←

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. We would like to thank graduate students Felix Burgstaller and Sebastian Hochgatterer for supporting us at the implementation of PEACE. Author's addresses: T. Furche, G. Grasso and C. Schallhart, Department of Computer Science, Oxford University, Parks Road, Oxford OX1 3QD; M. Huemer and M. Schrefl, Department of Business Informatics – Data & Knowledge Engineering, Johannes Kepler University, Altenberger Str. 69, 4040 Linz, Austria.

DOI: http://dx.doi.org/10.1145/0000000.0000000

PEACE-ful Web Event Extraction and Processing as Bi-Temporal Mutable Events

Tim Furche, Oxford University
Giovanni Grasso, Oxford University
Michael Huemer, Johannes Kepler University
Christian Schallhart, Oxford University
Michael Schrefl, Johannes Kepler University

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, consisting of event extraction, complex event processing, and action execution in response to these events. Given a schema of the events to be tracked, the framework populates this schema by extracting events from announcement sources. This extraction is performed by little programs called wrappers which produce the 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, as to distinguish between occurrence and detection time. This allows complex event specifications to track updates and to react upon differences in occurrence and detection time. In case of new, changing, or deleted events, PEACE allows to execute actions, such as tweeting or sending out email notifications. Actions are typically specified as web interactions, e.g., to fill and submit a form with attributes of the triggering event.

Our evaluation shows that PEACE's processing is dominated by the time needed for accessing the web to extract events and perform actions, allotting to 97.4%. Thus, PEACE requires only 2.6% overhead, and therefore, the complex event processor scales well even with moderate resources. We further show that simple and reasonable restrictions on complex event specifications and the timing of constituent events suffice to guarantee that PEACE only requires a constant buffer to process arbitrarily many event announcements.

Categories and Subject Descriptors: H.3.5 [**Information Storage and Retrieval**]: Online Information Services

General Terms: Languages, Design

Additional Key Words and Phrases: Complex event processing, web announcements, web automation, data extraction, web engineering

ACM Reference Format:
Tim Furche, Giovanni Grasso, Michael Huemer, Christian Schallhart, and Michael Schrefl, 2014. PEACE-ful Web Event Extraction and Processing
as Bi-Temporal Mutable Events. *ACM Trans. Web* V, N, Article A (January YYYY), 48 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Most events are announced at first and often exclusively on the web. 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 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 possibly revised later. Therefore, modern coordination tasks often boil down to continuously checking all relevant event sources for changes which might necessitate further actions in response. For example, awaiting a person on a flight with a stopover, one needs 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. Depending on all these conditions, we want to decide which actions to enact, e.g., notifying affected people with emails or changing hotel reservations. To differentiate these cases, we need 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.

1.1. Background and rationale of our approach to web event extraction and processing

It took about 25 years till the commercial database world (IBM's DB2, Oracle, and Teradata) took over the idea of temporal databases [Snodgrass and Ahn 1986] distinguishing transaction time (synonym: system time) and valid time (synonyms: application time, business time) in recording business events, such as the move of a person from e.g. Vienna to Oxford, the arrival of a flight, or the discharge of a patient from hospital. The valid time of such business events may be in the future, events may be recorded pro-actively (valid time > transaction time) and retro-actively (valid time < transaction time), and knowledge about events may be updated, i.e., the valid time of a business event may change.

Our work on bi-temporal web event processing is based on this event model of temporal databases, with *occurrence time* corresponding to *valid time* and *detection time* to *transaction time* of temporal databases. Most event processing systems do not consider high-level business events as described above, but model low-level system events that represent the notification about a new business event or the change of a business event by different kinds of events. Such system events are *immutable*, whereas business events are *mutable* (i.e., their occurrence times or their event attributes may change). *We believe that the processing of web events is best addressed by considering high-level business events that are mutable.*

We present a holistic approach to web-event extraction and processing. We extract events from web sources and consider them as high-level, bi-temporal business events. We introduce a high-level language to define complex events based over subscribed events that are extracted from the web and an accompanying business rule language that defines how to react to detected business events by initiating diverse actions, including publishing events on the web.

Snapshot databases assume that transaction time is identical to valid time, as many event processing systems do. Valid time database systems allow to record future events (i.e., to record events before they occur), but usually do not support recording events retro-actively (i.e., after they have occurred). This approach is sufficient for many applications, such as insurance, where one cannot take out insurance retro-actively.

In a first instance, our event model and our event processing rules (condition action statements associated with events) are based on this valid time database approach, thereby taking a *perfectness assumption*. The *perfectness assumption* assumes that the real world is perfect (perfect world assumption) in that the occurrence time of a business event does not change, i.e., an airplane lands as expected; a patient is discharged from hospital as planned and that the system world is perfect (perfect system assumption), i.e., there are no delays in detecting, communicating and recording events, and also in reacting to events. We will lift the perfectness assumption later.

From a design perspective, we abstract from an imperfect world when defining complex events and associated business rules in a first instance. This assists in focusing on the core business functionality. Such an approach is very common in system design: (1)

One designs a conceptual model of database abstracting from physical storage details and access structures (physical data independence), (2) Distributed database design may first concentrate on the global schema, abstracting from data fragmentation and data allocation (distribution independence), or (3) bi-temporal database design starts out with a snapshot or valid-time database first.

Our event-processing approach is *event-oriented*. Condition-action rules do not stand alone, but are always associated with an event class and define how to react to an event of the class once it occurs. Different to event-stream processing systems, where a complex event occurs at the time a certain event pattern is detected, our notion of event-occurrence is tied to real-world time (wall-clock-time). An event occurs when its occurrence time coincides with real-world time. *We believe such a* time-centric *and* event-oriented *approach best addresses the business perspective of handling web announcements.*

Condition-action rules are defined for event classes with the semantics that they are fired for an event of the event class if the event occurs (i.e., the time of the real world is the event's occurrence time). Thereby we assume a chronon-model, i.e., time changes at fixed-sized intervals (chronons). Time advancing, business rules are fired if there is an event that occurs at the current real-world time. For efficient processing, a time index may be used to initiate rule processing for an event class only at chronons for which an event of the event class occurs. If one wishes to act before the occurrence of an event, e.g., to leave for the airport an hour before the airplane arrives, one would define a relative temporal event to the arrival event that occurs one hour before the arrival.

We take a high-level *declarative approach* to specify complex events and we define their semantics by mapping complex events to SQL. Complex events are defined by predicates over attributes and occurrence-times of other events, primitive or complex. We represent primitive events by base relations and complex events by views over base relations and views. Thereby, we utilize the SQL-query optimizer for complex event detection and SQL index structures to index events on occurrence time or other attributes for efficient processing of events.

Lifting the perfectness assumption, we need to take into account that events may be detected or processed late (imperfect system) or that attributes and occurrence times of events may change in the real world (imperfect world). *This is a major difference to many complex event processing systems and mandated by the characteristics of event announcements on the web.*

Unlike many stream-processing systems, we do not process a continuous stream of events based on their detection time, but we compare the past and the current valid-time view of the event database, with potentially a bulk of new or changed events being available additionally once time advances a chronon.[1]

The underlying idea of bi-temporal event processing is taken from traditional office work. The metaphor we use is as follows: The office worker has a mail box used to deliver subscribed events (primitive events). He or she empties the mailbox at every chronon change (e.g., each day, each hour) and records with each event delivered a received time stamp (= detection time) that is the current real-world time. He or she uses the occurrence time stamps and other attributes of the subscribed events to determine complex events according to the specification of the complex event. If the occurrence time of an event is the real world time then he or she has to react to the occurrence of the event by processing the associated condition-action statement.

[1]A detailed comparison to event processing in Active Database Systems, Data Stream Management Systems, Complex Event Processing and Event Stream Processing Systems is given in the Related Work section

To deal with imperfectness, the office worker does not wish to make complicated comparisons involving detection time and occurrence time over the event history. A much simpler approach is sufficient and has been used for many years in office work based on paper-based systems. The office worker knows the previous state of the world for each event (i.e., the state before emptying the mail box) and knows the current state of the world for each event (i.e., the state after emptying the mail box). Thereby, new events may have been announced, some may have been canceled, others retroactively changed. The office worker can recognize anyone of these situations by comparing event occurrences with the same event identifier in the old and in the new event history. He or she will have for each type of event a set of business rules that tell him or her how to react to particular constellation of occurrence times of the old and the new event relative to *now* (the current reading of the wall-clock time). The office worker will react based on obsolete, additional, and changed information, considering it relative to the current time of his or her arm watch. *Our approach to web event extraction and processing mimics how a human agent would behave having looked at some web page some time ago and looking at the web page later again, taking into account the current real-world time.*

From a language design perspective, we provide for each event class a set of clauses to define how to react in the case of a perfect system and a perfect world and we provide a set of clauses to define how to deal with various cases of imperfectness (late announcement of an event, revocation of an event, retro-active change of an event; etc.). Typically, one would initially concentrate on the business problem itself and design the event processing for the perfect case first; thereafter one would define how to react to various cases that can arise due to an imperfect world or imperfect system.

1.2. Application domains

Our approach to high-level processing of events announced in inter- and intra-nets is applicable across most application domains where timely reaction or adaption to the events is necessary. Our running example is on flight arrival and departure announcements, a domain commonly known. Other examples include logistics and health care. In care at home, nurses visiting patients have to adapt their schedules if patients are discharged early or late from hospital or, if they should meet with a general practitioner jointly at some patient and the general practitioner is delayed. Discharge information of patients is made available at the hospital's intra-net, accessible to the home care organisation for its patients; as with flight arrivals, discharge events are bitemporal and may change. New, canceled or updated discharge events do not mean a complete re-scheduling of nurses. Overall scheduling tasks are rather complex and involve optimization of minimum travel distance of nurses, considering employment law as to work hours, and required support of patients. Re-acting to web announcements of discharge events does not involve a re-running of scheduling software, it rather means the adaption of the round trip on which the patient is typically visited (by squeezing another patient in or leaving the patient out). Similar, if a scheduled event (e.g., a joint meeting with a GP at a patient) is delayed (which will be again announced on the web, used as integration platform), the nurse would not unnecessarily wait there, but just see another patient in between. Our rule language is geared toward such common, but simple scenarios that require timely but simple action. In our running example on flight arrivals, we wish to advise that the passenger is late, a pick-up service is to be canceled, or a meeting he or she chairs is to be postponed or canceled. Re-planning a business trip is more complex, but typically not as time-constringent and would not be handled by our presented rule-language. However, the business rules determining whether the business trip should be replanned or be canceled may well be expressed by our rule language. The corresponding action events will be raised by the complex

event processor and handled by a human agent, who is relieved from the tedious task of observing and tracking web events and their interactions.

Our approach to web event extraction and processing is based on a chronon-based temporal event model in which time advances at the size of one chronon, which is typically between a second or a minute. We do not strive to support mission-critical real-time processing, but web-based business applications as described above in which timely reactions at second or minute level are sufficient and which otherwise would require application-specific complex programming or human processing.

