An event-based model to support distributed real-time analytics: finance case study

Zoran Milosevic[1],Weisi Chen[2], Andrew Berry[1], Fethi A. Rabhi[2]
[1]Deontik Pty Ltd, Brisbane, Australia
[2]University of New South Wales, Sydney, Australia
zoran@deontik.com, chenw@cse.unsw.edu.au, andyb@deontik.com, f.rabhi@unsw.edu.au

Abstract — **This paper describes key modelling concepts for events, event patterns and related concepts needed to develop a distributed software framework for real-time business analytics. These concepts are specified by means of a minimal meta-model, whose implementation can enable better interoperability between different event processing systems. This in turn can support better distributed, collaborative analytics applications in many domains. We show an implementation of our solution approach using a case study of several business analytics problems in finance.**

Complex Event Processing; real-time analytics; finance applications.

I. INTRODUCTION

The growing availability and access to data offers expanding opportunities for creating new insights through analytics. These new insights can be developed by applying various analytics techniques and appropriate tools to discover and communicate meaningful patterns in data. Our focus is on patterns of event instances as they occur in real time, which we refer to as *event pattern occurrences*. An event pattern occurrence signifies an occurrence of a particular set of events that satisfy the relationship, data and time constraints defined in an *event pattern type*. Examples include relationships between observations of patient condition in healthcare, particular combinations of buy and sell events in stock trading or correlation between social media postings and stock market activity. Note that the discovery mechanisms presented in this paper can also be applied to historical data, supporting a continuum of pattern detection across history and into the future for a set of data sources.

In a big data context, the detection of event pattern occurrences requires special technology infrastructure and techniques, such as complex event processing (CEP) [1]. CEP technology allows detection of event pattern occurrences against events arriving with high velocity, often from multiple data sources. CEP technology can be thus regarded as a form of machine analysis focused on detection of meaningful event occurrences, and which can be augmented with statistical analysis to support predictive capability. There are many platforms for event-processing systems, including various CEP platforms, but there is currently no standardised way of describing event types and event pattern types, and thus no standardised method for interchange of event pattern instances between systems [2].

The main contribution of this paper is in proposing a common language for consistently describing the event pattern types or event pattern instances implemented by or required by different systems. This in turn facilitates the definition of wire formats that can be consistently produced and consumed by those systems. We propose a meta-model defining key concepts and their relationships needed to precisely describe events, event pattern occurrences, event pattern types and other supporting concepts. The aim of this meta-model is to support interoperability between people involved in the design, development and integration of event processing systems, as well as interoperability between systems exchanging information. Such a meta-model can also provide the basis for the development of specific domain models, e.g. for finance, health, emergency management, utilities, etc., leveraging the power of model-driven development engineering techniques, and supporting tools such as Eclipse Modelling Framework [3].

The primary audience of this paper is computer scientists, solution architects, integrators and implementers involved in developing real-time analytics solutions. The ideas can be also of value for data scientists, analysts and researches involved in studying data in particular application domains, such as researchers involved in financial market analysis as discussed in [4] [5]. These subject matter experts can work together with computer scientists in defining rules that specify relationships between event occurrences of interest. This was indeed the approach taken in performing the finance case study described at the end of the paper.

This paper is structured as follows. The following section presents the motivation for this work arising from new opportunities and challenges related to real-time analytics, with particular emphasis on supporting business analytics and data science requirements. Section III introduces the foundational concepts for describing event patterns including events, event pattern types and event pattern occurrences. Section IV formalises the concepts though an event pattern meta-model. Section V provides a case study introducing several event pattern types from the financial market trading (equity) domain. Section VI describes how a set of pattern types have been implemented using a specific CEP engine, EventSwarm [6][7] to support researchers in finance domain. Section VII summarises key findings and describes our future work.

II. MOTIVATION

This paper is motivated by the need to better support researchers and data scientists in the finance domain who are interested in discovering important relationships between trading events. The data scientists need a fast and flexible

way of identifying and defining new event pattern types in order to support brokers and other interested market participants in detecting opportunities or threats associated with market trading. For example, brokers might be interested in detecting unusual spikes in price or volume of a specific stock, or detecting particularly poor or outstanding performance of a stock compared to others in its sector.

Due to the high velocity of trading, one needs an automated way to detect trading event pattern occurrences as well as an easy way to define the corresponding event pattern types. The complexity of financial market behaviour also requires that such pattern types support a wide range of pattern constructs, including mathematical, statistical and logical relationships between and across events, often within a specific time window of interest.

In the past, many such systems were developed commercially, either in house or by specialised vendors such as Apama [8], and primarily used by direct market participants such as investment banks, brokerage houses and stock exchanges. With the widespread availability of cloud services as a mechanism for delivery, there is an increasing interest from financial market researchers and the broader investment community to look at events signifying opportunities or threats.

Additionally, a number of regulatory organisations have recently introduced rules requiring listed companies to monitor potential causal relationships between social media postings and movement in company price or trading volume on a stock exchange. This is needed to protect investors from potentially damaging price manipulation or leaks of confidential information through social media postings. In this case one needs to monitor events coming from different sources, including market feeds and various social media channels. For example, the Australian Stock Exchange (ASX) recently published new guidelines for continuous disclosure, known as Guidance Note 8 [9]. These guidelines suggest that company executives, in particularly company secretaries of ASX listed companies, need to implement continuous monitoring of social media in order to detect and act upon social media postings which could have a material effect on the stock price of the company.

