Int J Data Sci Anal (2016) 2:13–27
DOI 10.1007/s41060-016-0029-7

REGULAR PAPER

An open architecture for event-based analytics

Zoran Milosevic[1] · **Weisi Chen**[2] · **Andrew Berry**[1] · **Fethi A. Rabhi**[2]

Received: 25 June 2016 / Accepted: 4 October 2016 / Published online: 17 October 2016

Abstract Event-based analytics is increasingly gaining prominence in business and social applications. Despite the availability of many solutions specializing in event processing systems (e.g. CEP technology), there is currently no commonly agreed way of describing event and event pattern types, and thus no standardized method for interchange of event pattern instances between systems. This paper advocates an open architecture for event-based analytics comprising a common model that supports interoperability of data between different systems. It introduces the foundational concepts for describing event patterns including events, event pattern types and event pattern occurrences. The event pattern meta-model is also formalized using a UML meta-model to facilitate its adoption and usage across the event analytics community. The paper provides a case study introducing several event pattern types from the financial market data analytics domain. This case study illustrates a number of specific event pattern types used by finance experts and an application that requires interoperability between two separate software component frameworks (a rule-based front-end and a CEP). Results show that the meta-model concepts are sufficient to represent and implement a class of real-life business analytics solutions. The paper also identified a number of semantic challenges in developing interoperability solutions for the event-based processing, in spite of the fact that we needed to merge only two separately developed event-based conceptual models.

✉ Weisi Chen
chenw@cse.unsw.edu.au; chenweisi.work@gmail.com

Zoran Milosevic
zoran@deontik.com

Andrew Berry
andyb@deontik.com

Fethi A. Rabhi
f.rabhi@unsw.edu.au

[1] Deontik Pty Ltd, Brisbane, Australia

[2] School of Computer Science and Engineering, University of New South Wales, Sydney, Australia

Keywords Complex event processing · Event-based analytics · Real-time analytics · Data modelling · Service-oriented architecture · ADAGE

1 Introduction

The growing availability and access to data offer 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 in this paper is on *event-based analytics*, i.e. analysing a set of events that are reflecting some changes in real world. Note that event-based analytics is taken in a broad context covering both real-time analytics and processing of historical event data. Examples of event-based analytics include determining relationships between observations of patient condition in health care, particular combinations of buy and sell events in stock trading or correlation between social media postings and stock market activity.

In a big data context, event-based analytics requires sophisticated technology infrastructure and techniques. This includes complex event processing (CEP) [1], visualization tools, statistics software packages, natural language processing tools, machine learning libraries and so on. Note that CEP technology primarily allows detection of event pattern occurrences against events arriving with high velocity, often from multiple data sources but can be also used for historical analytics, which is of high value in our case study in financial event analysis. CEP technology can be 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 CEP solutions as well as more general platforms for event processing systems, but there is currently no standardized way of describing event types and event pattern types, and thus no standardized method for interchange of event pattern instances between systems [2]. This presents a problem when various big data and analytics tools and techniques need to be linked together.

The main contribution of this paper is in advocating an open architecture for event-based analytics comprising a common model that supports interoperability of data between different systems, as mentioned above. 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 collaboration 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, leveraging the power of model-driven development engineering techniques, and supporting tools such as Eclipse Modelling Framework [3]. Note that this meta-model can be also considered in the context of growing set of tools and technologies developed to support real-time analytics requirements, as we have identified in [4]. This paper provides further detail to the initial results we presented in [5], including an extensive analysis of the benefits and some further challenges in developing interoperability solutions for the event-based processing. These challenges involve complex semantics issues associated with the interpretation of event patterns and their impact on the specification and implementation of event-based systems. These issues involve timestamp precision and ordering, bounding candidate matches, implementation of negation as well as discussion of the intricacies of the pattern expression semantics chosen to represent event patterns. These challenges have faced us both during the conceptualization and implementation efforts.

The primary audience of this paper consists of 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 researchers involved in studying data in particular application domains, such as domain experts involved in financial market analysis as discussed in [6] and [7]. 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 next section presents the motivation for this work arising from new opportunities and challenges related to event-based analytics including real-time aspects, with particular emphasis on supporting business analytics and data science requirements. Section 3 introduced our concept of an open architecture and interfaces to support event-based analytics. Section 4 introduces the foundational concepts for describing event patterns including events, event pattern types and event pattern occurrences and formalizes the concepts through an event pattern meta-model. Section 5 provides a case study introducing several event pattern types from the financial market trading (equity) domain and describes a prototype implementation using the EventSwarm software framework [8,9]. Section 6 presents results based on performing three different data analysis scenarios. Finally, Sect. 7 discusses related work and Sect. 8 summarizes key findings and describes our future work.

2 Motivation

This research was 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. These scientists needed a fast and flexible way of identifying and defining new event pattern types in order to detect opportunities or threats associated with market trading. For example, some event patterns can be associated with detecting unusual spikes in price or volume of a specific stock, or identifying particularly poor or outstanding performance of a stock compared to others in its sector. This type of analysis can be conducted in real time or as part of batch processing (historical analysis).

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. Further, for real-time applications in particular, the high velocity of trading requires an automated way of detecting trading event pattern occurrences and an easy way to define the corresponding event pattern types. Real-time requirements often require modifying traditional statistics and machine learning algorithms for the intricacies of real-time requirements, e.g. continual evaluation of standard deviation or regression over sliding time window.

