Bitemporal Complex Event Processing of Web Event Advertisements*

Tim Furche[1], Giovanni Grasso[1], Michael Huemer[2], Christian Schallhart[1], and Michael Schrefl[2]

[1] Department of Computer Science, Oxford University, Wolfson Building, Parks Road, Oxford OX1 3QD
firstname.lastname@cs.ox.ac.uk

[2] Department of Business Informatics – Data & Knowledge Engineering, Johannes Kepler University, Altenberger Str. 69, Linz, Austria
lastname@dke.uni-linz.ac.at

Abstract. The web is the largest bulletin board of the world. Events of all types, from flight arrivals to business meetings, are announced on this board. Tracking and reacting to such event announcements, however, is a tedious manual task, only slightly alleviated by email or similar notifications. Announcements are published with human readers in mind, and updates or delayed announcements are frequent. These characteristics have hampered attempts at automatic tracking.
PEACE provides the first integrated framework for event processing on top of web event ads. Given a schema of events to be tracked, the framework populates this schema through compact wrappers for event announcement sources. These wrappers produce events including updates and retractions. PEACE then queries these events to detect complex events, often combining announcements from multiple sources. To deal with updates and delayed announcements, PEACE's schemas are bitemporal so as to distinguish between occurrence and detection time. This allows complex event specifications to track updates and to react to differences in occurrence and detection time. Our evaluation shows that extracting the event from an announcement dominates the processing of PEACE and that the complex event processor deals with several event announcement sources even with moderate resources. We further show, that simple restrictions on the complex event specifications suffice to guarantee that PEACE only requires a constant buffer to process arbitrarily many event announcements.

1 Introduction

Most events are announced first and often only on the web these days. This trend is even more pronounced for time critical events, as the web is a ubiquitous and prompt information source. While the immediate availability of up-to-date information is a blessing

* The research leading to these results has received funding from the European Research Council under the European Community's Seventh Framework Programme (FP7/2007–2013) / ERC grant agreement DIADEM, no. 246858. Michael Huemer has been supported by a Marietta Blau Scholarship granted by the Austrian Federal Ministry of Science and Research (BMWF) for a research stay at Oxford University's Department of Computer Science.

in enabling much more complex, rapid interactions, it also imposes a challenge: The immediacy of web-published events allows for frequently and quickly distributed updates, leading to fragmentary and preliminary but inaccurate advertisements which are fixed later. Therefore, modern coordination tasks often boil down to continuously checking all relevant event advertisement sources for changes, ready to react on violations of our constraints. For example, awaiting a person on a flight with a stopover, we need to check that both flights are in time. If the second flight is delayed, the person will arrive late the same day; in contrast, if the first flight is late, the person might not even arrive on the same day, depending on the timeliness of the second flight and other available connections. But these distinctions do not suffice, as incomplete, incorrect, and late announcements complicate the situation even more. For example, not only may scheduled flights arrive late, but also the event announcements for such events may be advertised late themselves. This requires us to consider the *bitemporality* of events: Each event has an *occurrence time,* when it supposedly takes place, and a *detection time,* when its advertisement is detected on the web.

Checking all these cases is a boring, stressful, and repetitive task, ideally suited for automation – at least at first glance. Except for specific domains, however, such systems are rare as the available technologies do not allow for a rapid development of event announcement detection integrated with complex event processing: The employed complex event processor must be able to deal with event announcements that are unreliable, volatile, and out of time, as published on the web, i.e., the event processing must be fully bitemporal. While complex events have been studied extensively, see [8] for a survey, existing systems deal only with some aspects of the bitemporality. For example, [2] considers events with two time dimensions, but does not consider mutable events or delayed events; [17] only allows to deal with mutable events in terms of new and unconsolidated update events. Most systems [7, 9, 10, 18, 21] assume a single time dimension, i.e., events are detected instantaneously when they occur. To extract event announcements from the web, however, an easy to use, scalable extraction system is needed that is able to produce a continuous stream of extracted event announcements.

The PEACE *(Processing Event Ads into Complex Events)* framework introduced in this paper addresses these challenges by an integrated framework for extracting event announcements and detecting complex events over such announcements through a bitemporal complex event processor. PEACE is driven by event models that specify the attributes of events in a domain and are used both in the specification of wrappers (for extracting event announcements) and in the specification of complex event queries. For the former, we use an extension of OXPATH [11], a highly efficient web automation language, slightly adapted to the needs of event announcement extraction. OXPATH is particularly well suited for this task as it is able to extract data from even heavily scripted modern web sites such as Ebay. For the latter, we introduce a novel bitemporal complex event processing language BICEPL *(Bitemporal Complex Event Processing Language)* that can be evaluated directly on top of the event announcements extracted by OXPATH. As required, BICEPL distinguishes between detection and occurrence time of events. This bitemporal event handling allows for enacting different actions, depending on the events' properties and timing information, e.g., considering delays between event detection and occurrence time. For a long-running complex event

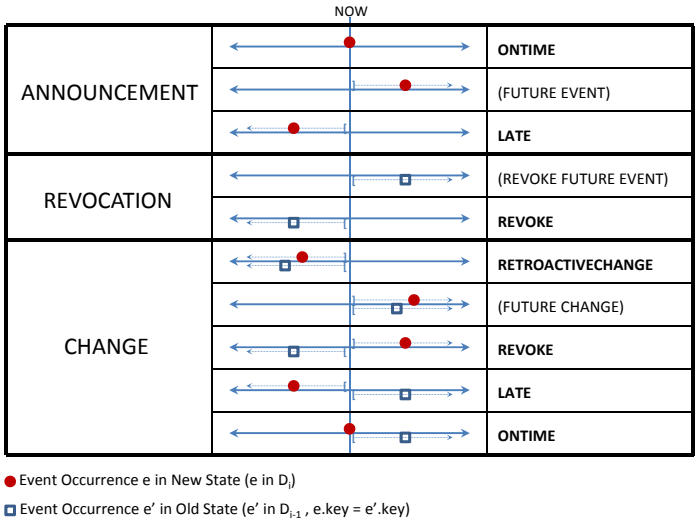

Fig. 1: Complex Event Publishing Cases

processing system it is essential that the memory use does not increase over time but remains constant (after a certain warm-up). This is guaranteed in BICEPL through the use of sliding windows, and, uniquely, in OXPATH through a novel buffer manager that guarantees constant memory regardless of the number of extracted events or web pages visited. This is key to an efficient implementation of BICEPL, since web event processing is dominated by loading and rendering web pages, as shown in Section 6.

