Acknowledgements

This research has been founded in part by the European Commission within the the project “EMILI — Emergency Management in Large Infrastructures” under grant agreement number 242438 and by the German Research Foundation (Deutsche Forschungsgemeinschaft) within the project “QONCEPT — Query Optimization in Complex Event Processing Technologies” under reference number BR 2355/1-1.

Contents

Two Semantics for CEP, no Double Talk: Complex Event Relational Algebra (CERA) and its Application to XChange^{EQ}

Michael Eckert, François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann

Abstract Complex Event Processing (CEP) denotes algorithmic methods for deriving higher-level knowledge, or complex events, from a stream of lower-level events in a continuous and timely fashion. High-level Event Query Languages (EQLs) are designed for expressing complex events in a convenient, concise, effective and maintainable manner. CEP differs fundamentally from traditional database or Web querying, as CEP continuously evaluates standing queries against a stream of incoming event data whereas traditional querying evaluates incoming ad hoc queries against (more or less) standing data.
However EQLs and traditional query languages share a need for clear formal semantics which typically consist of two parts: A declarative semantics specifying *what* the answer of a query should be and an operational semantics telling *how* this answer is actually computed. The declarative semantics serves as reference for the operational semantics which is the basis for query evaluation and optimization.
While formal semantics is well-understood for traditional query languages it has been rather neglected for EQLs so far. In this chapter we use the EQL XChange^{EQ} to demonstrate a general, easily transferable approach for defining both, the declarative and operational semantics of an EQL. The operational semantics on the one hand, bases on CERA, a tailored variant of relational algebra, and incremantal evaluation of query plans. Although the basic idea might sound familiar from previous approaches like [3, 12, 16], the way it is realized here is significantly different. The declarative semantics on the other hand, is defined using a Tarski-style model theory with accompanying fixpoint theory.

Michael Eckert
TIBCO Software, Balanstr. 49, 81669 Munich, Germany,
e-mail: meckert@tibco.com

François Bry, Simon Brodt, Olga Poppe, and Steffen Hausmann
Institute for Informatics, University of Munich, Oettingenstr. 67, 80538 Munich, Germany,
e-mail: {bry,brodt,poppe,hausmann}@pms.ifi.lmu.de

1 Introduction

In databases, relational algebra describes the order in which operators are applied to relations to compute answers for queries. It serves as a theoretical fundament for the operational semantics of database query languages (e.g., SQL) and query optimization.

A recent trend in information systems are continuous queries against event (or data) streams. This continuous querying of events is fundamentally different from the traditional ad hoc querying of databases or Web data, since event queries are standing queries evaluated continuously over time against changing event data received as an incoming stream.

Querying events often involves the notion of Complex Event Processing (CEP) which denotes algorithmic methods for making sense of events by deriving higher-level knowledge, or complex events, from lower-level events in a timely fashion and permanently. We refer the reader to the chapter "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed", in this volume, for a discussion of Event Query Languages (EQLs). Specific evaluation methods have been conceived for the efficient, stepwise evaluation of complex event queries against event data streams. We demonstrate the usage of one of these languages, XChange$^{\mathrm{EQ}}$ [5, 11], in a sensor network use case in Section 2.

We present Complex Event Relational Algebra (CERA), an extended and tailored variant of relational algebra, to represent execution plans for complex event queries in Section 3. Starting out with relational algebra and thus building on the foundation of database queries is not just helpful for understandability, it also lets event queries benefit from many results in database research (e.g., join algorithms, adaptive query evaluation). Further, the uniformity in the foundations of event queries and traditional queries is beneficial in systems and languages where event and non-event data is processed together — and queries should be optimized and evaluated together. This is quite common, especially for Event Condition Action (ECA) rules, where the E part is an event query, the C part a traditional query, and the parts share information through variable bindings.

The basic idea of transferring relational algebra to CEP is not new [3, 12, 16]. However we propose a significantly different way of doing so. Previous approaches like CQL [3, 12, 16] use stream-to-relation operators like time windows to conceptually convert the stream into a *finite* relation for *each point of time*. After that, quite ordinary relational algebra expressions are applied to this finite relation. In contrast to that, CERA views the whole stream as *one* potentially *infintite* relation. Tailored variant of relational operators are then applied to this infinite relation. The trick is that these operators are restricted in such a way that for each point of time it is sufficient to know the finite available part of the stream to compute the result up to that time point (see Section 3.4.1). Therefore CERA is suitable for an incremental, step-wise evaluation as required for complex event queries.

We illustrate CERA by its application to XChange$^{\mathrm{EQ}}$ [5, 11] which is one of the recently developed, expressive and easy-to-use high-level EQLs. We also provide details on how XChange$^{\mathrm{EQ}}$ rules are translated into CERA expressions and how query plans consisting of CERA expressions can be optimized and incrementally evaluated.

An important aspect of any query evaluation method is its correctness. Yet, this aspect has often been neglected in CEP so far. Proving the correctness of an operational semantics, which focuses on *how* an answer for a query is computed, entails the existence of a declarative semantics, which focuses on *what* the answers should be. To this end, in Section 4 we introduce a declarative semantics for XChange$^{\mathrm{EQ}}$ that is quite natural for event streams and in Section 5 we sketch a proof of the correctness of the operational semantics based on CERA with respect to the declarative semantics. Section 6 concludes this chapter.

The contributions of this chapter are:

1. Formal definition of CERA as the first corner stone of the operational semantics for EQLs
2. Description of incremental evaluation of query plans with materialization points as the second corner stone of the operational semantics for EQLs
3. Formal definition of the declarative semantics for EQLs by a Tarski-style model theory with accompanying fixpoint theory
4. Illustration of both semantics by an XChange$^{\mathrm{EQ}}$ program in the realm of sensor network use case
5. Proof of the correctness, i.e., soundness and completeness of the operational semantics for EQLs with respect to their declarative semantics

2 CEP Examples

Figure 1 contains an XChange$^{\mathrm{EQ}}$ program *P* which will be used as an example throughout this chapter. In this section we briefly describe the program and refer the reader to the chapter on “A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed”, in this volume, for the informal definitions of the basic notions in the realm of CEP (Section 2), the explanation of the syntax of XChange$^{\mathrm{EQ}}$ (Section 8), and the description of the sensor network use case (Section 1).

The first rule of the program in Figure 1 triggers fire alarm for an area if smoke and high temperature were measured in the area within one minute. For the second rule, assume all sensors of our network send their temperature measurements every 12 seconds. The second rule of the program infers that a sensor had been burnt down if it measured high temperature and did not send its measurements afterwards. And the last query computes average temperature reported by a sensor during the last minute every time the sensor sends a temperature measurement.

```
DETECT   fire { area { var A } }
ON and { event s: smoke {{ area {{ var A }} }},
         event t: temp {{ area {{ var A }}, value {{ var T }} }}
   } where { s before t, {s,t} within 1 min, var T > 40 }
END

DETECT   burnt_down { sensor { var S } }
ON and { event n: temp {{ sensor {{ var S }}, value {{ var T }} }},
         event i: timer:from-end [ event n, 12 sec ],
         while i: not temp {{ sensor {{ var S }} }}
   } where { var T > 40 }
END

DETECT   avg_temp { sensor { var S }, value { avg(all var T) } }
ON and { event m: temp {{ sensor {{ var S }} }},
         event i: timer:from-start-backward [ event m, 1 min ],
         while i: collect temp {{ sensor {{ var S }}, value {{ var T }} }} }
END
```

Fig. 1 XChange$^{\text{EQ}}$ program *P*

3 CERA: An Operational Semantics for Event Query Languages

This section is devoted to the formal definition of the operational semantics for EQLs. In Section 3.1 we clarify the purpose of an operational semantics for query languages in general and describe its desiderata for an EQL in particular. Complex Event Relational Algebra (CERA) builds the first corner stone of the operational semantics. CERA bases on relational algebra and extends it with operators which are able to treat notions specific for events such as event occurrence time. In Section 3.2 we define the operators formally and illustrate their usage by translating the XChange$^{\text{EQ}}$ rules in Figure 1 into CERA expressions. Section 3.3 is devoted to the formal specification of the translation of a single XChange$^{\text{EQ}}$ rule into a CERA expression. But keep in mind that CERA is independent from a particular EQL. Finally, Section 3.4 explains how query plans consisting of CERA expressions can be optimized and incrementally evaluated (the second corner stone of the operational semantics).

3.1 Purpose and Desiderata

In general, the purpose of an operational semantics is to provide an abstract description of an implementation of the evaluation engine of a language. For EQLs in particular, an operational semantics must fulfill the following core desiderata:

1. It should be an incremental, data-driven evaluation method storing and updating intermediate results instead of computing them anew in each step. The notion “in-

cremental” derives from the idea that in each step we compute only the changes relative to the previous steps.
2. Since the incoming event stream is unbounded, a naive query evaluation engine storing all intermediate results forever, runs out of memory sooner or later. Hence an operational semantics must enable garbage collection of irrelevant events, i.e., events which cannot contribute to a new answer any more. For the sake of brevity, we do not formalize garbage collection of events in this chapter and refer the reader to [7].
3. An operational semantics should provide a framework for query optimization since it must be able to capture the whole space of different query plans. In this chapter we only provide a general idea of event query optimization and refer the reader to [6, 4, 10, 18] for more details.
4. An operational semantics for EQLs must be correct with respect to the declarative semantics of EQLs and it must terminate for each incremental step.

3.2 CERA: Complex Event Relational Algebra

For the operational semantics we will restrict ourselves to hierarchical rule programs, i.e., programs that are free of any recursion of rules (see [11] for the formal definition). Note that this is a common restriction not just for event queries but also database views and queries, and causes no problems in most practical applications.

We want to base CERA on traditional relational algebra. There are three problems arising when doing so: the treatment of XML data carried by events, the incorporation of the time axis and the infinity of streams. We approach these problems in the following way: Streams are regarded from an omniscient perspective, i.e. as if all events ever arriving on the stream were already known. From this point of view the (complete infinite) stream is just a relation with tuples representing the events arriving on the stream. Each tuple representing an event consists of two timestamp attributes representing the begin and end of the occurrence interval of the event and a data-term attribute representing the XML data carried by the event. Of course this relation is potentially infinite and never known completely at any point of time. However this can be ignored for now. We will see below that due to a special property of CERA called “temporal preservation” (Section 3.4.1) and the “finite differencing” technique (Section 3.4.3), the evaluation is nevertheless able to incrementally compute the desired result working only on the finite part of the relation known up to some point of time.