1.3. Contributions and Organization

PEACE*(Processing Event Ads into Complex Events)* is the first integrated framework for complex event processing on event announcements in the web, designed around the following five contributions:

— *Bitemporal Event Model (Subsection 2.4).* Our approach distinguishes 10 timing cases, arising by different relationships of occurrence, detection, and wall-clock time. These cases are programmatically accessible to condition the action taken in response.
— *Integrated Event Extraction and Action Execution (Sections 3).* Instances of events are extracted with OXPATH wrappers, a declarative language for web interaction and data extraction built atop XPATH, able to fully interact with modern, scripted sites. The actions triggered during event processing are also implemented in OXPATH as web interactions, parameterized with the attributes of the triggering event.
— *Bitemporal and Mutable Complex Event Processing (Section 4).* PEACE provides BICEPL (BI-temporal Complex Event Processing Language) to specify condition-action statements for all event classes and to define complex event classes based on other constituent classes. The condition-action statements may relate event attributes, occurrence and wall-clock time to decide which action to execute. The definition of complex events extends familiar SQL-select statements to provide a powerful yet easily understood way for describing event schemata. We define the bitemporal event processing for a buffering and windowing semantics and show their equivalence under reasonable assumptions.
— *Efficient and Flexible Implementation (Section 5).* The prototype implementation of PEACE is highly memory-efficient, requiring in windowing semantics only constant memory regardless of the number of events or sources to extract from. Our architecture allows mixed deployments to develop mobile, light-weight front ends which connect to server systems bearing the application's load. Our implementation also comprises a development and a simulation and visualization environment for debugging and demonstrations.
— *Evaluation (Section 6).* We demonstrate by a comprehensive evaluation and extensive performance tests of the prototype implementation that our novel approach for web event extraction and processing as bi-temporal complex events can be put effectively into practice using web- and off-the-shelf relational technology. We show that our implementation only requires 2.6% overhead over loading and rendering for web access and that it is also well-suited for mobile devices with limited resources.

This paper extends and details a preliminary version of PEACE presented at WISE 2013 [Furche et al. 2013b].

2. COMPONENTS OF PEACE AND RUNNING EXAMPLE

In this section we introduce the components of PEACE along a running example used throughout the paper.

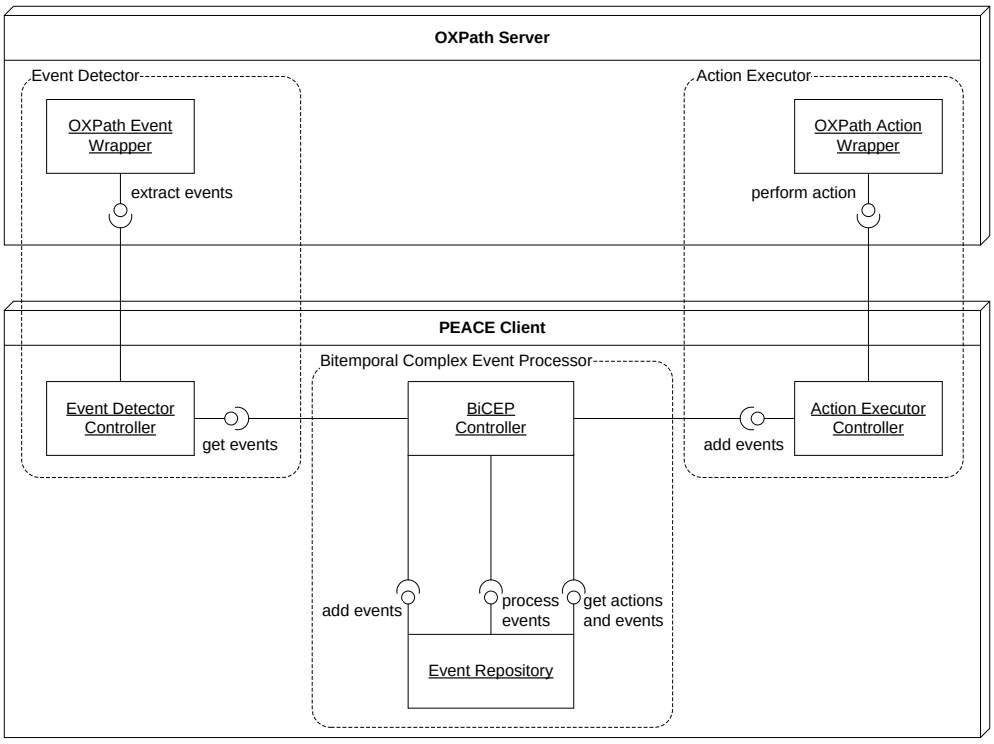


Fig. 1: Components of PEACE.

A PEACE application consists of BICEPL event classes for event processing and OXPATH wrappers for event extraction and action execution. All event classes in BICEPL have a schema of typed attributes, a subset thereof as key and a set of condition-action statements. The condition-action statements describe when an event triggers an action to be executed. The involved conditions refer to the preceding and current version of the event, thus allowing for reactions on changes. Subscribed event classes and complex event classes only differ in the way their event instances are produced: Subscribed events originate from OXPATH wrappers and database triggers, while complex events are computed from extended SQL queries over subscribed and other complex events. Thus, in BICEPL, subscribed classes do not require any further treatment, while complex classes need to include their constitutive queries.

Our running example deals with business travelers and their arrangements. To keep up with tight schedules, business travelers observe and quickly react upon many events, e.g., upon learning about flight delays, they need to check for connecting flights, or change hotel reservations. Observing flight data and reacting on deviations from the schedule is not only costly in time and effort but sometimes impossible, e.g., when the traveling person is on a flight without access to communication services. But having PEACE, all this hassle can be dealt with automatically. Aside managing the daily nuisances of business travel, PEACE can keep friends and family informed about ongoing trips by posting, e.g., Twitter messages.

2.1. Components of PEACE

PEACE consists of three different components, namely its *event detector*, *bitemporal complex event processor*, and *action executor*. Each of these components is an *event*

consumer and/or an *event supplier,* i.e., they take events as input for further processing and/or produce events to serve consumers. Therein, detected events are pulled from the detectors to the processors, and actions are pushed from the processors to the executors. Figure 1 shows these components of PEACE in a distributed deployment:

Event Detector. An event detector assumes the role of an event supplier which observes one event source, e.g., a web site publishing flight arrivals, to detect and supply events belonging to one subscribed event class, e.g., `FlightArrival`. Each subscribed event can be pulled by and delivered to several event consumers. A web-based event detector consists of an OXPATH *event wrapper* and an *event detector controller*. The event wrapper is started at predefined intervals by its controller. After each run, the wrapper returns the extracted events to the controller which returns them on request by its subscribed event processors. As depicted in Figure 1, the event detector controller and its OXPATH wrapper may run on different nodes, e.g., for workload distribution.

Bitemporal Complex Event Processor. Event processors pull events from event detectors or other event processors and derive from these input events new complex events. Accordingly, each processor acts as event consumer and supplier at the same time. As a supplier, it can serve several action executors or other event processors: The *derived complex events* are supplied to other event processors for *pulling,* while the *actions* triggered by these complex events are *pushed* to the action executors. The event processor contains an *active event repository* and a *controller*. The active event repository not only stores events but also processes subscribed events into complex ones, checks condition-action statements and creates actions if condition-action statements evaluate to true. For this purpose the event repository offers three different interfaces to its controller, as shown in Figure 1: With *add events,* the controller adds pulled events to the event repository. Next, with *process events,* the controller activates the condition-action evaluation and creates events and actions if necessary. Finally, with *get actions and events,* the actions resulting from the condition-action statements are pushed to the respective action executor and the newly generated complex events are stored for being pulled subsequently by other processors.

Action Executor. An action executor processes actions, supplied and pushed by event processors. Actions received by an action executor must be of the same class but may originate from different event processors. An action executor features an OXPATH *action wrapper* and a *controller*. The controller buffers received actions before it processes these actions sequentially. Alike event detectors, an action executor's wrapper and its controller may be deployed on different nodes.

2.2. Event Classes in the Running Example

We describe this scenario with the schemata of the subscribed event classes, forming the input to the system, and the corresponding action classes, representing the output. For completeness, we also specify the schema of complex event classes. Figure 2 presents these classes in UML notation. Note, for layout reasons, we display some associations as foreign keys.

Subscribed Event Classes. In our scenario, PEACE extracts flight arrival and departure events from web sites, such as www.flightarrivals.com (Figure 3), and gathers business trip information from internal information systems, all represented with the following event classes (Figure 2). **(1)** For each aircraft arrival, an event of the subscribed class `FlightArrival` is issued, with attributes `flightDay` and `flightNo` as key, and with attributes `fromLocation` and `toLocation` to describe the departure and destination locations. **(2)** An aircraft's take off is represented by an event of subscribed class `FlightDeparture` with the same attributes as `FlightArrival`. **(3)** A business

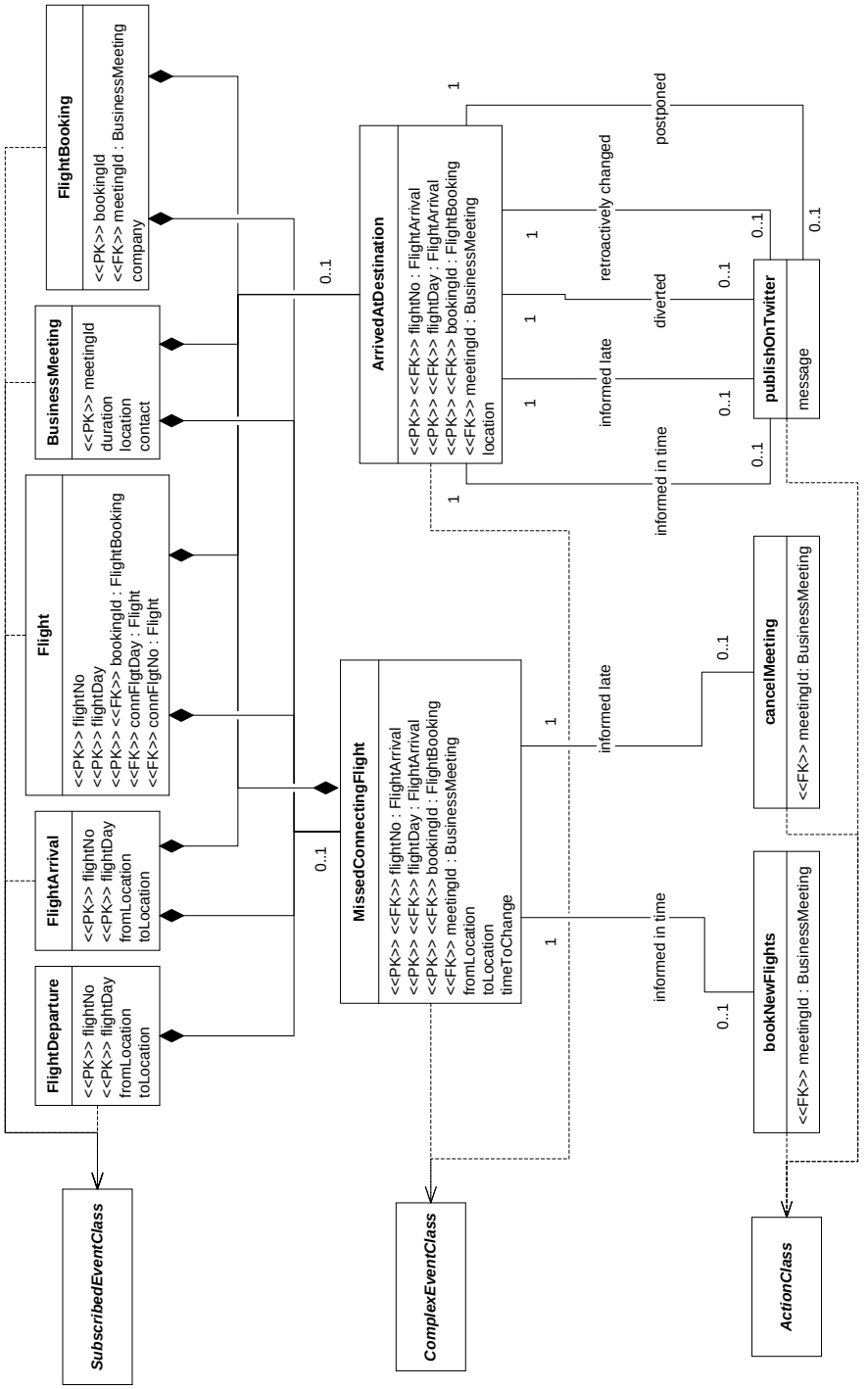


Fig. 2: UML Representation of Event Classes and Action Classes of our Running Example