PEACE is designed to support developers in adding new event announcement sources or new complex event queries. Both parts are driven by the event model and are minimal extensions of established query languages: OXPATH extends XPATH for web extraction. Most wrappers in OXPATH are a series of XPATH expressions, interspersed with action (such as clicking a button) and extraction instructions, that specify the attribute of the event model to populate with the selected web data. The XPATH portions can be created by a myriad of mature developer tools in Browsers such as Firebug, and then only (simple) actions and extraction instructions must be added to complete the wrapper. OXPATH also provides a visual interface that further simplifies this task [15].

BICEPL extends SQL-select statements to define *complex events* described with SQL-select statements that are extended, first with temporal comparisons between events, and second, with a definition of the occurrence time of the complex event in relation to the timing information of the involved events. The resulting events are continuously updated in their attributes, such as location or ticket price, and in their timing information. This leads to 10 different cases when complex events should be published, updated, or retracted, see Figure 1 and Section 5. Notice, Allan [3] defined possible relations between immutable, *unitemporal* intervals; in contrast Figure 1 displays possible cases in a life cycle of a single mutable, *bitemporal* event.

Contributions and Organization. PEACE is the first integrated framework for complex event processing on event announcements in the web, designed around the following four contributions:

(1) *Integration* (Section 3). PEACE integrates extraction and complex event processing through a joint event model. The event model not only provides the interface between the PEACE parts, it also drives the development of OXPATH wrappers just as much as BICEPL complex event queries.

(2) *Bitemporality* (Section 5). PEACE inherently relies on two time dimensions: It distinguishes between events' occurrence and detection time and allows different actions, depending on the relation between those times and the current time. Despite this powerful event model, PEACE's complex event language BICEPL is a small extension atop SQL-select statements to provide a powerful yet easily understood way for describing event schemata.

(3) *Integrated event extraction* (Section 4). The event schemata declared in BICEPL are populated by wrappers written in OXPATH. OXPATH is also built atop a commonly known declarative language, namely XPATH.

(4) *Lightweight, memory efficient implementation* (Section 6). The prototype implementation of PEACE is highly efficient and lightweight, requiring only constant memory regardless of the number of events or sources to extract from.

1.1 Running Example

We illustrate PEACE through a simple scenario taken from the daily live of a business man, who travels regularly by plane to business meetings (we discuss the details of the code shown in Sections 4 and 5). Flights are often delayed and hence, he must update his business partners frequently about delays. This is not only costly in time and effort but sometimes impossible, e.g., when he is on a flight without access to communication services. However, this task could be delegated to PEACE, such that it informs his colleagues whenever he would get late to a meeting. To simplify the example for presentation the business man only takes direct flights. To detect potential delays, the web sites of airlines and airports are observed continuously.

In this example we identify the following event classes: `FlightArrival` signifies the landing of a plane with the attributes `flightDay`, `flightNo`, `fromLoc`, and `toLoc`, the latter describing the departure and destination locations. Instances of `FlightArrival` are extracted from the web by wrappers covering airline and airport sites. `OneDayToArrival` is a complex event that occurs one day before a flight arrival associated with a business meeting, with attributes `flightDay`, `flightno`, `meetingId`, and `toLoc`. It announces relevant flight arrivals one day before the actual expected arrival takes place and is kept up-to-date on changes of the estimated arrival time. All classes have an additional implicit attribute `occ` for the occurrence time of the event and `det` for the detection time. Finally, the `BusinessMeeting` is provided through some database for this example, though they could also be extracted from a web calendar or some other meeting planning system.

To detect subscribed events on the web, PEACE integrates OXPATH wrappers that are executed on (possibly several) target pages. A wrapper for a certain event class produces data compliant with the schema of the event it observes. For this example, the wrapper in Figure 2 detects `FlightArrival` events from `flightarrivals.com`. This

```
1 doc("http://www.flightarrivals.com")
2   //a#panel0/{click /}//form#qbaForm/descendant::field()[1]/{$airport }
3   /following::field()[3]//option{select }/following::field()[1]/{click /}
4    /(/descendant::a[string(.)='Next >'][1]/{click /})*
5       //table#flifo//tr[position()>1]/self():<FlightArrival>
6         [./td[1]:<fromLoc=string(.)>]        [./td[2]:<flightNo=string(.)>]
7         [./td[3]/div:<flightDay=string(.)>]  [.:<toLoc=$airport>]
8         [./td[3]/text()[1]:<occTime=toUnixTime(.)>]
```