All three points, the integration of XML-data, the explicit representation of the time axis by means of timestamp attributes and the way to cope with the infinity of streams, are fundamentally different to previous approaches. The chapter “A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed”, in this volume, shows that composition operator based languages, data stream languages and production rule languages have none or only weak support for XML. These language groups also lack an explicit representation of time (though time plays a role of course) limiting the temporal relations expressible in these approaches. Data stream languages like CQL also intend to use a kind of relational algebra for their semantics however they approach the infinity of streams by (conceptually) *converting* parts of the stream into a finite relation for every point of time by, for example, time windows. After that, relational algebra is applied and the resulting relation is then converted back into a stream using another operator. (We refer the reader to the chapter "A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed", in this volume, for more details). This significantly differs from our approach as we do not do any conversion at all but just view the complete potentially infinite stream as a relation and directly apply relational algebra to that relation.

With regards to the XML integration we are not finished yet at this point as we need a possibility to access the data contained in the data-term attribute of an event (or its tuple respectively), for example, for selections or joins. Therefore we introduce the matching operator Q^{X} which extracts the desired data into attributes of the resulting relation. The complementary operator to Q^{X} is the construction operator C^{X} which is used to construct the new data-terms for the derived complex events from a number of attributes of a relation. It will be explained more closely on Page 10. In this way applying a complete CERA expression, i.e., an expression with Q^{X} at the bottom and C^{X} at the top, to an event stream, or more precisely to the relation representing the stream, results in a relation with the same schema, or an event stream, again. Thus CERA is answer closed. In the following we will mainly describe the effect of Q^{X} and C^{X} to the schema of a relation as XML matching and construction are not in the focus here. For details on the exact semantics of the XML matching in Q^{X} and XML construction in C^{X} see [11].

The **event matching** operator Q^{X} takes two arguments, an event stream (i.e. the corresponding relation) of course and a simple event query *event* $i:q$. The result of applying Q^{X} to the event stream E using the simple event query *event* $i:q$ is the relation $\mathsf{R_i} = \mathsf{Q}^{\mathsf{X}}_{[\ i:\ q\ ]}(E)$. Each event (or tuple) in the stream (or relation) matched by q corresponds to one or many tuples in $\mathsf{R_i}$ (see the definition of Q^{X} in [11]).

The schema $sch(\mathsf{R_i})$ of $\mathsf{R_i}$ corresponds directly to the free variables of q. Each free variable of q is a data attribute of $\mathsf{R_i}$. Furthermore $\mathsf{R_i}$ contains an event reference attribute $i.ref$ identifying the event a tuple was derived from, and two timestamp attributes $i.begin, i.end$ representing the occurrence time interval of this event. Consequently, $\mathsf{R_i}$ has the schema $sch(\mathsf{R_i}) = \{i.begin, i.end, i.ref, X_1, \ldots, X_n\}$, where $X_1, \ldots, X_n$ are the free variables of q. We denote the set of data attributes of $\mathsf{R_i}$ with $sch_{data}(\mathsf{R_i})$, the set of timestamp attributes with $sch_{time}(\mathsf{R_i})$ and the set of event reference attributes with $sch_{ref}(\mathsf{R_i})$.

Note that we use the named perspective on relations here, i.e., tuples are viewed as functions that map attribute names to values. This is more intuitive than the unnamed perspective identifying attribute values by their positions in ordered tuples.

XChange$^{\mathrm{EQ}}$ program P in Figure 1 gives rise to the relations in Figure 2. These relations will be the input for the CERA expressions into which we will translate the rules of P.

$$\begin{array}{lll}
\mathsf{Smoke_s} = \mathsf{Q}^{\mathsf{X}}_{[\,s\text{: } \mathit{smoke}\{\{\mathit{area}\{\{\mathit{var}\ A\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Smoke_s}) & = \{s.\mathit{begin}, s.\mathit{end}, \\
& & \quad s.\mathit{ref}, A\} \\
\mathsf{Temp_t} = \mathsf{Q}^{\mathsf{X}}_{[\,t\text{: } \mathit{temp}\{\{\mathit{area}\{\{\mathit{var}\ A\}\},\mathit{value}\{\{\mathit{var}\ T\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Temp_t}) & = \{t.\mathit{begin}, t.\mathit{end}, \\
& & \quad t.\mathit{ref}, A, T\} \\
\mathsf{Temp_n} = \mathsf{Q}^{\mathsf{X}}_{[\,n\text{: } \mathit{temp}\{\{\mathit{sensor}\{\{\mathit{var}\ S\}\},\mathit{value}\{\{\mathit{var}\ T\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Temp_n}) & = \{n.\mathit{begin}, n.\mathit{end}, \\
& & \quad n.\mathit{ref}, S, T\} \\
\mathsf{Temp_v} = \mathsf{Q}^{\mathsf{X}}_{[\,v\text{: } \mathit{temp}\{\{\mathit{sensor}\{\{\mathit{var}\ S\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Temp_v}) & = \{v.\mathit{begin}, v.\mathit{end}, \\
& & \quad v.\mathit{ref}, S\} \\
\mathsf{Temp_m} = \mathsf{Q}^{\mathsf{X}}_{[\,m\text{: } \mathit{temp}\{\{\mathit{sensor}\{\{\mathit{var}\ S\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Temp_m}) & = \{m.\mathit{begin}, m.\mathit{end}, \\
& & \quad m.\mathit{ref}, S\} \\
\mathsf{Temp_w} = \mathsf{Q}^{\mathsf{X}}_{[\,w\text{: } \mathit{temp}\{\{\mathit{sensor}\{\{\mathit{var}\ S\}\},\mathit{value}\{\{\mathit{var}\ T\}\}\}\}\,]}(\mathsf{E}), & \mathit{sch}(\mathsf{Temp_w}) & = \{w.\mathit{begin}, w.\mathit{end}, \\
& & \quad w.\mathit{ref}, S, T\}
\end{array}$$

Fig. 2 Input relations for the CERA expressions for the program *P* in Figure 1

Beside the relations shown in Figure 2, the program *P* makes use of **relative timer events**. Figure 3 contains generic definitions of the relative timer events used in *P*. Relative timer events are expressed by means of auxiliary event streams S and auxiliary relations X. The definitions take an event stream E and a relative timer specification as parameters. Each auxiliary event stream S contains one relative timer event s for each event r of the event stream E. The timestamps of s are defined relatively to the timestamps of r. The matching operator Q^{X} sets the value of attribute $j.\mathit{ref}$ of X to a reference to event r (denoted $\mathit{ref}(r)$). We need the attribute $j.\mathit{ref}$ to join X with another relation to drop superfluous tuples of X.

$$\begin{array}{ll}
\mathsf{X}_{[\text{ i: from-end[j,d] }]}(\mathsf{E}) & := \mathsf{Q}^{\mathsf{X}}_{[\text{ i: rel-timer-event[var j.ref] }]}(\mathsf{S}_{[\text{ from-end[d] }]}(\mathsf{E})), \text{ where} \\
\mathsf{S}_{[\text{ from-end[d] }]}(\mathsf{E}) & := \{s \mid \exists\, r \in \mathsf{E} \text{ with } s := \begin{cases} s(\mathit{begin}) & = r(\mathit{end}), \\ s(\mathit{end}) & = r(\mathit{end}) + d, \\ s(\mathit{term}) & = \text{rel-timer-event}(\mathit{ref}(r)) \end{cases} \quad \} \\
\mathsf{X}_{[\text{ i: from-start-backward[j,d] }]}(\mathsf{E}) & := \mathsf{Q}^{\mathsf{X}}_{[\text{ i: rel-timer-event[var j.ref] }]}(\mathsf{S}_{[\text{ from-start-backward[d] }]}(\mathsf{E})), \text{ where} \\
\mathsf{S}_{[\text{ from-start-backward[d] }]}(\mathsf{E}) & := \{s \mid \exists\, r \in \mathsf{E} \text{ with } s := \begin{cases} s(\mathit{begin}) & = r(\mathit{begin}) - d, \\ s(\mathit{end}) & = r(\mathit{begin}), \\ s(\mathit{term}) & = \text{rel-timer-event}(\mathit{ref}(r)) \end{cases} \quad \}
\end{array}$$

Fig. 3 Generic definitions of the relative timer events used in the program *P* in Figure 1

Beside event matching Q^{X}, CERA allows the following operators: natural join $\bowtie$, selection σ, projection π, temporal join $\bowtie_{i \sqsupseteq j}$, temporal anti-semi join $\overline{\ltimes}_{i \sqsupseteq j}$, merging of time intervals μ, renaming ρ, and event construction C^{X}. The definitions of natural join, selection and projection are just the same as in traditional relational algebra. There is only one important limitation of projection. It is only allowed to discard data attributes (e.g., X), but it is not allowed to discard time attributes (e.g., $i.begin$) and event references (e.g., $i.ref$).

The translation of while/collect in XChange$^{\text{EQ}}$ is expressed by temporal join. Temporal join is a new operator of CERA, it does not exist in traditional relational algebra and cannot be expressed as a combination of operators of traditional relational algebra. Let R and S be relations. In a temporal join $R \bowtie_\theta S$, the condition θ has the form $i.begin \leq j.begin \wedge j.end \leq i.end$, where $\{i.begin, i.end\} \subseteq sch_{time}(R)$ and $\{j.begin, j.end\} \subseteq sch_{time}(S)$. We abbreviate these conditions with $i \sqsupseteq j$. To achieve the right implicit groupng by event references (see the description of the event construction operator C^{X} below), the reference $j.ref$ must be dropped. This is possible because the temporal restriction $i \sqsupseteq j$ guarantees that the groups stay finite after $j.ref$ is dropped.

Definition 0.1 (Temporal join). Let R and S be relations such that $\{i.begin, i.end\} \subseteq sch_{time}(R)$, $i.ref \in sch_{ref}(R)$, $\{j.begin, j.end\} \subseteq sch_{time}(S)$, and $j.ref \in sch_{ref}(S)$.

$$\begin{aligned}
R \bowtie_{i \sqsupseteq j} S = \{o \mid\ & \exists r \in R\ \exists s \in S \text{ such that} \\
& r(X) = s(X) \text{ if } X \in sch(R) \cap Sch, \\
& r(i.begin) \leq s(j.begin) \text{ and } s(j.end) \leq r(i.end), \\
& o(X) = r(X) \text{ if } X \in sch(R) \text{ and } o(X) = s(X) \text{ if } X \in Sch, \\
& o(X) = \perp \text{ otherwise } \},
\end{aligned}$$

$$\text{where } Sch := sch(S) \setminus \{j.begin, j.end, j.ref\},$$

$$sch(R \bowtie_{i \sqsupseteq j} S) = sch(R) \cup Sch.$$

In order to express negation of an event in CERA we introduce a θ-anti-semi-join that uses the θ condition to define the event accumulation window (just analogously to the above definition of temporal join). The temporal anti-semi-join $R \overline{\ltimes}_{i \sqsupseteq j} S$ takes two relations R and S as input, where $\{i.begin, i.end\} \subseteq sch_{time}(R)$ and $\{j.begin, j.end\} = sch_{time}(S)$. (Note that it is "$\subseteq$" for the timestamps of the left side of the anti-semi-join and "$=$" for timestamps on the right side!) Its output is R with those tuples r removed that have a "partner" is S, i.e., a tuple $s \in S$ that agrees on all shared attributes with r and whose timestamps $s(j.begin)$, $s(j.end)$ are within the time bounds $r(i.begin)$, $r(i.end)$.