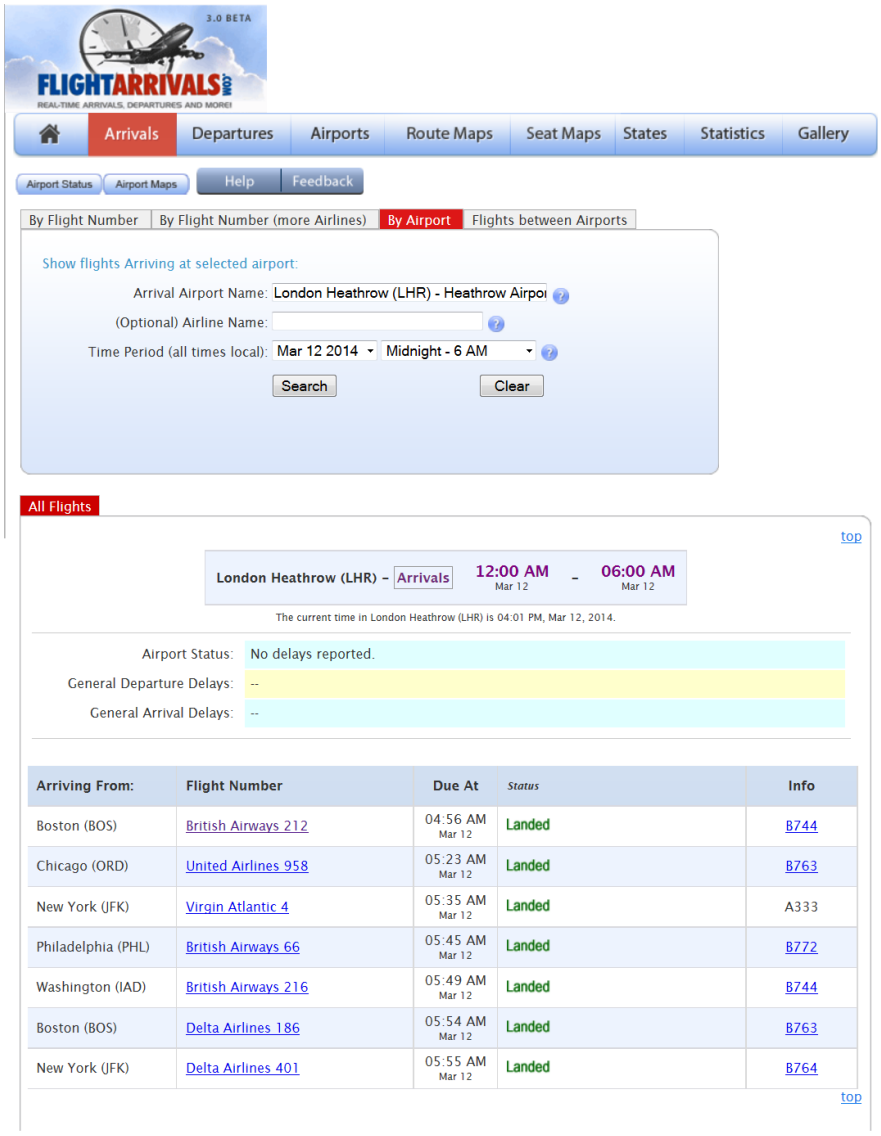


Fig. 3: Event Source for FlightArrival events: www.flightarrivals.com

meeting is represented by subscribed event class `BusinessMeeting`, with `meetingId` as key. **(4)** If flights are booked for a meeting, then a `FlightBooking` event is issued with a `bookingId` as key. **(5)** Scheduled flights are described by subscribed event class `Flight`, with attributes `flightDay`, `flightNo`, and `bookingId` (refering to the flight booking) as key, and `connFlgtDay` and `connFlgtNo` to link this flight to its connecting flight `Flight`, if any.

Since PEACE deals with bitemporal events, all extracted event classes (and the complex event classes following below) have two additional, implicit attributes, namely `occ` for the occurrence time of the event and `det` for its detection time.

Action Classes. In response to missed flights, PEACE advices to book new flights or cancels business meetings. **(1)** If PEACE learns about an unavailable flight at least a day in advance, it advises to book a new flight with an action of class `bookNewFlights`. **(2)** If the flight becomes unavailable less than a day in advance, then PEACE cancels the corresponding meeting with a `cancelMeeting` action. **(3)** On status changes of the business trip, PEACE also tweets notifications, represented by action class `publishOnTwitter`: *(a)* Within one hour delay after arrival, PEACE tweets the arrival at our final location. *(b)* If PEACE learns about an arrival more than one hour late, e.g., due to a downtime of the airport's web site, then it also includes the arrival time in the tweet. *(c)* If a flight is diverted, PEACE tweets the new destination. If PEACE has sent out notifications too early, e.g., learning only after the scheduled arrival time, that a flight has been delayed, then PEACE corrects the wrong tweet, distinguishing the case *(d)* when a traveler is still in the air and *(e)* when the traveler has arrived already.

Complex Event Classes. The actions above relate to two kinds of events, namely *missed flights* and *arrivals* at the final trip destinations. Both event types are represented by corresponding complex event classes: **(1)** Missed flights are represented by complex event class `MissedConnectingFlight` which is defined upon its constituent subscribed event classes, `FlightArrival`, `FlightDeparture`, `Flight`, `BusinessMeeting`, and `FlightBooking`. It holds attributes `flightNo`, `flightDay`, `bookingId`, `meetingId`, `fromLocation`, `toLocation`, and calculates the time for catching the connecting flight `timeToChange` from its constituent event classes. A flight is considered to be missed if the business traveler has less than 30 minutes between his/her connecting flights (`timeToChange` $<$ 30 minutes). **(2)** The arrival at the trip destination is represented by the complex event class `ArrivedAtDestination` holding attributes from its constituent subscribed event classes `FlightArrival`, `Flight`, `BusinessMeeting`, `FlightBooking`, and complex event class `MissedConnectingFlight`. The business traveler is considered to have arrived at his destination if all of his booked connecting flights took place and he did not miss any of them.

In our example the occurrence time of an `ArrivedAtDestination` event of a `FlightBooking` is the occurrence time of the `FlightArrival` event at the trip destination (not shown in Figure 2, but explained later in Example 4.3). In general, the occurrence time of a complex event is defined upon the occurrence times of its constituent events. The syntax of BICEPL, explained detailed in Subsection 4.1, provides for choosing *(i)* the occurrence time of a particular constituent event, or *(ii)* the maximum or minimum of the occurrence times of several constituent events. BICEPL allows also for defining a temporal offset relative to the occurrence time of a constituent event (such that one can define relative temporal events like `30MinutesBeforeFlightArrival`).

2.3. Application Setup

Taking the components described in Subsection 2.1, the setup of a PEACE application involves only the following four tasks:

(T1) *Event wrappers* in OXPATH, specifying web interactions leading to the events and the location of the attributes to be extracted per event. For sources outside the web, we use database triggers, e.g., to retrieve background information on a business meeting.

(T2) *Action wrappers* in OXPATH, performing some web interaction, such as posting a Twitter message. For actions beyond the web, service invocations and remote procedure calls are possible.

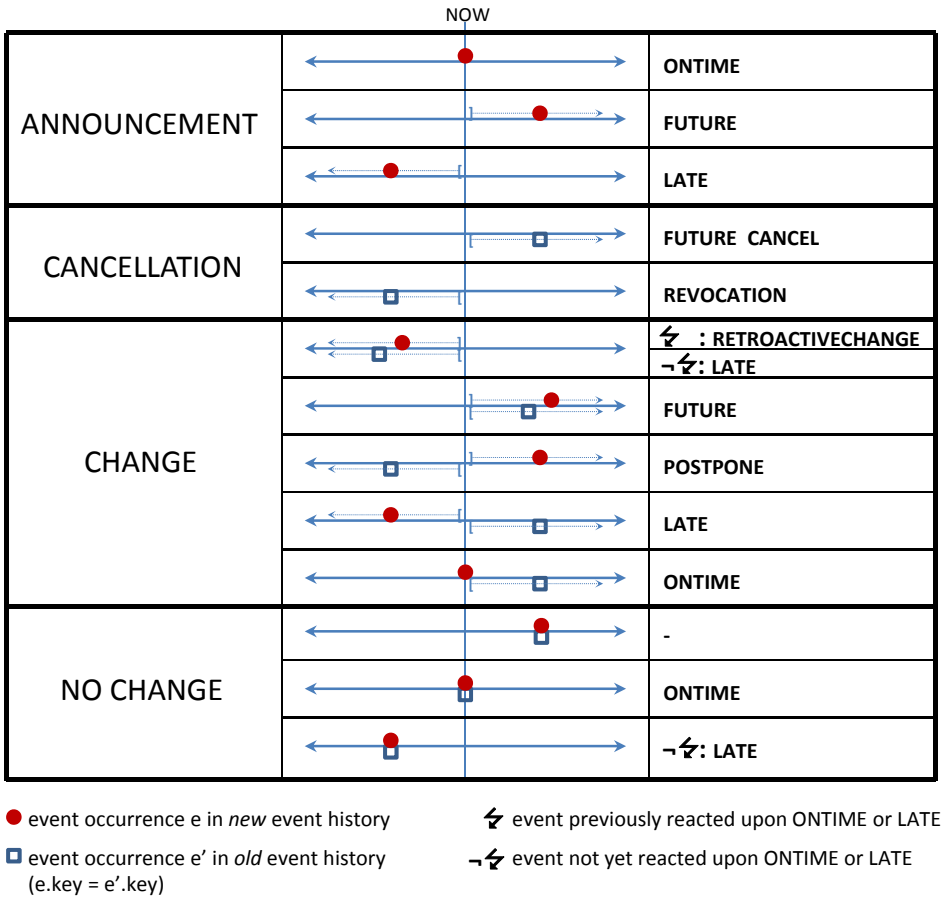


Fig. 4: Timing primitives

(T3) *Subscribed event classes* in BICEPL, specifying the schema of their events, capturing the event attributes extracted by the event wrappers, optionally with *condition-action statements*.

(T4) *Complex event classes* in BICEPL, specifying queries to aggregate subscribed events into complex events and *condition-action statements* applied on the arising events.

We discuss tasks **(T1-2)** in Section 3 on OXPATH, and tasks **(T3-4)** in Section 4 on BICEPL. In the respective sections, we give examples for all four tasks, all part of our running example on business travelers. In particular, we define **(T1)** the OXPATH event wrapper for flight arrivals in Example 3.1, discuss **(T2)** an OXPATH action wrapper for Twitter in Example 3.2, specify **(T3)** the subscribed event class for flight arrivals in Example 4.1, and define **(T4)** complex event classes for missing connecting flights and successful arrivals in Examples 4.2 and 4.3.

2.4. Specifying Conditional Actions with PEACE

In its core, PEACE allows to specify actions on how to react to events under the perfect system assumption. Events are immutable and known before or at the time they occur. The action is fired at the time the wall clock has advanced to the occurrence time of the

event. The event may be simple, i.e., correspond to a subscribed event or be a complex event such that the action can be conditional on a single subscribed event or several subscribed events. This behavior is defined by an ONTIME statement.

Additional statements handle various deviations from a perfect world or system, e.g., the event processor may have been down when it should have reacted to an event and can be late in processing it, the event may have been delivered late (i.e., only some time after it has occurred), or the event may be mutable from a business perspective such that event attributes or the occurrence time of an event changes after it has been announced or the event is retracted. This may happen pro- or retro-actively.

PEACE conceptually splits at any time its event repository into two event collections, one reflecting the last or old state of processing before new events (including cancellations) have been collected, the other one being the current or new state of processing. Each scenario can be characterized according to (1) whether the event is in the new collection, the old collection, or in both, (2) whether the occurrence time of the new event is in the past or future, (3) whether the occurrence time of the old event is in the past or future, and (4) whether the old and new event version are different or not.

Figure 4 gives an overview of the different constellations and the event publishing cases identified with them. Figure 4 classifies the possible event constellations on the left side according to whether the event is only in the new collection (ANNOUNCEMENT), only in the old collection (CANCELLATION), or in both collections but different (CHANGE), or in both collections and the same (NOCHANGE).

