Towards Formal Foundations of Event Queries and Rules

François Bry
Institute for Informatics
University of Munich
http://www.pms.ifi.lmu.de/
francois.bry@ifi.lmu.de

Michael Eckert
Institute for Informatics
University of Munich
http://www.pms.ifi.lmu.de/
michael.eckert@pms.ifi.lmu.de

ABSTRACT

The field of complex event processing still lacks formal foundations. In particular, event queries require both declarative and operational semantics. We put forward for discussion a proposal towards formal foundations of event queries that aims at making well-known results from database queries applicable to event queries. Declarative semantics of event queries and rules are given as a model theory with accompanying fixpoint theory. Operational semantics are then obtained by translating the considered queries into relational algebra expressions. We show the suitability of relational algebra for the kind of incremental evaluation usually required for event queries. With the aim of generating further discussion of formal foundations in the research community, we reflect openly upon both strengths and weaknesses of the presented approach.

1. INTRODUCTION

The emergence of event-driven architectures, automation of business processes, drastic cost-reductions in sensor technology, and a growing need to monitor IT systems due to legal, contractual, or operational considerations lead to an increasing generation of events. The tasks involved in making sense of all these events are commonly summarized under the term "Complex Event Processing" (CEP).

In particular, CEP involves monitoring event streams and clouds for complex (or composite) events, that is, events or situations that cannot be inferred from looking at single events but manifest themselves in certain combinations of several events. Combinations that are of interest are usually expressed in an (composite) event query language. While a considerable number of such event query languages have been proposed both from research and industry, there is still a lack of formal foundations. This lack of formal foundations has also been a discussion topic on a recent Dagstuhl seminar on event processing [37]. Most notably, both declarative and operational semantics for event queries are desirable.

VLDB '07, September 23-28, 2007, Vienna, Austria.

1.1 Expressive Event Query Languages

As we argue in [8, 7, 9], expressive event query languages should cover at least the four dimensions of data extraction, event composition, temporal and causal relationships, and event accumulation for non-monotonic features such as negation (absence of events) and aggregation. Accordingly, formal semantics for event query languages must be able to fully accommodate all four dimensions.

Further, a language should allow the use of (deductive) rules to define new events from the answers to a composite event query [26, 9], similar to the way views are used in database systems. Accordingly, the formalism used to give semantics to an event query language should support deductive rules. We now detail these requirements further.

Data extraction: Events contain data that is relevant for applications to decide whether and how to react to them. For events that are received as XML messages, the structure of the data can be quite complex and might vary (semi-structured). The data of events must be extracted and provided (typically as bindings for variables) to test conditions (e.g., arithmetic expressions) inside the query, combine event data with persistent, non-event data (e.g., from a database), construct new events (e.g., by deductive rules, see below), or trigger reactions (e.g., database updates).

Event composition: To support composite events, i.e., events that consist 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 stock transactions from the *same* customer for composition). Since reactions to events are usually sensitive to timing and order, an important question for composite 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 (and causal) relationships: Time plays an important role in event-driven applications. Event queries must be able to express temporal conditions such as "events A and B happen within 1 hour, and A happens before B." For some applications, it is also interesting to look at causal relationships, e.g., to express queries such as "events A and B happen, and A has caused B." An in-depth discussion of causality, especially defining and maintaining causal relationships, would go beyond the scope of this article, and we concentrate on temporal relationships here.

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. Application examples where event accumulation is required are manifold. A business activity monitoring application might watch out for situations where "a customer's order has *not* been fulfilled within 2 days" (negation). A stock market application might require notification if "the *average* of the reported stock prices over the last hour raises by 5%" (aggregation).

Rules: (Deductive) rules allow to define new, "virtual" events from the existing one (i.e., those that are received in the incoming event stream), much in the same fashion as one uses view (or rules) in databases to define new, derived data from existing base data. 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 [26].

In addition to deductive rules, event-based systems usually also require reactive rules, typically Event-Condition-Action (ECA) rules, to specify reactions to the occurrences of certain events. 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 additional actions. Accordingly and importantly, deductive rules are free of side-effects. Implementing deductive rules using reactive rules blurs this distinction and often strongly restrict optimization: techniques that are applicable for deductive rules, such as backward chaining or program rewriting, are not generally applicable to reactive rules. We refer to [7, 9, 6] for further discussion of reactive rules and their differences to deductive (event) rules.

1.2 Contributions

We put forward for discussion a proposal towards formal foundations of event queries that aims at making well-known results from database queries applicable to event queries.

It is beneficial to put event querying on the wheels of query answering for several reasons. Query answering is well-developed regarding both theoretical foundations and practical systems. As this article tries to emphasize, many (foundational and practical) results apply, or apply with some changes, to composite events, too. This includes formal semantics, chaining of rules, program and query transformations, join algorithms, and index structures. It is desirable to recognize where new concepts and methods are needed – and where they are not needed. Finally, query answering is undergoing a renaissance with Web and Semantic Web research and applications raising novel issues such as incomplete queries (e.g., against HTML and XML data) and queries that require ontology reasoning (e.g., against RDF data with RDFS and OWL ontologies) [12]. These issues can be expected to become very relevant for event queries, too, in the near future.

We show that declarative semantics of event queries and rules can be given as a (Tarski-style) model theory with accompanying fixpoint theory (Section 2). This approach accounts well for (1) data in events and (2) deductive rules defining new events from existing ones, two aspects often neglected in previous work of semantics of event query languages. To make the discussion concrete, we use a datalog-like rule language for querying events. This language is an abstraction of the high-level event query language XChange$^{\mathrm{EQ}}$ [7, 9] but preserves the essential issues. The main difference is that XChange$^{\mathrm{EQ}}$ supports querying events that are received as XML messages (e.g., SOAP [21] or CBE [15]), while the simplified rule language considered here deals only with events that are represented as simple relational facts. Although queries against XML and other Web data are highly relevant to complex event processing, we abstract this aspect out in this article for the sake of simplicity.

We then provide operational semantics (Section 3) by translating event queries into relational algebra (Section 3.1) and showing that these can be evaluated in the incremental, data-driven (or event-driven) manner that is generally required (Section 3.2). Relational algebra as a foundation for operational semantics has a number of advantages: (1) It is well-understood, in particular with regard to equivalences of expressions that give rise to query rewriting to obtain more efficient query plans and multi-query optimization by sharing of subexpressions. (2) Efficient evaluation has been studied for a long time and we can build upon many results from databases, including works on main memory and distributed databases. (3) It can incorporate non-event data (e.g., from a database) easily.

We also discuss optimization techniques in the form of query rewriting as well as using specialized join operators that make use of temporal conditions as they are common in event queries (Section 4).