To facilitate the automation of financial data analysis, our previous research work proposed the ADAGE framework [5], in which there are three types of services (see Figure **1**) that can be flexibly composited into a workflow to support event processing for data analysis, namely:

– Event import service: the service to extract and process native event data from event data repositories. (Input: events; Output: events).
– Event processing service: the service to transform imported event data in a variety of ways; examples are removal of duplicate events, handling data quality issues, combining two sets of processed data together; each of this transformation is essentially producing new information which can be regarded as a complex event [1] (Input: events; Output: (complex) events).
– Event export service: the service to transform processed data into alternate formats suitable for external application use. For example, processed data can be converted into comma separated value (CSV) files so that it can be imported into spreadsheets; also, charts can be created from processed data and saved as image. (Input: events; Output: csv / image).

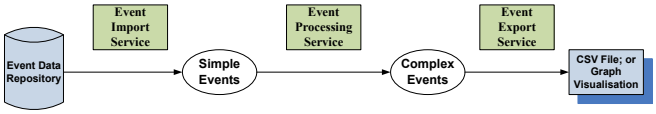

Figure 1: ADAGE processing

One significant limitation of ADAGE is that complex events generated by an "event processing service" do not contain detailed information regarding how they are generated, i.e. what pattern occurrence was detected signifying this complex event, and which simple events constitute the complex event. Further, since the services in the ADAGE framework are developed by different people, and they use various techniques (e.g. different programming languages) during the development, it is almost impossible to track the process of detecting a complex event.

In order to address this limitation we refined the ADAGE framework by splitting the event processing services into two separate type of services (Figure 2):

– Event pattern detection service: the service to detect occurrences of event patterns. (Input: events; Output: event pattern occurrences)
– Event pattern processing service: the service to process event pattern occurrences and convert them to complex events with all detailed information of the generation of the complex events, i.e. the final output required by the user. (Input: event pattern occurrences; Output: events).

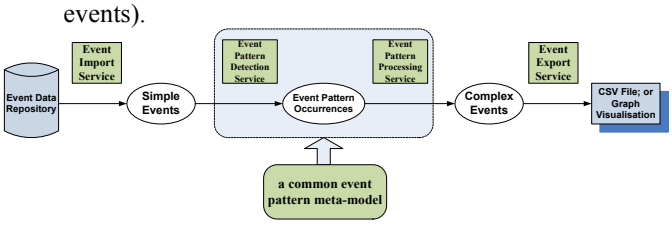

Figure 2: Extended ADAGE: CEP capability

With this refinement, the extended ADAGE framework has the ability to capture detailed information about the complex events, while retaining the original advantages, e.g. automation of data analysis, and the flexibility of building workflows. Further, the stream-oriented delivery of events to the event pattern processing service allows for connection of real-time data streams and consequently real-time analytics.

In order to capture complex event details we propose a common event pattern meta-model, which is described in Section IV. This meta-model benefits both developers and researchers. For event processing service developers, it is easier to implement services without thinking about how to represent the event pattern occurrences detected. For event export service developers (e.g. visualisation developers), the event pattern occurrences provide further valuable

information including the details of how the complex events are constructed. For external developers who want to call these services (e.g. to develop a rule-based system that needs to detect event pattern occurrences), the model provides the basis for defining a wire format for the event pattern occurrences. For researchers, with a more informative output such as visual representation based on the event pattern meta-model, the tracking of complex event generation is facilitated. In addition, if two services that require data interchange have different data models, the common meta-model makes it easier to relate those models and support different wire formats that reflect the same underlying meta model.

The work illustrated in this paper extends the results reported for the ADAGE framework in [4][5] in two ways: firstly, it provides extended *analysis* features to ADAGE owing to the ability to define trading patterns of interest and to capture how complex events matching those patterns are constructed from the input events; secondly, it allows real-time implementation of event pattern detection. Through applying these capabilities to stock market data and other relevant data sources, market participants can detect current or emerging insights to support trading and operational decisions.

III. FOUNDATIONAL CONCEPTS FOR BEHAVIOUR

This section introduces several fundamental concepts that will be formalised in the context of an event pattern meta-model in next section. They take into account relevant concepts from the RM-ODP standard [10], which provides precise definitions of foundational behavioural concepts such as actions, interactions, events and services in distributed systems, augmented with the definition of event patterns described in [1] and [2].

Note that the RM-ODP standard includes the description of various behavioural constraints, including deontic policy constraints [11][12], which are important for monitoring conditions associated with business policies [16]. These are not addressed in this paper but are described in detail elsewhere [11][13][19]. The fundamental concepts for behavior are required to ensure establishing a common understanding about modelling and downstream implementation of distributed systems and applications. This agreement on standard concepts is a necessary condition to ensuring interoperability among people and systems, in an open environment.

A. Action

RM-ODP defines *action* as 'something that happens'. Every action of interest for modelling purposes is associated with at least one object. The set of actions associated with an object is partitioned into internal actions and interactions. An internal action always takes place without the participation of the environment of the object. An interaction takes place with the participation of the environment of the object. Note here that "Action" means "action occurrence" not "action type". That is to say, different actions within a specification may be of the same type but still distinguishable in a series of observations. Depending on context, a specification may express that an action has occurred, is occurring or may occur [10].

B. Event

Event is described as 'the fact that an action has taken place'. When an action occurs, the information about the action that has taken place is captured in an event, and that event becomes part of the state of the system. An event may subsequently be communicated in interactions and this communication is called an *event notification*: it carries the information about the action from the object that performs or observes it to other objects that have a need to take action as a result of it [10].