This classification can be further discriminated along the positions of the occurrence times of the old and new event version relative to NOW, the current reading of the wall-clock. Some of the cases give natural rise to handle them the same way and are thus collected under a common timing primitive for use as condition in condition-action statements, e.g., it is irrelevant if an event announced originally for the future is moved into the past or if the event is originally announced in the past; in both cases the point in time when the event occurred was missed (LATE).

We define the conditions of the different cases formally in Subsection 4.3, where we also explain how to avoid acting to some constellation multiple times by use of toggle fired.

EXAMPLE 2.1 (SPECIFYING CONDITIONAL ACTIONS). *Following up on our running example (Section 2), let us assume that the arrival of your flight is published for the first time at 10am with an expected arrival time of 2pm. This case would be matched by* ANNOUNCEMENT *and* FUTURE*, possibly triggering a notification informing your business partners. Later, at 1pm, when the arrival time is updated to 2:15pm,* CHANGE *is satisfied, triggering again a notification. At 2:15pm,* ONTIME *is satisfied and it is assumed the flight has finally arrived. In contrast, if the flight did actually not arrive at 2:15pm but is rescheduled thereafter for 3pm, then* POSTPONE *matches, since the event has already triggered some actions before and the occurrence time moves from the past into the future.*

3. EXTRACTING EVENT ANNOUNCEMENTS & EXECUTING ACTIONS IN OXPATH

For detecting events announcements and executing actions on the web, PEACE integrates OXPATH, a recent state-of-the-art language for highly efficient web automation and data extraction [Furche et al. 2013a]. OXPATH's main strengths lie in its ability to deal with modern scripted websites, supporting the interaction with complex AJAX-enabled forms, and its high scalability in handling even millions of pages and extracted results at ease. While a full description of OXPATH can be found in [Furche et al. 2013a], we only discuss the most relevant features of OXPATH, which boils down

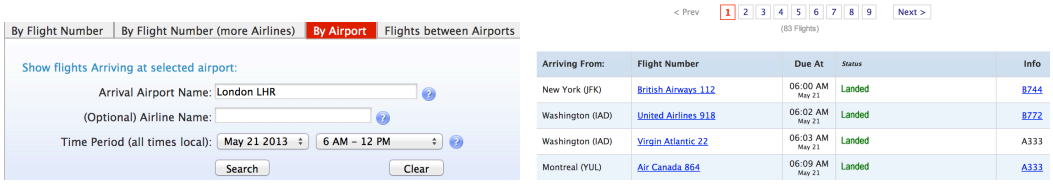


Fig. 5: Form and result page on www.flightarrivals.com.

to a four-fold extension of XPATH, which is executed over a live DOM taken from a browser:

(1) *Actions*, such as mouse events or typing, to simulate user interactions.
(2) *Kleene stars* to iterate, e.g., to access multiple pages within a paginated result.
(3) *Style axis* to query visual attributes to select, e.g., all elements colored green.
(4) *Extraction markers* to extract data from the DOM into (nested) records and attributes.

Aside its aptness for web automation, we have chosen OXPATH for integration within PEACE for a number of reasons: First, OXPATH provides a *single language* to facilitate both, event extraction and action execution. And since OXPATH is only slightly extending XPATH, many web developers are *easily capable* of manually writing new expressions or fixing inadequately generated expressions. The necessary XPATH expressions are quickly obtained with standard web developer tools, such as Firebug. Beyond manual editing, OXPATH provides an *entire ecosystem,* encompassing a visual IDE to obtain a wrapper from recorded web interactions and manually marked data to be extracted, an automated expression generator to induce XPATH expressions for matching a single node or a set of nodes from a given start node, as well as a scalable execution environment for distributed and parallel extraction and action execution. We have integrated these building blocks into PEACE, employing the visual IDE in the PEACE's client, and running the expression generator and execution environment within PEACE's server systems. We have reported about an early version of the visual IDE in [Kranzdorf et al. 2012]. Finally, the *loose coupling* between OXPATH and BICEPL allows for an independent execution of individual OXPATH queries arising in event extraction or action execution. Thus, PEACE exploits OXPATH's scalability well, offloading its most resource dependent tasks to background services, thereby also enabling mobile applications of PEACE. This is particularly important, as an overwhelming fraction of runtime is spent in accessing the web (>97%, see Section 6).

3.1. Detecting Events

To illustrate OXPATH's capabilities, we derive a wrapper for our running example. There are two steps in the construction of a wrapper for a specific event class: First, we define the navigation to the event announcements, and second, we specify how to map each of the event class attributes to HTML fragments of the event announcements. The resulting wrapper is employed by PEACE in polling the website repeatedly to extract the subscribed events fed to the complex event processor.

EXAMPLE 3.1 (OXPATH EVENT DETECTOR FOR FLIGHTARRIVAL EVENTS). *We extract flight arrivals from http://www.flightarrivals.com, shown in Figure 5. The left hand side shows the form reached after selecting the "By airport" tab, while the right hand side shows some of the results obtained after filling and submitting this form. Below we show the* OXPATH *wrapper extracting for a given airport all arrival times accessible on this site.*

```
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]:<occ=toUnixTime(.)>]
```

The wrapper consists of two parts, first navigating to the relevant data (Lines 1–4) and then extracting this data (Lines 5–8). After loading http://www.flightarrivals.com (Line 1), the wrapper clicks on the tab "Airport" and fills in the airport. This is done with OXPATH*'s* actions*, more specifically, a click action and a fill action taking variable* `airport` *as input (Line 2). Then, with the expression* `/following::field()[3]//option`*, the wrapper matches* all *possible arrival times and performs a select action on all of them – which means that the wrapper selects the corresponding options iteratively and executes the remaining part of the wrapper for each of them individually (Line 3). After form submission, the wrapper iterates through all paginated results with a* Kleene star *expression (Line 4), clicking zero or more times on the next button, before extracting data.*

Each result page reached contains a table with flight arrival entries in its rows. We turn all table rows, except the first one containing header information only, into `FlightArrival` *records, relying on* OXPATH*'s* extraction markers*. The first marker* `:<FlightArrival>` *creates a record for each table row except the first one (Line 5), which is then filled with attributes for departure location, flight number, flight day, and destination location. For example,* `:<fromLoc=string(.)>` *is the extraction marker for the departure location which is taken from the entire contents of the first column (Line 6).*

The construction of such a wrapper requires, except the schema for the data to be extracted, only the XPATH expressions for matching the form elements and the data to be extracted. These XPATH expressions are easily produced with standard web developer tools or the OXPATH IDE, as described above.

If a web site changes its structure, a new wrapper must be induced to update the old one. The new wrapper is typically generated with the OXPATH IDE again. However, using, e.g., DIADEM [Furche et al. 2014] and a suitable domain description, the wrapper is induced fully unsupervised. This enables PEACE to update its wrappers automatically once they stop delivering correct data.

3.2. Executing Actions

PEACE uses OXPATH wrappers not only for event detection but also for action execution: Such action wrappers fill forms and click buttons without extracting information, except for information such as confirmation or booking codes. An action wrapper is guaranteed to be executed from left to right and allows the execution of OXPATH actions only on single nodes. This contrasts with standard OXPATH [Furche et al. 2013a], which also performs actions on multiple nodes. We introduce this restriction to ensure that the OXPATH wrapper performs a single user interaction – clicking on multiple nodes with a single click-action would cause the execution to branch.

EXAMPLE 3.2 (OXPATH ACTION WRAPPER FOR POSTING ON TWITTER). *We continue our running example with the* OXPATH *action wrapper for posting Twitter messages.*

```
1 doc("https://mobile.twitter.com/session/new")
2   //input#username/{"user@example.com"}
3   /following::input#password/{"passme"}
4   //a[@title="Tweet"]/{click /}
5   //textarea[@class="tweetbox"]/{$p1}
6   /following::input[@value="Tweet"]/{click /}
```

The OXPATH *action wrapper loads the target web site (Line 1), fills in username and password (Lines 2-3), and submits the login form (Line 4). For Submitting the message, the wrapper fills in the message text (Line 5) and submits the message form (Line 6).*

Besides actions defined with OXPATH, PEACE allows for the integration of other actions by providing a plugin which implements the action executor interface.

4. EVENT PROCESSING WITH BICEPL

A PEACE systems works essentially in rounds consisting of three steps: First, new subscribed events are collected, second, these events are processed into complex events to determine the set of triggered actions, and third, these actions are executed. In this section, we focus on the second step, which is controlled by a BICEPL program defining subscribed and complex events, along with their condition-action statements. More specifically, BICEPL associates subscribed and complex event classes with a schema and optional condition-action statements, as discussed in the previous chapter. While subscribed events are produced by OXPATH wrappers, thus requiring in BICEPL only a schema declaration, complex events are defined with a query to compute the event instances from constituent events. Such queries are formulated as extended SQL select statements which aggregate constituent events into complex events.

To react on different changes of the observed events, BICEPL not only provides the current revision of a particular event, but also retains the preceding revision of each event (if any) for comparison. In BICEPL, condition-action statements may access these via keywords `NEW` and `OLD` and select relevant events with abbreviations for the timing predicates introduced in Subsection 2.4. However, provisioning preceding event revisions for comparison with all possible future updates requires PEACE to buffer events indefinitely. While such an approach leads to a most natural BICEPL semantics, the *buffering semantics,* it becomes quickly infeasible on systems with high event throughput. Thus, we provide a slightly weaker semantics, the *sliding window semantics,* which purges events when they have exceeded their life span, determined from a freezing time indicated with subscribed event classes and an observation span with complex event classes (which is explained later).

We give an informal high-level illustration of complex event detection in PEACE in the following, before we formally introduce syntax and semantics of PEACE in detail throughout this section.

Figure 6 illustrates the semantics of complex event processing based on the buffering semantics in which events are never purged. The illustration consists of events of three subscribed event classes s, c, and g, depicted by different shapes: square, circle, and triangle. Event instances of an event class are identified by the frame color, which is immutable and constitutes the key attribute of an event. Events are described by event attributes filling color and filling pattern. Two versions of a mutable event have the same frame color (denoting the value of the key attribute) and shape (denoting the class) but carry different detection time stamps. For simple reference to an event in textual representation, we use a unique event number, denoted as subscript to the event class name, as alternative key. Note: In our simple illustration, all event classes have the same key and descriptive attributes; this need not be so in general. The event number identifies a mutable event, but not a particular version of it (which would

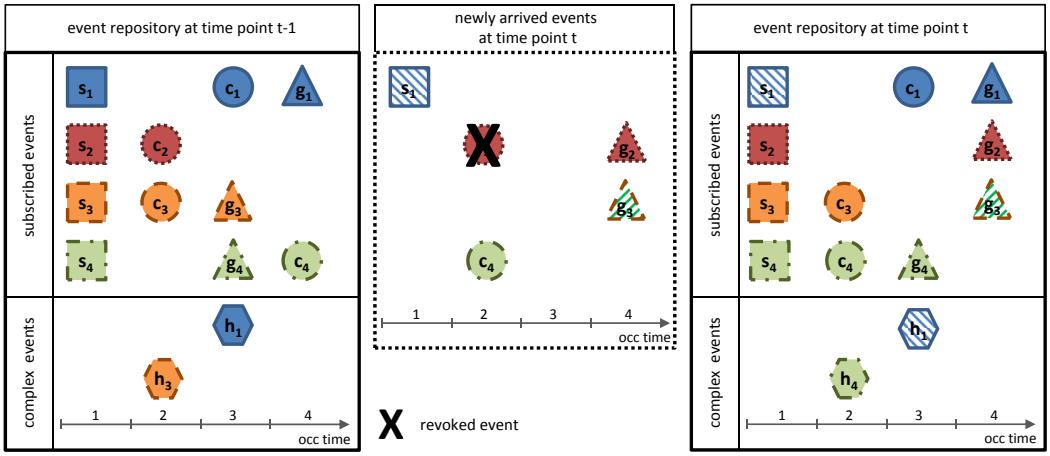


Fig. 6: Illustration of Complex Event Detection

be identified by the event number and its detection time). To improve readability on black and white printouts, events with the same frame color are depicted with a frame of the same line type (e.g., dotted, dashed). A complex event of complex event class h, depicted as hexagon, occurs if there are constituent events of event classes s, c, and g that strictly occur in this order and have the same frame color and same filling color. The composite event takes on the common frame and filling color; it receives its filling pattern from the event occurrence of s and its occurrence time from the event occurrence of c.