There are many existing CEP solutions, e.g. Apama [10], but the main problem with them is high costs and difficulty in adapting them to specific domains. For the types of the users that we are targeting one would need an open approach with an ability to leverage existing technologies such as R [11,12], Web services, REST [11], JSON [13], Open Calais [14], Amazon Kinesis [15], cloud analytics from Azure [16].

The open architecture suggested in this paper extends an earlier architecture that was based on the ADAGE framework [6,7]. It provides extended *analysis* features owing to the ability to integrate complex event processing services to detect patterns of interest from event data. This allows event pattern detection to occur in real time. Through applying these capabilities to stock market data and other relevant data sources, market participants can detect current or emerging insights into support trading and operational decisions.

3 Proposed open architecture

As mentioned earlier, the proposed open architecture is an extension of the one presented in [6,7]. In the original architecture, there are three types of services (see Fig. 1) that can be flexibly composed 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: event data query; Output: simple 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 high-level event [1] (Input: events; Output: (high-level) 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: high-level events; Output: csv/image).

One significant limitation of this architecture is that high-level events generated by an "event processing service" do

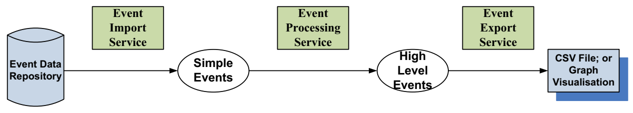


Fig. 1 Initial ADAGE architecture

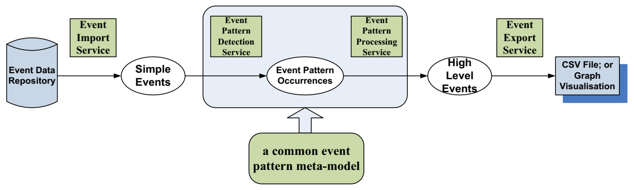

Event Data Repository
Event Import Service
Simple Events
Event Pattern Detection Service
Event Pattern Occurrences
Event Pattern Processing Service
High Level Events
Event Export Service
CSV File; or Graph Visualisation
a common event pattern meta-model

Fig. 2 Extended ADAGE: CEP capability

not contain detailed information regarding how they are generated, i.e. what pattern occurrence was detected in relation to this high-level event, and which simple events constitute a high-level event. Further, since the services in this architecture are developed by different people, and they use various techniques (e.g. different programming languages) during the development, it is almost impossible to track how a high-level event was originally detected and possibly change some pattern detection parameters.

The proposed architecture addresses this limitation by splitting the event processing services into two separate type of services (Fig. 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 high-level events with all detailed information of the generation of the high-level events, i.e. the final output required by the user (Input: event pattern occurrences; Output: events).

With this refinement, the extended architecture has the ability to capture detailed information about high-level events, while retaining the original advantages, e.g. automation of data analysis, and the flexibility of building workflows.

In order to capture high-level event details, we propose a common event pattern meta-model, which is described in Sect. 4. 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. visualization developers), the event pattern occurrences provide further valuable information including the details of how the high-level events are constructed. For external developers who want to invoke 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 domain experts, with a more informative output such as visual representation based on the event pattern meta-model, the tracking of high-level 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.

4 Event pattern meta-model

This section introduces several fundamental concepts that will be formalized in the context of an event pattern meta-model that was developed to support interoperability for event-based analytics, and initially proposed in [5]. These fundamental concepts are based on the relevant concepts from the RM-ODP standard [17], 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 [18], which are important for monitoring conditions associated with business policies [19]. These are not addressed in this paper but are described in detail elsewhere [18,20,21]. The fundamental concepts for behaviour 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.

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 [17].

4.1 Overview of the proposed meta-model

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 [17].

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. The definition of such a combination of events is referred to as an *event pattern type*, and a CEP engine is thus utilized 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].

The event pattern meta-model formalizes the concepts of event type, event pattern type, event pattern occurrences and related concepts, using a UML meta-model, depicted in Fig. 3. The structure and informal semantics of events and event patterns are 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 [22], 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 [23]. 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. 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.