C. Event Pattern Types and Ocurrences

Our interest is in using event processing systems to facilitate analytics activities, such as identifying data quality issues, performing exploratory analytics and ultimately developing an infrastructure to support predictive analytics in real time. The use of a special kind of event processing systems, i.e. Complex Event Processing (CEP) systems, allows the application of sophisticated techniques to *define* and *detect* interesting combination of events that have specific business meaning. A definition of such combination of events is referred to as an *event pattern type* and a CEP engine is thus utilised to detect occurrences of specific combinations of events that satisfy event pattern types. Such a combination is referred to as an *event pattern occurrence*. It is typically the combination itself, rather than individual events that carry business semantics.

An event pattern type is defined as a 'template specifying one or more combinations of events'. Given any collection of events, a CEP detection engine can find one or more subsets of those events that match a particular pattern type and thus satisfy this pattern type [2].

IV. EVENT-PATTERN META-MODEL

This section provides further elaboration of the concepts introduced in the previous section, using a UML meta-model, depicted in Figure **3**. The structure and informal semantics of events and event patterns is thus expressed as a combination of abstract syntax of the meta-model and narrative definition of the semantic concepts. Note that the concepts proposed as part of the RM-ODP framework have a formal semantics [12], but this is not discussed in this paper. There have been some other attempts for providing mathematic and logic based formalism for event pattern types and their detection, most notably [15]. Such formalism of detection semantics can constrain expressiveness, particularly in relation to parallel behaviour and time, without necessarily assisting in interoperability. Through providing narrative "hooks" our model allows formalism to be added as required to support implementation.

This meta-model represents a conceptual model focusing on that part of complex event processing that is concerned with describing events and event patterns (including event pattern types and event pattern occurrences). The meta-model does not capture other elements more related to the processing of events such as filtering, event aggregation,

channels for notifications etc. These are necessary elements for deploying any CEP implementation, but the focus of this paper is on the use of CEP event pattern matching as a data analysis or machine analytics technique.

A. Event

As introduced in section III.B, the concept of event signifies the fact that some action has happened in the real world. In an information system, an event is thus a record of some action that occurred in the real-world and it captures information about this occurrence. An event conforms to an event type as defined in the following section.

B. Event type

An event type characterizes a set of events that share particular properties. Typically, this includes information such as:

- when the action occurred, e.g. at particular point in time (instantaneous) or during particular interval (in which an action has start and end time associated with its occurrence)
- what action the event signifies, e.g. trade buy or sale,
- the source, that is where the action happened or was observed, e.g. on particular network, device, or particular software component
- event id, typically assigned by an information system when it creates or receives the events
- other application- or domain-specific data.

We model event type as an abstract type with two concrete classes: simple event type and complex event type. These are used to distinguish two key properties of event types and their corresponding instances. The following sections describe these two event types in terms of their components and corresponding instances.

C. Simple Event

A simple event is an event that signifies an occurrence of a single action. While in many cases a simple event is instantaneous, e.g. News event, it can also have duration, e.g. Intraday trade event.

A simple event typically has data modelled as an AtomicData element.

D. AtomicData

This modelling element is a generic attribute that captures business related data associated with an event, e.g. information about price of stock trade, volumes of trade etc. It is referred to as 'atomic' to signify the fact that event captures information about single action occurrence.

E. AtomicDataType

This modelling element specifies the type of AtomicData, defined by the format of data (e.g. CSV) and schema for the data (e.g. column definitions).

F. Functor

The concept of Functor is inspired by the Haskell functional programming language and is introduced to provide access to relevant information captured in an event regardless of its wire format or schema. For example there may be events capturing stock related announcements from different data streams, e.g. twitter, ASX, Google news etc, but we want to extract the relevant stock code in a consistent way for all streams. Thus we define a functor for each data source that satisfies a StockCode functor type.

G. Functor Type

Functor Type describes functions over events that return a value of a particular data type and semantics extracted or derived from an event instance (Event). Typically the value is extracted from the AtomicData element of an event, implying that functor instances are aware of the format and schema of the AtomicData element. Specific Functor instances can then be defined to access different data sources with different formats or schemas such as price or news item identifiers from providers such as Thomson Reuters. In many respects a Functor Type is a generalisation of the more usual notion of "attribute" that allows us to abstract over the way a CEP implementation accesses data in events, and also focus our modelling and programming effort on the data necessary for rule definition. Unused data can be ignored. This is particularly important for data arriving from different sources with different schemas and wire formats, as it allows us to establish relationships across different data streams in an abstract but unambiguous and implementable manner.

H. Complex Event

A complex event signifies occurrences of a combination of related events that have some business semantics. A complex event is used to capture event pattern occurences that satisfy event pattern types.. An event pattern type defines relationships between contained events. Note that A complex event can satisfy multiple event pattern types or none, so while event pattern occurrence implies complex event, the reverse is not true. A complex event can also include another complex event as part of that event combination.

I. Event Pattern Ocurrence

This concept signifies occurrence of a set of events which are related through some expression. The form of this expression is defined by an EventPattern Type.

J. Event Pattern Type

Event Pattern Type concept defines a specific relationship between events of some business significance. An Event Pattern Type is used to specify relationship, data and time constraints across constituent events.

There may be different type of pattern expression languages, combining mathematical, logical, statistical and various temporal constructs to define an event pattern. There are also a number of different event patterns types that were for example identified in [2].

K. Event Pattern Implementation

An Event Pattern Implementation refers to a specific implementation of an EventPatternType. One such implementation described in this paper makes use of the

concept of a pattern directed acyclic graph or P-DAG, as shown in Figure **4** and described next.