With the advancement of time, new subscribed events arrive, previously notified events are retracted, or event attributes or occurrence times of mutable events are updated. For each time chronon t we consider the event repository at time t. Relating to bi-temporal database systems, the event repository at transaction time t represents the valid time view of events as of transaction time t; the event repository at time t contains for each mutable event its current version at time t. The event repository at time t is split into the event repository of subscribed (or simple) events and the event repository for complex events. The event repository of complex events at time t can be derived from the event repository of subscribed events at time t. The event repository of subscribed events at chronon 0 is empty. For a time point $t > 0$, the event repository of subscribed events at time t can be built from the event repository of subscribed events at some previous point in time $t-1$ and the arrivals of subscribed events between $t-1$ and t. These time points are ideally two successive chronons, but need not be, if the event processor has been down (imperfect system).

Figure 6 shows event repositories at times $t-1$ and t. The event repository consists of subscribed events and of complex events. In event repository at time t, events s_1, c_1, and g_1 appear in that order and have the same frame color (blue, unbroken line) and the same filling color (blue). They are constituent events for complex event h_1 that takes its occurrence time from c_1 and its filling pattern from s_1. Likewise, events s_3, c_3, and g_3 are constituent events for complex event h_3. Note that complex events may be defined upon subscribed events and other complex events (as long as complex event definitions are not recursive), but this is not shown in the figure for simplicity.

At time point t, the following events arrive new, are updated or revoked: event g_2 arrives new, event c_2 is retracted, the filling patterns of mutable events s_1 and g_3 are updated to stripes, and the occurrence time of mutable event c_4 is updated such that

it occurs now between s_4 and g_4. These updates are depicted in the event repository named "newly arrived events at time point t", where an event retraction is denoted by showing a cross over the retracted event. The event repository of subscribed events at time t reflects these changes relative to event repository at time $t-1$. Complex events at time t are re-derived from the event repository of subscribed events at time t: Complex event h_1 receives a new filling pattern (stripes) as its constituent event s_1 changed its filling pattern to stripes; complex event h_3 is retracted since event g_3 changed its filling color to green and, hence, no longer matches with filling color orange of c_3 and g_3. Complex event h_4 is new to event repository at time t since events s_4, c_4, and g_4 with frame and filling color green appear now in the required sequence of occurrence times.

Once in a processing round complex events have been derived, for each event (subscribed or complex), found in either one or both repositories, condition-action statements defined for the event are evaluated. Each condition of such a statement is based on the timing primitives introduced previously in Figure 4.

Before we explain event processing with BICEPL in detail, we relate the use of occurrence time stamps of complex events in BICEPL to the use of interval timestamps by complex event processing languages for determining the "next" event in event processing [White et al. 2007]. Each complex event in BICEPL has a single occurrence time that indicates the wall-clock time at which the event is defined to occur and should be reacted upon. The occurrence time of a complex event is user-defined by a time expression over the occurrence-times of its constituent events (cf. previous Subsection 2.2 and syntax details in subsequent Subsection 4.1). Various event calculi support complex events that represent intervals, e.g., a 'purchase event' that starts with adding a first item to a shopping basket and ends with payment at check out. Different ways to choose a successor ("next") exist for intervals (cf. [White et al. 2007]). BICEPL does not provide an explicit "next" operator or event intervals. However, the semantics of "next" can be defined explicitly and, on a case to case bases, differently for each complex event by comparing (in an SQL query) the timestamps of the subscribed events in the event history relevant for detecting the complex event. Such an approach requires in general maintaining a complete event history (cf. [White et al. 2007]). Therefore, we initially define an idealized buffering semantics of BICEPL assuming a complete history of subscribed events (Subsection 4.4). As the idealized buffering semantics is impracticable for implementation in terms of memory requirements and performance, we thereafter (Subsection 4.5) introduce the sliding window semantics that purges subscribed events when their (user-defined) life span has exceeded and show that windowing and buffering semantics behave identically under certain reasonable conditions (Subsection 4.6).

4.1. Syntax of BICEPL

BICEPL's syntax in Figure 7 defines a program ⟨*program*⟩ as sequence of event class declarations, describing simple and complex event classes via ⟨*sclass*⟩ and ⟨*cclass*⟩, respectively.

We declare *subscribed event classes* ⟨*sclass*⟩ as mutable or immutable, with an event schema ⟨*schema*⟩, a ⟨*freezing time*⟩, and a sequence of condition-action statements ⟨*cond_action*⟩. The schema ⟨*schema*⟩ describes with ⟨*attributes*⟩ its typed attributes and with ⟨*key*⟩ the attribute subset forming the key of the class. The freezing time expresses how long a mutable event may change relative to the occurrence time given with its first version. The freezing time is given as ⟨*time_literal*⟩ which consists of a positive integer with a single character to indicate the time unit, referring to either seconds, minutes, hours, or days.

⟨*program*⟩ ::= { ⟨*sclass*⟩ | ⟨*cclass*⟩ }

⟨*sclass*⟩ ::= 'CREATE' ('MUTABLE' | 'IMMUTABLE') 'SUBSCRIBED EVENT CLASS' ⟨*schema*⟩
'FREEZING TIME' ⟨*time_literal*⟩ [⟨*cond_action*⟩ { ',' ⟨*cond_action*⟩ }] ';'

⟨*cclass*⟩ ::= 'CREATE' 'COMPLEX EVENT CLASS' ⟨*schema*⟩
'OBSERVATION SPAN' ⟨*time_literal*⟩ 'AS' ⟨*selection*⟩ [⟨*cond_action*⟩ { ',' ⟨*cond_action*⟩ }] ';'

⟨*schema*⟩ ::= ⟨*name*⟩ '(' ⟨*attributes*⟩ ')' 'ID' '(' ⟨*key*⟩ ')'
⟨*attributes*⟩ ::= ⟨*name*⟩ ⟨*type*⟩ { ',' ⟨*name*⟩ ⟨*type*⟩ }
⟨*key*⟩ ::= ⟨*name*⟩ { ',' ⟨*name*⟩ }

⟨*cond_action*⟩ ::= 'ON' ⟨*cond*⟩ 'DO' ⟨*action*⟩
⟨*cond*⟩ ::= ⟨*atom*⟩ | 'NOT' ⟨*cond*⟩ | ⟨*cond*⟩ 'AND' ⟨*cond*⟩ | ⟨*cond*⟩ 'OR' ⟨*cond*⟩
⟨*atom*⟩ ::= ⟨*value*⟩ ⟨*predicate*⟩ ⟨*value*⟩
| 'ANNOUNCEMENT' | 'CANCELLATION' | 'CHANGE' | 'FUTURECANCEL'
| 'ONTIME' | 'LATE' | 'LATE(' ⟨*min_delay*⟩ ',' ⟨*max_delay*⟩ ')'
| 'FUTURE' | 'RETROACTIVECHANGE' | 'REVOCATION' | 'POSTPONE'
⟨*value*⟩ ::= ('OLD.' | 'NEW.') ⟨*name*⟩ | ⟨*literal*⟩ | 'NOW'
⟨*action*⟩ ::= ⟨*name*⟩ '(' ⟨*value*⟩ { ',' ⟨*value*⟩ } ')'

⟨*selection*⟩ ::= 'SELECT' ⟨*select_clauses*⟩ 'OCCURRING AT' ⟨*time*⟩
⟨*time*⟩ ::= ⟨*table_ref*⟩ | ⟨*time*⟩ [('+' | '-') ⟨*time_literal*⟩]
| ('MAX' | 'MIN') '(' ⟨*time*⟩ { ',' ⟨*time*⟩ } ')'

⟨*time_literal*⟩ ::= ⟨*integer*⟩ ('s' | 'm' | 'h' | 'd')

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

Fig. 7: BICEPL Syntax.

Each *condition-action statement* contains a condition ⟨*cond*⟩ over the current and preceding version of an event and an action ⟨*action*⟩ to be performed if the condition holds. A condition is a Boolean combination of predicates where each predicate either **(1)** compares two values which may be attributes, belonging to the preceding or current event version, marked with `OLD` and `NEW`, respectively, literals, or the wall-clock time `NOW`; or **(2)** checks some timing condition, as described in Section 2.4. The new and old values in **(1)** only differ, if the event is updated at the current wall clock time, i.e., in case of a `CHANGE`. During an `ANNOUNCEMENT`, the old values, and during a `CANCELLATION`, the new values are null.

We declare *complex event classes* via ⟨*cclass*⟩, sharing with subscribed classes the schema, an ⟨*observation span*⟩, and condition-action statements, but also featuring a *selection statement* ⟨*selection*⟩. This ⟨*selection*⟩ is an SQL select statement referring to subscribed and complex classes freely, only avoiding circular dependencies. Our selection statements are extended with an `OCCURRING AT` clause to determine the event's occurrence time with a time expression ⟨*time*⟩. A time expression refers to the occurrence time of a constituent class via ⟨*table_ref*⟩ and may involve recursively min/max and increment/decrement computations. The observation span indicates how far the occurrence times of the constituent events of a complex event may be at most apart.

Examples 4.1–4.3 continue our running example and show event declarations with BICEPL. We declare the subscribed event class `FlightArrival` (Example 4.1), and complex event classes `MissedConnectingFlight` (Example 4.2), and

`ArrivedAtDestination` (Example 4.3). The latter declaration builds upon subscribed and complex event classes.

EXAMPLE 4.1 (SUBSCRIBED EVENT CLASS FLIGHTARRIVAL). *The* BICEPL *statement below declares the subscribed event class* `FlightArrival` *with its explicit attributes for flight day and number, and departure and destination location.* `FlightArrival` *is defined mutable (Line 1) since departure times or arrival locations may change, with* `flightDay` *and* `flightNo` *as key (Line 2), and a freezing time of two days (Line 3).*

```
1 CREATE MUTABLE SUBSCRIBED EVENT CLASS FlightArrival(flightDay TEXT,
    flightNo TEXT, fromLoc TEXT, toLoc TEXT)
2   ID (flightDay, flightNo)
3   FREEZING TIME (2d);
```

Since the running example's other subscribed event classes are very similar, we omit them here. Instead, we show two complex event classes, where the first one on missing connections serves as a constituent class for the second one on successful arrivals at the final destination.

