Rules for Making Sense of Events: Design Issues for High-Level Event Query and Reasoning Languages (Position Paper)

François Bry and **Michael Eckert**
Insitute for Informatics, University of Munich
http://www.pms.ifi.lmu.de

Abstract

Events play an essential role in business processes and some forms of business rules. Often they require detection of complex events, that is, events or situations that cannot be inferred from looking only at single events but that manifest themselves in certain combinations of several events. This entails a natural need for high-level query and reasoning languages for complex events. This position paper explores issues related to the design of such languages.

Introduction

Events play an essential role in business processes and some forms of business rules. A business process is initiated by some event from the outside world. For example, a customer sending a purchase order event will initiate a corresponding business process at the seller. Progress and control flow in a running process also depend on events. For example, a purchase order process will have to wait for an event generated by the warehouse system confirming the product's availability and proceed according to the availability information.

Business processes not only consume events in order to drive their own execution, they also produce and publish events to the outside world. First, such events can initiate further processes, both as sub-processes or independent processes. For example, order processing might generate an event requesting to check product availability, which is then performed as a sub-process. Similarly, order processing might generate an event that the purchase order has been accepted, which initiates shipping as an independent process. Second, events can also be used by other systems outside the context of a business process. For example, acceptance of a purchase order might cause the warehouse system to adjust the number of items on stock accordingly. Third, events can be used for monitoring purposes. For example, a business activity monitoring (BAM) application might count all purchase order events as well as all events where an order cannot be satisfied. Their ratio can serve of a key performance indicator and warnings can be generated automatically when a high ratio of orders cannot be fulfilled.

Also some forms of business rules explicitly refer to events. For example, a rule might state that a purchase order will be immediately rejected if the customer has an outstanding debt of more than $10 000. Often these are rules that guide, constrain, or implement parts of business processes, but they can also be rules that govern the behavior of services and applications. Again business rules can be both consumers and producers of events, i.e., both react to and publish events.

This position paper argues that high-level query and reasoning languages for events considerably help taking full advantage of events and the information they provide. They are particularly needed to detect and react to complex events, that is, situations that cannot be inferred from looking only at single events but that manifest themselves in certain combinations of several events over time. We discuss in this paper design issues for these languages, presenting our positions as five "rules":

- Business processes and rules have a natural need for expressive high-level complex event query and reasoning languages. Traditional query languages and knowledge representation formalisms are not suitable for events.
- Deductive rules about events are an essential abstraction and reasoning mechanism. Resorting to reactive rules for implementing such deductive rules is undesirable.
- An event query language must support at least the four complementary dimensions of data extraction, event composition, temporal relationships, and event accumulation.
- Expressiveness and ease of use of an event query language can only be achieved through a separated treatment of the four dimensions in syntax and semantics.
- Querying events is a young field that has still many emerging research issues. Among current approaches, there is no silver bullet, yet.

The issues and positions presented here have influenced and are influenced by our current work on the high-level event query language XChange$^{\mathrm{EQ}}$ (Bry & Eckert 2006; 2007a; 2007b; 2007c) as well as by earlier, ongoing work on the reactive Web language XChange (Bry, Eckert, & Pătrânjan 2006a) and on the Web query language Xcerpt (Schaffert & Bry 2004).

Rule 1: Need for High-Level Event Languages

Business processes and rules have a natural need for expressive high-level complex event query and reasoning languages. Traditional query languages and knowledge representation formalisms are not suitable for events.

In many scenarios, it is not sufficient to query and react to only single, "atomic" events. Instead, events have to be considered with their relationship to other events in streams of events. Such events (or situations) that do not consist of one single atomic event but have to be inferred from some pattern of several events over time are called *complex events* (or composite events). Such situations are especially common in business processes and rules, because their context requires integration of applications (Hohpe & Woolf 2003) and they have little control over which (types of) events are generated by these applications.

As an example where complex event detection is needed, consider a scenario where whenever an order has been completed some reaction is necessary (e.g., initiate billing process). Involved systems might not generate an "order completed" event; however it might be inferred from a combination of other events: an order has been taken (with number n), followed by a shipping (of order n with tracking number t), followed by a delivery (with tracking number t). Note that for it would not be sound to react only to the last event ("delivery") in the sequence. First, the "delivery" event here contains only a tracking number, but for the "order completed" event we would be interested in the order number. Second, "delivery" events might also be generated for items that are not actually orders (e.g., gifts, replacements).