4.2 Main concepts

4.2.1 Event

As introduced above, 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

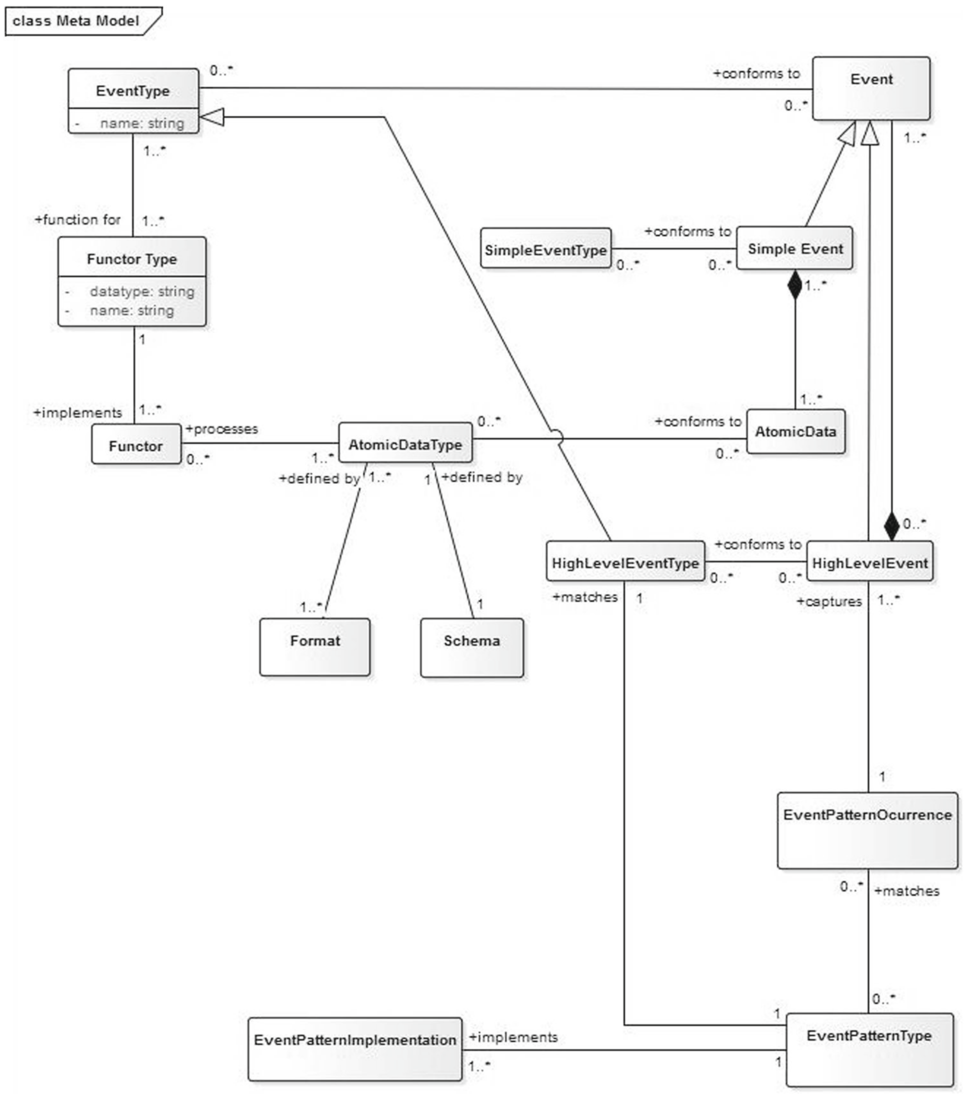


Fig. 3 Event pattern meta-model

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.

4.2.2 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 high-level 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.

4.2.3 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.

4.2.4 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. It is referred to as "atomic" to signify the fact that event captures information about single action occurrence.

4.2.5 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).

4.2.6 Functor

The concept of functor is inspired by the functional programming community 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, 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.

4.2.7 Functor type

A 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). As shown in Fig. 4, 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 generalization 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.

4.2.8 Event pattern occurrences, event pattern types and high-level events

An 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].

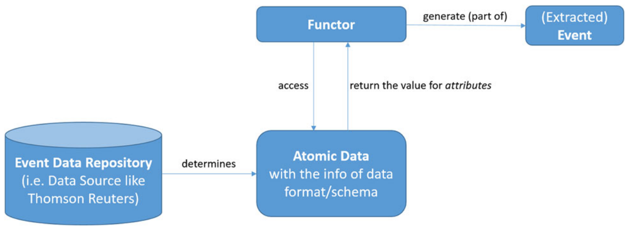

Functor
generate (part of)
(Extracted) Event
access
return the value for *attributes*
Event Data Repository (i.e. Data Source like Thomson Reuters)
determines
Atomic Data with the info of data format/schema

Fig. 4 How a functor works

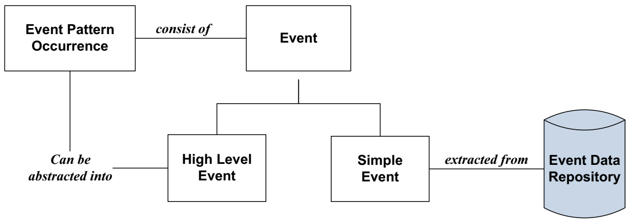


Fig. 5 Relationship between events and event pattern occurrences

An event pattern occurrence signifies the occurrence of a set of events which are related through some expression. The form of this expression is defined by an EventPatternType.

A high-level event is an event derived from event pattern occurrences rather than obtained from a data source. It is used to simplify event pattern occurrences so that they be used in further analysis. A high-level event may include high-level events as well as simple events as part of that event combination. These relationships are illustrated in Fig. 5.

4.2.9 Event pattern representation

An event pattern representation refers to the way events are interlinked within a specific implementation of an EventPatternType. The implementation described in this paper makes use of the concept of a pattern directed acyclic graph or P-DAG, as shown in Fig. 6 and described next.