Definition 0.2 (Temporal anti-semi-join). Let R and S be relations with $\{i.begin, i.end\} \subseteq sch_{time}(R)$ and $\{j.begin, j.end\} = sch_{time}(S)$.

$$\begin{aligned}
R \overline{\ltimes}_{i \sqsupseteq j} S = \{r \in R \mid \nexists s \in S \text{ such that } & \forall X \in sch(R) \cap sch(S).\ r(X) = s(X) \\
& \text{and } r(i.begin) \leq s(j.begin),\ s(j.end) \leq r(i.end)\},
\end{aligned}$$

$$sch(R \overline{\ltimes}_{i \sqsupseteq j} S) = sch(R).$$

In contrast to temporal join and temporal anti-semi-join, natural join maintains the time attributes of both input relations in order to ensure temporal preservation, an important property of CERA operators allowing them to work on finite available parts of streams.[1] But as the reader will see in Section 3.4.1, to guarantee temporal presevation, CERA operators must not maintain all timestamps of an input event but only the *greatest* timestamp of each input event. To reduce the number of timestamps of an event, merging operator is used. It computes a single time interval of an event out of many time intervals it carries. Let R be a relation. A merging operator $\mu[[begin, end] \leftarrow j_1 \sqcup \cdots \sqcup j_n](R)$ computes a new time interval $[begin, end]$ from existing ones $j_1, \ldots, j_n$ so that the time interval $[begin, end]$ covers all the intervals $j_1, \ldots, j_n$, i.e., $begin = min\{j_1.begin,\ \ldots,\ j_n.begin\}$ and $end = max\{j_1.end, \ldots, j_n.end\}$. Further, the merging operator extracts the beginning $begin$ and the end end of the new time interval as well as all attributes of R except of $j_1.begin, j_1.end, ..., j_n.begin, j_n.end$ in the manner of a projection. Merging of time intervals is not really a new operation in CERA. It is equivalent to an extended projection [14], a common practical extension of relational algebra used to compute new attributes from existing ones.

Definition 0.3 (Merging of time intervals). Let R be a relation and $j_1, \ldots, j_n$ time intervals such that $\{j_1.begin, j_1.end, \ldots, j_n.begin, j_n.end\} \subseteq sch_{time}(R)$. Merging of $j_1, \ldots, j_n$ is a relation defined as follows:

$$\begin{aligned}
&\mu[[begin, end] \leftarrow j_1 \sqcup \cdots \sqcup j_n](R) = \{t \mid \exists r \in R \text{ with}\\
&\quad t(begin) = \min\{r(j_1.begin), \ldots, r(j_n.begin)\},\\
&\quad t(end) = \max\{r(j_1.end), \ldots, r(j_n.end)\},\\
&\quad t(X) = r(X) \text{ if } X \in sch(R) \setminus \{begin, end, j_1.begin, j_1.end, \ldots, j_n.begin, j_n.end\},\\
&\quad t(X) = \bot \text{ otherwise}\},
\end{aligned}$$

$$\begin{aligned}
&sch(\mu[[begin, end] \leftarrow j_1 \sqcup \cdots \sqcup j_n](R)) =\\
&\quad \{begin, end\} \cup (sch(R) \setminus \{j_1.begin, j_1.end, \ldots, j_n.begin, j_n.end\}).
\end{aligned}$$

For technical reasons (i.e., usage of natural join), the named perspective of relational algebra sometimes requires a renaming operator, which changes the names of attributes without affecting their values. Renaming is denoted by $\rho[a'_1 \leftarrow a_1, \ldots, a'_n \leftarrow a_n](R)$. It renames attributes $a_1, \ldots, a_n$ of the relation R to $a'_1, \ldots, a'_n$ respectively. Note that if an attribute a_i is a data attribute of R it must be a data attribute in the resulting relation, if it is a timestamp, then a'_i must also be a time attribute, and if it is an event reference so also a'_i. Note also that timestamps must always occur pairwise; accordingly, they can only be renamed pairwise.

Definition 0.4 (Renaming). Let R be a relation such that $\{a_1, \ldots, a_n\} \subseteq sch(R)$ and $\{a'_1,\ \ldots, a'_n\} \cap sch(R) = \emptyset$.

$$\begin{aligned}
\rho[a'_1 \leftarrow a_1, \ldots, a'_n \leftarrow a_n](R) = \{t \mid \exists r \in R \text{ such that } & t(a'_i) = r(a_i) \text{ and}\\
& \forall X \notin \{a_1, \ldots, a_n\}.\ t(X) = r(X)\},
\end{aligned}$$

[1] Temporal restriction of temporal join and temporal anti-semi-join ansures temporal preservation.

$$\begin{aligned}\forall i \in \{1,\dots,n\}.\ & a_i \subseteq sch_{time}(R) \ \text{ iff } a'_i \subseteq sch_{time}\ (\rho[a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n](R)),\\ & a_i \subseteq sch_{ref}(R) \ \text{ iff } a'_i \subseteq sch_{ref}\ (\rho[a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n](R)),\\ & a_i \subseteq sch_{data}(R) \ \text{ iff } a'_i \subseteq sch_{data}\ (\rho[a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n](R)),\end{aligned}$$

$$\begin{aligned}& j.s \leftarrow i.s \in \{a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n\} \text{ iff } j.e \leftarrow i.e \in \{a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n\},\\ & \text{where } \{i.s,i.e\} \subseteq sch_{time}(R) \text{ and } \{j.s,j.e\} \subseteq sch_{time}(\rho[a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n](R)),\end{aligned}$$

$$sch(\rho[a'_1 \leftarrow a_1,\dots,a'_n \leftarrow a_n](R)) = (sch(R) \setminus \{a_1,\dots,a_n\}) \cup \{a'_1,\dots,a'_n\})$$

We will see the necessity of the operator in Section 3.4.

Finally, the **event construction** operator C^{X} serves for the construction of the data-terms carried by the derived complex events. The output schema of C^{X} is the same as the input schema of Q^{X}, i.e. the output relation of C^{X} can be regarded as a stream of derived events. The operator takes two arguments, a relation R with $sch_{time}(R) = \{begin, end\}$ and a rule head h. The result of applying C^{X} to the relation R using the rule head h is the stream (or relation) $E' = \mathsf{C}^{\mathsf{X}}_{[h]}(R)$. One event (or tuple) in E' corresponds to one or many tuples in R depending on whether h contains grouping and aggregation constructs or not.

If h does not contain grouping and aggregation constructs like `all`, then C^{X} constructs for each tuple $r \in R$ one event represented as the data term annotated with the time interval $[r(begin), r(end)]$. The data term results from substituting each free variable X of h by $r(X)$, i.e., by the value of attribute X of the tuple r.

$$\begin{aligned}&\mathsf{C}^{\mathsf{X}}_{[\ fire\{area\{var\ A\}\}\ ]}(\\ &\quad \mu[[begin,end] \leftarrow s \sqcup t](\\ &\quad\quad \sigma[\max\{s.end,t.end\} - \min\{s.begin,t.begin\} \leq 1\ min](\\ &\quad\quad\quad \sigma[s.end < t.begin](\\ &\quad\quad\quad\quad \sigma[T > 40](\\ &\quad\quad\quad\quad\quad \mathsf{Smoke}_{\mathsf{s}} \bowtie \mathsf{Temp}_{\mathsf{t}})))))\end{aligned}$$

Fig. 4 CERA expression for the first rule of the program *P* in Figure 1

For example, the first rule of the program *P* in Figure 1 corresponds to the CERA expression in Figure 4. Remember that the query triggers fire alarm for an area when smoke and high temperature are both detected in the area within one minute. Note that all temporal conditions (such as *s before t* and $\{s,t\}$ *within 1 min*) of the query have been turned into selections. Because temporal information is simply data in tuples (as *s.begin, s.end*, etc.), no special temporal operators are needed as part of the algebra; e.g., there is no need for a sequence operator as found in many event algebras.

The second rule of the program *P* corresponds to the CERA expression in Figure 5. Recall that the rule infers that a sensor had burnt down if it reported high temperature and did not send its measurements afterwards any more. For this rule we assume that all temperature sensors send their measurements every 12 seconds.

$$\begin{array}{l}
\mathsf{C}^{\mathsf{X}}_{[\ \mathit{burnt\_down}\{\mathit{sensor}\{\mathit{var}\ S\}\}\ ]}(\\
\quad \mu[[\mathit{begin},\mathit{end}] \leftarrow n \sqcup i \sqcup v](\\
\quad\quad \sigma[T > 40](\\
\quad\quad\quad (\mathsf{Temp_n} \ltimes \mathsf{X}_{[\text{ i: from-end[n,12 sec] }]}(\mathsf{E}))\ \overline{\ltimes}_{i \sqsupseteq v}\ \mathsf{Temp_v})))
\end{array}$$

Fig. 5 CERA expression for the second rule of the program *P* in Figure 1

If the rule head *h* contains grouping and aggregation constructs, the operator C^{X} does the required grouping of tuples and computes the values of aggregation functions. Note that explicit grouping happens after an implicit grouping of the tuples of *R* by event references and time attributes. (Since time attributes are functionally dependent on event references, they do not have any effect on the result of grouping but they must be part of each resulting tuple to guarantee temporal preservation.) As temporal joins are restricted to finite time intervals and projections may not discard event references, after the implicit grouping by event references and time attributes, each group is finite (but there may be of course infinitely many groups because the stream is potentially infinite). Therefore grouping initiated by *h* does not need any treatment specific for CEP.

$$\begin{array}{l}
\mathsf{C}^{\mathsf{X}}_{[\ \mathit{avg\_temp}\{\mathit{sensor}\{\mathit{var}\ S\},\mathit{value}\{\mathit{avg}(\mathit{all}\ \mathit{var}\ T)\}\}\ ]}(\\
\quad \mu[[\mathit{begin},\mathit{end}] \leftarrow m \sqcup i \sqcup w](\\
\quad\quad (\mathsf{Temp_m} \ltimes \mathsf{X}_{[\text{ i: from-start-backward[m,1 min] }]}(\mathsf{E}))\ \ltimes_{i \sqsupseteq w}\ \mathsf{Temp_w}))
\end{array}$$

Fig. 6 CERA expression for the third rule of the program *P* in Figure 1

Consider the last rule of *P* in Figure 1 corresponding to the CERA expression in Figure 6. Recall that the rule computes the average temperature reported by a sensor during the last minute every time the sensor sends a temperature measurement. C^{X} takes the tuples of the joined input relations, groups them (first implicitly by the event references and time attributes and then) according to the value of attribute *S* (denoting a sensor). For each group the average value of attribute *T* (denoting a temperature measurement) is computed and saved as the value of data attribute *value* of the resulting event.