L. Pattern DAG (P-DAG)

When designing the implementation of event pattern types, we consider the constituent events in an event pattern type, and the dependency between constituent events, including the temporal order of events, as the key factors. Thus, we adopt a directed acyclic graph called P-DAG (Pattern DAG) as a specific implementation of an event pattern type. P-DAG is thus used as an expression for specifying occurrences of a certain event pattern type. The advantage of P-DAG is that it can capture constituent events in an event pattern type as well as the dependencies between them. This leads to an easy representation of the event pattern type in a graphical way that is easy to understand for both developers and end users.

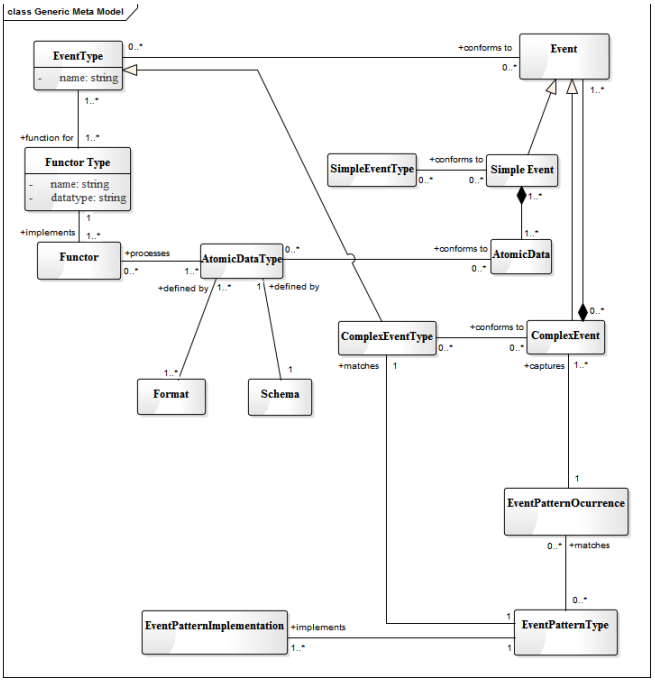

Figure 3: Core CEP meta-model

Formally, a P-DAG instance is defined as a DAG of Edge Types (*E*) connecting Node Types (*N*) as follows:

- *P-DAG* = < *N, E* >
- *N* = <n_1, n_2, ... >
  - n_i is a set of event(s), n_i = <e_1, e_2, ... >
  - e_i is an event (i = 1, 2, …)
- E = <edge$_1$, edge$_2$, … >
  - $edge_i$ is an edge between an ordered pair of nodes
  - $edge_i$ is defined by the ordering semantics (*source*, *target* and the *ordering option*)
  - *source* and *target* are two nodes in the P-DAG instance, which specify the order of two nodes
  - *ordering options* specifies additional rules of the ordering, e.g. the start time of the source node must be earlier than the start time of target node

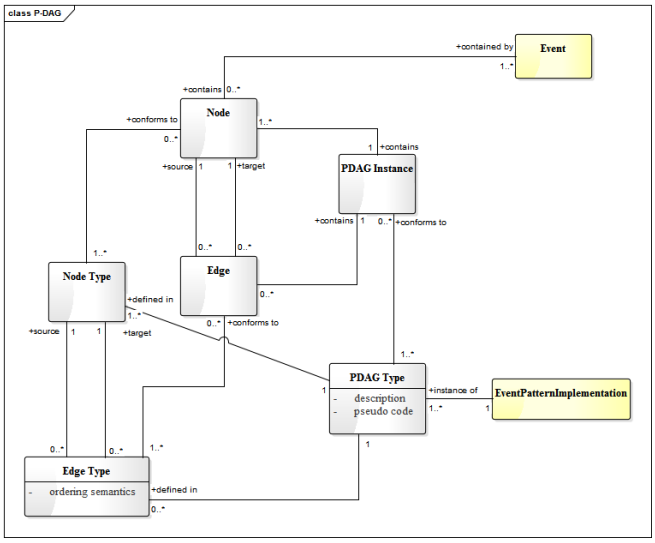

Figure 4: Event Pattern Type implementation: PDAG expressions

To sum up, a *node* depicts a constituent event in an event pattern occurrence; an *edge* represents the ordering dependencies between nodes (constituent events).

In a P-DAG, N is a non-empty set of nodes depicting all constituent events in an event pattern occurrence. The number of events represented by a node is called Node Cardinality. By default, a node denotes a single event; otherwise, a box with an annotation indicating the cardinality of the node will be attached to the box. The cardinality is an integer range, which can be from 0 to infinity.

V. CASE STUDY

This case study illustrates a number of specific event pattern rules used in finance research groups and a rule-based front-end application [14] that invokes EventSwarm.

A. Testing application

Figure **5** shows the implementation of an analytics application (AA) developed to support researchers/domain experts interested in experimenting with different event pattern types.

The application has a Front-End which is a known rule-based application identified by the researcher organisation. This Front-End has applied a data model built upon the proposed meta-model. It plays the role of managing the process of data analysis. The EventSwarm software framework is used as an event pattern detection service. This software framework was introduced in [7] and its use is described in section VI.B. It is important to note that both the Front-End component and EventSwarm component implement concepts that are compliant with the meta-model introduced in the previous section.