Fig. 2: OXPath Wrapper

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS FlightArrival(flightDay TEXT, flightNo
      TEXT, fromLoc TEXT, toLoc TEXT)
2    ID (flightDay, flightNo)
3    LIFESPAN (2d);
4
5 CREATE COMPLEX EVENT CLASS OneDayToArrival(flightDay TEXT, flightNo TEXT,
      toLoc TEXT, meetingId TEXT)
6    ID (flightDay, flightNo, meetingId)
7    AS SELECT fa.flightDay, fa.flightNo, fa.toLoc, bm.meetingId
8       FROM FlightArrival fa, BusinessMeeting bm
9       WHERE fa.flightNo = bm.flightNo AND fa.flightDay = bm.flightDay
10      OCCURRING AT fa - 1d
11   PUBLISH OneDayToArrival_OnTime
12      CASE LATE(0s,1m) OneDayToArrival_OnTime
13      CASE LATE(1m,1h) OneDayToArrival_Late
14      CASE RETROACTIVECHANGE OneDayToArrival_RAChanged
15      CASE REVOKE OneDayToArrival_Revoked;
```

Fig. 3: Event Definition for Flight Arrivals

wrapper fetches the target page (Line 1), then selects and submits the form for flights arriving at the parameterised arrival airport at any of the possible time windows in the day (Lines 2–3). The wrapper deals with paginated results by repeatedly clicking on "next" (Line 4). On each result page, `FlightArrival` objects are extracted along with their attributes (Lines 5–7) for every flight arrival entry listed, resulting in an event tuple as shown below:

flightDay	flightNo	fromLoc	toLoc	occ
[redacted]	BA 112	New York JFK	London LHR	1369116000

Figure 3 shows the BiCEPL specification for importing the subscribed event class `FlightArrival` and defining the complex event class `OneDayToArrival`. The first statement (Lines 1-3) declares the subscribed event class `FlightArrival` with its explicit attributes `flightDay`, `flightNo`, `fromLoc` and `toLoc`, with `flightDay` and `flightNo` as key (Line 2). `FlightArrival` is defined mutable (Line 1), since estimated flight arrival times change over time, and with a lifespan of `2d`, meaning 2 days (Line 3).

The second statement (Lines 5-15) defines the complex event type `OneDayToArrival` based on the constituent event classes `FlightArrival` and `BusinessMeeting`. It features as attributes `flightDay`, `flightNo`, `toLoc`, and `meetingId`, with `flightDay`, `flightNo`, and `meetingId` as key (Line 6). A `OneDayToArrival` event occurs one day before the flight arrives, as defined with the **`OCCURRING AT`** clause (Line 10). When the complex event is detected on time, or with up to one minute delay (`0s,1m`), the complex event publishes `OneDayToArrival_OnTime` (Lines 11-12). If the event is detected within a delay of more than one minute (`1m`) up to one hour(`1h`), `OneDayToArrival_Late` is published (Line 13).

If an already published event must be revised, e.g.. the flight arrives later than assumed beforehand, `OneDayToArrival_RAChanged` is published (Line 14). If an already published event is revoked, e.g., if the flight is canceled less than a day before it is planned to arrive, `OneDayToArrival_Revoked` is published (Line 15). All these following the schema of the associated complex event `OneDayToArrival`.

2 Related Work

To the best of our knowledge, PEACE is the first system that addresses the complex event processing for event announcements from the Web. This differs from mining events from Twitter or other sources [5, 13] but is related to typical complex event processing where many event sources are integrated to detect complex events and react to these. Therefore, we focus on the difference of PEACE and its complex event processor and language BICEPL with existing complex event processing systems.

Event processing approaches can be classified into three different approaches: Active Database Management Systems (ADMS) [1, 12, 20], Data Stream Management Systems (DSMS) [4, 6, 16] and Complex Event Processing Systems (CEP) [2, 7, 9, 10, 18, 21]. Unfortunately, all ADMS and DSMS approaches do not fit PEACE for not supporting bitemporality, as these systems make the perfect technology assumption [23], i.e., they assume an event is known instantaneously after it occurred. Even most CEPs make this assumption: The occurrence time of an event is given to it when entering the system. This restrictive time model disables reasoning over events in an imperfect world, as for this task the occurrence time and detection time are essential. To the best of our knowledge AMIT [2] is the only CEP system thinking about two time dimensions. Yet, these time dimensions are not supported to be used for temporal comparisons in complex event processing. Situations, as they call complex events, are defined by their highly expressive, imperative complex event language which is conceptually similar to the ones used in active database systems, e.g., in [20].

Mapping our complex event declarations to existing complex event processing approaches is only partly possible: The *on time* case, defining the reaction if there are no delays, changes, or errors in the event planning and announcement, is the standard scenario assumed by all approaches. This is the only case that is directly supported. The *late* case, is not supported at all by existing approaches due to a missing second time dimension. Notice, the occurrence time is event inherent (implicit) and may not be compared with user defined (explicit) attributes. Mapping the *retroactive change* case as well as the *revoke* case to existing languages is possible, though a tenuous task and requires the subscription to multiple event types.

Beside event processing, also Event Calculus (EC) [14, 19, 22] in knowledge representation deal with events, representing knowledge about events for reasoning purposes. Originally EC is unitemporal, though, there exist extensions [19, 22] to implement bitemporal deductive database systems. EC rules are expressive enough to model complex events, but would require a number of low-level rules to represent a single complex event declaration of BICEPL: EC provides a general event model while BICEPL is a succinct yet expressive language tailored for event management.

3 PEACE Approach and Event Model