Finally, we reflect openly upon both strength and weaknesses of the presented approach (Section 5) and also compare it to existing proposals. We do this with the aim of generating further discussion of formal foundations for event queries in the research community and making preliminary steps towards a research agenda on formal foundations of event queries.

We would like to emphasize that the datalog-like rule language used in this article has the primary purpose is to make the discussion concrete. It has been chosen because its syntax is close to traditional rule languages (and we emphasize the need for rules here) as well as because of its closeness to XChange$^{\mathrm{EQ}}$. The approaches presented throughout the article are however quite general and *not* language specific. In particular, an efficient, algebra-based query evaluation is not tied to queries being expressed in any given language.

2. DECLARATIVE SEMANTICS

We now introduce a rule language for composite event queries together with its declarative semantics.

2.1 Basic Concepts

As **time model**, we assume a linearly ordered set of time points $(\mathbb{T}, <)$. Extensions to other time models, in particular where there is only a partial order, are not difficult. We show time points as integers in this article for simplicity; of course the approach can deal with more realistic time models of human calendars as well.

An **event** e^t is a simple relational fact e together with an occurrence time t. Following [17, 2], events occur over *time intervals* rather than just at time points. It turns out that time intervals are also very appropriate for (composite) events that have been derived by rules. The time intervals over which events happen are closed and convex, i.e., have the form $t = [b, e] = \{p \mid b \leq p \leq e\}$ (where $b \in \mathbb{T}$ and $e \in \mathbb{T}$). The set of all events is denoted $Events$.

For convenience we define: $begin([b, e]) = b$, $end([b, e]) = e$, $[b_1, e_1] \sqcup [b_2, e_2] = [min\{b_1, b_2\}, max\{e_1, e_2\}]$, and $[b_1, e_1] \sqsubseteq [b_2, e_2]$ iff $b_2 \leq b_1$ and $e_1 \leq e_2$.

Throughout this paper, we will use a simple e-shopping application that provides event such as orders, shipping, or delivery of items for illustration. For example the event $\text{order}(47, \text{"muffins"}, 11)^{[3,3]}$ indicates that an order with number 47 for 11 muffins with has been made at time 3. Note that we allow time intervals that degenerate to time points, when there is no duration associated with an event as in the case of an order.

A further example of an interesting event would be $\text{comp}(42, \text{"muffins"})^{[3,7]}$, which indicates that the order with number 42 for muffins has been completed over time interval $[3, 7]$. Often an event such as "order completed" is not provided by the involved systems (here: the e-shopping application) as an atomic event in the incoming event stream or cloud. Rather it has to be derived from existing events such as (1) a customer placing an "order" with number n for a given quantity q of a product p followed by (2) this order (again with order number n) being "shipped" as a package with tracking number t followed by (3) the "delivery" of the package (again with tracking number t).

2.2 Querying Event with Rules

Deductive rules are an appropriate way of capturing such knowledge about inferred events. They resemble the notion of views in database systems or rules in expert and other knowledge-based systems. This leads to a first example of our rule language, which formalizes the above knowledge about "order completed" events:

$$\text{(1)} \quad \begin{array}{lcl} \text{comp}(id, p) & \leftarrow & o : \text{order}(n, p, q), \\ & & s : \text{shipped}(n, t), \\ & & d : \text{delivered(t)} \\ & & o \ \mathsf{before}\ s, s\ \mathsf{before}\ d, \end{array}$$

The head (left side) of the rule defines new events "comp." The occurrence time for each new "comp" event is the time interval covering all events that contributed to an answer of the composite event query in the body (right side).

The query illustrates the first three querying dimensions of Section 1.1. The atomic event queries $\text{order}(n, p, q)$, $\text{shipped}(n, t)$, and delivered(t) extract data from events in the form of bindings for the variables n, p, q, and t. (dimension "data extraction").

The query combines events of types "order," "shipped," and "delivered," respectively, in a conjunction,[1] meaning that all three events must happen (dimension "event combination"). The combination is sensitive to data: we only consider those events that are related by order number n and tracking number t.

Finally, the query also has two temporal conditions: the "order" event must happen before the "shipped," and the "shipped" before the "delivered" event. Note that the atomic event queries have been prefixed with event identifiers o, s, d that are then used to refer to their events in temporal conditions. The relation before is one of Allen's temporal interval relations [4]. In addition to such qualitative conditions, one can also use metric conditions, e.g., $\{o, s, d\}$ within $48h$ to specify that all events must happen with 48 hours.

Non-monotonic features such as negation, i.e, detecting the absence of events, and aggregation of data from multiple events over time are important requirements for event query languages.

The crux of such non-monotonic features is that they cannot be simply applied to unbounded event streams, i.e., event streams that extend indefinitely into the future. We therefore need a way to restrict the event stream to a finite temporal extent (i.e., to a finite time interval) and apply negation and aggregation only to the events accumulated in this window (querying dimension "event accumulation").

It should be possible to determine the accumulation window dynamically depending on other events. Typical cases of such accumulation windows are: "from event a until event b," "one minute until event b," "from event a for one minute," and (since events can occur over time intervals, not just time points) "while event c." Here we only look at the last case because it subsumes the first three (they can be defined as composite events, once more emphasizing the advantage of using time intervals rather than points for events).

Proceeding further with examples, let's say that an order is overdue if it has not been shipped within 6 hours in the case that less than 10 items were ordered and within 12 hours in the case of 10 or more items. Detecting "overdue" events involves a negation, i.e., the absence of "shipped" events in a given accumulation window. The accumulation window is defined in relation to the "overdue" event (6 and 12 hours on from the "overdue" event) and accordingly called a **relative temporal event**. The rules for the two cases are:

$$\text{(2)} \quad \begin{array}{lcl} \text{overdue}(id) & \leftarrow & o : \text{order}(id, p, q), w : \text{extend}(o, 6h), \\ & & \mathsf{while}\ w :\ \mathsf{not}\ \text{shipped}(id, t), q < 10 \end{array}$$

$$\text{(3)} \quad \begin{array}{lcl} \text{overdue}(id) & \leftarrow & o : \text{order}(id, p, q), w : \text{extend}(o, 12h), \\ & & \mathsf{while}\ w :\ \mathsf{not}\ \text{shipped}(id, t), q \geq 10 \end{array}$$

The relative temporal event is specified as $\text{extend}(o, 6h)$ and begins with the start of an "order" event (o being its event identifier) and ends 6 hours after the end of "order." Such relative timer events are particularly useful when time-outs are involved (as in the example above) or when counts, averages and other aggregations should be computed over sliding time windows (as we will see further down).

The event identifier w of the temporal event is then used to specify the accumulation window (keyword while) for the negation (keyword not) of shipped events. Note that the event negation is (and must be!) sensitive to variable bindings. Only the absence of a "shipped" event with the same id as the "order" is relevant.