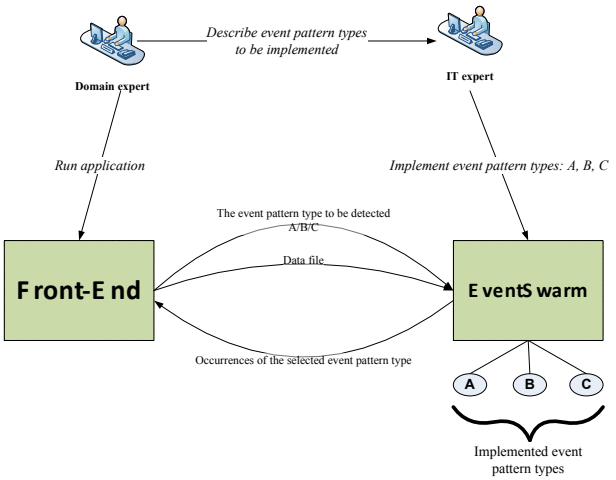

Figure 5: Implementation Environment

The following steps were applied in deploying and testing the finance event pattern types:

1. A researcher specifies a set of interesting event pattern types. The specifications of the event pattern types are described in a natural language in writing or verbally to an IT expert.
2. The IT expert implements the event pattern types in the event pattern detection service and makes them available for invocation by the Front-End application.
3. The researcher invokes the event pattern detection service exposed via the Front-End. To invoke the event pattern detection service, the researcher needs to select a desired event pattern type (e.g. Duplicate dividend rule with a specific ID) from the list of event pattern types that have been implemented, and provide a data file to be analysed (e.g. SGB.csv data set). The Front-End then passes these parameters to EventSwarm using an HTTP GET request to a configurable URL. Alternatively, the researcher can conduct event pattern detection tasks via the EventSwarm pattern detection service GUI.
4. The EventSwarm engine then returns detected event pattern occurrences in JSON format reflecting the model described in IV. Finally, the occurrences are further processed by the Front-End.

It is worth noting that the results can be both displayed on the EventSwarm GUI or delivered to a known application identified by the researcher organisation. This programmatic delivery is done using an HTTP POST request to a configurable URL.

Rules are implemented as Ruby classes in a Rails application at present, but we anticipate developing a domain specific language or user interface allowing researchers to define new rules in the future. This domain specific language will support a limited subset of the possible pattern types that can be expressed in the model from [7].

B. *Event patterns considered*

Table **1** describes the event pattern types related to financial market data that have been identified as candidates for analysis. There are 24 event pattern types that have been taken into consideration, of which 14 event pattern types have been selected for implementation due to the high interest for the research teams involved (see Table 1). The rest of the event pattern types considered are of less interest in this project; and we will implement them as part of our future work.

C. *Results*

Experiments were conducted by a researcher with some assistance of an IT expert. The researcher used the implemented analytics application (AA) to manage and conduct data analysis. First, the Front-End was used to define the processes of three different data analysis scenarios. They are:

- Detecting occurrences of event pattern type No. 2 (1 event pattern type involved)
- Detecting occurrences of event pattern types No. 1-7 (Handling data quality issues of dividend events; 7 simple event pattern types involved)
- Detecting occurrences of event pattern types No.8, 10-14 (Calculating earnings; 6 more sophisticated event pattern types involved)

The same analysis processes were implemented as a local bespoke program for comparison purposes. The researcher then executed these three analysis processes in AA and in the bespoke program respectively and inspected the results. For all the three scenarios, the analysis processes were conducted successfully with identical results. This indicates that the AA performed well and yielded the same results. Most importantly, the interoperability between the Front-End and EventSwarm was ensured due to the fact that the underlying data models in both the Front-End and EventSwarm are built on the same meta-model.

Table 1: Implemented event pattern types

	Name	*Event Pattern Description*
1	Dividend event	An event is a "Dividend" event.
2	Duplicate Dividends	Two events with Type "Dividend" have the same timestamp, the same "Div Amt." and the same "Div Ex Date".
3	Missing EOD Event on Dividend Ex Date	No "End Of Day" event exists with "Div Ex Date" of a "Dividend" event as the timestamp.
4	Div Missing Div Amt or Ex Date	An event with the type "Dividend" has null or empty value in the field "Div Amt." or "Div Ex Date".
5	Dividends with Different Div IDs	A pair of duplicate dividends (pattern type No. 4) have different "Div Mkt Lvl ID".
6	Status is not 'APPD'	A "Dividend" event has a value other than 'APPD' in the field "Payment Status".
7	Delete Marker is not '0'	A "Dividend" event has '1' in the field "Div Delete Marker".