3.3 Translation into CERA

We now turn to the formal specification of the translation of a single XChange$^{\mathrm{EQ}}$ rule into a CERA expression. The rules are first normalized, which means that `or` is eliminated and the literals in the rule body are ordered in a specific way. Figure 7 shows the general structure of a normalized XChange$^{\text{EQ}}$ rule.[2] Note that all rules of the program in Figure 1 are normalized.

$$
\begin{array}{l}
\texttt{DETECT} \\
\quad h \\
\texttt{ON} \\
\quad \texttt{and}\{\underbrace{b_1,\ldots,b_{i_s}}_{\substack{\text{Simple}\\\text{Event}\\\text{Queries}}},\underbrace{b_{i_s+1},\ldots,b_{i_t}}_{\substack{\text{Relative}\\\text{Timer}\\\text{Spec}}},\underbrace{b_{i_t+1},\ldots,b_{i_a}}_{\substack{\text{Accumulation}\\\text{for}\\\text{Collection}}},\underbrace{b_{i_a+1},\ldots,b_n}_{\substack{\text{Accumulation}\\\text{for}\\\text{Negation}}}\}\ \texttt{where}\{c_1,\ldots,c_k\} \\
\texttt{END}
\end{array}
$$

Fig. 7 Normalized XChange$^{\text{EQ}}$ rule

Figure 8 shows the translation of a single normalized XChange$^{\text{EQ}}$ rule. The translation of rule sets requires additionally the notion of query plans which will be introduced in Section 3.4.2.

$$
\begin{array}{rcl}
b_1 & \mapsto & B_1 \\
b_1,\ldots,b_{i+1} & \mapsto & B_{i+1} \\
\texttt{and}\{b_1,\ldots,b_n\} & \mapsto & B_n \\
\texttt{and}\{b_1,\ldots,b_n\}\ \texttt{where}\{c_1,\ldots,c_k\} & \mapsto & C \\
\texttt{DETECT}\ h\ \texttt{ON}\ \texttt{and}\{b_1,\ldots,b_n\}\ \texttt{where}\{c_1,\ldots,c_k\}\ \texttt{END} & \mapsto & Q
\end{array}
$$

$$B_1 := \mathsf{Q}^{\mathsf{X}}_{[\,j:\,q\,]} \qquad \text{for } b_1 = \texttt{event}\, j : q$$

$$
B_{i+1} := \begin{cases}
B_i \bowtie \mathsf{Q}^{\mathsf{X}}_{[\,j:\,q\,]} & \text{if } b_{i+1} = \texttt{event}\, j : q \\
B_i \bowtie X_{[\,j:\,\mathit{REL\text{-}TIMER\text{-}SPEC}[\,j',d\,]\,]} & \text{if } b_{i+1} = \texttt{event}\, j : \mathit{REL\text{-}TIMER\text{-}SPEC}[j',d] \\
B_i \bowtie_{j \sqsupseteq i'} \mathsf{Q}^{\mathsf{X}}_{[\,i':\,q\,]} & \text{if } b_{i+1} = \texttt{while}\, j : q \\
B_i \overline{\ltimes}_{j \sqsupseteq i'} \mathsf{Q}^{\mathsf{X}}_{[\,i':\,q\,]} & \text{if } b_{i+1} = \texttt{while}\, j : \texttt{not}\ q
\end{cases}
$$

where $1 \le i < n$ and i' is a fresh event identifier

$C := \sigma[c'_1 \wedge \cdots \wedge c'_q](B_n)$ where c'_i is the translation of c_i (see Figure 13.4 in [11] for details)

$Q := \mathsf{C}^{\mathsf{X}}_h(\mu[[\mathit{begin},\mathit{end}] \leftarrow j_1 \sqcup \cdots \sqcup j_l](C))$

Fig. 8 Translation of a single normalized XChange$^{\text{EQ}}$ rule into a CERA expression

[2] Note that the normalization of a single rule usually yields a set of rules not a single rule due to the elmination of `or`.

3.4 Incremental Evaluation

Till now we pretended to have a kind of "omniscience": The relations contain conceptually all events that ever happen and are probably infinite for that reason. In the actual evaluation of event queries it is not possible to foresee future events. Event queries are evaluated *incrementally* on finite event histories. This section explains the details on incremental evaluation of programs in an EQL. We start with temporal preservation of CERA, a property allowing incremental evaluation of CERA expressions (Section 3.4.1). Then, we introduce the notion of query plans with materialization points (Section 3.4.2) and explain their incremental evaluation (Section 3.4.3).

3.4.1 Temporal Preservation

The restrictions that CERA imposes on expressions (compared to an unrestricted relational algebra), make this approach reasonable since we do not need any knowledge about future events when we want to obtain all results of an expression with an occurrence time until *now*. More precisely, to compute all results of a CERA expression Q with an occurrence time before or at time point *now*, we need to know (the finite part of) its input relations up to this time point *now*. In order to formally define and prove this property of CERA, called temporal preservation, we need the following auxiliary definition.

Definition 0.5 (Occurrence time of a tuple). The occurrence time of a tuple r in the result of a CERA expression Q is a time interval given by

$$\begin{aligned} occtime(r) = [\ &\min\{r(i.begin) \mid i.ref \in sch_{ref}(\mathsf{Q})\}, \\ &\max\{r(i.end) \ \ \mid i.ref \in sch_{ref}(\mathsf{Q})\}\ ] \end{aligned}$$

To refer to the end of the occurrence time of a tuple, i.e., $end(occtime(r))$ in selections we introduce the shorthand $\mathsf{END_Q} := max\{\ i_1.end,\ \dots,\ i_n.end\ \}$ where $\{\ i_1.end, \dots, i_n.end\ \} \cup \{\ i_1.begin, \dots, i_n.begin\ \} = sch_{time}(\mathsf{Q})$.[3]

Theorem 0.1 (Temporal preservation). *Let Q be a CERA-expression with input relations $R_1, \dots, R_n$. Then for all time points now : $\sigma[END_Q \leq now](Q) = Q'$, where Q' is obtained from Q by replacing each R_k with $R'_k := \{r \in R_k \mid end(occtime(r)) \leq now\}$.*

Proof (Sketch). By induction. For event matching, event construction, selection, and projection, the claim is obvious since timestamps are not changed at all. By definition, natural join maintains the timestamps of both input relations without change. By definition, merging does not change the maximum value over all timestamps. Temporal join and temporal anti-semi-join are only allowed with temporal restrictions that also ensure that the maximum value is maintained. We refer to [11] for details.

[3] Note that $\mathsf{END_Q}$ is a syntactical expression which can be used in selections whereas $end(occtime(r))$ denotes a mathematical function on the semantic level.

3.4.2 Query Plans with Materialization Points

In traditional relational databases, a query plan describes the order in which operators are applied to base relations to compute answers for queries and serves as a bases for query optimization techniques such as “push selection” and storage and reuse of shared subqueries. In CEP, queries are evaluated continuously over time against changing event data received as an incoming stream and therefore a query plan should additionally account for storage of intermediate results of CERA expressions to avoid their re-computation in later evaluation steps. To this end, query plans with so-called materialization points are introduced in this section. A materialization point is a relation which saves and updates the results of a CERA expression instead of computing them anew in each evaluation step.

Definition 0.6 (Query plan with materialization points). A query plan is a sequence $QP = \langle M_1 := \mathsf{Q}_1, \ldots, M_n := \mathsf{Q}_n \rangle$ of materialization point definitions $M_i := \mathsf{Q}_i$. M_i is called a materialization point. Q_i is either a basic stream,[4] a CERA (sub-)expression or a union $R_1 \cup \ldots \cup R_n$ of materialization points. Each materialization point M_i is defined only once in QP, i.e., $M_i \neq M_j$ for all $1 \leq i < j \leq n$. The materialization point definitions must be acyclic, i.e., if M_j occurs in Q_i then $j < i$ for all $1 \leq i \leq n$ and all $1 \leq j \leq n$.

Since a query plan is acyclic, its semantics is straightforward: compute the results of its expressions from left right, replacing references to materialization points with their (already computed) result.

$$\begin{array}{l}
\mathsf{Fire_f} := \mathsf{C}^{\mathsf{X}}_{[\,fire\{area\{var\ A\}\}\,]}(\\
\qquad \mu[[begin, end] \leftarrow s \sqcup t](\\
\qquad\quad \sigma[\max\{s.end, t.end\} - \min\{s.begin, t.begin\} \leq 1\ min](\\
\qquad\qquad \sigma[s.end < t.begin](\\
\qquad\qquad\quad \sigma[T > 40](\\
\qquad\qquad\qquad \mathsf{Smoke_s} \bowtie \mathsf{Temp_t}))))) \\
\\
\mathsf{Burbt\_down_b} := \mathsf{C}^{\mathsf{X}}_{[\,burnt\_down\{sensor\{var\ S\}\}\,]}(\\
\qquad\qquad \mu[[begin, end] \leftarrow n \sqcup i \sqcup v](\\
\qquad\qquad\quad \sigma[T > 40](\\
\qquad\qquad\qquad (\mathsf{Temp_n} \bowtie \mathsf{X}_{[\text{ i: from-end[n,12 sec] }]}(\mathsf{E}))\ \overline{\ltimes}_{i \sqsupseteq v}\ \mathsf{Temp_v}))) \\
\\
\mathsf{Avg\_temp_a} := \mathsf{C}^{\mathsf{X}}_{[\,avg\_temp\{sendor\{var\ S\}, value\{avg(all\ var\ T)\}\}\,]}(\\
\qquad\qquad \mu[[begin, end] \leftarrow m \sqcup i \sqcup w](\\
\qquad\qquad\quad (\mathsf{Temp_m} \bowtie \mathsf{X}_{[\text{ i: from-start-backward[m,1 min] }]}(\mathsf{E})) \bowtie_{i \sqsupseteq w} \mathsf{Temp_w}))
\end{array}$$

Fig. 9 Query plan for the program *P* in Figure 1

Figure 9 shows a query plan for the program *P*. The plan can be significantly improved. First, it does not account for the materialization of shared subqueries.