PEACE is designed for quickly instantiating a complex event detection system on top of event announcements from new sources. The setup of PEACE application requires the provision of the following: **(1)** *Subscribed event classes,* specifying the schema of their events. **(2)** OXPATH *wrappers* for those subscribed event classes which feed from web sources, matching the schema of their subscribed event classes. **(3)** *Wrappers* for other sources, if any, matching again the schema of their subscribed event classes; examples include database triggers, e.g., to retrieve background information on a business meeting. **(4)** *Complex event classes* to aggregate the subscribed events into complex events for capturing the conditions and information required by the application and for driving the publication of events to be delivered to the client of PEACE.

In our running example, for **(1)**, we show the subscribed event class `FlightArrival` in Lines 1-3 of Figure 3, omitting the `BusinessMeeting` which is similar. For **(2)**, we show the OXPATH wrapper in Figure 2 while, for **(3)**, `BusinessMeeting` is filled via a database trigger. At last, Lines 5-15 of Figure 3, show the definition of a complex event. In total, the entire example takes 19 lines of BICEPL and 13 lines of OXPATH plus a database tigger to implement.

Event Model. We identify such a BICEPL program with the event classes $\mathbb{E}$ it declares. Every event e processed by $\mathbb{E}$ belongs to one such class $e.\mathsf{class} = E \in \mathbb{E}$. Depending on the concrete class E, the event features certain attributes $e.attr$ for the attributes $attr \in E.\mathsf{schema}$, as specified in the schema $E.\mathsf{schema}$ of E. Each event schema contains a set of key attributes $E.\mathsf{key} \subseteq E.\mathsf{schema}$ and we denote with $e.\mathsf{key}$ the values of the key attributes in event $e \in E$. Further, the occurrence time and detection time of each event e is accessible via the implicit attribute $e.\mathsf{occ}$ and $e.\mathsf{det}$, respectively. These attributes are implicit, since we want to control the way they are computed and accessed, providing occurring-at and checking-at clauses. Note that the key of an event identifies the event but not the announcement, i.e., there may coexist several announcements e and e' referring to the same event $e.\mathsf{key} = e'.\mathsf{key}$ detected at different times, i.e., with $e.\mathsf{det} \neq e'.\mathsf{det}$.

Event classes in $\mathbb{E}$ are partitioned into subscribed and complex events $\mathbb{S}$ and $\mathbb{C}$. In addition to the schema, subscribed event classes $S \in \mathbb{S}$ also have a lifespan $S.\mathsf{lifespan}$, determining how long an event is retained before being purged, and an associated wrapper $S.\mathsf{wrap}$ or trigger expression. In contrast, complex event classes $C \in \mathbb{C}$ have a set of publication statements $C.\mathsf{pub}$ and a query function $C.\mathsf{query}$. The publication statements in $C.\mathsf{pub}$ are subdivided into $C.\mathsf{pub}[O]$, $C.\mathsf{pub}[L]$, $C.\mathsf{pub}[C]$, $C.\mathsf{pub}[R]$ for

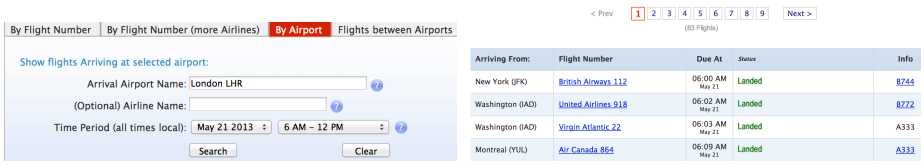

Fig. 4: Form and Result Page on www.flightarrivals.com.

on-time, late, change, and revoke publication events. If the query C.query depends on observed subscribed event classes or other complex event classes (derived from those observed events), then we call these classes *constituent event classes* of C. The constituent classes do not only include direct dependencies but also indirect dependencies via other constituent complex classes. If $E \in \mathbb{E}$ is a constituent event class of $C \in \mathbb{C}$, we write $E \subset C$; hence $\subset$ is transitive by definition. Additionally, we require $\subset$ to be irreflexive and asymmetric, i.e., we allow *no cyclic dependencies* in complex event queries. Likewise, we write $e \subset c$ for concrete event instances e and c, if e is a constituent event of c. For the set of C-events obtained by evaluating C.query, we write $C.\text{query}(O,D,t)$, where O is the set of so far observed subscribed events, D is the set of derived constituent complex classes, and t is the wall clock time.

4 Extracting Event Announcements

For detecting events announcements on the web, PEACE integrates OXPATH, a recent state-of-the-art tool for highly efficient data extraction [11]. OXPATH's main strengths lie in its ability to deal with modern scripted websites necessitating complex interaction (e.g., Ajax forms, auto-completion fields), and the capability to scale well in time and memory, handling even millions of pages and extracted results at ease. Indeed, OXPATH's output handling requires no buffer as the extracted data is streamed out once matched on the page, see [11]. This behaviour fits perfectly with the needs of PEACE. A full description of OXPATH is out of this paper's scope, and can be found in [11]. In short, OXPATH is an extension of XPATH with four features: **(1)** *actions*, such as mouse events, form filling, for simulating user interactions with web pages; **(2)** iteration via *Kleene stars,* e.g., to deal with sites that present their results across several pages or use any pagination techniques; **(3)** more expressive node selection through the *style axis*, querying the actual visual attributes as rendered by the browser, to select e.g. all elements coloured green; and, **(4)** specification of data to be extracted in terms of (nested) records and attributes, via *extraction markers.* To illustrate OXPATH's capabilities, we discuss how to derive a wrapper from a PEACE event schema $\mathbb{E}$ in OXPATH along the running example from Figure 2.

There are two steps in the derivation for a specific event class $E \in \mathbb{E}$: First we define the navigation on the event source to the event announcements. Second, we specify how to map each of the event model's attributes to fragments of the event announcements. In our example, Lines 1–4 perform the navigation and Lines 5–8 the attribute mapping. Figure 4 shows the form on the left hand that the navigation part first fills by selecting the "By airport" tab and then filling the airport into the arrival airport field. It then iterates over all options of the second select in the time period and submits the form once