EXAMPLE 4.2 (COMPLEX EVENT CLASS MISSEDCONNECTINGFLIGHT). *We detect connecting flights which are probably missed with the complex event class* `MissedConnectingFlight`.

```
1 CREATE COMPLEX EVENT CLASS MissedConnectingFlight(flightDay TEXT,
    flightNo TEXT, fromLoc TEXT, toLoc TEXT, bookingId TEXT, meetingId
    TEXT, timeToChange NUMBER)
2   ID (flightDay, flightNo, bookingId)
3   OBSERVATION SPAN (2d)
4   AS SELECT fd.flightDay, fd.flightNo, fd.fromLoc, fd.toLoc,
        fb.bookingId, bm.meetingId, (fd - fa) AS timeToChange
5       FROM FlightArrival fa, FlightDeparture fd, BusinessMeeting bm,
            Flight f, FlightBooking fb
6       WHERE fa.flightNo = f.flightNo AND fa.flightDay = f.flightDay AND
            bm.meetingId = fb.meetingId AND f.bookingId = fb.bookingId AND
            f.connFlgtNo = fd.flightNo AND f.connFlgtDay = fd.flightDay
            AND (fa + 30m) > fd
7       OCCURRING AT fd
8   ON ANNOUNCEMENT AND (NEW - 1d) < NOW DO
        bookNewFlights(NEW.meetingId),
9   ON ANNOUNCEMENT AND (NEW - 1d) >= NOW DO
        cancelMeeting(NEW.meetingId);
```

A `MissedConnectingFlight` *event features as attributes some attributes of the missed flight, the corresponding booking and meeting id, and the expected time available to reach this flight at the airport. A flight is included in this complex event class if this (possibly negative) time is below 30 minutes (Line 6). The occurring-at clause defines the occurrence time of a* `MissedConnectingFlight` *event as the missed flight's departure time (Line 7). To react appropriately to a missed connection, it is necessary to react immediately upon its detection: We execute action* `bookNewFlights` *with* `meetingId` *as parameter when a* `MissedConnectingFlight` *event is detected for the first time and more than one day in advance (Line 8). If the complex event is detected not more than 1 day in advance, then the meeting is canceled by calling the action* `cancelMeeting` *with parameter* `meetingId` *(Line 9).*

EXAMPLE 4.3 (COMPLEX EVENT CLASS ARRIVEDATDESTINATION). *We also report the successful arrivals at the final destination with the complex event class* `ArrivedAtDestination`.

```
1 CREATE COMPLEX EVENT CLASS ArrivedAtDestination(flightDay TEXT,
     flightNo TEXT, bookingId TEXT, meetingId TEXT, location TEXT)
2    ID(flightDay, flightNo, bookingId)
3    OBSERVATION SPAN (3d)
4    AS SELECT fa.flightDay, fa.flightNo, fb.bookingId, bm.meetingId,
         fa.toLoc AS location
5       FROM FlightArrival fa, BusinessMeeting bm, Flight f,
            FlightBooking fb
6       WHERE f.connFlgtNo = NULL AND fa.flightNo = f.flightNo AND
            fa.flightDay = f.flightDay AND bm.meetingId = fb.meetingId AND
            f.bookingId = fb.bookingId AND NOT EXISTS (SELECT * FROM
            MissedConnectingFlight mcf WHERE mcf.bookingId = f.bookingId)
7       OCCURRING AT fa
8    ON ONTIME OR LATE(0s, 1h) DO publishOnTwitter(
         format("Just arrived at %s!", NEW.location)),
9    ON LATE(1h, 1d) DO publishOnTwitter(
         format("Have arrived in %s at %t!", NEW.location, NEW.occ)),
10   ON CHANGE AND OLD.location <> NEW.location DO publishOnTwitter(
         format("Diverted from %s to %s!", OLD.location, NEW.location)),
11   ON POSTPONE DO publishOnTwitter(
         format("Not landed yet, expected to arrive at %t!", NEW.occ)),
12   ON RETROACTIVECHANGE DO publishOnTwitter(
         format("Revise, arrived in %s at %t!", NEW.location, NEW.occ));
```

This complex event class refers to subscribed classes, such as `FlightArrival`*, and the complex class* `MissedConnectingFlight`*, featuring attributes on flight, booking, and meeting, together with the location of the final arrival. An* `ArrivedAtDestination` *event is defined to occur at the arrival time of the final leg of a trip, which is a flight without further connecting flight, given that no connection of the booking has been missed. (Lines 4-7). Depending on the specific situation, the five condition-action statements of the class post different Twitter messages. The message is conveniently produced with function* `format`*, producing the message from a format string, referring to numeric value with* `%n`*, string value with* `%s`*, and time stamps with* `%t`*.*

— *On time or within less than 1 hour, we post that the traveler just arrived (Line 8).*
— *With more than one hour but with less than one day delay, we post that the traveler has arrived, together with the arrival time (Line 9).*
— *If the destination location changes, then the flight deviation is reported (Line 10).*
— *If the arrival is rescheduled from the past to the future, then the new expected arrival time is posted (Line 11).*
— *We also report retroactive changes, when an already reported arrival has in fact been diverted or delayed (Line 12).*

4.2. Event-Condition-Action Model underlying BICEPL

The event-condition-action model comes in three parts, first describing the properties common to both subscribed and complex event classes, then discussing the specifics of subscribed and complex events, and finishing with actions and their properties. We identify a point in time t with the number of *chronons* that passed between a reference time point and t. The length of a chronon is a system wide parameter that is fixed at system startup and remains constant until the system shuts down.

Event Classes. We identify 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. Further, each event schema contains attributes occ, occurrence time, and det, detection time, accessed via $e.\mathsf{occ}$ and $e.\mathsf{det}$, respectively. Each event schema contains a set of key attributes $E.\mathsf{key} \subseteq (E.\mathsf{schema} \setminus \{\mathsf{det}\})$. We denote with $e.\mathsf{key}$ the values of the key attributes in event $e \in E$. Note that the key of an event identifies the event but not its announcement, i.e., there may coexist announcements e and e' referring to the same event $e.\mathsf{key} = e'.\mathsf{key}$, detected at different times, i.e., with $e.\mathsf{det} \neq e'.\mathsf{det}$. An event class $E \in \mathbb{E}$ has a set of condition-action statements $E.\mathsf{ca}$. Each concrete condition-action statement $s \in E.\mathsf{ca}$ contains a condition $s.\mathsf{cond}$ and an action $s.\mathsf{action}$ which is performed when $s.\mathsf{cond}$ holds. This condition is evaluated as $s.\mathsf{cond}(t, e, e')$, where events e and e' refer to the event state at time points t and t' with $t > t'$, with t as the current wall clock time. The action is executed with the very same arguments as $s.\mathsf{cond}(t, e, e')$.

Subscribed and Complex Event Classes. Event classes in $\mathbb{E} = \mathbb{S} \cup \mathbb{C}$ are partitioned into subscribed and complex event classes $\mathbb{S}$ and $\mathbb{C}$. Subscribed event classes $S \in \mathbb{S}$ are associated with a wrapper $S.\mathsf{wrap}$ or a trigger expression. In our running example, we use wrappers for flight arrival and departure events, and database triggers for business meeting events. In contrast, complex event classes $C \in \mathbb{C}$ have a query function $C.\mathsf{query}$. For the set of C-events obtained by evaluating $C.\mathsf{query}$, we write $C.\mathsf{query}(O, D)$, where O is the set of observed subscribed events and D is the set of already derived complex events. To ensure a well-defined order for evaluating the functions $C.\mathsf{query}$ for $C \in \mathbb{C}$, we require *acyclic dependencies* between classes (and their functions): We denote with the *constituent event classes* of C all classes involved in $C.\mathsf{query}$, i.e., C depends on its constituent classes. 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 direct constituent class of $C \in \mathbb{C}$, we write $E \subset C$; we write $E \subset^+ C$ if E is direct or indirect constituent class of C. $\subset^+$ is transitive by definition. We denote by $\subset^*$ the transitive and reflexive cover of $\subset$. Additionally, we require $\subset^+$ to be irreflexive ($E \not\subset^+ E$ for all $E \in \mathbb{E}$) and asymmetric ($E \subset^+ E'$ implies $E' \not\subset^+ E$ for all $E, E' \in \mathbb{E}$). Thus, we rule out cyclic dependencies, as we required. We also write $e \subset c$ for concrete event instances e and c, if $e.\mathsf{class} \subset c.\mathsf{class}$.

Action Classes. Next to event classes, each BICEPL program declares action classes $\mathbb{A}$. Every action $a \in \{p.\mathsf{action} \mid p \in E.\mathsf{ca} \text{ and } E \in \mathbb{E}\}$ belongs to exactly one action class $a.\mathsf{class} = A \in \mathbb{A}$. The schema $A.\mathsf{schema}$ of an action class A contains a set of typed attributes, allowing for integers, floats, and strings. Each concrete action a with $a.\mathsf{class} = A$ must match its schema $A.\mathsf{schema}$, i.e., having a correctly typed attribute $a.attr$ for all $a.attr \in A.\mathsf{schema}$. Moreover, each action class has an action wrapper $A.\mathsf{exec}$ which is a parameterized OXPATH wrapper with $A.\mathsf{schema}$ as invocation signature.

4.3. Mapping BICEPL into its Event-Condition-Action Model

We map BICEPL class definitions into the event classes of our model from the preceding section. Given a BICEPL event class, we obtain an event class $E \in \mathbb{E}$ with schema attributes $E.\mathsf{schema}$ and key $E.\mathsf{key}$ directly from the class declaration and the `ID` clause of the event class. It remains to add for general event classes $E \in \mathbb{E} = \mathbb{S} \cup \mathbb{C}$ the condition-action statements in $E.\mathsf{ca}$, and for complex event classes $C \in \mathbb{C}$, the query functions $C.\mathsf{query}$.

SQL-Query Rewriting. Given a complex event class in BICEPL, we rewrite its query into standard SQL and use the resulting query to define C.query. We obtain this query by rewriting the select clause in the BICEPL class with the following three steps.

(1) We expand all table references in time expressions to access the implicit occurrence time, e.g., 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 occ ...`, defining the occurrence time as first attribute in the newly created event.

EXAMPLE 4.4 (SQL-QUERY REWRITING). *We rewrite the query of Example 4.3, not showing the unchanged* `FROM` *and* `WHERE` *clauses. The projection of the original select statement is extended with occurrence, where the occurrence time refers to the occurrence of the constituent* `FlightArrival` *event.*

```
1  SELECT fa.occ AS occ,
2   fa.flightDay AS flightDay, fa.flightNo AS flightNo, fb.bookingId AS
       bookingId, bm.meetingId AS meetingId, fa.toLoc AS location
3    FROM ...  WHERE ...
```

Condition-Action Rewriting. Each BICEPL condition-action statement translates into one condition-action statement $ca \in E$.ca with ca.cond and ca.action. To obtain ca.cond, we first rewrite the timing primitives in the condition-action statements of the BICEPL classes as follows. In this rewriting, we express the timing primitives in terms of `OLD` and `NEW` which allow to access the preceding and current revisions of the event in concern. For simplicity, `OLD <> NEW` stands for comparison of all attributes of the new and old event version.

(1)	`ANNOUNCEMENT`	`(NEW.key IS NOT NULL AND OLD.key IS NULL)`
(2)	`CANCELLATION`	`(NEW.key IS NULL AND OLD.key IS NOT NULL)`
(3)	`CHANGE`	`(NEW <> OLD)`
(4)	`ONTIME`	`(NEW.occ = NOW)`
(5a)	`LATE`	`(NEW.occ < NOW AND NOT FIRED)`
(5b)	`LATE(min,max)`	`(NOW - NEW.occ)> min AND (NOW - NEW.occ)<= max AND NOT FIRED`
(6)	`FUTURE`	`(NEW.occ > NOW AND OLD.key IS NULL)OR (NEW <> OLD AND NEW.occ > NOW AND OLD.occ > NOW)`
(7)	`FUTURE CANCEL`	`(NEW.key IS NULL AND OLD.occ > NOW)`
(8)	`RETROACTIVECHANGE`	`(NEW <> OLD AND OLD.occ < NOW AND NEW.occ < NOW AND FIRED)`
(9)	`POSTPONE`	`(OLD.occ < NOW AND NEW.occ > NOW)`
(10)	`REVOCATION`	`(NEW.key IS NULL AND OLD.occ < NOW)`

Then, condition ca.cond(t, e, e') is evaluated by mapping `NOW` to t and `NEW` to e, `OLD` to e' and `FIRED` to $e \in$ fired. Note: As a side-effect of `ONTIME` or `LATE` having been evaluated to true, the event is entered into a *fired* list, and as a side effect of the condition for `CANCELLATION` or for `POSTPONE` evaluating to true it is removed from this list. The toggle fired is used to discriminate between a "late processing" situation (in which an event was not handled on-time due to the event processor being unavailable at that time)

and a "retro-active change" situation; it is also used to avoid reacting multiple times to a "late processing" situation, which, different to other situations, may appear as such in consecutive processing cycles.