Finally the two rules also show that we can apply conditions on the event data such as $q < 10$ and $q \geq 10$ just like

[1] As usual in rule languages, we write conjunctions as a comma ",", though we will use the more logic-oriented notation of "$\wedge$" in the model theory.

one can in any database query language.

As an example of an aggregation, consider reporting the number of all "shipped" events that have taken place in the last 24 hours whenever an "overdue" event is detected. A report of a high number could indicate that the shipping department is overloaded, a lower number that the problem is elsewhere.

$$(4)\quad \begin{array}{lll} \mathrm{rep}(\mathsf{count}(sid)) & \leftarrow & o : \mathrm{overdue}(oid), \\ & & w : \mathrm{extend\_backward}(o, 24h), \\ & & \mathsf{while}\ w :\ \mathsf{collect}\ \mathrm{shipped}(sid, t) \end{array}$$

This rule uses event accumulation (while) to collect all "shipped" events over a given time window. (Note that different variables *oid* and *sid* are used!) The time window is specified as a window going 24 hours into the past from the current "overdue" event (extend_backward(o, 24h)). The count aggregate function in the rule head is used to yield and report the desired number.

Note that this rule queries events that have been generated by other rules, illustrating the benefit of rules as an abstraction mechanism put forward in Section 1.1.

2.3 Model Theory

Having described various requirements on expressive event query languages using the simple rule language informally, we now turn to giving declarative semantics for that rule language. An established approach for rule languages is to provide a model theory with an accompanying fixpoint theory. In case the reader is not familiar with this approach, as might be likely since it is not (yet) common for event query languages, we try to recall the necessary concepts as we go along. More in-depth introductions can be found in the literature on logic programming and (deductive) databases (e.g., [12, 25, 1, 18]). We start with the model theory and then will see why we additionally need a fixpoint theory.

The basic problem of giving semantics can be described as follows: given a set of rules (also called program) together with a set of incoming events (those events that happen in the outside world and are not derived by rules; also called the stream of incoming events), we want to know all events that are derived by the rules. (This approach also covers the issue of finding the answers for a given event query w.r.t. a program and an incoming event stream.)

The idea is to relate sentences (rules, queries) from the language to an *interpretation* by defining an *entailment relation*. An interpretation is basically a set of events (we will refine this in an instant). The entailment relation is defined *recursively over the structure* of the sentences. Of interest for the semantics are those interpretations that (1) satisfy all rules of a program w.r.t. the entailment relation and (2) contain the stream of incoming events. These interpretations are then called *models*.

Since rules contain variables which have to be bound, we also need the notion of a **substitution** and its application. A substitution σ is a mapping from variable names to values. The application of a substitution σ to an atomic event query q yields a ground (relational) fact that is denoted $\sigma(q)$. For example applying σ with $\sigma(t) = 42$ to $q = \mathrm{delivery}(t)$ yields $\sigma(q) = \mathrm{delivery}(42)$. When the constructor in the rule head contains an aggregation function (e.g., count) we have to apply a **set of substitutions** Σ rather than a single substitution. For example applying $\Sigma = \{\sigma_1, \sigma_2\}$, where $\sigma_1(sid) = 42$, $\sigma_1(sid) = 43$, to the rule head $c = \mathrm{rep}(\mathsf{count}(sid))$ from example (4) yields $\Sigma(c) = \mathrm{rep}(2)$.

An **interpretation** is 3-tuple $M = (I, \sigma, \tau)$, where (1) $I \subseteq Events$ is the set of events e^t that "happen," i.e., are either in the stream of incoming events or derived by some deductive rule. (2) σ is a grounding substitution containing substitutions for the "normal" variables (i.e., data variables, but not event identifiers). (3) τ is a substitution for the event identifiers, i.e., a mapping from event variables to *Events*. Since τ signifies the events that contributed to the answer of some query, we also call it an "event trace."

The **entailment** (or **satisfaction**) $M \models F^t$ of an expression F (rule, query etc.) over an occurrence time t in an interpretation M is defined recursively in Figure 1.

Given a program P (i.e., a set of rules) and a stream of incoming events E, we call an interpretation $M = (I, \sigma, \tau)$ a **model** of P under E if (1) M satisfies all rules $(c \leftarrow Q) \in P$ for all time intervals t and (2) contains the stream of incoming events, i.e., $E \subseteq I$. Note that here the incoming event stream simply corresponds to the notion of base facts or extensional data found of traditional model theories. When we consider whole programs P, only the set I of events that happen is relevant and σ and τ are unimportant. In those cases we identify M with I.

In the following points our model theory differs slightly from well-known model theories of logic programming and deductive databases: (1) The model theory directly accommodates use of aggregate functions in the rule head (via the last line in the definition). This is not specific for event query rule languages, but needed in any kind of rule language supporting aggregation (e.g., Xcerpt [34, 33]). (2) We use the event trace τ in addition to normal substitutions σ. It ensures that for each separate combination of events a different new event is generated and also gives rise to reasoning about (vertical) causality [26], i.e., allows to recognize which events constituted to the detection of which event. (3) Events are satisfied (entailed) over a time interval t (written in the superscript). Accommodating occurrence times of events is an obvious requirement for event query languages. (4) As we will see in Section 2.5 (second theorem), our model theory makes sense on unbounded incoming streams of events. In other words E may be infinite as long as for each time interval considered its restriction to this interval is finite.

2.4 Fixpoint Theory

A model theory, such as the one presented above, has the issue of allowing many models for a given program. If we consider as an example a program P containing the example rules (1) through (4) from this section and the empty event stream $E = \emptyset$, then $M = \emptyset$ is a model (the one that is intended), but so would be for example $M = \{\mathrm{overdue}(20)^{[2,78]}\}$. To give precise semantics we need a unique (and intended) model.

A common and convenient way to obtain such a unique model is to define it as the solution of a fixpoint equation (which is based on the model theory). A fixpoint theory also describes an abstract, simple, forward-chaining evaluation method, which can easily be extended to work incrementally as is required for event queries [8].