8	Earning before End Of Day	*E -> EOD* An event with type "Earning" (E) happens before an event with type "End Of Day" (EOD). (Find only one closest occurrence for each EOD if it exists.)
9	12-month Earning before End Of Day	*E_{12} -> EOD* An event with type "Earning" (E_{12}) happens before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of both E_{12} is 12 (Find only one closest occurrence for each EOD if it exists.)
10	Two 6-month Earnings before End Of Day	*$E_{6(2)}$ -> $E_{6(1)}$ -> EOD* Two events $E_{6(1)}$ and $E_{6(2)}$ with type "Earning" ($E_{6(2)}$ before $E_{6(1)}$) happen before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of both $E_{6(1)}$ and $E_{6(2)}$ is 6; • $E_{6(2)}$.epsEndDate + $E_{6(2)}$.epsLength = $E_{6(1)}$.epsEndDate • Find only one closest occurrence for each EOD if it exists.
11	Two 3-month earnings and one 6-month earning before End Of Day	*$E_{3(2)}$ -> $E_{3(1)}$ -> E_6 -> EOD* Three events with type "Earning" ($E_{3(2)}$ before $E_{3(1)}$ before E_6) happen before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of $E_{3(2)}$ and $E_{3(1)}$ is 3; The "EPS Period Length" of E_6 is 6; • $E_{3(2)}$.epsEndDate + $E_{3(2)}$.epsLength = $E_{3(1)}$.epsEndDate • $E_{3(1)}$.epsEndDate + $E_{3(1)}$.epsLength = E_6.epsEndDate • Find only one closest occurrence for each EOD if it exists.
12	One 3-month earning and one 9-month earning before End Of Day	*E_3 -> E_9 -> EOD* Two events E_3 and E_9 with type "Earning" (E_3 before E_9) happen before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of E_3 is 3 and The "EPS Period Length" of E_9 is 9; • E_3.epsEndDate + E_3.epsLength = E_9.epsEndDate • Find only one closest occurrence for each EOD if it exists.
13	Four 3-month earnings before End Of Day	*$E_{3(4)}$ -> $E_{3(3)}$ -> $E_{3(2)}$ -> $E_{3(1)}$ -> EOD* Four events $E_{3(1)}$, $E_{3(2)}$, $E_{3(3)}$, and $E_{3(4)}$ with type "Earning" ($E_{3(4)}$ before $E_{3(3)}$ before $E_{3(2)}$ before $E_{3(1)}$) happen before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of $E_{3(1)}$, $E_{3(2)}$, $E_{3(3)}$, and $E_{3(4)}$ is 3; • $E_{3(i)}$.epsEndDate + $E_{3(i)}$.epsLength = $E_{3(i-1)}$.epsEndDate (i=2,3,4) • Find only one closest occurrence for each EOD if it exists.
14	One 9-month earning and one 3-month earning before End Of Day	*E_9 -> E_3 -> EOD* Two events E_3 and E_9 with type "Earning" (E_9 before E_3) happen before an event with type "End Of Day" (EOD) with: • The "EPS Period Length" of E_3 is 3 and The "EPS Period Length" of E_9 is 9; + E_9.epsLength = E_3.epsEndDate • Find only one closest occurrence for each EOD if it exists.

The researcher and the Front-End developer have reported a number of advantages from driving the detection of event pattern occurrences using the concepts from the meta-model. In particular, the meta-model facilitates portability, allowing the Front End to consume the output of EventSwarm and thus leverage its ability to detect certain types of patterns without being locked with a particular EPS. Indeed, the invoking platform can call other EPSs to detect other types of patterns without the need to adapt to the output type specified by that EPS. Further, the EventSwarm platform provides additional implementation benefits, including:

- simple and easy-to-use API: developers can easily integrate EventSwarm into their own applications.
- very fast and efficient complex event processing. The average speed of processing is more than 10,000 events per second (remote invocation time inclusive) on Thomson Reuters Tick History daily data provided by Sirca. This is almost as fast as a bespoke program dedicated to a fixed event processing process and executed locally.
- implementing event pattern types is generally very fast. It normally takes less than a day to implement 5 event pattern types.
- JSON as the output format is well structured and it is convenient for developers to parse and further analyse the results.

Some limitations include:

- Users are not able to define event pattern types via the GUI or an API, and thus the development cycle largely depends on the availability of the IT expert.
- The communication between the researcher and the IT expert can be very intensive so that the definition of event pattern types described by the researcher can be understood accurately by the IT expert. Any failure in the communication may cause issues that will be hard to diagnose in the future.

VI. IMPLEMENTATION

A. Front-End Application

The front-end application provides the capability to run event processing rules to process financial market data, and to manage these rules in an incremental way. The GUIs and all business logic of the application are implemented using Java. The rules are stored in a PostgreSQL database. The front-end application sends requests to EventSwarm using RESTful invocations and receives the responses encoded using the wire format described in section D below.

B. EventSwarm

The EventSwarm implements the meta-model concepts from section IV and is used as the pattern detection service. It provides both a user interface and a RESTful interface for matching patterns against data sets. Upon completion of processing, it displays the result on the user interface and passes the result back to the calling application using an HTTP POST request containing matches encoded using the wire format described in section D below.

The EventSwarm service is implemented using Ruby on Rails on the JRuby platform. The specified patterns were coded in Ruby, and are called in response to requests from a user or external application. The Ruby "patterns" are primarily constructors that build a CEP graph using EventSwarm core constructs. These constructs are provided through the EventSwarm core Java library with convenient Ruby wrappers to facilitate rapid development. Pattern matching execution primarily occurs in Java for maximum performance, although some elements of the earnings patterns are implemented in Ruby. Encoding of results into the wire format and sending is also implemented in Ruby.

Patterns are matched in an EventSwarm application by feeding events through one or more processing graphs that select matching events or sets of events. Processing graph nodes can include sliding windows, filters, splitters (powersets) and abstractions. Abstractions are values or data structures calculated or constructed from the stream of events, for example, the EventSwarm statistics abstractions maintains sum, mean, variance and standard deviation over numeric data extracted from events in a stream. Events can be added to and removed from the stream by a node, although downstream nodes can choose to ignore removals. For example, a sliding window works by instructing downstream nodes to remove events that have "fallen out" of the window.

C. Rule implementation

The rules are implemented using four common designs and one pattern-specific design as described below.

1) Simple filters

Some rules were implemented using a simple, single-event filter that collected events matching one or more static field values. For example, the "dividend deleted" rule matched events with a type of "Dividend" and a value of "1" in the "Div Delete Marker" field.

2) Duplicate detectors

Other rules needed to detect duplicates in the data stream. Duplicate detection can have high memory and processor overheads because it is necessary to hold a potentially unbounded set of "candidates", and each new event has to be compared with all of the preceding candidates. Thus we use a candidate filter to minimise the number of candidates, then use the EventSwarm `DuplicateEventExpression` to compare candidates with new events using one or more event comparators. An example of such a duplicate detector is the "Duplicate dividend" pattern, which filters the set of candidates so that only "Dividend" events are considered, and then compares candidate stock code, amount and div-ex date to identify duplicates.