Similarly, the action $ca.\mathsf{action}(t, e, e')$ yields a tuple, consisting of the procedure name and the arguments obtained by evaluating the argument expressions using the same mapping as for $ca.\mathsf{cond}$.

With these rules, we turn each condition-action statement into an SQL query computing the actions triggered by this statement. Since condition-action statements depend on the events known at the current and previous processing iteration, accessed via `NEW` and `OLD`, we employ for each event class two corresponding tables.

EXAMPLE 4.5 (CONDITION-ACTION REWRITING). *This example shows how the condition-action statements for* `ArrivedAtDestination` *of Example 4.3 are rewritten, relying on tables* `new_aad` *and* `old_aad` *for the events known at the current and previous processing iteration. We obtain* `new_aad` *as view with the query of Example 4.4 and update latter from this view after each processing step. We show the rewriting of the* `POSTPONE` *statement of Example 4.3 below.*

```
1 SELECT new_aad.occ
2   WHERE old_aad.occ < NOW AND new_aad.occ > NOW
3     AND old_aad.flightNo = new_aad.flightNo AND old_aad.flightDay =
        new_aad.flightDay AND old_aad.bookingID = new_aad.bookingID;
```

The occurrence time is the only attribute parameterizing the action `publishOnTwitter`*, as format strings are dealt with later during wrapper invocation. Thus, the generated action tuple contains only the occurrence time (Line 1). Following rule* ***(8)****, we rewrite the* `POSTPONE` *predicate and extended the resulting* `WHERE` *clause with join conditions to match the event keys (Line 3).*

4.4. Buffering 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. Depending on the changing observations and wall clock time, $\mathbb{E}$ triggers the execution of some actions. 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}), resulting in a set of action tuples at each time instant 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 set $O_i = \{e \in O_{i-1} \mid \nexists e' \in \Delta_i \text{ and } e.\mathsf{key} = e'.\mathsf{key}\} \cup \Delta_i^+ \setminus \Delta_i^-$ for $i > 0$, where $\Delta_i = \Delta_i^+ \cup \Delta_i^-$ contains the subscribed events observed (Δ_i^+) or retracted (Δ_i^-) between t_{i-1} and t_i. Note, we assume that at each subscription cycle, the event detector of an event class will provide at most one event version for each mutable event, if an event version is provided it is the most recent one. The detection time of a subscribed event is the time the event is collected from the event detector and entered into the event repository of the bi-temporal event processor (and not the time the event version is detected by the event detector).

Events of complex event classes are defined by queries upon subscribed event classes, although possible indirectly, like views over views over base tables in a relational database without recursion. Thus each complex event class in the end depends in one processing cycle only on subscribed event classes and the order in which complex events are processed is irrelevant, as long as the queries for constituent event classes of a complex event class are evaluated before the query of the complex event class.

To obtain the corresponding complex event, we fix an ordering $\mathbb{C} = \{C^1 \dots C^l\}$ with $C^k \not\subset C^j$ for all $j < k$, which must exist, since $\subset$ is irreflexive and asymmetric. Then, we derive the complex events D_i at time instant t_i with

$$D_i = \mathsf{derive}(O_i, t_i) = D_i^l \text{ with } D_i^0 = O_i \text{ and } D_i^j = D_i^{j-1} \cup C^j.\mathsf{query}(O_i, D_i^{j-1}) \ .$$

Depending on the differences between D_i and D_{i-1}, program $\mathbb{E}$ triggers a set of actions, as specified in the condition-action statements of the event classes in $\mathbb{E}$. More specifically, we check the condition-action statements for each distinct event id found in D_i or D_{i-1}, i.e., the set of relevant ids I is given with $I = \mathsf{id}(D_i) \cup \mathsf{id}(D_{i-1})$ for $\mathsf{id}(D_i) = \{\langle e.\mathsf{key}, e.\mathsf{class}\rangle \mid e \in D_i\}$. To access the unique event $e_i \in D_i$ with $e_i.\mathsf{key} = x.\mathsf{key}$ and $e_i.\mathsf{class} = x.\mathsf{class}$, we write $D_i[x] = e_i$. If no such event exists, we set $D_i[x] = \bot$. Then, the condition-action statements for event key $x \in I$ are found in $D_i[x].\mathsf{class.ca}$ or $D_{i-1}[x].\mathsf{class.ca}$ (if both exist, the statements agree, as $D_{i-1}[x].\mathsf{class} = D_i[x].\mathsf{class}$ holds), and thus, the triggered actions at time t_i are

$$\mathsf{triggered}(D_i, D_{i-1}) = \bigcup_{x \in I} \{\ ac.\mathsf{action}(t_i, D_i[x], D_{i-1}[x]) \ \mid\ ac \in (D_i[x].\mathsf{class.ca} \cup D_{i-1}[x].\mathsf{class.ca}) \text{ and } ac.\mathsf{cond}(t_i, D_i[x], D_{i-1}[x])\ \}$$

At last, $\mathsf{triggered}(D_i, D_{i-1})$ has a side effect: For each $ac.\mathsf{cond}(t_i, D_i[x], D_{i-1}[x])$, if ac = `ONTIME` or ac = `LATE` evaluated to true, we set $\mathsf{fired} = \mathsf{fired} \cup \{x\}$, if ac = `CANCELLATION` or ac <> `POSTPONE` evaluated to true, we set $\mathsf{fired} = \mathsf{fired} \setminus \{x\}$.

Taking these pieces together, we arrive at an idealized semantics, employing unlimited buffering, as defined next.

Definition 4.6 (*Buffering Semantics*). Given the sequence of subscribed events $O_0, O_1, \dots$ arising at time instants $t_0, t_1, \dots$, we define the *buffering semantics* at instant t_i

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

EXAMPLE 4.7 (EVENT REPOSITORY AND BUFFERING SEMANTICS). *To continue our running example, we* ***(1)*** *derive from the event histories O_{i-1} and O_i on subscribed events (Table I) the corresponding derived histories D_{i-1} and D_i (Table II), and* ***(2)*** *determine the triggered actions.*

To compute the complex events, we need to fix an order $\mathbb{C} = \{C^1 \dots C^l\}$ of the complex event classes $\mathbb{C}$. In this case, `MissedConnectingFlight` *must be evaluated before* `ArrivedAtDestination`*, since the former is a constituent class of the latter. We initialize D_{i-1}^0 with $D_{i-1}^0 = O_{i-1}$ and derive events of class C^1 =*`MissedConnectingFlight` *by evaluating $C^1.\mathsf{query}(O_{i-1}, D_{i-1}^0)$, resulting in the empty set, thus $D_{i-1}^1 = D_{i-1}^0$. Second, we derive event of class C^2 =*`ArrivedAtDestination`*, yielding two events (Table II) to be added to $D_{i-1}^2 = D_{i-1}$. D_i is derived analogously.*

Based on D_{i-1} and D_i, the condition-action statements `cond` *of our complex event classes may trigger actions. In this case, the only triggered statement is the* `POSTPONE` *case of class* `ArrivedAtDestination`*, since at 19:00 the* `ONTIME` *statement has fired and needs to be corrected now, as the flight turns out be delayed until 19:30.*

Note, the definition of the buffering semantics does not address possible implementations and potential performance improvements. E.g., it is not necessary to evaluate action-conditions for events at time points where no event occurs. Temporal indexes on the occurrence times of events can be introduced to skip chronons for which it is known that no event occurs. Also, in our implementation, we do not maintain the sequence of subscribed event sets $O_0, \dots, O_i$ or derive O_i from O_{i-1}. We keep a repository R of subscribed events, recording different versions of mutable events with different

FlightArrival at t_{i-1}

flightDay	flightNo	occ	det	fromLoc	toLoc
10.01.	AF1381	10.01.07:00	10.01-07:00	London	Paris
10.01.	AF1780	10.01.19:00	10.01-18:50	Paris	London

FlightArrival at t_i

flightDay	flightNo	occ	det	fromLoc	toLoc
10.01.	AF1381	10.01.07:00	10.01.07:00	London	Paris
10.01.	AF1780	**10.01.19:30**	**10.01.19:01**	Paris	London

FlightDeparture at t_{i-1} and t_i

flightDay	flightNo	occ	det	fromLoc	toLoc
10.01.	AF1381	10.01.05:30	10.01.05:30	London	Paris
10.01.	AF1780	10.01.17:00	10.01.17:00	Paris	London

BusinessMeeting at t_{i-1} and t_i

meetingID	occ	det	duration	location	contact
m1	10.01.08:00	08.01.10:15	1h	Paris	Ms. Martin
m2	10.01.20:00	08.01.10:15	1h	London	Mr. Smith

Flight at t_{i-1} and t_i

flightDay	flightNo	occ	det	bookingId	...
10.01.	AF1381	10.01.05:30	08.01.10:15	b1	...
10.01.	AF1780	10.01.17:00	08.01.10:15	b2	...

FlightBooking at t_{i-1} and t_i

bookingId	occ	det	company	meetingId
b1	08.01.10:00	08.01.10:15	Travel Star	m1
b2	08.01.10:10	08.01.10:15	Travel Planet	m2

Table I: Event repositories O_{i-1} and O_i for subscribed event classes at t_{i-1} =19:00 and t_i =19:01

	flightDay	flightNo	occ	det	bookingId	meetingId	location	fired
t_{i-1}	10.01.	AF1381	10.01.07:00	10.01.07:00	b1	m1	Paris	true
	10.01.	AF1780	10.01.19:00	10.01.18:50	b2	m2	London	true

	flightDay	flightNo	occ	det	bookingId	meetingId	location	fired
t_i	10.01.	AF1381	10.01.07:00	10.01.07:00	b1	m1	Paris	true
	10.01.	AF1780	**10.01.19:30**	**10.01.19:01**	b2	m2	London	true

Table II: Event repositories D_{i-1} and D_i for ArrivedAtDestination at t_{i-1} =19:00 and t_i =19:01

detection times and, if applicable, retractions of mutable events. For a given chronon i the set O_i is given as view over R, where O_i contains for each mutable event the most recent version up to and inclusive i (i.e., the version with the most recent detection time stamp less or equal i, or no event, if the most recent entry up to and inclusive i denotes a retraction). This approach simplifies coping with system outages or delays and permits to provide a historic AS OF AT TIME i VIEW at any time, a feature our systems shares with bi-temporal database systems. Further, events can be always loaded in parallel into the event repository. The event processor, after being down or having finished a processing cycle late, resumes at time point i that corresponds to NOW, irrespectively how far back time point $i-1$ of its last processing cycle was. Timely reactions to event occurrence may have been missed in between, but event processing

will properly resume according to the condition-action rules specified, which, e.g., will lead to apply the `LATE` action rather than the `ONTIME` action.

4.5. Sliding Window Semantics

As a more practical semantics, we introduce the *sliding window semantics* which sets a life span for subscribed events and purges them when their life span has expired.

The expiration time of an event is derived from its projected occurrence-time at inception inceptOcc, i.e., when its first version is detected, and upon the life span associated with its class. The occurrence time at inception is passed on for an event e from O_{i-1} to O_i such that we can simply write $e.\mathsf{inceptOcc}$ instead of $O_i[e].\mathsf{inceptOcc}$, and we have likewise, $e.\mathsf{expiration} = e.\mathsf{inceptOcc} + e.\mathsf{class.lifespan}$. Then we purge events with $\mathsf{purged}(O,t) = \{e \in O \mid e.\mathsf{expiration} \geq t\}$, keeping only unexpired events. Finally, we need apply the same time stamp t_{i-1} in purging both O_i and O_{i-1}. Otherwise, if we would use t_i and t_{i-1}, we would could purge different events from O_i and O_{i-1}, potentially causing cancelation events which only occur because of the purging.