An additional issue is introduced by non-monotonic features such as negation and aggregation when they are combined with recursion. Consider a program consisting of the following two rules:

$$
\begin{array}{ll}
I,\sigma,\tau \models (i : q)^t & \text{iff } \sigma(q)^t \in I \text{ and } \tau(i) = \sigma(q)^t \\
I,\sigma,\tau \models (i : \mathsf{extends}(j,d))^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ \tau(i) = e^t,\ t = t' + d \\
\ldots & \text{(Definitions for other temporal events are similar and skipped.)} \\
I,\sigma,\tau \models (i \ \mathsf{before}\ j)^t & \text{iff } \mathit{end}(\tau(i)) < \mathit{begin}(\tau(j)) \\
I,\sigma,\tau \models (i \ \mathsf{during}\ j)^t & \text{iff } \mathit{begin}(\tau(j)) < \mathit{begin}(\tau(i)) \text{ and } \mathit{end}(\tau(i)) < \mathit{end}(\tau(j)) \\
\ldots & \text{(Definitions for other temporal and non-temporal conditions are similar and skipped.)} \\
M \models (q_1 \wedge \cdots \wedge q_m \wedge C_1 \wedge \cdots \wedge C_n)^t & \text{iff } M \models q_1^{t_1}, \ldots M \models q_m^{t_m},\ t = t_1 \sqcup \cdots \sqcup t_m, \text{ and } M \models C_1^t, \ldots M \models C_n^t \\
& \text{where } q_1, \ldots q_m \text{ are atomic event queries and } C_1, \ldots C_n \text{ are conditions, } m \geq 1,\ n \geq 0 \\
I,\sigma,\tau \models (\mathsf{while}\ j :\ \mathsf{not}\ q)^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ t' = t, \text{ and for all } t'' \sqsubseteq t \text{ we have } I,\sigma,\tau \not\models q^{t''} \\
I,\sigma,\tau \models (\mathsf{while}\ j :\ \mathsf{collect}\ q)^t & \text{iff exists } e^{t'} \text{ with } \tau(j) = e^{t'},\ t' = t, \text{ and exits } t'' \text{ with } I,\sigma,\tau \models q^{t''} \\
I,\sigma,\tau \models (c \leftarrow Q)^t & \text{iff (1) exists } \tau' \text{ s.t. } \Sigma(c)^t \subseteq I \text{ for } \Sigma = \{\sigma' \mid I,\sigma',\tau' \models Q^t\}, \text{ or (2) for all } \sigma' \text{ and } \tau'\text{: } I,\sigma',\tau' \not\models Q^t.
\end{array}
$$

Figure 1: Model Theory

$$
\begin{array}{lll}
\mathrm{p}(x) & \leftarrow & w : \mathrm{s}(x), \mathsf{while}\ w :\ \mathsf{not}\ \mathrm{q}(x) \\
\mathrm{q}(x) & \leftarrow & w : \mathrm{s}(x), \mathsf{while}\ w :\ \mathsf{not}\ \mathrm{p}(x)
\end{array}
$$

It is not clear what the intended semantics of this program are. For example under the event stream $E = \{s(1)^{[1,3]}\}$ both $M_1 = \{s(1)^{[1,3]}, p(1)^{[1,3]}\}$ and $M_2 = \{s(1)^{[1,3]}, q(1)^{[1,3]}\}$ are models and, since they are symmetric, none is preferable. This is a common and inherent difficulty when rules and negation are combined. (In fact it is an adaption of the standard example $p \leftarrow \neg q, q \leftarrow \neg p$ from logic programming and deductive databases.) A simple and established solution is to avoid such situations by requiring programs to be stratifiable.

Stratification restricts the use of recursion in rules by ordering the rules of a program P into so-called strata (sets P_i of rules with $P = P_1 \uplus \cdots \uplus P_n$) such that a rule in a given stratum can only depend on (i.e., access results from) rules in lower strata (or the same stratum, in some cases).[2] One can then give a unique model for each stratum using as "input" (as stream of incoming events) the model of the next lower stratum.

The restriction to stratifiable programs could be partially lifted at the cost of a more involved semantics (and evaluation) and there has been much research on this issue. This is however outside the scope of this paper.

Three types of stratification are required for our event query rule language: (1) Negation stratification, i.e., events that are negated in the query of a rule may only be constructed by rules in lower strata, events that occur positively may only be constructed by rules in lower strata or the same stratum. (2) Grouping stratification, i.e., rules using aggregation constructs like count in the head (together with a collect in the body) may only query for events constructed in lower strata. (3) Temporal stratification, i.e., if a rule queries a relative temporal event like extends$(i, 1h)$ then the anchoring event (here: i) may only be constructed in lower strata. While negation and grouping stratification are fairly standard, temporal stratification is a requirement specific to complex event query programs like those expressible in our rule language.

The **fixpoint operator** T_P for a program P is defined as:

$$
\begin{aligned}
T_P(I) = I \cup \{e^t \mid \quad & \text{there exist a rule } c \leftarrow Q \in P, \\
& \text{a maximal set } \Sigma \text{ of substitutions,} \\
& \text{and an event trace } \tau \text{ such that} \\
& e \in \Sigma(c) \text{ and } \forall \sigma \in \Sigma. I, \sigma, \tau \models Q^t\}
\end{aligned}
$$

[2]Equivalently one can partition the relation names used in rule heads instead of rules.

The repeated application of T_P until a fixpoint is reached is denoted T_P^ω.

The **fixpoint interpretation**[3] $M_{P,E}$ of a program P with stratification $P = P_1 \uplus \cdots \uplus P_n$ under and event stream E is defined by computing fixpoints stratum by stratum: $M_0 = E = T_\emptyset^\omega(E)$, $M_1 = T_{\overline{P_1}}^\omega(M_0)$ $\ldots$, $M_{P,E} = M_n = T_{\overline{P_n}}^\omega(M_{n-1})$. Here, $\overline{P_i} = \bigcup_{j \leq j} P_j$ denotes the set of all rules in strata P_i and lower.

2.5 Theorems

We now give a theorem that shows that the declarative semantics for programs in our rule language provided by the fixpoint interpretation are well-defined and unambiguous.

Theorem For a stratifiable program P and an event stream E, $M_{P,E}$ is a minimal model of P under E. Further, $M_{P,E}$ is independent of the stratification of P.

Note that "minimal" in the theorem entails that all event in the model are either in the stream of incoming events or have been derived by rules, i.e., no events have been added without justification.

More interestingly, we can show that the model theory and fixpoint semantics are sensible on infinite event streams. The next theorem justifies a streaming evaluation, where answers to composite event queries are generated "online" and we never have to wait for the stream to end. This is especially important, since event streams can conceptually be infinite and thus not end at all.

In particular it ensures an event e^t can be detected at the time point $\mathit{end}(t)$ since no knowledge about any events in the future of $\mathit{end}(t)$ is required. Ensuring that evaluation methods are not expected to "crystal gaze" is of course an important requirement and one example where we can use the declarative semantics to prove interesting statements about a (composite) event query language.

Theorem Let $E \mid t$ denote the restriction of an event stream E to a time interval t, i.e., $E \mid t = \{e^{t'} \in E \mid t' \sqsubseteq t\}$. Similarly, let $M \mid t$ denote the restriction of an interpretation M to t. Then the result of applying the fixpoint procedure to $E \mid t$ is the same as applying it to E for the time interval t, i.e., $M_{P,E|t} \mid t = M_{P,E} \mid t$. In other words to evaluate a program over a time interval t, we do not have to consider any events happening outside of t.