for each of those options. On the result page, it iterates through the next links connecting the paginated results using a *Kleene star* expression. This entire expression can be easily obtained using standard web developer tools present in most browsers or in Firebug, though we here use some of OXPath's extensions to obtain a more readable expression. We also provide a visual tool that allows the recording of interaction sequences and automatically generates the corresponding navigation sequence so that the user only has to change the parameterized values or adjust the iteration parts.

For the attribute mapping, one can again use standard web developer tools to obtain XPath expressions for identifying which peace of a page maps to which attribute of E. This is shown in Lines 5–8. Each result page reached contains a table with flight arrival entries in its rows. We skip the first row as it only contains the column headers (Line 5), and for all the remaining we use OXPATH's *extraction markers* to shape the extracted data in compliance with the schema of the corresponding event class `FlightArrival` defined in Figure 3: One event `:<FlightArrival>` is created for each table row, along with attributes such as `<fromLoc=string(.)>` for the departure airport in the first column `./td[1]`; similarly for flight number, flight day, and occurrence time in adjacent columns. Notice, that all key attributes ($E.\mathsf{key}$) must be present explicitly on the web site to allow for tracking changes to the event (here flight day and number).

The resulting wrapper is used in PEACE to poll the website repeatedly and produce the input events for the complex event processor. The polling frequency and behaviour can be adjusted by the user, though PEACE provides a simple, but effective change detection based on a sample of the result pages by default. If within $E.\mathsf{lifespan}$ the same event is detected multiple times (in different pollings), it is only reported if its attributes have changed.

5 Event Processing with BICEPL

We designed BICEPL to handle event advertisements as they occur on the web. In a perfect world, each event, as identified by its key attributes, would be announced once and would have a single event occurrence time. Web announcements are certainly subject to frequent updates, hence BICEPL features events which are updated independently of the event's actual occurrence time, e.g., allowing for retro-actively changes or revocations. Moving into such an imperfect world, we still assume that events occur only once, but the attributes and occurrence times for past and future events may change. If there are different announcements e and e' for the same event $e.\mathsf{key} = e'.\mathsf{key}$, we assume that those announcements have different detection times $e.\mathsf{det} \neq e'.\mathsf{det}$. But given det and key, only a single event announcement e may exist with $e.\mathsf{det} = \mathsf{det}$ and $e.\mathsf{key} = \mathsf{key}$.

In BICEPL, all events belong to a *event class*, defining a schema for its events' explicit attributes. *Subscribed events* are produced by OXPATH wrappers or other sources, hence BICEPL only define their lifespan. *Complex events* are derived from constituent events which are either subscribed or other complex. The definition of a complex event class comes in two parts, **(1)** as an *event selector* based on an SQL select statement which aggregates constituent events into complex events. The event selector describes the attributes of the corresponding complex events in a perfect world, i.e., it disregards event updates. **(2)** For each complex event, we define publication statements in reac-

tion to certain update types. We consider situations when events **(a)** are announced on time, **(b)** have been detected late, after having already occurred, **(c)** are retro-actively changed, or **(d)** are retro-actively revoked.

Syntax of BICEPL BICEPL's syntax in Figure 5 defines a program ⟨*program*⟩ as sequence of event class declarations, each describing either a simple or complex event class. We declare a subscribed event class $S \in \mathbb{S}$ via ⟨*sclass*⟩ as either mutable or immutable, with an event schema S.schema, a key S.key, and a lifespan S.lifespan, given as ⟨*time_literal*⟩ which consists of a positive integer with `s`, `m`, `h`, or `d` for seconds, minutes, hours, or days. A complex event class $C \in \mathbb{C}$ is declared via ⟨*cclass*⟩, consisting of a schema C.schema, an SQL select statement C.query, and event publication statements C.pub. The SQL select statement may refer to subscribed event classes in $\mathbb{S}$ and to other complex event classes in $\mathbb{C}$, as long as no circular dependency arises. The schema ⟨*schema*⟩ describes with ⟨*attributes*⟩ the typed attributes E.schema and with ⟨*key*⟩ the attributes forming the key E.key of event class E. The select statements are extended with *occurring-at* and *checking-at* clauses. An occurring-at clause describes a complex event's occurrence time, while a checking-at clause describes when the event is checked for, parameterized with start and end times of those checks and the interval between two consecutive checks. Both clauses are based on time expressions ⟨*time*⟩ which refer to occurrence times of constituent events. The occurrence time of a constituent event is accessed via ⟨*table_ref*⟩, e.g., in a statement `SELECT FROM FlightArrival fa...`, we use `fa` to refer to the occurrence time of the selected flight arrival event. In occurring-at clauses only, time expressions may also contain `'NOW'` to refer to the current system time. This implies that the boundaries of checking-at clause are determined by occurrence times of constituent events. Based on these basic expressions, time expressions involve recursively min/max and increment/decrement computations. The where-clause of a select statement may also involve comparisons of ⟨*time*⟩ expressions (not shown in the grammar). Next to the select statement, complex events also describe publication events C.pub in ⟨*publication*⟩. We distinguish publication events for on-time, late, retro-actively changed, and revoked event announcements, referred to as C.pub[O], C.pub[L], C.pub[C], and C.pub[R], and declared with ⟨*ontime*⟩, ⟨*late*⟩, ⟨*change*⟩, and ⟨*revoke*⟩. In all four cases, BICEPL requires the name of the publication event to generate. Late events are optionally parameterized with an interval restricting the considered delay.