4.3 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

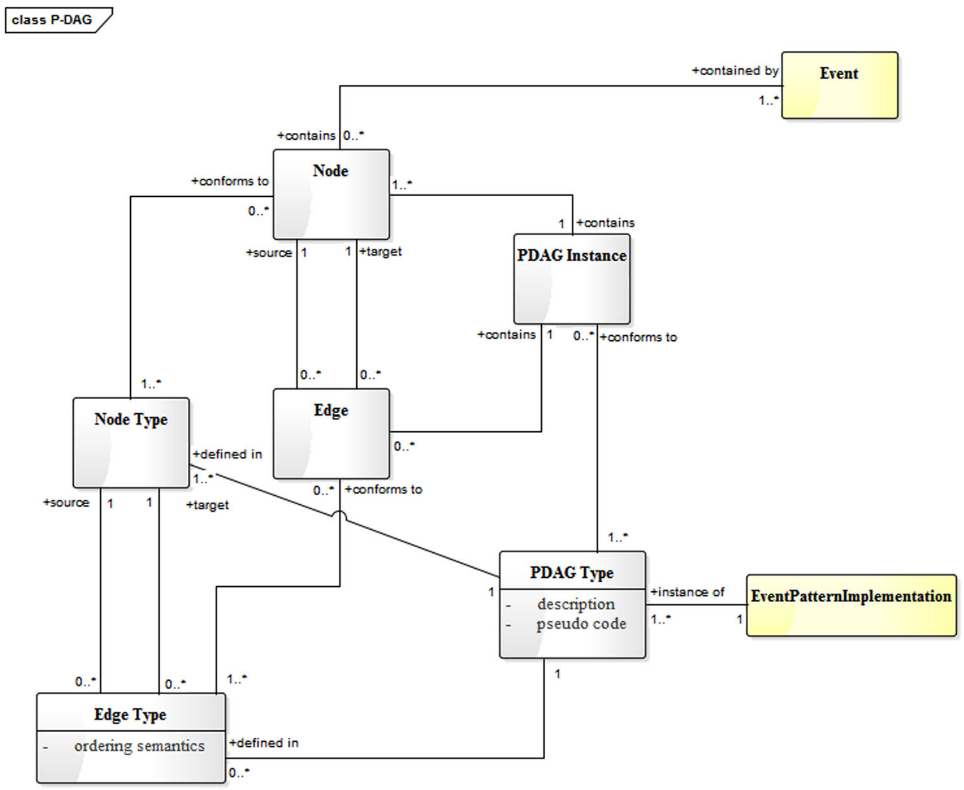


Fig. 6 Event Pattern Type implementation: PDAG expressions

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.

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 $=<$ $\mathrm{n}_1, \mathrm{n}_2, \ldots >$ n_i is a set of event(s), $n_i =< e_1, e_2, \ldots > e_i$ is an event ($i = 1, 2, \ldots$)
- E $= <$ $\mathrm{edge}_1, \mathrm{edge}_2, \ldots >$
  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

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.

As mentioned earlier, a P-DAG is only meant to represent temporal dependencies between constituent event nodes and is primarily designed to provide additional information about an event pattern occurrence. This information has been used for different purposes, such as visualization [24].

5 Implementation and case study

This section illustrates a testing application implemented to validate our proposed method, which requires interoperability between two separate software components (i.e. a rule-based front-end [25,26] and a CEP). In addition, this section also illustrates a case study that involves financial data analysis using this implementation. The reason why the finance domain is selected for the case study is that financial market data are representative as a type of event data and that finance experts frequently look into patterns in data to gain insights into the implications out of data, or to help make financial decisions. In this case study, a number of specific event pattern types were defined and implemented as required by non-IT finance experts, and one of the finance experts used the application to find occurrences of these event pattern types.

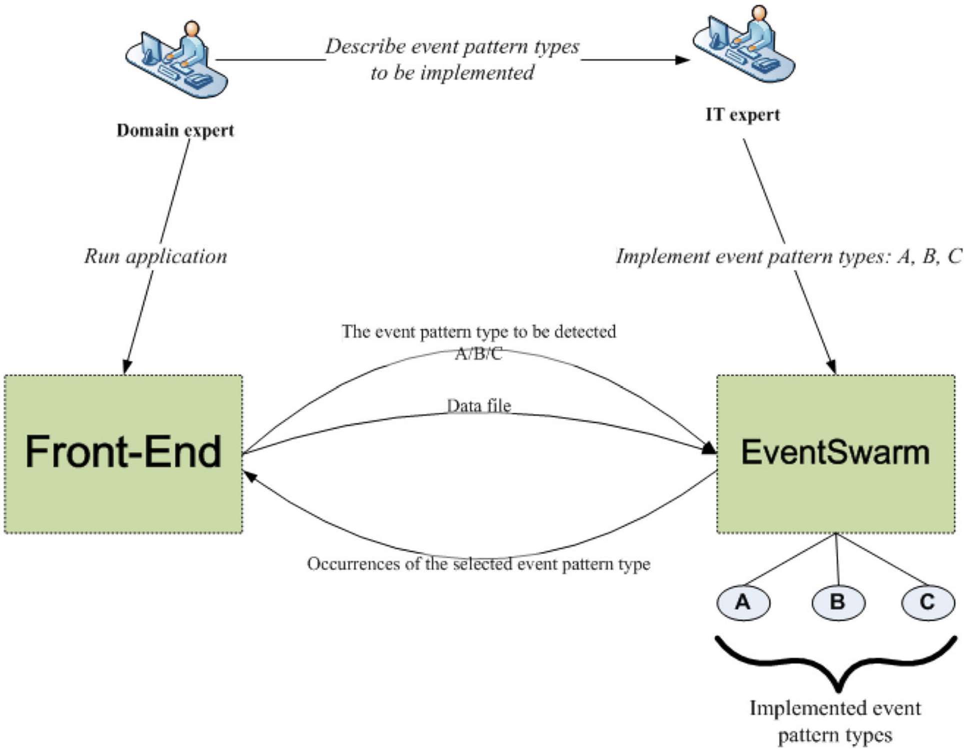

Describe event pattern types to be implemented
Domain expert
IT expert
Run application
Implement event pattern types: A, B, C
The event pattern type to be detected A/B/C
Data file
Front-End
EventSwarm
Occurrences of the selected event pattern type
A
B
C
Implemented event pattern types

Fig. 7 Implementation environment

5.1 Testing application

Figure 7 shows the implementation of an analytics application (AA) developed to support domain experts interested in implementing a decision-support system based on different event pattern types.