Proofs for both theorems are presented in [9] in a more general setting (queries against events in XML format instead of relational). The proof for the first theorem is an adaption of a standard proof in [25].

[3]As mentioned before, we consider whole programs P now and thus can skip σ and τ in the interpretation I.

3. OPERATIONAL SEMANTICS

While declarative semantics tell us *what* the answers to given event queries are, they tell us very little on *how* to actually evaluate them. We concentrate in this paper on evaluating queries in rule bodies, touching only briefly upon issues related to chaining of rules.

3.1 Relational Algebra for Events

We now first translate rule bodies into relational algebra expressions. These serve as a logical query plan and we can exploit query rewriting as an optimization technique (see Section 4). The actual incremental evaluation of (possibly rewritten) query plans will be the topic of Section 3.2.

Whenever an event (e.g., $\text{order}(42, \text{"muffins"}, 2)^{[3,3]}$) occurs that matches some atomic event query (e.g., o : $\text{order}(\text{id}, \text{p}, \text{q})$) this gives bindings for the free variables in the query (e.g., $id \mapsto 42, p \mapsto \text{"muffins"}, q \mapsto 2$) together with the event's occurrence time. We will represent the occurrence time as variable bindings with the special names $i.s$ and $i.e$, where i is the event identifier given in the query (e.g., $o.s \mapsto 3, o.e \mapsto 3$). This leads directly to representing the results of atomic event queries as relations of named tuples. Each atomic event query i : Q has an associated base relation R_i with schema $sch(R_i) = \{i.s, i.e\} \cup freevars(\text{Q})$.

By virtue of representing occurrence times as part of tuples, translating composite event queries of rule bodies into relational algebra expressions becomes quite straightforward. (Extended) projection is used to discard variables that do not occur in the rule head and to compute the occurrence time of the result. Combination of (atomic) event queries with conjunction is translated as a natural join. Conditions on the data are expressed as selections. Maybe a bit surprisingly, temporal conditions (such as o before s are also expressed as selections; this works because we made temporal information (i.e., occurrence times of events) part of the data of our base relations.

With this and R_o, S_s, T_d respectively denoting the relations for o : order(id, p, q), s : shipped(id, t), and d : delivered(t), the query from example (1) from the previous section can be expressed as:

$$\pi_{r.s \leftarrow \min\{o.s,s.s,d.s\}, r.e \leftarrow \max\{o.e,s.e,d.e\}, id, p}(\sigma_{o.e<s.s \wedge s.e<d.s}(R_o \bowtie S_s \bowtie T_d))$$

The starting time $r.s$ of the result is the minimum of all involved starting times ($o.s$, $s.s$, $d.s$), the ending time $r.e$ is the maximum of all ending times ($o.e$, $s.e$, $d.e$).

Negation of events must be, as mentioned earlier, sensitive to variable bindings. It can be expressed using a θ-anti-semi-join, which is defined as $R \overline{\ltimes}_\theta S = R \setminus \pi_{sch(R)}(\sigma_\theta(R \bowtie S))$.

Relative timer events require the construction of their occurrence times and can thus be expressed by an extended projection.

The expression for example (2) then is (analogous for (3)):

$$\pi_{r.s \leftarrow \min\{o.s,w.s\}, r.e \leftarrow \max\{o.e,w.e\}, id}(\sigma_{q<10}(\pi_{w.s \leftarrow o.s, w.e \leftarrow o.e+6h, sch(R_o)}(R_o) \overline{\ltimes}_{w.s<s.s \wedge s.e<w.e} S_s))$$

When event accumulation is used for aggregating data from events, this requires a θ-join between the accumulated events and the rest of the query, where the θ expresses the temporal condition given by the accumulation window. For the actual aggregation in the head, the grouping operator γ is used. (We follow the common notation and meaning for γ as given in [18]: it partitions the input tuples into groups of tuples having equal values on the grouping attributes and for each group outputs a single tuple with the grouping attributes and the additional aggregated attributed.)

With T_o and U_s denoting the relations for o : overdue(oid) and s : shipped(sid, t), the expression for example (4) is:

$$\gamma_{r.s,r.e,COUNT(sid)}(\pi_{r.s \leftarrow \min\{o.s,w.s\}, r.e \leftarrow \max\{o.e,w.e\}, sid}(\pi_{w.s \leftarrow o.s-24h, w.e \leftarrow o.e, sch(T_o)}(T_o) \bowtie_{w.s<s.s \wedge s.e<w.e} U_s))$$

The framework laid out in this section is fairly general. Most event queries expressible in other event query languages, e.g., [2, 3, 5, 11, 14, 19, 27, 29, 32], can be translated into both the rule language and the relational algebra quite easily. Note however that we have not discussed event consumption or instance selection [36, 23] here, so far; we will do this in Section 5.

3.2 Incremental Evaluation

The task of evaluating a composite event query Q_r (given as a relational algebra expression) is a step-wise procedure: In each step, we are given a set E of events that happen *at* the current time, which we will denote *now* (i.e., for all $e^t \in E : end(t) = now$). The required output then are all answers produces by Q_r that happen *at* the current time *now*, i.e., only $Q_{new} = \sigma_{r.e=now}(Q)$. The computation of Q_{new} usually requires knowledge of events that happened *before* the current time.[4] Accordingly, in each step we also have to maintain some data structures that store knowledge about these events for use in future evaluation step. Note that in this description of the task, we assume that events arrive in order of their occurrence times. We also say that the event stream is (temporally) ordered. A discussion on lifting this restriction can be found in [10].

A primitive way of evaluating a given event query Q_r would be based on maintaining as data structure simply a set S of all events received seen so far. In each step, add E to S, i.e., $S := S \cup E$ and evaluate the expression $\sigma_{r.e=now}(Q)$ against S. This approach is however undesirable for efficiency reasons: each step performs a considerable amount of redundant computation, recomputing (intermediate) results that had been computed also in previous steps. (These intermediate results are also called "partial answers" or "semi-composed events" in other works.)

An incremental, data-driven approach that "remembers" intermediate results across different evaluation steps (in the style of the rete algorithm [16]), is preferable. The primary concern in making evaluation incremental is on joins since these are "blocking" operators, i.e., their current input may have to be combined with future inputs. For selection and projection this is not the case; they apply on a per-tuple basis and can directly output their results. Note that the grouping operator γ as we used it in the previous section can be understood as non-blocking, since the blocking has already been performed by a join in its input.