Semantics of BICEPL We define the semantics of BICEPL in two variants – first as *idealized semantics* without ever purging observed events, and second as *sliding window semantics* by considering the lifespan of the subscribed events and purging them when they turn stale. As a technical prerequisite, we start with rewriting complex event queries into standard SQL.

Query rewriting: Given an expanded SQL select statement, we turn C.query into standard SQL by performing three rewriting steps: **(1)** We expand all table references so as to access the implicit occurrence time attribute, e.g., in our running example we rewrite `OCCURRING AT fa - 1d` into `OCCURRING AT fa.occ - 84600`, as 1 day equals 84600 seconds. **(2)** We rewrite occurring-at clauses into a definition of the implicit occurrence time attribute `occ`, e.g., `OCCURRING AT fa.occ - 84600` yields `SELECT fa.occ - 84600 as`

⟨*program*⟩ ::= { ⟨*sclass*⟩ | ⟨*cclass*⟩ }
⟨*sclass*⟩ ::= 'CREATE' ('MUTABLE' | 'IMMUTABLE') 'SUBSCRIBED EVENT CLASS'
⟨*schema*⟩ 'LIFESPAN' ⟨*time_literal*⟩ ';'
⟨*cclass*⟩ ::= 'CREATE' 'COMPLEX EVENT CLASS'
⟨*schema*⟩ 'AS' ⟨*selection*⟩ ['PUBLISH' ⟨*publication*⟩] ';'
⟨*schema*⟩ ::= ⟨*name*⟩ '(' ⟨*attributes*⟩ ')' 'ID' '(' ⟨*key*⟩ ')'
⟨*attributes*⟩ ::= ⟨*name*⟩ ⟨*type*⟩ { ',' ⟨*name*⟩ ⟨*type*⟩ }
⟨*key*⟩ ::= ⟨*name*⟩ { ',' ⟨*name*⟩ }
⟨*selection*⟩ ::= 'SELECT' ⟨*select_clauses*⟩ 'OCCURRING AT' ⟨*time*⟩
['CHECKING AT (' ⟨*time*⟩ ',' ⟨*time*⟩ ',' ⟨*time_literal*⟩ ')']
⟨*time*⟩ ::= ⟨*table_ref*⟩ | 'NOW' | ⟨*time*⟩ [('+' | '-') ⟨*time_literal*⟩]
| ('MAX' | 'MIN') '(' ⟨*time*⟩ { ',' ⟨*time*⟩ } ')'
⟨*publication*⟩ ::= ⟨*ontime*⟩ { ⟨*late*⟩ } { ⟨*change*⟩ } { ⟨*revoke*⟩ }
⟨*ontime*⟩ ::= ⟨*name*⟩
⟨*late*⟩ ::= 'CASE LATE' ['(' ⟨*min_delay*⟩ ',' ⟨*max_delay*⟩ ')'] ⟨*name*⟩
⟨*change*⟩ ::= 'CASE RETROACTIVECHANGE' ⟨*name*⟩
⟨*revoke*⟩ ::= 'CASE REVOKE' ⟨*name*⟩

(with ⟨*select_clauses*⟩ and ⟨*table_ref*⟩ taken from a SQL grammar)

Fig. 5: BICEPL Syntax.

`occ` ..., defining the occurrence time as first attribute in the newly created event. **(3)** We turn checking-at clauses into additional where-clauses.

Idealized Semantics: A BICEPL program, identified by its event classes $\mathbb{E} = \mathbb{S} \cup \mathbb{C}$, observes a sequence of pairs O_i, t_i, where O_i is the set of subscribed events detected up to time stamp t_i. We set t_0 to the system start-up time, and require $t_i > t_{i-1}$ for all $i > 0$. We start with $O_0 = \emptyset$, and for $i > 0$, we set $O_i = \{e \in O_{i-1} \mid \nexists e' \in \Delta_i \text{ and } e.\mathsf{key} = e'.\mathsf{key}\} \cup \Delta_i$ where Δ_i contains the subscribed events observed between t_{i-1} and t_i. Depending on the differences between O_i and O_{i-1}, program $\mathbb{E}$ publishes a set of publication events, as specified in $C.\mathsf{pub}$ for $C \in \mathbb{C}$. Hence we define the semantics $[\![\mathbb{E}]\!](O_i, t_i, O_{i-1}, t_{i-1})$ of program $\mathbb{E}$ over two pairs O_i, t_i and O_{i-1}, t_{i-1} to be compared. The distinction between mutable and immutable events has no semantic effect but enables an optimized treatment of immutable events.

The semantics

$$[\![\mathbb{E}]\!](O_i, t_i, O_{i-1}, t_{i-1}) = \mathsf{compare}(\mathsf{derive}(O_i, t_i), \mathsf{derive}(O_{i-1}, t_{i-1}))$$

is computed in two steps: **(A)** We *derive* the complex events $D_i = \mathsf{derive}(O_i, t_i)$ and $D_{i-1} = \mathsf{derive}(O_{i-1}, t_{i-1})$ with $\mathsf{derive}(O, t) = D^l$ for complex classes $\mathbb{C} = \{C^1 \dots C^l\}$. Herein, we set $D^0 = \emptyset$ and $D^j = D^{j-1} \cup C^j.\mathsf{query}(O, D^{j-1}, t)$. We assume $C^j \subset C^k$ for all $j < k$, as $\subset$ is irreflexive and asymmetric. **(B)** We determine the resulting publication events by *comparing* D_i and D_{i-1} and adding to $\mathsf{compare}(D_i, D_{i-1})$ the publication events arising in the following cases (see Figure 1):

(1) *Announcement* – there exists $e \in D_i$ but no $e' \in D_{i-1}$ with $e.\mathsf{key} = e'.\mathsf{key}$.
(a) For $e.\mathsf{occ} = t_i$ we add $p \in e.\mathsf{class.pub}[O]$ (on-time).
(b) For $e.\mathsf{occ} < t_i$ we add $p \in e.\mathsf{class.pub}[L]$ if $p.\mathsf{min} < t_i - e.\mathsf{occ} \leq p.\mathsf{max}$ (late).
(c) For $e.\mathsf{occ} > t_i$ we add nothing (future event).
(2) *Revocation* – there exists $e \in D_{i-1}$ but no $e' \in D_i$ with $e.\mathsf{key} = e'.\mathsf{key}$.

(a) For $e.\mathsf{occ} < t_i$ we publish $p \in e.\mathsf{class.pub}[R]$ (revoke).
(b) For $e.\mathsf{occ} \geq t_i$ we add nothing (future revoke).
(3) *Change* – there exists $e \in D_{i-1}$ and a $e' \in D_i$ with $e.\mathsf{key} = e'.\mathsf{key}$ but $e \neq e'$.
(a) For $e.\mathsf{occ} \leq t_i$ and $e'.\mathsf{occ} \leq t_i$ we add $p \in e.\mathsf{class.pub}[C]$ (retroactive change)
(b) For $e.\mathsf{occ} > t_i$ and $e'.\mathsf{occ} > t_i$ we add nothing (future change).
(c) For $e.\mathsf{occ} > t_i$ and $e'.\mathsf{occ} \leq t_i$ we add $p \in e'.\mathsf{class.pub}[R]$ (revoke).
(d) For $e.\mathsf{occ} < t_i$ and $e'.\mathsf{occ} > t_i$ we add $p \in e.\mathsf{class.pub}[L]$
with $p.\mathsf{min} < t_i - e.\mathsf{occ} \leq p.\mathsf{max}$ holds (late).
(e) For $e.\mathsf{occ} = t_i$ and $e'.\mathsf{occ} > t_i$ we add $p \in e.\mathsf{class.pub}[O]$ (on-time).

