in which it is implemented is rarely of concern to the trader.

Event Graph

Events are detected on the server using an event graph. An event graph which consists of nodes and directed edges is a graph constructed to reflect the primitive and complex events declared in an application. Each event is represented as an event node in the graph, and the event nodes are connected by their subscription relationships. An internal node of the event graph represents a complex event, and a leaf node represents a primitive event. Thus the event detector generates an event tree whose root node represents the complex event. An event tree is created for each complex event and these trees are merged to form an event graph for detecting a set of complex events. This will avoid the detection of common sub-events multiple times thereby reducing storage requirements.

For each node in the event graph, there is an event subscriber linked list containing all the complex events that use this event as its constituent one. An event node has a pointer to its subscriber which becomes its parent node. Complex events and rules can subscribe any number of primitive events. These events and rules are kept either in event-list or in rule-list of the primitive event. Leaf nodes pass the primitive events immediately to their parent nodes as the semantics of all contexts are identical for primitive events [10].

Whenever a primitive event is detected, it will propagate the event notification to its subscribers, that is, its parent nodes. Event occurrences flow upwards as in a data-flow computation. The parent nodes maintain the occurrence of its constituent events along with their parameter lists which are stored separately for each context set to the node. If the complex event occurs by the last notification, it is detected and further propogates to its subscribers. Each time an event is raised, it will check its "send back" flag. If the "send back" flag is true, the server will send this event notification to a specific application according to this event "site" attribute.

Complex Event Detections

An event detector has a linked list whose nodes hold one reactive class of an application. Each node, in turn, has two linked lists, begin list and end list. The lists have the subscribers to be notified at the beginning or the end of these methods's invocations. For example:

event begin (e1) int sellStock(int number);

The primitive event e1 is bound to a method named sell stock and the method notifies its occurrence at the beginning of its invocation. The event detector detects primitive events produced during an application processing. It detects only events for which event type subscription has been submitted and signals them and their environments to the event manager. The general principle for recognizing events is the following: primitive events are injected at the leaves of the event graph. Then these events flow upwards following edges through internal nodes which represent component events. When a triggering node is reached, the recognized triggering event is signaled and then taken into account for rule execution.

The event subscriber list records complex events that are related to this event. Each node has a pointer to each of its subscribers. Thus each subscriber of a global event becomes one of its parent nodes that the event tree is built from. By default a subscriber is inserted in the end-list if it does not specify when to be notified. This organization reduces the search which is based on the class. However, search for the class is sequential. An event graph is constructed while the event trees of complex events in the application are built by their subscribing relations. The event graph is connected to the reactive class list through primitive-leaf nodes. A primitive-leaf node is pointed by a method which is belonging to one of the reactive class nodes. This connected event graph and the reactive list constitute a local event detector for the application. A primitive event constructor registers itself to the reactive class list as a subscriber of a method in the list.

Esper Event Programming Language (EPL)

EPL processing by executing continuous queries on event streams is used to define complex events, similar to the concepts known from active databases [11]. These query languages execute operations similar to SQL, including:

SELECT (Selects event types, attributes of an event in the event stream)
WHERE (Define conditions for the events that should fulfill the query)
AGGREGATION (Min, Max and other aggregations known from SQL are available)
JOIN (Similar to our definition of event correlation, events can be joined via their attributes)
TIMEWINDOW (The queries are executed against the events in a specific sliding time window)

For example:
select avg(price)
from StockTickEvent.win:time(300)
where StockTickEvent.symbol='IBM';
select symbol, avg(price) as averagePrice
from StockTickEvent.win:length(100)
group by symbol;

The first query returns the average price of all IBM stock tick event within the last 300 seconds (with a sliding time window). The second query returns the average price per symbol for the last 100 stock ticks.

EVALUATION

In order to obtain an impression of the performance of our CEP system, we conducted an experimental evaluation. Two things are remarkable about our implementation and test environment. Our implementation is strictly single-threaded and consumes only few resources (in the worst case, one CPU core is fully utilized). We avoided implementing a multi-threaded CEP server, because our target use-cases are applications that need embedded CEP functionality on the same machine.

We have examined for each basic CEP operator the number of input events being processed per second. We also have evaluated the scalability of all basic CEP operators by running multiple of them at the same time. Our test machine had an Intel i7 CPU with 8 GB main memory. Figure 6 shows the results on a logarithmic y-axis.

In the experiments, exactly one event was pushed into all input event streams for every millisecond. So all adjacent events in an event stream had timestamps that differed exactly by one millisecond from each other. Therefore, all time windows of the parameterized test queries were filled with 500 events on average.

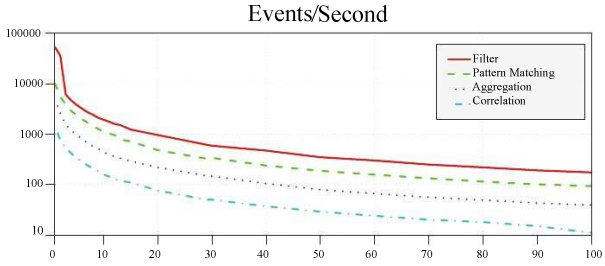

Figure 6. The results on a logarithmic y-axis

CONCLUSION

In this paper, we have investigated how events are defined, detected and managed and presented an expressive event specification language that supports AT. We have illustrated the detection of complex events and proposed an architecture for its implementation based on AT. Our approach clearly substantiates existing event-driven systems with declarative semantics. All the event detection algorithms we have developed extend readily when the identification of the object is allowed as an explicit parameter of a primitive event.

REFERENCES

[1] Zengqiang Ma; ShaZhong; Xingxing Zou; Yacong Zheng, *Journal of Chemical and Pharmaceutical Research*, **2014**(2), 145-150.
[2] ZhiyunFeng; Maozhu Jin; Renpei Yu, *Journal of Chemical and Pharmaceutical Research*, **2014**(2), 187-192.
[3] Trimurti Lambat; Sujata Deo; Tomleshkumar Deshmukh, *Journal of Chemical and Pharmaceutical Research*, **2014**(4), 888-892.
[4] Archit Bansal; Kaushik Mishra; Anshul Pachouri, *International Journal of Computer Applications*, **2010**(1), 1-5.
[5] Piyanath Mangkorntong; Fethi A. Rabhi, *IADIS International Conference e-Commerce*, **2009**, 69-86.
[6] Alan Demers; Johannes Gehrke; Biswanath P, Proceedings of the Conference on Innovative Data Systems Research(CIDR), **2007**, 412–422.
[7] Gianpaolo Cugola; Alessandro Margara, Proceedings of the 5th ACM international conference on Distributed event-based system, **2011**(5), 1-32.
[8] Nihal Dindar; Peter M. Fischer; Merve Soner; Nesime Tatbul, Proceedings of the 5th ACM international conference on Distributed event-based system, **2011**(5), 33-69.

[9] Erik Eiesland. Master Thesis, Department of Computer Science, Stfold University College, **2011**, 1-124.
[10] Marcelo R. N. Mendes; Pedro Bizarro; Paulo Marques, Proceedings of the second international conference on Distributed event-based systems, **2008**, 313–316.
[11] R. Baldoni; S. Bonomi; G. Lodi; M. Platania; L. Querzoni, International Workshop on Data Dissemination for Large scale Complex Critical Infrastructures, **2010**, 1-25