While it is of course possible to implement detection of such complex events using general purpose programming languages, it is much more desirable to use a high-level event query language. This allows to specify the event query on a high abstraction level that focuses on the query's logic rather than programming on a low level an actual detection algorithm. Even when they don't aim for high performance, detection algorithms are usually complicated since they involve state maintenance (events and partial query answers) and require a form of manual "garbage collection" (removing events and partial answers that become irrelevant). Also, high-level languages make the resulting code much easier to maintain, which is especially important since business processes and rules are expected to change frequently. Finally, high-level event query languages give rise to query compilers that do automatic performance optimization, thus taking this burden off the programmer's shoulders. This last point is especially important since many query optimization techniques (such as multi-query optimization which exploits similarities between several queries) conflict with maintainability when they are programmed manually.

Querying and reasoning with events has much in common with traditional (database) query languages and knowledge representation formalisms.[1] However, there are important differences which make those unsuitable for dealing with events and entail a need for *tailored* event query and reasoning languages:

- Events are received over time in a stream-like manner, while in a database all facts are available at once and usually stored on disk.
- Event streams are unbounded into the future, potentially infinite, whereas databases are finite. This has especially consequences for non-monotonic query features such as negation or aggregation.
- Relationships between events such as temporal order or causality play an important role for querying events. In databases, relationships between facts are usually part of the data (e.g., references with foreign keys).
- Timing of answers has to be considered when querying events: event queries are evaluated continously against the event stream and generate answers at different times. These answers may trigger actions such as updates to a database. Typically actions are sensitive to ordering; hence it is important *when* an answer is detected.
- Query evaluation and optimization for event streams require different methods than for databases. In event streams a large number of (standing) queries are evaluated against small pieces of incoming data (events). Evaluation is thus usually data-driven, rather than query-driven. Many optimizations rely on exploiting similarities between queries rather than clustering and indexing data.

Rule 2: Support for Deductive Rules

Deductive rules about events are an essential abstraction and reasoning mechanism. Resorting to reactive rules for implementing such deductive rules is undesirable.

Deductive rules allow to define new, "virtual" events from the existing ones (i.e., those that are received in the incoming event stream), much in the same fashion as one uses views (or rules) in databases to define new, derived data from existing base data. For example the "order completed" event from above could be defined by a deductive rule, and then in turn be used in other complex event queries.

Only very few event languages support such purely deductive rules, even though support is highly desirable for a number of reasons: Rules serve as an abstraction mechanism, making query programs more readable. They allow to define higher-level application events from lower-level (e.g., database or network) events. Different rules can provide different perspectives (e.g., of end-user, system administrator, corporate management) on the same (event-driven) system. Rules allow to mediate between different schemas for event data. Additionally, rules can be beneficial when reasoning about causal relationships of events (Luckham 2002).

Event-based systems usually provide reactive rules, typically Event-Condition-Action (ECA) rules or production rules, to specify reactions to the occurrences of certain events (Berstel *et al.* 2007). While deductive rules can be, and often are, implemented using reactive rules, we argue that deductive (event) rules are inherently different from reactive rules because they aim at expressing "virtual events," not actions. Accordingly and importantly, deductive rules

[1] We use in the following the term "database" as an umbrella term for all kinds of traditional knowledge representation systems.

are free of side-effects. Implementing deductive rules using reactive rules blurs this distinction. This has negative consequences for development and maintainability, and restricts optimization: techniques that are applicable for deductive rules, such as backward chaining or program rewriting, are not generally applicable to reactive rules.

Rule 3: Four Dimensions of Event Queries

An event query language must support at least the four complementary dimensions of data extraction, event composition, temporal relationships, and event accumulation.

A sufficiently expressive event query language should cover (at least) the following four complementary dimensions. How well a language covers each of these dimensions gives a practical measure for its expressiveness.

Data extraction: Events contain data that is relevant to decide whether and how to react to them. The data of events must be extracted and provided (typically as bindings for variables) to test conditions (e.g., arithmetic expressions) inside the query, construct new events (e.g., by deductive rules), or trigger reactions (e.g., updates). Often, events are transmitted as messages in XML formats; examples for such message formats include SOAP (Gudgin *et al.* 2007), Common Base Event (CBE) (IBM 2004), and the Facility Control Markup Language (FCML) (Bry *et al.* 2008). The structure of data in such XML messages can be quite complex, which gives a strong motivation to embed an XML query language into an event query language.