It is important to note that if we were analysing a continuous stream, we would also use a sliding time window or sliding N-sized-window to avoid infinite buffering of candidates. Use of such sliding windows can decrease the accuracy of pattern matching because some matches might be missed, but in most practical scenarios, there is no loss of accuracy because duplicates are close together in the data stream. For this implementation, we relied on the finite size of the data sets analysed rather than using a sliding window.

3) Simple event sequence

Two rules matched a simple sequence of events, with each event in the sequence satisfying certain static conditions. These rules were implemented using the EventSwarm `SequenceExpression` with simple attribute matchers for each event in the sequence. For example, the "Earnings event before EOD" used `SequenceExpression` to look for an "Earning" event followed by an EOD event. Note that normally, this will generate a match for *any* sequence that satisfies the sequence expression components, meaning an EOD event would be paired with *all* previous earnings events. To ensure that EOD events were only paired with the most recent earnings event, only one candidate earnings event was held (i.e., a sliding window of size 1).

4) Conditional event sequence

A number of sequence patterns were implemented where the events of each candidate sequence needed to satisfy an additional condition defining necessary relationships between the events in the sequence. In EventSwarm, we implement this by first matching the sequence, then applying the inter-event condition as a filter on the candidate sequence matches. For example, the "Bi yearly earnings before EOD" required that the two earnings events that started the sequence were contiguous in time. This was matched by looking for a sequence of two 6-month earnings events followed by an EOD event, then filtering the resulting sequences to match only those sequences where the earnings events were contiguous in time.

A duplicate event filter was used in front of these expressions to ensure that duplicate earnings events were removed and only a single match was generated for each earnings period.

5) Event not present in history

The "Dividend without a valid EOD for the div ex date" pattern required that we identify cases where a "Dividend" event div-ex date (the end date of the period for which the dividend was paid) did not have a valid EOD event for the div-ex date. The "Dividend" event normally occurs within a month of the div-ex date. To implement this pattern, we maintained an EventSwarm sliding time window to hold the last 31 days of valid EOD events (i.e. at least one month), then for each matching "Dividend" event, the time window was searched to determine if it contained an EOD event that matched the div-ex date.

Some elements of this expression were implemented in Ruby because no combination of existing EventSwarm components could implement the relatively obscure semantics. Note that this pattern implies negation, which is

not easily implemented in CEP systems generally. See the evaluation in section E further discussion of negation.

D. Wire format

The wire format used to express patterns is a direct reflection of the P-DAG model described in Section IV. It uses JSON [17] for simple and efficient cross-platform processing. A sample of a JSON file that saves event pattern occurrences is shown in Figure **6**. Each JSON file contains a number of event pattern occurrences that match a particular event pattern type. Each occurrence contains a *description* of the event pattern type and a *P-DAG instance*, which consists of a number of *nodes* and *edges*. Each *node* consists of the *event type*, id, source, start time, end time and a list of other *atomic data*. Each *edge* consists of *ordering*, *source* and *target*.

Figure 6: Sample JSON file of Pattern Occurrences

E. Evaluation

The implementation of the patterns using EventSwarm was mostly straightforward and required only minimal programmer effort. It did, however, continually highlight complex semantic issues in pattern specification and implementation. Key examples are identified in the following subsections.

1) Timestamp precision and ordering

Some of the data sets processed had timestamps with a precision of 1 day (i.e. no time component). Thus strict "before" relationships in patterns would not fire unless dates were different. For example, if one searched for a dividend announcement followed by an end-of-day event, the pattern would only match end-of-day events on subsequent days. This is a general problem of precision in timestamps: sequence patterns can only match for events separated by a period greater than or equal to the timestamp precision. So if timestamp precision is 1 second, events separated by less than 1 second cannot be distinguished in time and thus cannot be sequenced.

2) Bounding candidate matches

A pattern that requires two or more events to match (e.g. A AND B) requires the solution to hold candidate A matches for subsequent pairing with B events. For a continuous data stream, an explicit or implied bound is required to make the pattern scalable, because each A event needs to be held as a candidate match until it can be determined that no further B events are possible. Thus to make the required storage finite, we need a bound on the number of A events held as candidates (e.g. in the last hour) or through an indicator that implies no further B events are possible (e.g. end-of-data-set).

3) Negation

Negation can be particularly difficult to implement in CEP systems. Consider the example NOT(A). Over a continuous data stream, at what point can we assert that A has not occurred? Similar to the problem of candidate matches for sequence or conjunction queries, we need an explicit or implied bound for the evaluation semantics. For example, we could evaluate the pattern at regular intervals (i.e. every hour) or evaluate it over a limited time window (e.g. in the last hour). A further complication with negation is in deciding what to report as the match. What is the pattern matched by NOT(A)? Is it the set of events that *has* occurred? Or is it an empty result? This question becomes even harder to answer when conjunctions are used with negation. Thus a general pattern specification language that permits negation must provide mechanisms and semantics to address these issues.

4) Edge semantics

The P-DAG model for event pattern occurrences includes edges between events. These are defined in terms of the events and in this implementation, the edges reflect a strict "before" relationship between events. This is simple to implement and very general because it bears no relationship to the pattern specification. At present, the wire format also does not identify node types. Thus it can be difficult for a researcher to determine which event matched which element of a pattern without re-evaluating the pattern constraints locally (i.e. outside of EventSwarm).