Sliding Window Semantics: Ideally, we would never drop observed, un-revoked subscribed events, but due to limited resources, we have to drop events eventually. To avoid altering the semantics, we need not only consider the lifespan of each subscribed event e but also the lifespan of all subscribed events e' such that e and e' are involved into some complex event c. We set $c.\mathsf{expiration} = \max\{e.\mathsf{occ} + e.\mathsf{class.lifespan} \mid e \subset c\}$ and $e.\mathsf{expiration} = \max\{c.\mathsf{expiration} \mid e \subset c\}$. Then we purge events with $\mathsf{purge}(O,t) = \{e \in O \mid e.\mathsf{expiration} \geq t\}$, keeping only unexpired events. Finally, we need apply the same time stamp t_{i-1} in purging O_i and O_{i-1} (instead of t_i and t_{i-1}), leading to

$$[\![\mathbb{E}]\!]_{\mathsf{purge}}(O_i,t_i,O_{i-1},t_{i-1}) = [\![\mathbb{E}]\!](\mathsf{purge}(O_i,t_{i-1}),t_i,\mathsf{purge}(O_{i-1},t_{i-1}),t_{i-1}) \ .$$

This windowing and idealized semantics behave identically if $\mathbb{E}$ uses only **(A1)** *monotone queries,* i.e., $C.\mathsf{query}(O,D,t) \subseteq C.\mathsf{query}(O',D',t)$ for all $O \subseteq O'$, $D \subseteq D'$, and $C \in \mathbb{C}$, and is **(A2)** *key subsuming,* i.e., $E.\mathsf{key} \subseteq C.\mathsf{key}$ for all $E \subset C$; and if the subscribed events $O_0, O_1, \ldots$ fed to $\mathbb{E}$ have **(B1)** *timely updates,* never updating an event beyond its lifespan (for $e' \in O_{i-1}$ and $e \in O_i$ with $e'.\mathsf{key} = e.\mathsf{key}$, we have $e'.\mathsf{occ} + e.\mathsf{class.lifespan} \geq e.\mathsf{det}$), and **(B2)** *cohesive updates,* i.e., the constituent events of a complex event must share an overlapping lifespan as matching constituent events.

Theorem 1. *If a program $\mathbb{E}$ satisfies **(A1-2)** and if the events $O_0, O_1, \ldots$ satisfy **(B1-2)**, then $[\![\mathbb{E}]\!]_{\mathsf{purge}}(O_i,t_i,O_{i-1},t_{i-1}) = [\![\mathbb{E}]\!](O_i,t_i,O_{i-1},t_{i-1})$ holds.*

6 Implementation and Evaluation

PEACE should not only support server systems but also applications on small mobile devices. Therefore, our current implementation has been designed to be lightweight and portable, implemented in Java and SQLite in-memory database, and tested on Android and Ubuntu. We evaluate PEACE in three scenarios: The first two scenarios both extract and process flight arrivals from the web site of Heathrow Airport but differ in scale. The third scenario involves a stress test on PEACE's complex event processor to demonstrate its scalability.

In Scenario 1 we extract all flights from Frankfurt Airport to London Heathrow Airport, which were 36 flights on the day we performed the test. Requiring to load multiple pages, this test took 17 seconds on average. In Fig. 6a we show the time spent in different of PEACE modules; the initialization of the event detector (EDT), performing the OXPATH query, forwarding the events to the buffer (Event Buffer Maintenance), and processing the BICEPL programs. OXPATH dominates the complex event processor

with about 96,7% of the time needed to perform one processing step. As OXPATH spends 98% of its time in browser overhead [11], PEACE is dominated by browser overhead as well.

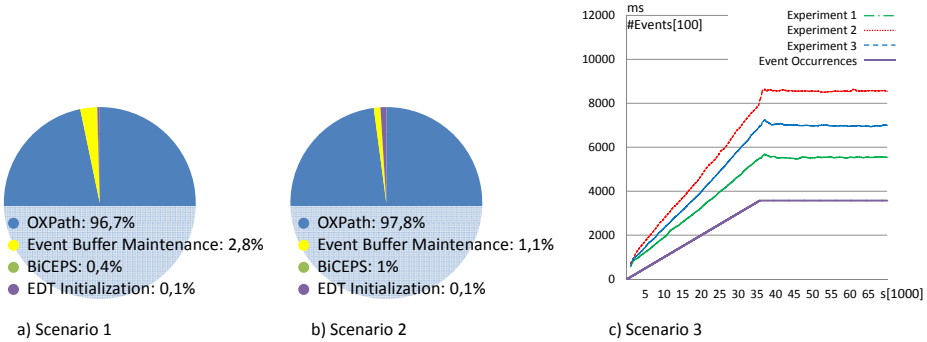

Fig. 6: Profiling PEACE components.

In Scenario 2 we increase the number of extracted and processed events to estimate the behaviour of the system when scaling up. We extracted all 1680 flight arrivals at London Heathrow Airport. As shown in Fig. 6b OXPATH dominates PEACE with 97,8% even stronger, as initialization overhead becomes less important.

In Scenario 3 we performed a stress test on BICEPL's complex event processor to show the scaling behaviour of the complex event processor. To significantly increase the number of events, we would have to deploy a multitude of OXPATH wrappers. Therefore, we perform three experiments: Experiment 1 involves 5 subscribed event classes and 3 complex classes, each producing 10.000 events per hour. In Experiment 2, we add 2 more complex classes, and in Experiment 3 we use 2 complex classes and another complex class atop the former two. Fig. 6c shows the performance curve in milliseconds per processing step, and the number of subscribed events present in the repository (in hundreds, the same in all three experiments). All complex events include on-time, late, retroactive change, and revoke event publication statements. As Fig. 6c shows, the runtime of the complex event processor increases with the number of events in the repository. Keeping the number of event occurrences in the repository constant, also the time needed for processing steps remains constant. Due to lifespan specifications of subscribed events and the subsequent purging of old events the processing time can be capped together with the number of events in the repository. In the three experiments the database size was bounded by 165 MB, 221 MB, and 158 MB, respectively.

7 Conclusion

We have presented PEACE as an integrated framework to extract and process event announcements on the web. PEACE relies on a bitemporal event model which supports BICEPL, a compact language for complex event processing, and an efficiently

implementable semantics, requiring limited buffering only. We are currently expanding BICEPL with action executors to react on occurring complex events. Furthermore, we will evaluate PEACE on mobile devices, e.g., tablet-computers or smartphones.

References

1. Adaikkalavan, R., Chakravarthy, S.: SnoopIB: Interval-based event specification and detection for active databases. Data Knowl. Eng. **59** (2006)
2. Adi, A., Etzion, O.: Amit - the situation manager. VLDB J. **13** (2004)
3. Allen, J.F.: Maintaining knowledge about temporal intervals. Commun. ACM **26** (1983)
4. Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: CIKM. (2006)
5. Boettcher, A., Lee, D.: EventRadar: A real-time local event detection scheme using twitter stream. In: GreenCom. (2012)
6. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In: SIGMOD Conference. (2000)
7. Cugola, G., Margara, A.: TESLA: a formally defined event specification language. In: DEBS. (2010)
8. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. **44** (2012)
9. Demers, A.J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., White, W.M.: Cayuga: A general purpose event monitoring system. In: CIDR. (2007)
10. Eckert, M., Bry, F.: Rule-based composite event queries: the language XChange^{EQ} and its semantics. Knowl. Inf. Syst. **25** (2010)