[4] When translating an XChange$^{\text{EQ}}$ program, the basic streams are the incoming event stream E and the auxiliary streams for relative timer events.

Relations $\mathsf{Temp_n}$ in the second expression and $\mathsf{Temp_w}$ in the third expression are equal except for the names of time attributes and event references, i.e., $\mathsf{Temp_w} = \rho[w.begin \leftarrow n.begin, w.end \leftarrow n.end, w.ref \leftarrow n.ref](\mathsf{Temp_n})$. (Compare their simple event queries in Figure 2.) But the same tuples of the relations are computed and saved twice. To avoid this, we introduce a new relation $\mathsf{Temp_{n'}}$ (where n' is a fresh event identifier), save all the respective tuples only once in it, and use this relation in both expressions.

The same holds for the relations $\mathsf{Temp_v}$ in the second expression and $\mathsf{Temp_m}$ in the third expression, i.e., $\mathsf{Temp_v} = \rho[v.begin \leftarrow m.begin, v.end \leftarrow m.end, v.ref \leftarrow m.ref](\mathsf{Temp_m})$. We use the relation $\mathsf{Temp_{m'}}$ (where m' is a fresh event identifier) in both expressions.

Second, selections should be as near to their respective relations as possible to reduce the number of tuples which must be further considered (e.g., joined with tuples of other relations). This optimization technique, usually called "push selection", is adopted from the traditional relational algebra. To this end we apply the selection $\sigma[T > 40]$ to the relation $\mathsf{Temp_t}$ before $\mathsf{Temp_t}$ is joined with the relation $\mathsf{Smoke_s}$ in the first expression of the query plan. We analogously modify the second expression of the query plan.

Third, intermediate results of a query should be materialized to avoid their recomputation in later evaluation steps. For example, in the first query, if a new event arrives and is saved in the relation $\mathsf{Smoke_s}$ we have to compute $\sigma[T > 40](\mathsf{Temp_t})$ anew in order to join it with the changed relation $\mathsf{Smoke_s}$. To avoid this recomputation we define a new materialization point $\mathsf{A_t} := \sigma[T > 40](\mathsf{Temp_t})$ and join it with the relation $\mathsf{Smoke_s}$. To avoid re-computations in the other expressions we introduce the materialization points $\mathsf{B_{n',i}}, \mathsf{C_{n'}}$, and $\mathsf{D_{m',i}}$. Consider Figure 10 for the improved query plan.

So far it might seem that this is just an insignificant change in notation. However, it will become clear in the next section that only those intermediate results are "materialized", i.e., remembered across individual evaluation steps, that have a materialization point. Therefore the query plans in Figures 9 and 10 are different in terms of incremental evaluation, although of course both yield the same results for the program *P*. Note that the efficiency of a query plan depends on characteristics of its event streams and there is no general principle to tell which one is more efficient.

3.4.3 Finite Differencing

Evaluation of an event query program, or rather its query plan *QP*, over time is a step-wise procedure. A step is initiated by some base event (an event which is not derived by a rule) happening at the current time, which we denote *now*. Then for each materialization point *M* in *QP*, the required output for this step is the set of all computed answers (tuples *r* representing materialized intermediate results and derived events) that "happen" at this current time *now*, i.e., where $end(occtime(r)) = now$.

$$\begin{aligned}
&\mathsf{Fire_f} := \mathsf{C}^{\mathsf{X}}_{[\ fire\{area\{var\ A\}\}\ ]}(\\
&\quad \mu[[begin,end] \leftarrow s \sqcup t](\\
&\quad\quad \sigma[\max\{s.end, t.end\} - \min\{s.begin, t.begin\} \leq 1\ min](\\
&\quad\quad\quad \sigma[s.end < t.begin](\\
&\quad\quad\quad\quad \mathsf{Smoke_s} \bowtie \mathsf{A_t})))), \text{ where} \\
&\\
&\quad\quad\quad \mathsf{A_t} := \sigma[T > 40](\mathsf{Temp_t}) \\
&\\
&\mathsf{Burnt\_down_b} := \mathsf{C}^{\mathsf{X}}_{[\ burnt\_down\{sensor\{var\ S\}\}\ ]}(\\
&\quad\quad \mu[[begin,end] \leftarrow n' \sqcup i \sqcup m'](\\
&\quad\quad\quad \mathsf{B_{n',i}} \ \overline{\ltimes}_{i \sqsupseteq m'} \ \mathsf{Temp_{m'}})), \text{ where} \\
&\\
&\quad\quad\quad \mathsf{B_{n',i}} := \mathsf{C_{n'}} \bowtie \mathsf{X}_{[\text{ i: from-end[n',12 sec] }]}(\mathsf{E}), \text{ where} \\
&\\
&\quad\quad\quad \mathsf{C_{n'}} := \sigma[T > 40](\mathsf{Temp_{n'}}) \\
&\\
&\mathsf{Avg\_temp_a} := \mathsf{C}^{\mathsf{X}}_{[\ avg\_temp\{sendor\{var\ S\}, value\{avg(all\ var\ T)\}\}\ ]}(\\
&\quad\quad \mu[[begin,end] \leftarrow m' \sqcup i \sqcup n'](\\
&\quad\quad\quad \mathsf{D_{m',i}} \bowtie_{i \sqsupseteq n'} \mathsf{Temp_{n'}})), \text{ where} \\
&\\
&\quad\quad\quad \mathsf{D_{m',i}} := \mathsf{Temp_{m'}} \bowtie \mathsf{X}_{[\text{ i: from-start-backward[m',1 min] }]}(\mathsf{E}),
\end{aligned}$$

Fig. 10 Improved query plan for the program P in Figure 1

In other words in each step, we are not interested in the full result of M, but only in $\triangle M := \sigma[\mathsf{END_Q} = now](M)$.[5]

A naive, non-incremental way of query evaluation would be: Maintain a stored version of each base event relation across steps. In each step simply insert the new event into its base relation and evaluate the query plan from scratch according to its non-incremental semantics (previous section). Then apply the selection $\sigma[\mathsf{END_Q} = now]$ to each materialization point to output the result of the step. This is, however, inefficient since we compute not only the required result $\triangle M = \sigma[\mathsf{END_Q} = now](M)$, but also all results from previous steps, i.e., also $\sigma[\mathsf{END_Q} < now](M)$.

It is more efficient to use an incremental approach, where we (1) store not only base relations but also some intermediate results, namely those of each materialization point M across steps and then (2) in each step only compute the changes of M that result from the step. It turns out that due to the temporal preservation of CERA (see Theorem 0.1), the change to each M involves only inserting new tuples into M and that these tuples are exactly the ones from $\triangle M$.

We can compute $\triangle M$ efficiently using the changes $\triangle R_i$ of the input relations R_i of $M := \mathsf{Q}$, together with $\circ R_i = \sigma[\mathsf{END_Q} < now](R_i)$, their materialized states from

[5] We assume for simplicity here that the base events are processed in the temporal order in which they happen, i.e., with ascending ending timestamps. Extensions where the order of events is "scrambled" (within a known bound) are possible, however. Note that while the time domain can be continuous (e.g., isomorphic to the real numbers), the number of evaluation steps is discrete since we assume a discrete number of incoming events.

the previous evaluation steps.[6] Using finite differencing, we can derive a CERA-expression $\triangle\mathsf{Q}$ so that $\triangle\mathsf{Q}$ involves only $\triangle R_i$ and $\circ R_i$ and $\triangle\mathsf{Q} = \triangle M$ (for each step). Finite differencing pushes the differencing operator $\triangle$ inwards according to the equations in Figure 11.

$$\begin{array}{ll@{\qquad}ll}
\triangle\mathsf{Q}^{\mathsf{X}}(\mathsf{Q}) & = \mathsf{Q}^{\mathsf{X}}(\triangle\mathsf{Q}) & \circ\mathsf{Q}^{\mathsf{X}}(\mathsf{Q}) & = \mathsf{Q}^{\mathsf{X}}(\circ\mathsf{Q}) \\
\triangle\mathsf{C}^{\mathsf{X}}(\mathsf{Q}) & = \mathsf{C}^{\mathsf{X}}(\triangle\mathsf{Q}) & \circ\mathsf{C}^{\mathsf{X}}(\mathsf{Q}) & = \mathsf{C}^{\mathsf{X}}(\circ\mathsf{Q}) \\
\triangle\sigma_C(\mathsf{Q}) & = \sigma_C(\triangle\mathsf{Q}) & \circ\sigma_C(\mathsf{Q}) & = \sigma_C(\circ\mathsf{Q}) \\
\triangle\rho_A(\mathsf{Q}) & = \rho_A(\triangle\mathsf{Q}) & \circ\rho_A(\mathsf{Q}) & = \rho_A(\circ\mathsf{Q}) \\
\triangle\pi_P(\mathsf{Q}) & = \pi_P(\triangle\mathsf{Q}) & \circ\pi_P(\mathsf{Q}) & = \pi_P(\circ\mathsf{Q}) \\
\triangle\mu_M(\mathsf{Q}) & = \mu_M(\triangle\mathsf{Q}) & \circ\mu_M(\mathsf{Q}) & = \mu_M(\circ\mathsf{Q}) \\
\triangle(\mathsf{Q}_1 \Join \mathsf{Q}_2) & = \triangle\mathsf{Q}_1 \Join \circ\mathsf{Q}_2 \cup \triangle\mathsf{Q}_1 \Join \triangle\mathsf{Q}_2 \cup & \circ(\mathsf{Q}_1 \Join \mathsf{Q}_2) & = \circ\mathsf{Q}_1 \Join \circ\mathsf{Q}_2 \\
 & \quad \cup \circ\mathsf{Q}_1 \Join \triangle\mathsf{Q}_2 & & \\
\triangle(\mathsf{Q}_1 \Join_{i \sqsupseteq j} \mathsf{Q}_2) & = \triangle\mathsf{Q}_1 \Join_{i \sqsupseteq j} \circ\mathsf{Q}_2 \cup \triangle\mathsf{Q}_1 \Join_{i \sqsupseteq j} \triangle\mathsf{Q}_2 & \circ(\mathsf{Q}_1 \Join_{i \sqsupseteq j} \mathsf{Q}_2) & = \circ\mathsf{Q}_1 \Join_{i \sqsupseteq j} \circ\mathsf{Q}_2 \\
\triangle(\mathsf{Q}_1 \overline{\ltimes}_{i \sqsupseteq j} \mathsf{Q}_2) & = \triangle\mathsf{Q}_1 \overline{\ltimes}_{i \sqsupseteq j} \circ\mathsf{Q}_2 \cup \triangle\mathsf{Q}_1 \overline{\ltimes}_{i \sqsupseteq j} \triangle\mathsf{Q}_2 & \circ(\mathsf{Q}_1 \Join_{i \sqsupseteq j} \mathsf{Q}_2) & = \circ\mathsf{Q}_1 \Join_{i \sqsupseteq j} \circ\mathsf{Q}_2
\end{array}$$

Fig. 11 Equations for finite differencing

Finite differencing is a method originating in the incremental maintenance of materialized views in databases, which is a problem very similar to incremental event query evaluation. We refer the reader to [11, 7] for more information on incremental evaluation and garbage collection enabled by CERA.