The basic idea for making a join $R \bowtie S$ incremental is to make it a stateful operator that stores the inputs that might be needed in future evaluation steps and in each step only outputs those results that are new (i.e., happen at the time of the current evaluation step and thus have not been computed in previous steps). For this, we use the basic fact that $R \bowtie S = (R_{old} \bowtie S_{old}) \cup (R_{new} \bowtie S_{old}) \cup (R_{old} \bowtie S_{new}) \cup (R_{new} \bowtie S_{new})$, where R_{new} and S_{new} contain only

[4]Note that the second theorem of Section 2.5 ensures that no knowledge of future events is required.

the events happening at the current time, and R_{old} and S_{old} any (relevant) events that happened before. When performing the join in an evaluation step, $R_{old} \bowtie S_{old}$ need not be computed because it has already been computed by the previous steps. Note that all parts of the union are disjoint due to the different occurrence times on their tuples.

The straightforward way of describing the incremental evaluation just outlined is to perceive each node in the expression tree as an object which has pointers to node objects for its subexpressions (arguments), some auxiliary data, and a method eval() which delivers the result of evaluating the expression (only those events happening at the current time). We additionally assume a global set E of all events happening at the current time as described above, which will be used for evaluating atomic events. The nodes for atomic event queries, selections, and joins are depicted in Figure 2 in pseudo code.

Nodes for relative timer events (extended projections) potentially generate tuples with an occurrence time $j.e$ that lie in the future (i.e., $j.e > now$). These should not be passed on to the parent node immediately, but only in a later evaluation step. They are therefore stored in a relation $R_{delayed}$ until time has progressed further. Pseudo code for the relative timer node is also depicted in Figure 2.

Stateful operators are easy to understand and implement. We would however like to also sketch a slightly different, more abstract perspective on incremental evaluation as well, which emphasizes the relationship with (incremental) maintenance of materialized views [22].

In the expression for example (1) from the Section 3.1, we would maintain the materialized view $V_{o,s} = R_o \bowtie S_s$ together with materializations of R_o, S_s, T_d.[5] The query then is $Q_r = \pi_X(\sigma_C(V \bowtie T_d))$ (subscripts abbreviated with X and C). In each evaluation step we are given the changes (i.e., the events that are added) to the base relations, which we denote $\triangle R_o$, $\triangle S_s$, $\triangle T_d$. The change to the materialized view $V_{o,s}$ then is given (as explained earlier) by $\triangle V_{o,s} = (\triangle R_o \bowtie \triangle S_s) \cup (R_o \bowtie \triangle S_s) \cup (\triangle R_o \bowtie S_s)$. In turn the change to Q_r is $\triangle Q_r = \pi_X(\sigma_C((\triangle V_{o,s} \bowtie \triangle T_d) \cup (V_{o,s} \bowtie \triangle T_d) \cup (\triangle V_{o,s} \bowtie T_d))$. This $\triangle Q_r$ is exactly the Q_{new}, which we require as output of the current step (see beginning of this section).

Derivation of expression accounting for the changes such as $\triangle V_{o,s}$ and $\triangle Q_r$ has been studied extensively in the literature. Usually these works are more general, accounting not only for adding of tuples ($\triangle R$) but also deletion of tuples ($\triangledown R$) and include a minimality condition for the change expressions. View maintenance has also been studied using bag algebras, which allow duplicates, instead of (set-oriented) relational algebra [20].

Advantages of this "view maintenance perspective" on incremental evaluation are: (1) It accounts well for multi-query optimization: the same materialized view can be used for different queries. (2) Such optimizations can be derived by rewriting rules on a set of view definitions (together with other optimizations). (3) View maintenance also accounts for deletion of tuples (in intermediate and final results). This is beneficial when the assumption that the event stream is temporally ordered is dropped and latency concerns make an evaluation preferable that generates at first incorrect or incomplete answers (when non-monotonic features such as negation or aggregation are considered) and produces corrections afterwards. (4) When storing *all* intermediate results is not possible or desirable, it is open to space-time trade-offs [31], i.e., storing only certain intermediate results and recomputing others. (5) Finally, being amenable to equational reasoning it simplifies theoretical investigations, in particular correctness proofs (see also [20]).

[5]Note that often we will not materialize a base relation R itself, as here, but an actual view $V' = \pi_{X'}\sigma_{C'}(R)$ of it.

4. OPTIMIZATION

An important motivation behind basing composite event query evaluation on relation algebra is to perform query optimization like in databases. We only scratch the surface of event query optimization here, briefly discussing physical optimization, logical optimization, and temporal optimization. While the first two are well-explored in databases, the third is rather novel and specific to event queries.

We would like to emphasize that the aim of this section is not to develop a full event query optimizer, but rather to demonstrate the potential of grounding event query optimization in relational algebra.

4.1 Physical Optimization

Physical Optimization encompasses choosing implementations for operators (e.g., join algorithms) as well as choosing index structures for intermediate results.

As an example for an operator implementation, (double) pipelined hash joins have proven particularly effective in scenarios that are similar to (main memory) event query evaluation [24].

Note that index structures for intermediate results (including the base relations) can be chosen at the compile-time of the query, i.e., an informed choice can me made based on a priori knowledge of the query. In a database, in contrast, index structures for base relations are determined earlier as part of database design. In this phase there is no (full) knowledge of the queries and usually a human database administrator has to choose index structures based on predictions of the queries that will be made.

4.2 Logical Optimization

Logical Optimization attempts to transform a given query plan (given in relational algebra) into an equivalent but more efficient plan. Rewriting rules can be used to express such transformations. They are based on well-known equivalences of relational algebra expressions such as $R \bowtie S = S \bowtie R$, $(R \bowtie S) \bowtie T = (R \bowtie S) \bowtie T$, $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$ (provided C contains only attributes from $sch(R)$). Arguably, choosing a good join order and pushing selections are equally important in event query evaluation as they are in traditional database query evaluation.

To the well-known equivalences, event querying potentially adds equivalences based on (simple) temporal reasoning to add or remove implied temporal conditions, e.g., $\sigma_{i.e<j.s}(R) = \sigma_{i.e<j.s \wedge i.s<j.s}(R)$ (since trivially $i.s \leq i.e$).

To illustrate query rewriting, the original expression of query (1) can be transformed to

$$(*) \quad \begin{array}{l} \pi_{r.s \leftarrow \min\{o.s,s.s,d.s\}, r.e \leftarrow \max\{o.e,s.e,d.e\}, id, p}(\\ \quad \pi_{o.s,o.e,n,p}(R_o) \bowtie_{o.e<s.s} (S_s \bowtie_{s.e<d.s} T_d)\ ) \end{array}$$

We will discuss in the next section that θ-joins where θ is a temporal condition (e.g., $\theta = o.e < s.s$) of a particular form allow for an interesting optimization based on temporal order of events.