It is anticipated that researchers defining pattern types will need to associate events with "placeholders" in the pattern type. For example, if a pattern A -> B AND A' -> C (A == A') is evaluated against a data stream, the researcher needs to know which event matched A, A', B and C respectively. To do this requires the association of explicit edge semantics with the pattern specification and/or explicit labelling of nodes, assuming re-use of node types (e.g. A, A'). This adds considerable complexity to pattern specification, the implementation and the wire format. The required semantics, likely complexity and implementation effort are currently being investigated.

VII. Conclusions

This paper has introduced a minimal set of modelling concepts related to the specification of event pattern types and event pattern occurrences. The meta-model was developed based on the foundational behavioural concepts from the RM-ODP standards, augmented with a number of concepts needed to support event pattern matching semantics for real-time analytics applications. We have shown that these concepts are sufficient to represent and implement a class of business analytics solutions for a number of use cases in finance related to market trading. In particular, the CEP EventSwarm framework, which is compliant with these

concepts, allowed quick deployment of new rules, even of significant complexity, while delivering high performance execution of these rules in real-time. The software was deployed in a cloud environment and its services were invoked through the Front End application, which is also compliant with the meta-model.

This case study has also revealed a number of semantics difficulties inherent to many CEP applications. In particular, the examples from section VI.E suggest that developing a general-purpose pattern language has limited value for *researchers* due to the inherent complexity of pattern specification and the difficulty of scaling without explicit bounds. We believe that researchers would be better served by limited, domain-specific pattern languages with a tractable matching semantics pre-determined (e.g. implied bounds, sequence and negation semantics that reflect natural data constraints, node labelling). The implication is that unless an event processing domain is strictly bounded, the complexity of an adequate pattern language is high enough that *programmers* will be required to implement patterns regardless of the language syntax and semantics.

There might be some value for *programmers* in using a general-purpose pattern language rather than Ruby or similar OO/procedural languages, but the development of such a language is a large and complex body of work. Witness both the complexity and limitations of the many language choices in this domain discussed in [18]. The approach applied by EventSwarm, which is to make core constructs available in a general purpose programming language, provides agility, extensibility and accessibility. In particular, it allows existing application development frameworks like Rails to be used in the construction of CEP applications. The native availability of database, user interface, testing and deployment capabilities, coupled with the wide availability of third-party modules makes this a compelling approach to building robust, usable and production-quality applications.

It should be noted that the semantic issues identified cannot be fully expressed in the recent formalism published in [15], justifying our decision not to capture formal detection semantics in the meta-model at this point in our work. Our experience from this implementation, however, will help in ensuring that future formalisation and domain languages have adequate expressiveness.

In future we plan to implement a full set of rules listed in section V.B. We also plan to look at more complex event pattern occurrences related to cross-correlation between stock market events and the social media postings, in order to be able to develop new insights into relation between streams of events coming from different sources. We will continue to monitor developments in event pattern formalisms, and leverage these developments to create suitable and tractable domain specific languages for pattern definition.

ACKNOWLEDGEMENTS

We would like to thank the Smart Services Cooperative Research Centre in Australia for sponsoring our research project and Sirca for providing data used in the case study.

REFERENCES

[1] D. Luckham, The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. MA, USA: Addison Wesley Professional, 2002.

[2] O. Etzion and P. Niblett, *Event Processing in Action*: Manning Publications Co., 2011.

[3] http://eclipse.org/modeling/emf/

[4] Yao, L. and Rabhi, F. A. (2014), Building architectures for data-intensive science using the ADAGE framework. Concurrency Computat.: Pract. Exper.. doi: 10.1002/cpe.3280

[5] Rabhi FA, Yao L, Guabtni A. ADAGE: a framework for supporting user-driven ad-hoc data analysis processes., Computing 2012; 94(6):489–519.

[6] http://www.deontik.com/Products/EventSwarm.html

[7] A. Berry, Z. Milosevic: Real-Time Analytics for Legacy Data Streams in Health: Monitoring Health Data Quality. IEEE EDOC 2013 conference, p. 91-100, 2011

[8] http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/default.asp

[9] www.asx.com.au/documents/about/guidance-note-8-clean-copy.pdf

[10] ITU-T/ISO, "ITU-T X.902 | ISO/IEC 10746-2, Information Technology Open Distributed Processing Reference Model – Foundations", 2010.

[11] ITU-T/ISO, "ITU T Rec. X.911 | ISO/IEC 15414: Enterprise language DIS for ITU T Recommendation X.911| ISO/IEC 15414 Amd 1", 2013.

[12] ISO/IEC IS 10746-4, Information Technology — Open Distributed Processing — Reference Model: Architectural Semantics, 1998. Also published as ITU-T Recommendation X.904.

[13] P.F. Linington, Z. Milosevic, A. Tanaka and A. Vallecillo, *Building Enterprise Systems with ODP, An Introduction to Open Distributed Processing*, Chapman & Hall/CRC Press, 2011.

[14] Chen, W. & Rabhi, F. A., An RDR-Based Approach for Event Data Analysis. In *Proceedings of 3rd Australasian Symposium on Services Research and Innovation* (ASSRI'13).

[15] Sylvain Hallé, Simon Varvaressos: A Formalization of Complex Event Stream Processing. EDOC 2014: 2-11

[16] OMG, "Semantics of Business Vocabularies and Rules", Available at http://www.omg.org/spec/SBVR/

[17] ECMA, "The JSON Data Interchange Standard", http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[18] G. Cugola and A.Margara, "*Processing Flows of Information: From Data Stream to Complex Event Processing*", ACM Computing Surveys, Vol. 44 No. 3, June 2012.

[19] Guido Governatori, Zoran Milosevic, Shazia Sadiq: Compliance checking between business processes and business contracts. EDOC 2006.