The application has two components. A Front-End is used by the domain expert to manage the process of data analysis. The second component (EventSwarm) is used as an event pattern detection service. It is important to note that both the Front-End component and EventSwarm component interact via concepts that are compliant with the meta-model introduced in the previous section. The front-end application provides the capability to manage event processing rules for financial market data 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 following steps were applied in deploying and testing the event pattern types used in the case study:

1. A number of finance experts specify a set of interesting event pattern types. The specifications of the event pattern types are communicated in a natural language in writing or verbally to an IT expert.
2. The IT expert implements the rules for detecting the event pattern types in EventSwarm and makes them available for invocation by the Front-End application. Rules in EventSwarm are implemented as Ruby classes in a Rails application.
3. Using the Front-End, the one of the finance experts selects 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 provides 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 finance expert 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 Sect. 4. Finally, the occurrences are further processed by the Front-End, meanwhile corresponding high-level events are generated (e.g. a “duplicate dividend” event can be generated in accordance with the information provided by the detected “duplicate dividend” pattern occurrence).

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

5.2 EventSwarm

The EventSwarm implements the meta-model concepts from Sect. 4 and is used as a 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 Sect. 5.4 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.

5.3 Event patterns and event processing rules considered

Table 1 describes the event pattern types related to financial market data analysis case study that have been identified as candidates for analysis. Note that all these event pattern types are requested by finance experts who are doing research using financial market data and are interested in addressing data quality issues and conducting data processing using these rules. Also note that each event pattern occurrence can incur the generation of a high-level event, which stores the detailed information of this particular event pattern occurrence.

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

5.3.1 Simple filters

Some rules were implemented using a simple, single-event filter that collected events matching one or more static field values (rules 1, 4, 6 and 7 from Table 1). For example, the "dividend deleted" rule matched events with a type of "Dividend" and a value of "1" in the "Div Delete Marker" field.

5.3.2 Duplicate detectors

Other rules needed to detect duplicates in the data stream (rules 2 and 5 from Table 1). 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 minimize the number of candidates and 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.

5.3.3 Simple event sequence

Two rules (rules 8 and 9 from Table 1) 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).

5.3.4 Conditional event sequence

A number of sequence patterns (rules 10, 11, 12, 13 and 14 from Table 1) 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

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." (Dividend amount) and the same "Div Ex Date" (Dividend 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" (Dividend market-level ID)
6	Status is not "APPD"	A "Dividend" event has a value other than "APPD" (Approved) 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 $\rightarrow$ 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} $\rightarrow$ 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	$E_{6(2)}$ $\rightarrow E_{6(1)}$ $\rightarrow$ 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	$E_{3(2)} \rightarrow E_{3(1)}$ $\rightarrow E_6$ $\rightarrow$ 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 $\rightarrow E_9$ $\rightarrow$ 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	$E_{3(4)} \rightarrow E_{3(3)} \rightarrow E_{3(2)}$ $\rightarrow E_{3(1)}$ $\rightarrow$ 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 $\rightarrow E_3$ $\rightarrow$ 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

"EPS" (finance term) in this table stands for "Earnings Per Share" 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.

5.3.5 Event not present in history

The "Dividend without a valid EOD for the div ex date" pattern (rule 3 from Table 1) 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 whether 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 Sect. 6.2.3 further discussion of negation.

5.4 Wire format

The wire format used to express patterns is a direct reflection of the P-DAG model described in Sect. 4. It uses JSON [13] for simple and efficient cross-platform processing. A sample of a JSON file that saves event pattern occurrences is shown in Fig. 8. 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*.

6 Results

6.1 Evaluation

Experiments were conducted by one of the finance experts who had specified the event pattern types with the assistance of an IT expert. The finance expert used the Front-End to define the following three different data analysis scenarios:

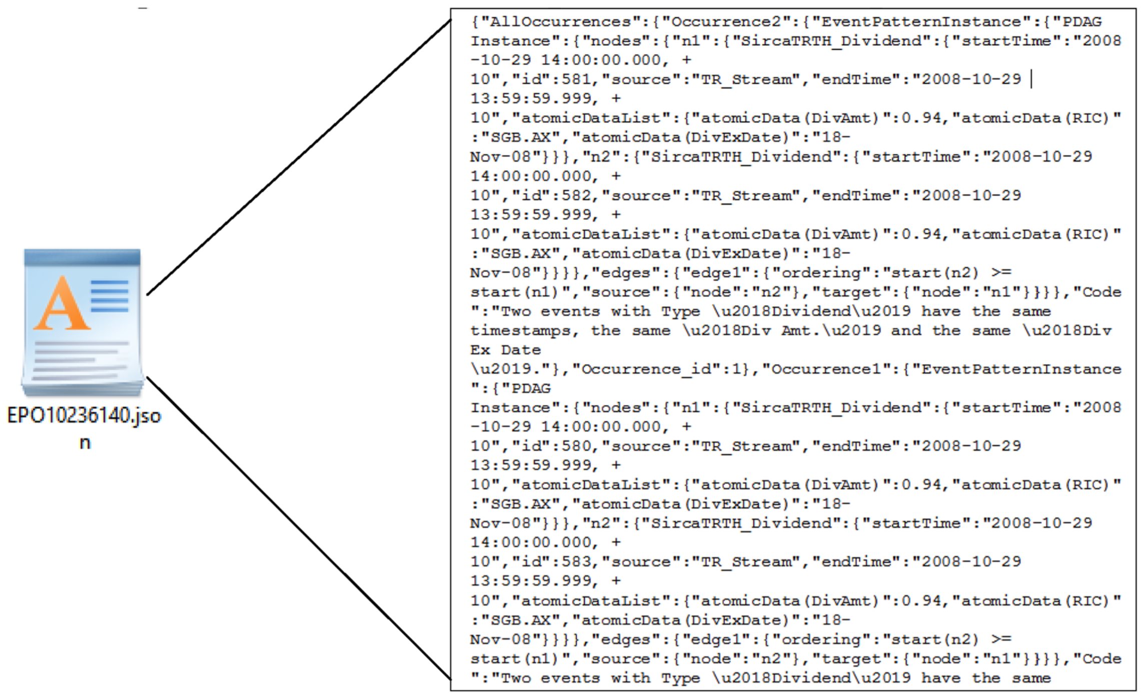
– Detecting occurrences of event pattern type No. 2 (1 event pattern type involved)
– Detecting occurrences of event pattern types No. 1–7 (Handling data quality issues of dividend events; 7 simple event pattern types involved)
– Detecting occurrences of event pattern types No.8, 10–14 (Calculating earnings;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

These analysis processes were also implemented as a local bespoke program for comparison purposes. For all the three scenarios, the analysis processes produced identical results in both cases. This indicates that the interoperability between the Front-End and EventSwarm was successfully achieved due to the fact that the underlying data models in both the Front-End and EventSwarm are built on the same meta-model.

The finance expert and the Front-End developer have also 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, some additional implementation benefits the EventSwarm platform provides have also been reported, 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.

Last but not least, the results produced by the testing application (based on the extended ADAGE architecture) contain detailed information about high-level events, which could not be achieved by the old ADAGE architecture.



```
{"AllOccurrences":{"Occurrence2":{"EventPatternInstance":{"PDAG
Instance":{"nodes":{"n1":{"SircaTRTH_Dividend":{"startTime":"2008
-10-29 14:00:00.000, +
10","id":581,"source":"TR_Stream","endTime":"2008-10-29
13:59:59.999, +
10","atomicDataList":{"atomicData(DivAmt)":0.94,"atomicData(RIC)"
:"SGB.AX","atomicData(DivExDate)":"18-
Nov-08"}}},"n2":{"SircaTRTH_Dividend":{"startTime":"2008-10-29
14:00:00.000, +
10","id":582,"source":"TR_Stream","endTime":"2008-10-29
13:59:59.999, +
10","atomicDataList":{"atomicData(DivAmt)":0.94,"atomicData(RIC)"
:"SGB.AX","atomicData(DivExDate)":"18-
Nov-08"}}}},"edges":{"edge1":{"ordering":"start(n2) >=
start(n1)","source":{"node":"n2"},"target":{"node":"n1"}}}},"Code
":"Two events with Type \u2018Dividend\u2019 have the same
timestamps, the same \u2018Div Amt.\u2019 and the same \u2018Div
Ex Date
\u2019."},"Occurrence_id":1},"Occurrence1":{"EventPatternInstance
":{"PDAG
Instance":{"nodes":{"n1":{"SircaTRTH_Dividend":{"startTime":"2008
-10-29 14:00:00.000, +
10","id":580,"source":"TR_Stream","endTime":"2008-10-29
13:59:59.999, +
10","atomicDataList":{"atomicData(DivAmt)":0.94,"atomicData(RIC)"
:"SGB.AX","atomicData(DivExDate)":"18-
Nov-08"}}},"n2":{"SircaTRTH_Dividend":{"startTime":"2008-10-29
14:00:00.000, +
10","id":583,"source":"TR_Stream","endTime":"2008-10-29
13:59:59.999, +
10","atomicDataList":{"atomicData(DivAmt)":0.94,"atomicData(RIC)"
:"SGB.AX","atomicData(DivExDate)":"18-
Nov-08"}}}},"edges":{"edge1":{"ordering":"start(n2) >=
start(n1)","source":{"node":"n2"},"target":{"node":"n1"}}}},"Code
":"Two events with Type \u2018Dividend\u2019 have the same
```

Fig. 8 Sample JSON file of pattern occurrences

6.2 Limitations

The experiments did, however, highlight complex semantic issues in pattern specification, implementation and representation. Key examples are identified in the following subsections.

6.2.1 *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.

6.2.2 *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).

6.2.3 *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.

6.2.4 *Edge semantics*

The P-DAG model for event pattern occurrences defines edges between events which 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 domain expert 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 domain experts 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 domain expert 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.