Event composition: To support complex events, i.e., events that consist out of several events, event queries must support composition constructs such as the conjunction and disjunction of events (more precisely, of event queries). Composition must be sensitive to event data, which is often used to correlate and filter events (e.g., consider only "order" and "shipping" events with the *same* order number for composition). Since reactions to events are usually sensitive to timing and order, an important question for complex events is *when* they are detected. In a well-designed language, it should be possible to recognize when reactions to a given event query are triggered without difficulty.

Temporal relationships: Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "shipping happens more than 24 hours after the order." Qualitative relationships concern only the temporal order of events (e.g., "shipping after order"). Quantitative (or metric) relationships concern the actual time elapsed between events (e.g., "shipping and order more than 24 hours apart").

Event accumulation: Event queries must be able to accumulate events to support non-monotonic features such as negation of events (understood as their absence) or aggregation of data from multiple events over time. The reason for this is that the event stream is (in contrast to extensional data in a database) unbounded (or "infinite"); one therefore has to define a scope (e.g., a time interval) over which events are accumulated when aggregating data or querying the absence of events. Event accumulation is particularly required in many queries from business activity monitoring, e.g., to watch for situations where "a customer's order has *not* been shipped within 2 days" (negation) or where "the *sum* of all orders on a business day is below \$1M" (aggregation).

Rule 4: Separation of Concerns

Expressiveness and ease of use of an event query language can only be achieved through a separated treatment of the four dimensions in syntax and semantics.

Many event query languages are based on composition operators constructing complex event queries from atomic event queries (in this context often also called event types) and composition operators (Gehani, Jagadish, & Shmueli 1992; Gatziu & Dittrich 1993; Chakravarthy *et al.* 1994; Zimmer & Unland 1999; Adi & Etzion 2004; Bry, Eckert, & Pătrânjan 2006a). Such languages are also often called event algebras, since they consist of a set (the event types) and operations on it (composition operators). Composition operators can be understood as functions whose input and output are streams of events. For example, binary composition operators such as conjunction (often written $A \wedge B$ or $A \triangle B$) or sequence (often written $A; B$) take as input two event streams, the results of the queries that are their arguments, and produce as output another event stream containing the complex events. There is a wide spectrum of further operators supported by different languages and no agreement on a standard set of operators.

Consider an event query asking for events A, B, C, and D to happen, with the constraints that A happens before B, A before C, and C before D. One might be tempted to write this query in an event algebra as $(A; B) \triangle (A; C) \triangle (C; D)$. This however does not yield the intended result since different instances of A and C can be used to match this expression (while the query requires using the same instances). A correct way to express the query would be $A; (B \triangle (C; D))$. Consider now just adding an additional constraint that B happens before D. The new expression bears only little resemblance to the old: $A; (B \triangle C); D$. In fact, even though we have *added* a constraint in our specification, the query has the *same* number of operators in the event algebra! This is quite unnatural and might easily cause programming errors.

To deal with queries that involve also metric temporal constraints such as "events A and B happen within 1 hour," many event algebras extended the basic operators with temporal constraints. For example, $(A; B)_{1h}$ would denote that B happens within 1 hour after A. However, a query asking for A to happen, then B to happen within 1 hour of that A, and then C to happen within 1 hour of that B, is not expressible with such an operator.

Note that in this argument, we take the interval-based interpretation of the sequence operator as in (Zhu & Sethi 2001; Galton & Augusto 2002; Adaikkalavan & Chakravarthy 2005; Bry, Eckert, & Pătrânjan 2006b), where the occurrence time of $(A; B)$ here the time interval covering both A and B. Would we understand as occurrence time of $(A; B)$ only the time point of B, as in some earlier works (Gehani, Jagadish, & Shmueli 1992; Gatziu & Dittrich 1993; Chakravarthy *et al.* 1994), we actually could express the query as $((A; B)_{1h}; C)_{1h}$. However then the modified query where C is supposed to happen after B and within 2 hours of A (as opposed to within 1 hour of B) cannot be expressed.