AtomicNode extends QueryNode:
  AtomicQuery A;
  Relation eval():
    return $A(E)$;

SelectionNode extends QueryNode:
  QueryNode Q;
  Condition C;
  Relation eval():
    return $\sigma_C(Q.\text{eval}())$;

JoinNode extends QueryNode:
  QueryNode Q_L, Q_R;
  Relation L_{old}, R_{old};
  Relation eval():
    $L_{new} := Q_L.\text{eval}()$;
    $R_{new} := Q_R.\text{eval}()$;
    $J := (L_{new} \bowtie R_{old})$
      $\cup (L_{old} \bowtie R_{new})$
      $\cup (L_{new} \bowtie R_{new})$;
    $L_{old} := L_{old} \cup L_{new}$;
    $R_{old} := R_{old} \cup R_{new}$;
    return J;

RelativeTimerNode extends QueryNode:
  QueryNode Q;
  Relation $R_{delayed}$;
  EventAttribute i,j;
  Duration s', e';
  Relation eval():
    $N := Q.\text{eval}()$;
    $R_{new} := \pi_{j.s\leftarrow i.s+s', j.e\leftarrow i.e+e', sch(Q)}(N)$;
    $J := \sigma_{j.e\leq now}(R_{delayed} \cup R_{new})$
    $R_{delayed} := \sigma_{j.e>now}(R_{delayed} \cup R_{new})$
    return J;

Figure 2: Implementation of operator nodes for incremental evaluation

Query rewriting based on both heuristics and cost-estimation is well-studied in databases. In the context of event query evaluation, appropriate cost measures (e.g., throughput, latency) and cost-estimation techniques (e.g., to estimate throughput of operators or sizes of intermediate results) might still have to be developed, though. A particular issue is that there are scenarios in event processing where statistics for the event stream (i.e., the base relations) that would be necessary for a cost-estimation are not available at the time of the query compilation. (Keep in mind that a database has all data –and thus also necessary statistics– available at the time a given query is executed. In contrast in an event processing system, data –and thus statistics– might only arrive *after* the query has been posed.) One solution for such situations is to generate several query plans based solely on heuristics, start all plans in parallel, and drop over time those plans that turn out to be inefficient.

4.3 Temporal Optimization

Recall from Section 3.2 that we assume event streams to be temporally ordered. This assumption gives rise to optimizations based on temporal conditions specified in queries. (Slightly more complex variants of the optimizations are still possible when the order assumption is lifted.)

When we look at how the rewritten relational algebra expression $(*)$ from example query (1) is evaluated, we can see that the joins evaluate and store both inputs according to the pseudo code from Figure 2. (The same applies to the original expression from Section 3.1.) By considering the temporal conditions in the query, which manifest as conditions of the θ-joins, we could do better: It is unnecessary to *store* the right input (T_d) of the second join ($\bowtie_{s.e<d.s}$) — the temporal condition $s.e < d.s$ will discard any tuples generated by the joins ($S_{new} \bowtie T_{old}$) and ($S_{new} \bowtie T_{new}$). With similar justification, it is not necessary to store the right input (results of $\bowtie_{s.e<d.s}$) in the first join ($\bowtie_{s.e<d.s}$). Further, in both joins it is unnecessary to even *evaluate* the right input as long as the left input is empty.

Temporal θ-joins that make the described optimizations have been introduced in [10]. The first temporal θ-join streams (i.e., does not store) one of its inputs (either left or right). The second temporal θ-join suppresses (avoids) evaluation of its right argument whenever the current state of query evaluation permits. The optimizations can also be combined into a left-streaming/right-suppressing and a right-streaming/right-suppressing θ-join. The implementation of these θ-joins as well as the conditions (prerequisites on θ) when they may be applied are given in detail in [10].

Another important optimization is concerned with using metric temporal conditions such as $\{o, s, d\}$ within $48h$ (cf. Section 2.2) to purge tuples (events) that have "timed-out" from stores. To illustrate this, consider a variant of query (1) for orders that have been completed in a timely fashion, so that order and shipping happen within 12 hours, while shipping and delivery happen within 36 hours:

$$\begin{aligned} \text{timely}(id, p) \quad \leftarrow \quad & o : \text{order}(n, p, q), s : \text{shipped}(n, t), \\ & d : \text{delivered(t)}, o \textsf{ before } s, s \textsf{ before } t, \\ & \{o, s\} o \textsf{ within } 12h, \{s, d\} \textsf{ within } 36h, \end{aligned}$$

A (rewritten) relation algebra expression for this query is:

$$\pi_{r.s\leftarrow\min\{o.s,s.s,d.s\}, r.e\leftarrow\max\{o.e,s.e,d.e\}, id, p}(\sigma_{d.e-s.s\leq 36} \\ (R_o \bowtie_{o.e<s.s, s.e-o.s\leq 12h} S_s) \bowtie_{s.e<d.s, d.e-o.s\leq 48h} T_d) \ )$$

Note that the condition $d.e - o.s \leq 48h$ in the second join has been inferred from the original conditions in the query. Rewriting rules, as argued for in Section 4.2, are well-suited for making such inferences.

For the first join, we can use the condition $s.e - o.s \leq 12h$ to purge any tuples that are older than 12 hours from the left store (which contains "order" events coming from R_o). For the second join, we can similarly use the condition $d.e - o.s \leq 48h$ to purge any tuples that are older than 48 hours from the left store. Keep in mind that, as argued earlier, the right argument in both joins can be made streaming, i.e., is not stored at all. Accordingly it is not necessary to purge tuples on the right side (though the conditions would allow that).

Removing stored events is an important requirement in event querying. If events are not removed at all, storage for intermediate results typically grows at least linearly in the number of events received so far [13]. For event queries against unbounded streams this is not tolerable. Note however that in order to remove stored events, temporal conditions (such as the one above) have to be given with the queries. This can lead to notions such as a restricted class of "legal event queries" as introduced in [11].

5. DISCUSSION

While we have used a concrete event query language to give declarative and operational semantics, the general approach is *not* language specific. We now discuss strengths and weaknesses of our approach.

A number of features found in other event query languages are not fully covered by the approach presented. Of particular concern are event instance selection and event consumption [36, 23]. (The general issue has been introduced first in

[14] with so-called "parameter contexts.")

Consider a query for generating a congestion warning for cars. Each car car sends an event position(car, loc) whenever it enters the region loc. A congestion in area loc is reported with an event congestion(loc). Whenever a congestion event happens, we would like to warn all cars that are in area loc. A rule such as

$$\begin{array}{lll} \text{warn}(car) & \leftarrow & l : \text{position}(car, loc), j : \text{congestion}(loc), \\ & & l \;\mathsf{before}\; j \end{array}$$

would not capture the intended application semantics: a given "congestion" event would be paired up with *all* "position" events that car car has sent so far. However, for the intended application only the most recent "position" event is relevant for each car.