4 A Declarative Semantics for Event Query Languages

Section 4.1 explains the purpose and necessity of a declarative semantics for a programming language in general and its desiderata for an EQL in particular. Section 4.2 is devoted to the formal definition of the declarative semantics of XChange$^{\mathrm{EQ}}$ with a model-theoretic approach in order to prove the correctness of its operational semantics (Section 5).

4.1 Purpose, Necessity and Desiderata

In general, a declarative semantics relates the syntax of a language to mathematical objects and expressions that capture the intended meaning. In other words, a declarative semantics focuses on expressing *what* a sentence in the language means, rather

[6] Note that some previous results can become irrelevant in later evaluation steps, i.e., they cannot contribute to new answers any more. Therefore they should be deleted to speed up the later evaluation steps. See [7] for the formal definition of garbage collection enabled by CERA.

than *how* that sentence might be evaluated (which is the purpose of an operational semantics).

A declarative semantics thus provides a convenient basis to prove the correctness of various operational semantics. In particular in the area of query languages there are usually a myriad of equivalent ways to evaluate a given query, that is, of possible operational semantics. If, on the other hand, the formal semantics of a language were specified only in an operational way, proving the correctness of other operational semantics would be significantly harder: since an operational semantics focuses on how the result is computed not on what is the result, we have to reason about the equivalence of two computations. When we prove correctness of an operational semantics with respect to a declarative semantics, we instead just reason about properties of the output of one computation. This use of a declarative semantics to prove correctness of evaluation methods is particularly useful in research on optimization.

A declarative semantics have often been neglected in EQLs so far. Our goal is a declarative semantics that is natural on event streams, i.e., does not require a conversion from streams to relations and back, like SQL-based EQLs do [3, 12, 16], and as declarative as possible and thus avoids any notion of state.

Because of these reasons we specify the declarative semantics by a Tarski-style model theory with accompanying fixpoint theory in Section 4.2. This approach has another important advantage, namely it accounts well for data in events and rule chaining, two aspects that have often been neglected in the semantics of EQLs till now.

4.2 Model Theory and Fixpoint Theory

While the model-theoretic approach is well-established for traditional, non-event query and rule languages, its application to EQLs is novel and we highlight the extensions that are necessary in this section. We also show that our declarative semantics is suitable for querying events that arrive over time in unbounded event streams and illustrate this statement by the declarative semantics of the XChange$^{\text{EQ}}$ program P in Figure 1.

The idea of a model theory, as it is used in traditional, non-event query languages [15, 1, 17], is to relate expressions to an *interpretation* by defining an *entailment relation*. Expressions are syntactic fragments of the query language such as rules, queries, or facts viewed as logic sentences. The interpretation contains all facts that are considered to be true. The entailment relation indicates whether a given interpretation entails a given sentence in the language, that is, if the sentence is logically true under this interpretation. For the semantics of a given query program and a set of base facts are those interpretations of interest that (1) satisfy all rules of the program and (2) contain all base facts. Because it satisfies all rules, such an interpretation particularly contains all facts that are derived by rules. We call these interpretations *models*.

When we replace facts that are true with events that happen, this approach can also be applied to EQLs. The problem, of course, is that events are associated with *occurrence times* and event queries are evaluated *over time* against a potentially *infinite event stream*. At each time point during the evaluation we know only which events have happened (i.e., been received in the event stream) so far, not any events that might happen in the future. We start of with some basic definitions that explain how we represent time and events in the semantics of XChange$^{\text{EQ}}$.

Time is represented by a linearly ordered **set of time points** $(\mathbb{T},<)$. The **set of time intervals** is $\mathbb{TI} = \{t = [begin, end] \mid begin \in \mathbb{T}, end \in \mathbb{T}, begin \leq end\}$. For an interval t, $begin(t)$ denotes its beginning and $end(t)$ its end, i.e., $t = [begin(t), end(t)]$. We omit t in the notation if the interval is clear from the context, i.e., we write *begin* and *end* instead of $begin(t)$ and $end(t)$ respectively.

The **set of data terms** is denoted *DataTerms* (see [17, 13] for the full grammar of Xcerpt data terms). Recall that data terms are used to represent data and type information for events. An event is a tuple of a time interval t and a data term e, written e^t. The **set of events** is denoted $Events$; $Events = DataTerms \times \mathbb{TI}$. Let $E \subseteq Events$ denote an **event stream** and $EventIdentifiers$ the **set of event identifiers**.

To explain how simple event queries are matched against incoming events and how events derived by rules are constructed, we have to explain some concepts of the Web query language Xcerpt, whose query and construct terms are used in XChange$^{\text{EQ}}$. We try to keep these explanations brief and refer the reader to [9, 17] for details.

An **Xcerpt query term** is a pattern that accesses data to extract relevant portions of it. An **Xcerpt construct term** is a pattern that constructs new data. Consider the first rule in Figure 1. *fire*$\{$*area*$\{$*var A*$\}\}$ is a construct term, *smoke*$\{\{$*area*$\{\{$*var A*$\}\}\}\}$ and *temp*$\{\{$ *area*$\{\{$*var A*$\}\}$,*value*$\{\{$*var T*$\}\}\}\}$ are query terms. Since query and construct terms of a rule contain variables (A and T in this case) that are bound to values during the application of the rule, we need the concept of substitution.

Let *Vars* denote the **set of variable names**. A **substitution** σ is a partial mapping from variable names to data terms, i.e., $\sigma : Vars \to DataTerms$. We write substitutions as $\sigma = \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$, meaning that $\sigma(X_i) = v_i$ for $i \in \{1, \ldots, n\}$ and $\sigma(Y) = \perp$ for $Y \notin \{X_1, \ldots, X_n\}$.

The **application of a substitution** σ to a query term q replaces the occurrences of variables V in q with their values $\sigma(V)$. The result is denoted $\sigma(q)$. If $\sigma(q)$ is a **ground term**, i.e., a term without variables, we call σ a **grounding substitution** of q.

Simple event queries in XChange$^{\text{EQ}}$ are Xcerpt query terms that are matched against data terms e of incoming events e^t. This matching of simple event queries is based on **simulation** between ground terms as defined for Xcerpt [9, 17]. Intuitively, a ground query term q simulates into a data term d, denoted $q \preceq d$, if the nodes and the structure of the graph that q represents, can be found in the graph of d. This simulation relationship of Xcerpt is especially designed for the variations and incompleteness in semi-structured data.

A non-ground query term q' simulates into a data term d, $q' \preceq d$, if there is a grounding substitution σ such that $\sigma(q') \preceq d$. Note that for a given non-ground

$I,\sigma,\tau \models q^t$	iff exists $e^{t'}$ with $e^{t'} \in I$, $t' = t$ and $\sigma(q) \preceq e$
$I,\sigma,\tau \models (\texttt{event}\ i : q)^t$	iff exists $e^{t'}$ with $\tau(i) = e^{t'}$, $e^{t'} \in I$, $t' = t$ and $\sigma(q) \preceq e$
$M \models (q_1 \wedge q_2)^t$	iff exist t_1 and t_2 with $t = t_1 \sqcup t_2$ and $M \models q_1^{t_1}$ and $M \models q_2^{t_2}$
$M \models (q_1 \vee q_2)^t$	iff $M \models q_1^t$ or $M \models q_2^t$
$I,\sigma,\tau \models (Q\ \texttt{where}\ C)^t$	iff $I,\sigma,\tau \models Q^t$ and $W_{\sigma,\tau}(C) = true$
$I,\sigma,\tau \models (\texttt{while}\ j:\ \texttt{not}\ q)^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $t' = t$, and not exist $t'' \sqsubseteq t$ such that $I,\sigma,\tau \models q^{t''}$
$I,\sigma,\tau \models (\texttt{while}\ j:\ \texttt{collect}\ q)^t$	iff exists $e^{t'}$ with $\tau(j) = e^{t'}$, $t' = t$, and exist σ and $t'' \sqsubseteq t$ such that $I,\sigma,\tau \models q^{t''}$
$I,\sigma,\tau \models (h \leftarrow B)^t$	iff for all τ with $\Sigma_\tau := \{\sigma \mid I,\sigma,\tau \models B^t\}$ $\Sigma_\tau = \emptyset$ or $\Sigma_\tau(h) \subseteq I$

Fig. 12 Entailment relation defining the model theory for XChange$^{\text{EQ}}$

query term q' and a given data term d, there are often several substitutions that allow a simulation between the two. We denote the **substitution set** of q' and d by $\Sigma := \{\sigma \mid \sigma(q') \preceq d\}$.

An XChange$^{\text{EQ}}$ rule head contains an Xcerpt construct term h for constructing new, derived events. This construction uses the substitution set Σ obtained from the evaluation of the query in the respective rule body to replace variables with values. Application of Σ to h, defined in [17], returns a set of data terms representing derived events.

Now we can define interpretation and entailment, which are the core of the model theory of XChange$^{\text{EQ}}$.

Definition 0.7 (Interpretation). An interpretation for a given XChange$^{\text{EQ}}$ query, rule, or program is a 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 rule.
2. σ is a grounding substitution for (data) variables.
3. $\tau : EventIdentifiers \rightarrow Events$ is a substitution for event identifiers.

The substitution τ for event identifiers is, compared to model theories of traditional, non-EQLs, unusual. It is needed for evaluating temporal conditions and relative timer events. Since τ signifies the events that contribute to the answer of some query, we also call it an "event trace."

The **entailment** (or satisfaction) $M \models F^t$ of an XChange$^{\text{EQ}}$ expression F over a time interval t in an interpretation M is defined in Figures 12 and 13. (We require the programs to be range restricted [8].)