7 Related work

Event-based analytics is increasingly gaining prominence in business and social applications as a result of the need to deal with the volume and diversity of data and the need to act instantly in response to data triggers [27]. This is fuelled by the growing audience for data insights, increasing use of cloud computing and commodity infrastructure, and the proliferation of new data sources, such as social media and mobile devices [4]. Event-based analytics requires sophisticated infrastructure and techniques. According to a recent survey of the technology landscape [4], there are different layers in the technologies stack comprising infrastructure platforms, e.g. Tibco [28], IBM [29], Oracle [30], Azure [16], data processing platforms, e.g. Hadoop [31], Storm [32], Kafka [33], data analytics services, e.g. Drools Fusion [34], Esper [35] and advanced tools, e.g. statistical, machine learning, text processing tools. For the types of the users that we are targeting one would need an open approach with an ability to leverage existing technologies such as R [11,12], Web services, REST [11], JSON [13], Open Calais [14], Amazon Kinesis [15], cloud analytics from Azure [16].

There are many approaches that allow multiple technologies and tools to be used together to define complex analytics processes. The most important effort in providing "integrated solutions" that link such heterogeneous tools is in the area of scientific workflow management systems or SWfMSs [36]. Such systems (e.g. Taverna [37], Kepler [38], Galaxy [38], Grid Nexus [39]) provide facilities for the composition of workflows that abstract at least some of the technical details of data analysis and allow domain experts to focus on what workflow components are required and not how the components will be executed. However, current SWfMs are "not abstract enough" [40]. Non-technical users struggle to construct workflows that involve complex control-flow operations such as repeating iterations of tasks or defining parallel tasks. The ADAGE Framework [6] gives a set of architectural guidelines based on service-oriented computing principles that restrict the choice of services depending on their role in the analytics pipeline. This idea has been applied in facilitating the definition of event data analysis processes. However, the underlying event model was based on a simple event model [41] that mirrors the ubiquitous row–column CSV format used by finance experts. This model could not cope with event-based analytics involving patterns of event instances, which we refer to as *event pattern occurrences*. The main contribution of this paper is to address this limitation by proposing a common event pattern meta-model (see Sect. 4) as it will enable the exchange of high-level events and event pattern occurrences between different systems. This meta-model has been formalized as a UML meta-model to facilitate its adoption and usage across the event analytics community.

8 Conclusions and future work

This paper has advocated an open architecture approach to support better interoperability of various analytics tools concerned with event-based analytics. The paper was motivated by needs identified in our previous work to support event-based analysis in financial market, in particular the need to express high-level events, in terms of different type of relationships between events. While this previous work was focused on historical analysis of events, the architecture and tools currently being developed [42] can be also applied to real-time analytics applications. The central part of our interoperability approach is the development of a precise event-based meta-model which was refined through extensive testing [4,42], while leveraging the increasing popularity of the Event Swarm CEP engine, in several vertical domains [8,9,43].

The meta-model presented in this paper includes a minimal set of modelling concepts related to the specification of event types, event pattern types, event pattern occurrences and related concepts. 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. The use of functors to access individual constituents of event instances provides an additional abstraction mechanism for integrating existing event models and event-based repositories.

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 financial case study has also identified a number of semantic difficulties inherent in CEP platforms and EPS languages, and we believe this is an important result in its own right. The issues identified in Sect. 6.2 suggest that a general-purpose EPS language has limited value for domain experts due to the inherent complexity of pattern specification and the difficulty of scaling without explicit bounds. It also supports our decision to focus on a meta-model for capturing pattern matches, which provides interoperability without constraining the pattern specification and matching semantics. With regard to future development of EPS languages for domain experts, our evaluation suggests that domain experts would be better served by limited, domain-specific pattern languages with a tractable matching semantics predetermined (e.g. implied bounds, sequence and negation semantics that reflect natural data constraints, node labelling).

It should be noted that the semantic issues identified in Sect. 6.2 are not addressed in the recent event processing formalism published in [23] nor in RETE-based rule implementation environments like Drools [34]. A broad discussion of EPS languages and platforms, including the associated limitations and complexities, can be found in [44]. The programmer-oriented 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 [42]. These languages and platforms do not, however, address the needs of domain experts in specifying patterns.

In future, we plan to implement a full set of rules listed in Sect. 5.2. By then, we will conduct experiments with more representative researcher candidates, so that our work will become more solid. 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.

Acknowledgments 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.