Event instance selection addresses this issue by given language constructs to select only specific instances of events, e.g., the last (as required above), first, n-th event, or all events. Note that defining what constitutes the first, last, or n-th event can be complicated because there might not be a total order on events (e.g., when the events occur over time intervals and overlap). See also discussion in [35].

Further, depending on the involved event sources, the same congestion might be reported several times. When this happens, we do not want to generate further warnings. 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 later answers. In the example, generating a warning could consume the "location" event. As a consequence, future "congestion" events (with the same location) would not generate further warnings because there is no corresponding "location" event anymore.

Event instance selection and consumption tend to be rather difficult to use and understand. Moreover, queries that are stated in natural language often leave the intended selection and consumption semantics implicit (as purposefully illustrated with the congestion example). It requires significant application domain knowledge to derive them.

Formal semantics for an event query language that incorporates instance selection and consumption are presented in [23]. However, it does not account for data in events and events that are correlated by their data. Considering event data can be quite important: in the above example consumption and instance selection should only be done on a per-car basis, i.e., we only want to select the last instance of a "location" event for each car, not the last out of all "location" events from all cars. Further [23] is based on time points rather than intervals and does not support rules.

Dealing with event data is a particular strength of both the declarative and the operational semantics put forward in this work. This issue has been neglected in related work, where events have typically been treated as simple propositional facts that have no associated data. We have seen in the examples throughout this paper that data is important and affects query semantics. Particularly we would like to point out that there are queries, such as example (1), where events are not correlated on a single parameter (variable). Along the same line, aggregation of data from events has rarely been considered in related work on event queries, even though it is clearly needed in many applications. (A notable exception is [30].)

Our approach extends quite straightforwardly to incorporate queries that also access non-event data, e.g., data from databases. The need to access non-event data provides a strong motivation to apply the foundations of database to querying events, as done in this work. To give an example where combination of event and non-event data is necessary, reconsider the "overdue" queries (2) and (3). Assume that perishable products have to be shipped within only 3 hours and that there is a database relation perishable(p) listing all perishable products. We can use the database relation as a condition, not different from other conditions such as $q < 10$ in (2), leading to a rule

$$\begin{array}{lll} \text{overdue}(id) & \leftarrow & o : \text{order}(id, p, q), w : \text{extend}(o, 6h), \\ & & \mathsf{while}\; w :\; \mathsf{not}\; \text{shipped}(id, t), \\ & & \text{perishable}(p) \end{array}$$

Declarative semantics can be extended to include in its interpretation a set D of database facts in addition to the event stream E. Note that deductive rules about non-event data can also be accommodated. For operational semantics, this simply leads to a join between event tuples and tuples from the database. Note that the join should usually be evaluated with event tuples driving retrieval of database tuples.

Combining non-event data with event data raises a potential issue when the non-event data changes during the detection of the (composite) event. It then becomes relevant when the non-event data is accessed. Our declarative semantics could be extended so that it has not a single set D of database facts but a sequence of such sets with associated validity times. Operational semantics would either have to obey restrictions that make sure that non-event data is retrieved at the correct time or incorporate constructs that allow to retrieve a specific version of a database. An example for the latter is the "rollback operator" of [28].

Maybe the most important strength of using relational algebra as a basis for operational semantics it that it allows for well-known optimizations, in particular based on query rewriting, as well as new optimizations based on temporal conditions. Optimization can be expected to become as important for event queries as it is for database queries. So far however, there has been little study on rewriting event queries. Formal foundations for event queries, in particular with a clean separation of declarative and operational semantics as in this article, can be argued to be a key enabler for future research on optimization.

6. CONCLUSION

The results outline in this article are part of ongoing work on the high-level event query language XChange$^{\text{EQ}}$, which supports querying events in XML formats and deductive rules. We have presented declarative and operational semantics for event queries that aim at making well-known approaches and results from the database field applicable to events queries. Our approach emphasizes the necessity to access data in events when querying and correlating events. It includes support for deductive rules, an important simple and effective abstraction and reasoning mechanism.

7. 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`).

8. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data and Knowledge Eng.*, 59(1), 2006.

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

[4] J. F. Allen. Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), 1983.

[5] R. S. Barga and H. Caituiro-Monge. Event correlation and pattern detection in CEDR. In *Proc. Int. Workshop Reactivity on the Web*, 2006.

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

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

[8] F. Bry and M. Eckert. Twelve theses on reactive rules for the Web. In *Proc. Int. Workshop Reactivity on the Web*, 2006.

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

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

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

[12] F. Bry, N. Eisinger, T. Eiter, T. Furche, G. Gottlob, C. Ley, B. Linse, R. Pichler, and F. Wei. Foundations of rule-based query answering. In *Reasoning Web, Int. Summer School*, LNCS. Springer, 2007.

[13] A. P. Buchmann, J. Zimmermann, J. A. Blakeley, and D. L. Wells. Building an integrated active OODBMS: Requirements, architecture, and design decisions. In *Proc. Int. Conf. on Data Engineering*, pages 117–128, 1995.

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

[15] Common Base Event. www.ibm.com/developerworks/webservices/library/ws-cbe.

[16] C. L. Forgy. A fast algorithm for the many pattern/many object pattern match problem. *Artif. Intelligence*, 19(1), 1982.

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

[18] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

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

[20] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1995.

[21] M. Gudgin et al. SOAP 1.2. W3C recomm., 2003.

[22] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2), 1995.

[23] A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proc. Int. Symp. on Temporal Representation and Reasoning*, 2002.

[24] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML query engine for network-bound data. *VLDB J.*, 11(4), 2002.

[25] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.

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

[27] M. Mansouri-Samani and M. Sloman. GEM: A generalised event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2), 1997.

[28] L. E. McKenzie and R. T. Snodgrass. Extending the relational algebra to support transaction time. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1987.

[29] D. Moreto and M. Endler. Evaluating composite events using shared trees. *IEE Proceedings — Software*, 148(1), 2001.

[30] I. Motakis and C. Zaniolo. Temporal aggregation in active database rules. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1997.

[31] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In *Proc. Int. Conf. on Management of Data (SIGMOD)*, 1996.

[32] C. Sánchez, M. Slanina, H. B. Sipma, and Z. Manna. Expressive completeness of an event-pattern reactive programming language. In *Proc. Int. Conf. on Formal Techniques for Networked and Distrib. Systems*, 2005.

[33] S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Inst. f. Informatics, U. of Munich, 2004.

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

[35] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is 'next' in event processing? In *Symp. on Principles of Database Systems (PODS)*, 2007.

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

[37] Dagstuhl Seminar 07191 Event Processing, `http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=2007191`.