Figure 12 defines the more salient cases of the model theory. For the sake of brevity, the expressions in this figure use binary "and" with symbol $\wedge$ and binary

$I, \sigma, \tau \models (\texttt{event}\ i : \textit{from-end}[j,d])^t$ iff exists $e^{t'}$ with $\tau(j) = e^{t'}$,
$\tau(i) = \text{rel-timer-event}(e,t')^t$,
$\mathit{begin}(t) = \mathit{end}(t')$, $\mathit{end}(t) = \mathit{end}(t') + d$

$I, \sigma, \tau \models (\texttt{event}\ i : \textit{from-start-backward}[j,d])^t$ iff exists $e^{t'}$ with $\tau(j) = e^{t'}$,
$\tau(i) = \text{rel-timer-event}(e,t')^t$,
$\mathit{begin}(t) = \mathit{begin}(t') - d$, $\mathit{end}(t) = \mathit{begin}(t')$

Fig. 13 Entailment of the relative timer events in XChange$^{\text{EQ}}$ used in Figure 1

$W_{\sigma,\tau}(i\ \mathit{before}\ j) = \mathit{true}$ iff $\mathit{end}(\tau(i)) < \mathit{begin}(\tau(j))$

$W_{\sigma,\tau}(\{i_1,\dots,i_n\}\ \mathit{within}\ d) = \mathit{true}$ iff $E - B \leq d$ with $E := \max\{\mathit{end}(\tau(i_1)),\dots,\mathit{end}(\tau(i_n))\}$
and $B := \min\{\mathit{begin}(\tau(i_1)),\dots,\mathit{begin}(\tau(i_n))\}$.

Fig. 14 Fixed interpretation for conditions in the *where* clause used in Figure 1

"or" with symbol $\vee$ instead of the multi-ary *and*$\{\dots\}$ and *or*$\{\dots\}$. Also, rules are written as $h \leftarrow B$ instead of *DETECT h ON B END*.

Figure 13 defines entailment of the relative timer events used in the program in Figure 1. (See [11] for the complete version of the figure.) For the sake of brevity, the prefix "*timer:*" and the keyword "*event*" within the relative timer specification have been skipped.

Our entailment relation uses a fixed interpretation W for all conditions that can occur in the *where*-clause of a query. This includes the temporal relations like *before* as well as conditions on data such as arithmetic comparisons. This fixed interpretation of the temporal conditions is another feature of our model theory that is not common in model theories for traditional, non-EQLs.

W is a function that maps a substitution σ, an event trace τ, and an atomic condition C to a Boolean value (true or false). We usually write σ and τ in the index of W. $W_{\sigma,\tau}$ extends straightforwardly to Boolean formulas of conditions. Figure 14 gives the definitions of W for the temporal conditions of XChange$^{\text{EQ}}$ that have been used in Figure 1. (See [11] for the complete version of the figure.) The definition of W is deliberately left outside the "core model theory" to make it more modular and demonstrate that it is easy to integrate further conditions or even a separate, external temporal reasoner.

Recall our primary goal in specifying declarative semantics for XChange$^{\text{EQ}}$: given an XChange$^{\text{EQ}}$ program P and an event stream E, we want to find out all events that are derived by the rules of P. This means that we must find an interpretation that contains the event stream E and satisfies all rules of P. Such an interpretation is called a model.

Definition 0.8 (Model). Given an XChange$^{\text{EQ}}$ program P and a stream of incoming events E, we call an interpretation $M = (I, \sigma, \tau)$ a model of P under E if

- M satisfies all rules $r = (h \leftarrow B) \in P$ for all time intervals t, i.e., $M \models r^t$ for all $t \in \mathbb{TI}$ and all $r = (h \leftarrow B) \in P$, and
- M contains the stream of incoming events, i.e., $E \subseteq I$.

On close inspection of the entailment relation, we can see that σ and τ are actually irrelevant to whether a given interpretation M is a model or not; it depends only on I. We therefore can identify the notion of a model with just the I part of an interpretation, $M = I$.

Consider the XChange$^{\text{EQ}}$ program P in Figure 1 and the event stream

$$\begin{aligned} E = \{\ & \mathit{temp}\ \{\ \mathit{area}\{a\},\ \mathit{sensor}\{s\},\ \mathit{value}\{40\}\ \}^{[60,63]}, \\ & \mathit{smoke}\ \{\ \mathit{area}\ \{a\}\ \}^{[65,68]}, \\ & \mathit{temp}\ \{\ \mathit{area}\{a\},\ \mathit{sensor}\{s\},\ \mathit{value}\{41\}\ \}^{[70,80]}\ \ \}. \end{aligned}$$

We assume timestamps to be time intervals. The bounds of the intervals denote minutes since the beginning of the current 10-minutes-long-window. In this case an event with timestamp [0,0] happens 10 minutes before an event with timestamp [599,599]. The life-time of all events is restricted to the window they happened within. These assumptions are not suitable for real-life applications but they help to keep the example simple.

The interpretation

$$\begin{array}{ll} M_1 = \{ & \\ \mathit{temp}\{\mathit{area}\{a\},\mathit{sensor}\{s\},\mathit{value}\{40\}\}^{[60,63]}, & \mathit{fire}\{\mathit{area}\{a\}\}^{[65,80]}, \\ \mathit{smoke}\{\mathit{area}\{a\}\}^{[65,68]}, & \mathit{burnt\_down}\{\mathit{sensor}\{s\}\}\}^{[70,92]}, \\ \mathit{temp}\{\mathit{area}\{a\},\mathit{sensor}\{s\},\mathit{value}\{41\}\}^{[70,80]}, & \mathit{avg\_temp}\{\mathit{sensor}\{s\},\mathit{value}\{40\}\}^{[10,80]} \\ \} & \end{array}$$

is a model for P under E: by applying the recursive definition of $\models$ we can check that $M_1 \models r^t$ for all $t \in \mathbb{TI}$, $r \in P$, and we also have $E \subseteq M_1$. Note that each rule of P derives exactly one complex event of M_1 and the timestamp of a complex event comprises the timestamps of all events this complex event was derived from. Note also that the second temperature measurement does not fall into the time window for aggregation of the last rule of P. That is why the avarage temperature is 40, not 40,5.

The interpretation

$$\begin{array}{ll} M_2 = \{ & \\ \mathit{temp}\{\mathit{area}\{a\},\mathit{sensor}\{s\},\mathit{value}\{40\}\}^{[60,63]}, & \mathit{fire}\{\mathit{area}\{a\}\}^{[65,80]}, \\ \mathit{smoke}\{\mathit{area}\{a\}\}^{[65,68]}, & \mathit{burnt\_down}\{\mathit{sensor}\{s\}\}\}^{[70,92]}, \\ \mathit{temp}\{\mathit{area}\{a\},\mathit{sensor}\{s\},\mathit{value}\{41\}\}^{[70,80]}, & \mathit{avg\_temp}\{\mathit{sensor}\{s\},\mathit{value}\{40\}\}^{[10,80]}, \\ \mathit{temp}\{\mathit{area}\{b\},\mathit{sensor}\{t\},\mathit{value}\{20\}\}^{[1,2]} & \\ \} & \end{array}$$

where we "added" the event $temp\{area\{b\}, sensor\{t\}, value\{20\}\}^{[1,2]}$ in comparison to M_1, is also a model of P under E. Clearly, however, M_2 is not the model we intend for our program to have, because the "additional" event is "unjustified." More precisely, this event is neither in the event stream E nor derived by a rule of P. M_1 is the intended model, because all events in it are justified.

To unambiguously settle on a single, intended model, we will use the fixpoint theory which builds upon the model theory. Note that the problem of specifying the intended model out of the (infinitely) many possible models is also a common part of the traditional model-theoretic approach. It is not specific for EQLs.

The intended model is the (least) fixpoint of the immediate consequence operator, which derives new events from known events (based on the model theory). Non-monotonic features such as negation and aggregation introduce well-known issues when they are combined with recursion of rules. In particular, there might be no fixpoint or several. To ensure that a single fixpoint exists, we restrict XChange$^{\text{EQ}}$ 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).

Restriction to stratifiable programs is a common approach from logic programming introduced first in [2]. But in contrast to logic programming, in CEP three types of **stratification** are required:

1. *Negation stratification*: Events that are matched by a negative simple event query of a rule r (e.g., *not temp*$\{\{$*sensor*$\{\{$*var S*$\}\}\}\}$ of the second rule in Figure 1) may only be constructed by rules in lower strata than the stratum of r. Events which are matched by a positive simple event query of a rule r may be constructed by rules in lower strata or the same stratum as that of r.
2. *Grouping stratification*: Rules r with grouping constructs like *all* in the construction may only query events constructed by rules in lower strata than the stratum of r. Therefore the last rule in Figure 1 must be in a higher stratum then all rules the head of which is matched by *temp*$\{\{$*sensor*$\{\{$*var S*$\}\}$*, value* $\{\{$*var T*$\}\}$ $\}\}$.
3. *Temporal stratification*: If a rule r defines a relative timer event, e.g., *timer: from-end [event n, 12 sec]* in the second rule in Figure 1, then the anchoring event (here: n) may only be constructed by rules in lower strata than the stratum of r.

While negation and grouping stratification are fairly standard, temporal stratification is a requirement specific to complex event query programs like those expressible in XChange$^{\text{EQ}}$. We are not aware of former consideration of the notion of temporal stratification. See [11] for the formal definitions of the notions.

The basic idea for obtaining the fixpoint interpretation of a stratifiable XChange$^{\text{EQ}}$ program is to apply the rules stratum by stratum: first apply the rules in the lowest stratum to the incoming event stream, then apply the rules in the next higher stratum to the result, and so on until the highest stratum. This requires the definition of the immediate consequence operator T_P for an XChange$^{\text{EQ}}$ program.

Definition 0.9 (Immediate consequence operator). The immediate consequence operator T_P for an XChange$^{\text{EQ}}$ program is defined as:

$$T_P(I) = I \cup \{e^t \mid \text{there exist a rule } h \leftarrow B \in P, \text{ and } \tau \text{ such that } e \in \Sigma_\tau(h) \text{ where } \Sigma_\tau := \{\sigma \mid I, \sigma, \tau \models B^t\} \}$$

The operator is obviously monotonic [8]. Hence according to Knaster-Tarski theorem, it has a fixpoint [8]. The repeated application of T_P until a fixpoint is reached is denoted T_P^ω. A fixpoint means here an interpretation I such that $T_P(I) = I$.