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison Wesley Professional, Reading (2002)
2. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co., Greenwich (2011)
3. EMF: Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/ (2015)
4. Milosevic, Z., Chen, W., Berry, A., Rabhi, F.A.: Real-Time Analytics. In: Buyya, R., Calheiros, R.N., Dastjerdi, A.V. (eds.) Big Data: Principles and Paradigms. Morgan Kaufmann/Elsevier (2016)
5. Milosevic, Z., Berry, A., Chen, W., Rabhi, F.A.: An event-based model to support distributed real-time analytics: finance case study. In: Enterprise Distributed Object Computing Conference (EDOC), 2015 IEEE 19th International, 21–25 Sept. 2015, pp. 122–127
6. Yao, L., Rabhi, F.A.: Building architectures for data-intensive science using the ADAGE framework. Concurr. Comput. Pract. Exp. **27**(5), 1188–1206 (2014). doi:10.1002/cpe.3280
7. Rabhi, F.A., Yao, L., Guabtni, A.: ADAGE: a framework for supporting user-driven ad-hoc data analysis processes. Computing **94**(6), 489–519 (2012). doi:10.1007/s00607-012-0193-0
8. Deontik: EventSwarm. http://deontik.com/Products/EventSwarm.html (2015)
9. Berry, A., Milosevic, Z.: Real-time analytics for legacy data streams in health: monitoring health data quality. In: Paper presented at the 17th IEEE International Enterprise Distributed Object Computing Conference (EDOC), Vancouver (2013)
10. SoftwareAG: Apama. http://www.softwareag.com/corporate/products/apama_webmethods/analytics/overview/default.asp (2015)
11. Wikipedia: REST. https://en.wikipedia.org/wiki/Representational_state_transfer
12. R: The R Project for Statistical Computing. http://www.r-project.org/ (2015)
13. ECMA: The JSON Data Interchange Standard. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (2015)
14. Calais, O.: http://new.opencalais.com (2015)
15. Kinesis, A.: https://aws.amazon.com/kinesis/ (2015)
16. Azure. https://azure.microsoft.com/en-us/documentation/articles/stream-analytics-introduction/ (2015)
17. ITU-T/ISO: ITU-T X.902 | ISO/IEC 10746-2, Information Technology Open Distributed Processing Reference Model – Foundations. (2010)
18. ISO/IEC: ISO/IEC IS 15414, Information technology - Open distributed processing - Reference model - Enterprise language, 3rd edn. (2015)
19. OMG: Semantics of Business Vocabularies and Rules. http://www.omg.org/spec/SBVR/ (2015)
20. Linington, P., Milosevic, Z., Tanaka, A., Vallesillo, A.: Building enterprise systems with ODP. In: Linington, P.F., Milosevic, Z., Tanaka, A., Vallecillo., A. (eds.) An Introduction to Open Distributed Processing, 1st edn. CRC Press, Chapman Hall (2011)
21. Governatori, G., Milosevic, Z., Sadiq, S.: Compliance checking between business processes and business contracts. In: Paper presented at the EDOC (2006)

22. ISO/IEC-IS-10746-4: Information Technology—Open Distributed Processing—Reference Model: Architectural Semantics (1998)
23. Hallé, S., Varvaressos, S.: A formalization of complex event stream processing. In: Paper presented at the EDOC (2014)
24. Perry, R.T., Kutay, C., Rabhi, F.: Using complex events to represent domain concepts in graphs. In: Kim, J.K. (ed.) Information Science and Applications, pp. 303–311. Springer, Berlin Heidelberg (2015)
25. Chen, W., Rabhi, F.: An RDR-based approach for event data analysis. In: Paper presented at the Third Australasian Symposium on Service Research and Innovation (ASSRI'13), Sydney, Australia
26. Chen, W., Rabhi, F.A.: Enabling user-driven rule management in event data analysis. Inf. Syst. Front. **18**(3), 511–528 (2016). doi:10.1007/s10796-016-9633-2
27. Luckham, D.: Event Processing for Business: Organizing the Real-Time Enterprise. Wiley, Hoboken (2012)
28. TIBCO: TIBCO BusinessEvents. http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/default.jsp (2015)
29. IBM: InfoSphere Streams. http://www-03.ibm.com/software/products/en/infosphere-streams/ (2015)
30. Oracle: CQL. http://docs.oracle.com/cd/E17904_01/apirefs.1111/e12048/intro.htm (2015)
31. Apache: Hadoop. http://hadoop.apache.org/ (2015)
32. Apache: Storm. http://storm.apache.org/ (2015)
33. Kafka. https://kafka.apache.org/ (2015)
34. RedHat: Drools Fusion. http://drools.jboss.org/drools-fusion.html (2015)
35. Esper. http://esper.codehaus.org/ (2015)
36. McPhillips, T., Bowers, S., Zinn, D., Ludascher, B.: Scientific workflow design for mere mortals. Future Gener. Comput. Syst. **25**, 541–551 (2009)
37. Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A., Li, P.: Taverna: a tool for the composition and enactment of bioinformatics workflows. Bioinformatics **20**(17), 3045–3054 (2004). doi:10.1093/bioinformatics/bth361
38. Kepler. https://kepler-project.org (2015)
39. Nexus, G.: http://www.gridnexus.com/ (2015)
40. Withers, D., Kawas, E., McCarthy, L., Vandervalk, B., Wilkinson, M.: Semantically-guided workflow construction in Taverna: the SADI and BioMoby plug-ins. In: 4th international conference on Leveraging applications of formal methods, verification, and validation, pp. 301–312 (2010)
41. Rabhi, F.A., G, A., Yao, L.: A data model for processing financial market and news data. Int. J. Electron. Finance **3**(4), 387–403 (2009)
42. Milosevic, Z., Chen, W., Berry, A., Rabhi, F.A.: An event-based model to support distributed real-time analytics: finance case study. In: Paper presented at the EDOC (2015)
43. ASX: guidance-note-8-clean-copy. www.asx.com.au/documents/about/guidance-note-8-clean-copy.pdf (2015)
44. Cugola, G., Margara, A.: Processing flows of information: from data stream to complex event processing. ACM Comput. Surv. **44**(3), 1–62 (2012). doi:10.1145/2187671.2187677