11. Furche, T., Gottlob, G., Grasso, G., Schallhart, C., Sellers, A.: OXPath: A Language for Scalable Data Extraction, Automation, and Crawling on the Deep Web. VLDB Journal (2013)
12. Gehani, N.H., Jagadish, H.V.: Ode as an active database: Constraints and triggers. In: VLDB. (1991)
13. Ilina, E., Hauff, C., Celik, I., Abel, F., Houben, G.J.: Social event detection on twitter. In: ICWE. (2012)
14. Kowalski, R.A., Sergot, M.J.: A logic-based calculus of events. New Generation Comput. **4** (1986)
15. Kranzdorf, J., Sellers, A., Grasso, G., Schallhart, C., Furche, T. In: Proc. of WWW
16. Liu, L., Pu, C., Tang, W.: Continual queries for internet scale event-driven information delivery. IEEE Trans. Knowl. Data Eng. **11** (1999)
17. Luckham, D.: Event Processing for Business. John Wiley & Sons, Inc., Hoboken, New Jersey (2012)
18. Luckham, D.C.: Rapide: A language and toolset for causal event modeling of distributed system architectures. In: WWCA. (1998)
19. Mareco, C.A., Bertossi, L.E.: Specification and implementation of temporal databases in a bitemporal event calculus. In: ER (Workshops). (1999)
20. McCarthy, D.R., Dayal, U.: The architecture of an active data base management system. In: SIGMOD Conference. (1989)
21. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.R.: Distributed complex event processing with query rewriting. In: DEBS. (2009)
22. Sripada, S.M.: A logical framework for temporal deductive databases. In: VLDB. (1988)
23. Wieringa, R.: Design methods for reactive systems - Yourdon, Statemate, and the UML. Morgan Kaufmann (2003)

A Proof of Theorem 1

Proof. We need to show that

$$[\![\mathbb{E}]\!]_{\mathsf{purge}}(O_i,t_i,O_{i-1},t_{i-1}) = [\![\mathbb{E}]\!](O_i,t_i,O_{i-1},t_{i-1})$$

holds, if conditions **(A1-2)** and **(B1-2)** are met. Following the notation in Section 5, we have $[\![\mathbb{E}]\!](O_i,t_i,O_{i-1},t_{i-1}) = \mathsf{compare}(D_i,D_{i-1})$. Analogously, for the purged case, with $O_i^P = \mathsf{purge}(O_i,t_{i-1})$ and $O_{i-1}^P = \mathsf{purge}(O_{i-1},t_{i-1})$, and with $D_i^P = \mathsf{derive}(O_i^P,t_i)$ and $D_{i-1}^P = \mathsf{derive}(O_i^P,t_{i-1})$, we obtain $[\![\mathbb{E}]\!]_{\mathsf{purge}}(O_i,t_i,O_{i-1},t_{i-1}) = \mathsf{compare}(D_i^P,D_{i-1}^P)$. Thus, we rewrite the theorem statement as

$$\mathsf{compare}(D_i^P,D_{i-1}^P) = \mathsf{compare}(D_i,D_{i-1}) \ .$$

Since **(A1)** allows only monotone queries $C.\mathsf{query}(O,D,t)$ for complex event types $C \in \mathbb{C}$, $\mathsf{derive}(O_i,t_i)$ is *monotone* with $\mathsf{derive}(O,t) \subseteq \mathsf{derive}(O',t)$ for all $O \subseteq O'$. In particular, because of $O_i^P \subseteq O_i$, we have $D_i^P = \mathsf{derive}(O_i^P,t_i) \subseteq \mathsf{derive}(O_i,t_i) = D_i$ for all $i \geq 0$. Further, to show $c \in D_i^P$, it suffices to show $f \in O_i^P$ for all constituent events $f \subset c$, as additional events cannot invalidate c. Since **(A2)** requires $e.\mathsf{class.key} \subseteq c.\mathsf{class.key}$ for all constituent events $e \subset c$, each complex event c identifies its constituent events $e \subset c$, i.e., no other constituent set can support the same complex event.

To show the theorem, we distinguish changing, newly announced, revoked, and unchanging events. In the first three cases, we show that the relevant complex events are not purged. In case of unchanging events, we show that PEACE purges the complex event eventually without emitting publication events.

Changing events. A complex event c changes into c', if $c \in D_{i-1}$ and $c' \in D_i$ with $c \neq c'$ but $c.\mathsf{key} = c'.\mathsf{key}$. For equivalence in case of changing events, we need to show $c \in D_{i-1}^P$ and $c' \in D_i^P$. Since $c \neq c'$, some constituent event $e \subset c$ must have changed into e'. We obtain for all constituent events $f \subset c$

$$\begin{aligned} f.\mathsf{expiration} &\geq c.\mathsf{expiration} \\ &\geq e.\mathsf{occ} + e.\mathsf{class.lifespan} \\ &\geq e'.\mathsf{det} = t_i \quad . \end{aligned}$$

The first two inequalities hold by definition (see the discussion before Theorem 1), and the third one holds by **(B1)**. Thus, no constituent event $f \subset c$ is purged at time $t_{i-1} < t_i$; all are kept either unchanged or updated, leading to $c \in D_{i-1}^P$ and $c' \in D_i^P$, as claimed.

Announcing events. A complex event $c' \in D_i$ is announced, if there exists no $c \in D_{i-1}$ with $c.\mathsf{key} = c'.\mathsf{key}$. Thus, at least one constituent event $e' \subset c'$ has been announced or updated at t_i, now forming with other events a constituent set for c'. Because of **(B2)**, all constituent events must share an overlapping lifespan, and hence none of them can be purged before t_i, when the constituent is formed for the first time. Thus for all $f' \subset c'$, we find $f' \in O_i^P$ and hence $c' \in D_i^P$.

Revoking events. A complex event $c \in D_{i-1}$ is revoked, if there exists no $c' \in D_i$ with $c.\mathsf{key} = c'.\mathsf{key}$. Since c is revoked, there must be a constituent event $e \subset c$ which has been revoked or updated at t_i such that c has no constituent set anymore. Because of **(B1)**, $f.\mathsf{expiration} \geq t_i$ (as in case of changing events) for all constituent events $f \subset c$, and hence none of them can be purged before t_i, leading to $c \in D_{i-1}^P$, as required.

Unchanging events. If an event c remains unchanged with $c \in D_{i-1}$ and $c \in D_i$, we show that either *(i)* $c \in D_{i-1}^P$ and $c \in D_i^P$ holds, or *(ii)* $c' \notin D_{i-1}^P$ and $c' \notin D_i^P$ for any c' with $c'.\mathsf{key} = c.\mathsf{key}$: For case *(i)*, assume $c \in D_{i-1}^P$. Then $f.\mathsf{expiration} \geq t_{i-1}$ for all constituent events $f \subset c$. Since we compute O_i^P with $O_i^P = \mathsf{purge}(O_i, t_{i-1})$, we obtain $f \in O_i^P$, implying $c \in D_i^P$. Now for *(ii)*, assume $c \notin D_{i-1}^P$. Then, at least one constituent event $f \subset c$ has been purged already at t_{i-1} with $f \notin O_{i-1}^P$. This constituent event f remains purged at t_i, hence $f \notin O_i^P$ and thus $c \notin D_i^P$. □