Composition operators mix the event querying dimensions (e.g., event composition and temporal relationships in the case of the sequence operator). This leads to the exemplified difficulties in correctly expressing and understanding some event queries and also to a certain lack in expressiveness. Furthermore there is some confusion in the interpretation of operators: even for the seemingly simple sequence operator, at least four different interpretations are conceivable (Zhu & Sethi 2001).

We therefore argue that an expressive high-level event query language should treat the querying dimensions separated from each other in syntax and semantics. We have applied this principle in the language design of XChange$^{\mathrm{EQ}}$ with good results in terms of expressiveness and ease of use. All query examples can be expressed in XChange$^{\mathrm{EQ}}$ and in a manner quite similar to the natural language descriptions given above.

Rule 5: No Silver Bullet, yet

Querying events is a young field that has still many emerging research issues. Among current approaches, there is no silver bullet, yet.

While a lot of progress in research and industry has been made on querying events, it is still a young field. We detail now some of emerging issues that are query features which are difficult to handle with current approaches.

Event instance selection and event consumption (Zimmer & Unland 1999) have been introduced early (Chakravarthy *et al.* 1994), but are still little understood. *Event instance selection* allows to restrict the instances of one event type that are considered for a composition with other events so that, e.g., only the first, n-th, or last instance is considered. *Event consumption* allows to invalidate certain events so that, once they have been used in one answer to a query, they cannot be used in other, later answers. Note that consumption has direct impact on semantics and should be distinguished from event deletion as a "garbage collection" mechanism in evaluation algorithms. In general, both instance selection and event consumption should be sensitive to event data (e.g., for each sensor identifier select the last event), an aspect that has usually been ignored so far. Further, the necessary notions such as first, last, or next event only have an intuitive meaning when there is a linear order on the occurrence times of events; however usually there is no clear linear order when events happen over time intervals not points. Also instance selection and event consumption can be argued to have a rather imperative, non-declarative flavor, making queries hard to use and hard to optimize. We also refer to (Bry & Eckert 2007c) for deeper explanations.

Another issue is whether certain situations should be modeled and queried as (complex) events or rather as *states*. An inventory application might generate events when items come in or go out. Expressing a situation where less than 10 items are in store as a complex event is hard with current approaches, while it is relatively easy with a state-focused approach such as production rules. The state is modeled as a counter, which is increased and decreased by events, and a production rule will fire when the counter sinks below 10. On the other hand, current complex event approaches are much better suited when situations involve event pattern matching or negation and aggregation over time.

It is often necessary to combine *non-event data* with event data. For example, inventory tracking events might contain item identifiers and queries require a database lookup to find out the product group of a given item. When the non-event data changes during the detection of the (complex) event, it then becomes relevant when the non-event data is accessed. This is an emerging issue for both language design and evaluation algorithms and closely related to the issue states from the preceeding paragraph.

An important issue is whether or how *knowledge representation mechanisms* can be applied to events and event types. We have already argued in rule 2 that deductive rules are indispensable, and they are already supported by some query languages (Bry & Eckert 2007a). However other approaches can be interesting as well. For example it can be useful to have an ontology that formalizes type hierarchies of events (e.g., inheritance like "a delivery event is a package tracking event") and relationships such as causality, cooccurrence, or temporal order between events. Whether current ontology languages are suited for this task, however, is still an open question. Representation of knowledge about events is also discussed in (Adi, Botzer, & Etzion 2000).

The ability to query causal relationships between events is interesting especially for diagnostic queries (e.g., to tell apart abortions of business processes caused by user request or by network failure). One can distinguish horizontal and vertical causality (Luckham 2002). Vertical causality is between events on the same abstraction level, e.g., a shipping event is the cause for a delivery event, and entails that the causing events happens before the caused event. It is often modeled as "key" in event data (e.g., the tracking number in the shipping and delivery events), though other approaches (e.g., an external table or computable values) are conceivable, too. Horizontal causality is between events on different abstraction levels, e.g., a delivery events is (together with other events) a cause an order completed event. Horizontal causality is often modeled by means of deductive rules as illustrated previously. With the notable exception of the Rapide EPL (Luckham 2002), current event query languages do not offer an explicit construct for querying causal relations between events and it has to be done implicitly (e.g., by correlating event data on a key). One can argue that a construct for querying causal relationship can be integrated into an event query language in the same fashion as the already supported temporal relationships (Bry & Eckert 2006). However the remaining primary problem is the modeling of causality (esp. vertical causality), which is still little understood. Furthermore, causality relationships between events are sometimes simply unknown or unobservable.