Definition 0.10 (Fixpoint interpretation). Let $\overline{P_i} = \bigcup_{j \leq j} P_j$ denote the set of all rules in strata P_i and lower. The fixpoint interpretation $M_{P,E}$ of an XChange$^{\text{EQ}}$ program P with stratification $P = P_1 \uplus \cdots \uplus P_n$ under event stream E is defined by computing fixpoints stratum by stratum:

$$\begin{aligned} M_0 &= E = T_\emptyset^\omega(E), \\ M_1 &= T_{\overline{P_1}}^\omega(M_0), \\ &\ldots, \\ M_{P,E} = M_n &= T_{\overline{P_n}}^\omega(M_{n-1}). \end{aligned}$$

The fixpoint interpretation $M_{P,E}$ is also called the intended model of P under E and specifies the declarative semantics.

Consider, for example, the XChange$^{\text{EQ}}$ program P in Figure 1 and the event stream E above.

$$\begin{aligned} M_0 = E = \quad & \{\ \mathit{temp}\ \{\ \mathit{area}\{a\},\ \mathit{sensor}\{s\},\ \mathit{value}\{40\}\ ^{[60,63]}, \\ & \quad \mathit{smoke}\ \{\ \mathit{area}\ \{a\}\ \}\ ^{[65,68]}, \\ & \quad \mathit{temp}\ \{\ \mathit{area}\{a\},\ \mathit{sensor}\{s\},\ \mathit{value}\{41\}\ \}\ ^{[70,80]}\ \} = T_\emptyset^\omega(E) \\ M_{P,E} = M_1 = M_0 \cup & \{\ \mathit{fire}\ \{\ \mathit{area}\ \{a\}\ \}\ ^{[65,80]}, \\ & \quad \mathit{burnt\_down}\ \{\ \mathit{sensor}\ \{s\}\ \}\ \}\ ^{[70,92]}, \\ & \quad \mathit{avg\_temp}\ \{\ \mathit{sensor}\ \{s\},\ \mathit{value}\{40\}\ \}\ ^{[10,80]}\ \} = T_{\overline{P_1}}^\omega(M_0) \end{aligned}$$

In addition to giving unambiguous semantics to stratifiable XChange$^{\text{EQ}}$ programs, the fixpoint theory also describes an abstract, simple, forward-chaining evaluation method, which can easily be extended to work incrementally as it is required for event queries.

5 Two Semantics, no Double Talk

We now want to show that semantics of XChange$^{\text{EQ}}$, the declarative one from Section 4.2 and the operational semantics given by the translation to CERA in Section 3.3, are equivalent. In other words we now want to show the correctness of the translation of a normalized rule $h \leftarrow B$.[7] We only give the main ideas here, for details see [11].

We consider only hierarchical programs with a single rule.[7] Recall that the declarative semantics for a program $\{h \leftarrow B\}$ with a single rule $h \leftarrow B$ is given by $M_{\{h \leftarrow B\},E}$. Therefore we have

$$M_{\{h \leftarrow B\},E} = T^{\omega}_{\{h \leftarrow B\}}(E) = T_{\{h \leftarrow B\}}(E)$$

meaning that the semantics is given by applying the fixpoint operator to the program $\{h \leftarrow B\}$ and the event stream E once. That is, the declarative semantics of a single rule $h \leftarrow B$ is given by:

$$T_{\{h \leftarrow B\}}(E) = E \cup \{e^t \mid \exists \tau \text{ such that } e \in \Sigma_\tau(h) \text{ where } \Sigma_\tau := \{\sigma \mid I, \sigma, \tau \models B^t\}\}$$

Let Q be the CERA expression that translates the rule $h \leftarrow B$. We identify events e^t with tuples r of $Q(E)$ and E by $r(begin) = begin(t)$, $r(end) = end(t)$, $r(term) = e$. In the other direction we identify tuples r with events $r(term)^{[r(begin),r(end)]}$. With this identification correctness of the translation means that

$$Q(E) \cup E = T_{\{h \leftarrow B\}}(E).$$

For this it suffices to show that

$$Q(E) = \{e^t \mid \exists \tau \text{ such that } e \in \Sigma_\tau(h) \text{ where } \Sigma_\tau := \{\sigma \mid I, \sigma, \tau \models B^t\}\} \qquad (*)$$

Next, we have to find a correspondence between the elements r of the relations $S(E)$ generated by subexpressions S of Q, i.e., $B_1, \ldots, B_n, C$ defined in Section 3.3, and the combined σ, τ used by the declarative semantics in Section 4.2. In other words we want to have a corresponding tuple $r_{\sigma,\tau}$ for each pair σ, τ and a corresponding pair σ_r, τ_r for each tuple r. Figure 15 shows the correspondence.

We use this correspondence to show a lemma about each subexpression S of Q that translates a subexpression F of the rule body B (see Figure 8).

Lemma 0.1 (Equivalence of subexpressions). *Let F be the subexpression translated by S and be $t = occtime(r)$. Then following equivalence holds:*

$$r \in S(E) \quad \Leftrightarrow \quad E, \sigma_r, \tau_r \models F^t$$

The left to right part of Lemma 0.1 is soundness ("results produced by the operational semantics are results according to the declarative semantics"). The right to left part is completeness ("results according to the declarative semantics are actually produced by the operational semantics").

Consider the definitions of B_n and C in Figure 8. The proof of Lemma 0.1 is done by induction on n over B_n, and then by additionally showing that Lemma 0.1 also holds for C.

[7] Note that the proof for a hierarchical program with a single rule immediately applies to arbitrary hierarchical programs. The reason is that there exits a topological ordering of the rules for hierarchical programs. Applying the proof for a single rule to the rules of the hierarchical program in topological order yields a proof for arbitrary hierarchical programs.

$$r_{\sigma,\tau} := \begin{cases} r(X) &= \sigma(X) \quad \text{for all } X \in sch_{data}(S) \\ r(j.ref) &= ref(\tau(j)) \quad \text{for all } i.ref \in sch_{ref}(S) \\ r(j.begin) &= begin(\tau(j)) \quad \text{for all } i.begin \in sch_{time}(S) \\ r(j.end) &= end(\tau(j)) \quad \text{for all } i.end \in sch_{time}(S) \\ r(X) &= \bot \quad \text{otherwise} \end{cases}$$

$$\sigma_r := \begin{cases} \sigma_r(X) = r(X) & \text{for all } X \in sch_{data}(S) \\ \sigma_r(X) = X & \text{otherwise} \end{cases}$$

$$\tau_r := \begin{cases} \tau_r(i) = s_i(term)^{[s_i(begin), s_i(begin)]} & \text{for all } i.ref \in sch_{ref}(S) \text{ , where } s_i := ref^{-1}(i.ref) \\ \tau_r(i) = \bot & \text{otherwise} \end{cases}$$

Fig. 15 Correspondence between tuples and substitutions

The proof of (*) is done applying Lemma 0.1 to rule body B as used in (*) and the translation C of B as defined in Figure 8. The details of the proof are given in [11].

6 Conclusion and Outlook

In this chapter, we have formally defined the operational and declarative semantics for XChange$^{\text{EQ}}$, illustrated them on the sensor network use case and proved their equivalence. Both semantics are generic and easily transferable to an arbitrary EQL. On the bases of the approach described here, some other points essential for CEP can be immediately implemented. Two of them, garbage collection and query optimization, are addressed in this section.

As mentioned above, evaluation of complex event queries over time involves storing events in materialization points. Naive query evaluation simply stores all events forever. Since the event stream is not bounded into the future every naive query evaluation engine can run out of memory sooner or later. Therefore there is a need for garbage collection of irrelevant events, i.e., events which cannot contribute to the derivation of new events (any more). We refer the reader to [7] for an approach on static determination of temporal relevance for incremental evaluation of complex event queries. Temporal relevance is particularly suitable for garbage collection because one of the main principles of a reasonable CEP engine is that no rule must wait for an event forever.

The so-called general relevance of events including temporal, causal, structural relevance as well as relevance with regards to event data, addresses query optimization. In addition to the consideration of general relevance, automatic query optimization is possible by means of application specific knowledge formalized as a model. With the help of the model one can, e.g., recognize the unsatisfiability of a (sub-) query in order to suspend or delete it and to avoid the storage of irrelevant events.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
3. A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.
4. S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29(3):545–580, 2004.
5. F. Bry and M. Eckert. Rule-Based Composite Event Queries: The Language XChange$^{\mathrm{EQ}}$ and its Semantics. In *Proc. Int. Conf. on Web Reasoning and Rule Systems*, volume 4524 of *LNCS*, pages 16–30. Springer, 2007.
6. F. Bry and M. Eckert. Temporal order optimizations of incremental joins for composite event detection. In *Proc. Int. Conf. on Distributed Event-Based Systems*. ACM, 2007.
7. F. Bry and M. Eckert. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proc. Int. Conf. on Distributed Event-Based Systems*, pages 289–300. ACM, 2008.
8. 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, Third Int. Summer School 2007*, volume 4636 of *LNCS*, pages 1–153. Springer, 2007.
9. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proc. Int. Conf. on Logic Programming, Copenhagen, Denmark (29th July–1st August 2002)*, volume 2401 of *LNCS*. Springer, 2002.
10. L. Ding, S. Chen, E. A. Rundensteiner, J. Tatemura, W.-P. Hsiung, and K. S. Candan. Runtime semantic query optimization for event stream processing. In *Proc. Int. Conf. on Data Engineering*, pages 676–685. IEEE Computer Society, 2008.
11. M. Eckert. *Complex Event Processing with XChange^{EQ}: Language Design, Formal Semantics and Incremental Evaluation for Querying Events*. PhD thesis, Institute for Informatics, University of Munich, 2008.
12. EsperTech Inc. Event stream intelligence: Esper & NEsper. `http://esper.codehaus.org`.
13. T. Furche. *Implementation of Web Query Languages Reconsidered: Beyond Tree and Single-Language Algebras at (Almost) No Cost*. PhD thesis, Institute for Informatics, University of Munich, 2008.
14. H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.
15. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1993.
16. J. Morrell and S. D. Vidich. Complex Event Processing with Coral8. White Paper. `http://www.coral8.com/system/files/assets/pdf/Complex_Event_Processing_with_Coral8.pdf`, 2007.
17. S. Schaffert. *Xcerpt: A Rule-Based Query and Transformation Language for the Web*. PhD thesis, Institute for Informatics, University of Munich, 2004.
18. P. A. Tucker, D. Maier, T. Sheard, and P. Stephens. Using production schemas to characterize strategies for querying over data streams. *IEEE Transactions on Knowledge and Data Engineering*, 19(9):1227–1240, 2007.

Index