To sum up, there are still many emerging issues related to querying events. At least so far, there is no approach that satisfies all needs. Since there is no silver bullet, the choice of a particular event query language to solve a concrete problem is an important one and anything but easy.

Conclusion

There is an obvious need for expressive high-level event query and reasoning languages in the context of business processes and rules, but also other areas such as sensor networks, service level agreement (SLA) monitoring, and supervisory control and data acquisition (SCADA). In this position paper, we have presented five "rules" for the design of such languages. Purposefully, some of these rules might be controversial. Language design as well as algorithmic issues still need research and discussions.

Acknowledgments

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (`http://rewerse.net`).

References

Adaikkalavan, R., and Chakravarthy, S. 2005. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Eng.* In press.

Adi, A., and Etzion, O. 2004. Amit — the situation manager. *Int. J. on Very Large Data Bases* 13(2).

Adi, A.; Botzer, D.; and Etzion, O. 2000. Semantic event model and its implication on situation detection. In *Proc. Europ. Conf. on Information Systems*.

Berstel, B.; Bonnard, P.; Bry, F.; Eckert, M.; and Pătrânjan, P.-L. 2007. Reactive rules on the web. In *Reasoning Web, Int. Summer School*.

Bry, F., and Eckert, M. 2006. A high-level query language for events. In *Proc. Int. Workshop on Event-driven Architecture, Processing and Systems*.

Bry, F., and Eckert, M. 2007a. Rule-Based Composite Event Queries: The Language XChange^{EQ} and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*.

Bry, F., and Eckert, M. 2007b. Temporal order optimizations of incremental joins for composite event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*.

Bry, F., and Eckert, M. 2007c. Towards formal foundations of event queries and rules. In *Proc. Int. Workshop on Event-Driven Architecture, Processing and Systems*.

Bry, F.; Lorenz, B.; Ohlbach, H. J.; Roeder, M.; and Weinberger, M. 2008. The Facility Control Markup Language FCML. In *Proc. Int. Conf. on the Digital Society*.

Bry, F.; Eckert, M.; and Pătrânjan, P.-L. 2006a. Querying composite events for reactivity on the Web. In *Proc. Int. Workshop on XML Research and Applications*.

Bry, F.; Eckert, M.; and Pătrânjan, P.-L. 2006b. Reactivity on the Web: Paradigms and applications of the language XChange. *J. of Web Engineering* 5(1).

Chakravarthy, S.; Krishnaprasad, V.; Anwar, E.; and Kim, S.-K. 1994. Composite events for active databases: Semantics, contexts and detection. In *Proc. Int. Conf. on Very Large Data Bases*.

Galton, A., and Augusto, J. C. 2002. Two approaches to event definition. In *Proc. Int. Conf. on Database and Expert Systems Applications*.

Gatziu, S., and Dittrich, K. R. 1993. Events in an active object-oriented database system. In *Proc. Int. Workshop on Rules in Database Systems*.

Gehani, N. H.; Jagadish, H. V.; and Shmueli, O. 1992. Composite event specification in active databases: Model & implementation. In *Proc. Int. Conf. on Very Large Data Bases*.

Gudgin, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.-J.; Nielsen, H. F.; Karmarkar, A.; and Lafon, Y. 2007. SOAP version 1.2 part 1: Messaging framework. W3C recommendation.

Hohpe, G., and Woolf, B. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

IBM. 2004. Common Base Event. `www.ibm.com/developerworks/webservices/library/ws-cbe`.

Luckham, D. C. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley.

Schaffert, S., and Bry, F. 2004. Querying the Web reconsidered: A practical introduction to Xcerpt. In *Proc. Extreme Markup Languages*.

Zhu, D., and Sethi, A. S. 2001. SEL, a new event pattern specification language for event correlation. In *Proc. Int. Conf. on Computer Communications and Networks*.

Zimmer, D., and Unland, R. 1999. On the semantics of complex events in active database management systems. In *Proc. Int. Conf. on Data Engineering*.