Definition 4.8 (*Sliding Window Semantics*). Given a sequence of subscribed events $O_0, O_1, \ldots$ for time instants $t_0, t_1, \ldots$, we define the *sliding window semantics* at instant t_i

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

There are two strategies in setting the lifespan of a subscribed event class: (1) It may be set by the designer and subscribed events are no longer considered for action triggering or complex event detection once they have expired. This choice has been taken by others, e.g., CEDR [Barga et al. 2007; Goldstein et al. 2007] by defining a valid period for events (after which they are no longer considered for event detection), but has the drawback that events may be purged prematurely leading to a result different to the buffering semantics, which keeps an accumulated history of events. (2) It may be determined and set according to the semantics definition by the event processor based on system guarantees (e.g., an assertion on the maximum latency of retro-active event arrivals) and domain guarantees (e.g., an assertion on how much the occurrence time might differ in later event versions from the initially announced occurrence time or how much time might pass between the occurrences of constituent events of a complex event).

We have chosen the second, more user friendly approach and set a freezingTime for subscribed and an observationSpan for complex event classes. Based on a practical set of assertions on our queries and the occurrence of subscribed events, we derive the life span of subscribed event classes.

Under the perfectness assumption, events are not mutable and they are known before or at the time they occur. We relax the perfectness assumption not freely, but within boundaries. Events may be detected late due to network or communication delays. The latency represents a *system guarantee* on the maximal latency of retro-active event detection: $e.\mathsf{det} - e.\mathsf{inceptOcc} \leq e.\mathsf{class.latency}$.

Mutable events may change in the future. But there will be eventually some point in time, when the event no longer changes. The mutabilitySpan represents a service guarantee indicating how long, relative to the occurrence time at its inception (inceptOcc), event attributes or the occurrence time itself may change: $O_{i-1}[e] \neq O_i[e] \Rightarrow (O_i[e].\mathsf{det} \leq e.\mathsf{inceptOcc} + e.\mathsf{class.mutabilitySpan})$. Mutable events may not only change their attributes but also their occurrence times. The tempvariabilitySpan represents a domain guarantee on how far the occurrence time of a mutable event may be moved forward or backward in the occurrence time dimension with regard to its occurrence

time at inception, e.g., how much earlier or later a flight arrival may occur relative to its first announcement: $|e.\mathsf{inceptOcc} - e.\mathsf{occ}| \leq e.\mathsf{class.tempvariabilitySpan}$.

To avoid maintaining three time spans we use only a freezingTime and set mutabilitySpan = latency = tempvariabilitySpan = freezingTime. We call an event *frozen* after it may not mutate any more in the future and we call this point in time freezeTime of the event: $e.\mathsf{freezeTime} = e.\mathsf{inceptOcc} + e.\mathsf{class.freezingTime}$.

For complex events, we introduce a domain guarantee, the observationSpan, indicating in the occurrence time dimension the maximum spread of the occurrence times of constituent events that make up the complex event. E.g., for a missed flight connection, the observationSpan expresses how far the arrival event and the departure event may be at most apart (otherwise it is a stop over and one would definitely catch the next flight leg). The observationSpan assertion is satisfied for a complex event c, if $\max\{e.\mathsf{occ} \mid e \subset c\} - \min\{e.\mathsf{occ} \mid e \subset c\} \leq c.\mathsf{class.observationSpan}$.

We now explain how the lifespan of subscribed event classes is determined from the freezingTime of subscribed event classes and the observationSpan of complex event classes. A complex event changes if one of its subscribed events changes. At the schema level, we derive the freezing time of a complex event class based on the freezing times of its direct or indirect constituent event classes: $C.\mathsf{freezingTime} = max\{E.\mathsf{freezingTime} \mid E \subset^+ C\}$.

We define the spread of a complex event class C as an upper bound on how far for a complex event of C, the occurrence times of the subscribed events at the leaves of the event composition tree and the occurrence time of the complex event are at most apart. For each subscribed event class E at the bottom of the complex event tree, we have $E.\mathsf{spread} = 0$. For an event of complex event class C, its direct constituent events may be at most $C.\mathsf{observationSpan}$ apart. Considering the occurrence times of indirect constituent events, this interval may be extended at its beginning and at its end by at most the maximum of the spread of its constituent events. Further, the occurrence time of the complex event may be shifted in the `OCCURRING AT` clause by an offset relative to the occurrence time of some its constituent events. Thus we have: $C.\mathsf{spread} = 2\max\{E.\mathsf{spread} \mid E \subset C\} + C.\mathsf{observationSpan} + C.\mathsf{offset}$, where offset denotes the future or back shift of events of a complex event relative to some event of its direct constituent event classes.

Based on spread which is defined on the occurrence time of most recent event versions, we define inceptSpread as a corresponding upper bound on the occurrence times inceptOcc when mutable events are first detected that will later form a complex event. If the occurrence time of an event is not updated outside its freezingTime ($|e.\mathsf{inceptOcc} - O_i[e].\mathsf{occ}| \leq e.\mathsf{class.freezingTime}$), the inceptOcc of the subscribed events of a complex event of C may stretch at most by the maximum freezingTime of its constituent events, $C.\mathsf{freezingTime}$, into the past and into the future of spread. Thus, we have $C.\mathsf{inceptSpread} = C.\mathsf{spread} + 2C.\mathsf{freezingTime}$.

Subscribed events must be kept at least as long as (1) needed to participate in the formation of any new complex event and (2) once detected, to detect any new versions of the complex event if one of its constituent events changes. Thus, we set $E.\mathsf{lifespan} = E.\mathsf{maxFreezeTime} + E.\mathsf{maxInceptSpread}$, with $E.\mathsf{maxFreezeTime} = \max\{C.\mathsf{freezingTime} \mid E \subset^* C\}$ and $E.\mathsf{maxInceptSpread} = \max(\{C.\mathsf{inceptSpread} \mid E \subset^+ C\} \cup \{0\})$.

4.6. Semantic Equivalence

While the buffering semantics will always produce intuitive behavior as it maintains an accumulated event history, the sliding windows semantics may lead to non-intuitive behavior for cases in which events are purged prematurely.

The buffering and windowing semantics behave identically, if the program $\mathbb{E}$ and the subscribed events $O_0, O_1, \ldots$ fed at time instants $t_0, t_1, \ldots$ to $\mathbb{E}$ satisfy the following sanity conditions:

(P) $\mathbb{E}$ is *m*onotone, i.e., (i) $D = \mathsf{derive}(O,t) \subset D' = \mathsf{derive}(O',t)$ for all $O \subseteq O'$ and (ii) $e \in D, e' \in D', e.key = e'.key \Rightarrow e = e'$.

(E1) Each subscribed event is detected with an occurrence time within its freeze time (*latency guarantee*), i.e., $e.\mathsf{det} - e.\mathsf{inceptOcc} \leq e.\mathsf{class.freezingTime}$.

(E2) Events are not modified or cancelled beyond their freeze time (*mutability guarantee*), i.e., $O_{i-1}[e] \neq O_i[e] \Rightarrow (t_i - e.\mathsf{inceptOcc} \leq e.\mathsf{class.freezingTime})$ where t_i is the time events of O_i are fed into ECs.

(E3) Events do not update their occurrence time to values outside their freeze time (*occ-time variability guarantee*), i.e, $|e.\mathsf{inceptOcc} - e.\mathsf{occ}| \leq e.\mathsf{class.freezingTime}$.

(E4) Constituent events of a complex event spread at most across the observation span of the complex event class (*observation span guarantee*), i.e., $\max\{e.\mathsf{occ} \mid e \subset c\} - \min\{e.\mathsf{occ} \mid e \subset c\} \leq c.\mathsf{class.observationSpan}$.

Meeting condition **(P)** should be easy to achieve in most scenarios. $\mathbb{E}$ is monotone if (i) the composition of event queries does not directly or indirectly include negation (or all quantification) over events of a subscribed event class and (ii) the value of event attributes of a complex event are calculated only from events whose key is subsumed by the schema of the complex event. Applications in web-based settings, which we have in mind, frequently apply the open world assumption and do not support the closed world semantics of negation as failure (as known for Prolog or Datalog and supported by BICEPL through using SQL). But we do have examples where negation as failure is practicable and we show how our approach can be extended to non-monotone programs. The keys of constituent events can be included into the schema of the complex event during event class design. Note that a sufficient and easy check for $\mathbb{E}$ to not include negation over subscribed events is that each $\mathbb{E}$ contains only monotone queries $(C.\mathsf{query}(O,D) \subseteq C.\mathsf{query}(O',D')$ for all $O \subseteq O', D \subseteq D'$, and $C \in \mathbb{C}$.

Conditions **(E1)** to **(E4)** can be typically met by identifying proper values for freezingTime and observationSpan during system design based on service guarantees derived from service level agreements or a domain analysis.

Notice that, while the sliding window semantics and the buffering semantics show the same observable behavior if the above sanity conditions are met, different deviations from the perfectness assumption can effect the observable behavior of triggered actions. Different late arrivals of subscribed events or different down times of the event processor may lead to different actions to be triggered as events may, for example, be processed according to the `LATE` statement rather than by the `ONTIME` statement associated with an event. But exactly this is desired!

THEOREM 4.9 (SEMANTIC EQUIVALENCE). *If program* $\mathbb{E}$ *and events* $O_0, O_1, \ldots$ *satisfy* ***(P, E1-4)****, then* $[\![\mathbb{E}]\!]_{\mathsf{purged}}(O_i, t_i, O_{i-1}, t_{i-1}) = [\![\mathbb{E}]\!](O_i, t_i, O_{i-1}, t_{i-1})$ *holds.*

PROOF. First, we rewrite the theorem statement: Following the notation in Section 4, we have $[\![\mathbb{E}]\!](O_i, t_i, O_{i-1}, t_{i-1}) = \mathsf{triggered}(D_i, D_{i-1})$. Analogously, for the purged case, with $O_i^P = \mathsf{purged}(O_i, t_{i-1})$ and $O_{i-1}^P = \mathsf{purged}(O_{i-1}, t_{i-1})$, and $D_i^P = \mathsf{derive}(O_i^P, t_i)$ and $D_{i-1}^P = \mathsf{derive}(O_i^P, t_{i-1})$, we obtain $[\![\mathbb{E}]\!]_{\mathsf{purged}}(O_i, t_i, O_{i-1}, t_{i-1}) = \mathsf{triggered}(D_i^P, D_{i-1}^P)$. Thus, we rewrite the theorem claim as

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

To show the theorem, we show **(I)** $\mathsf{triggered}(D_i^P, D_{i-1}^P) \subset \mathsf{triggered}(D_i, D_{i-1})$ and **(II)** $\mathsf{triggered}(D_i, D_{i-1}) \subset \mathsf{triggered}(D_i^P, D_{i-1}^P)$.

(I): Since **(P)** is monotone, the deletion of any subscribed event e in O does (i) not introduce any new complex event c in D^P that is not already in D and (ii) does not alter in D^P any complex event c existing in D. Thus, for any event pair $(c, c') \in (D_i^P, D_{i-1}^P)$ we also have $(c, c') \in (D_i, D_{i-1})$ such that $[\![\mathbb{E}]\!]_{\mathsf{purged}} \subseteq [\![\mathbb{E}]\!]$ holds.

(II): We distinguish newly announced, revoked, changing, 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 only after the event has been fired by `ONTIME` or `LATE` (cf. Figure 4).

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'. To detect c' all other constituent events must still be alive:

$$\begin{array}{ll} f.\mathsf{expiration} = f.\mathsf{inceptOcc} + f.\mathsf{class.maxFreezeTime} + f.\mathsf{class.maxInceptSpread} & \geq_1 \\ f.\mathsf{inceptOcc} + c'.\mathsf{class.freezingTime} + c'.\mathsf{class.inceptSpread} & \geq_2 \\ f.\mathsf{inceptOcc} + e.\mathsf{class.freezingTime} + c'.\mathsf{class.inceptSpread} & \geq_3 \\ f.\mathsf{inceptOcc} + e.\mathsf{det} - e.\mathsf{inceptOcc} + c'.\mathsf{class.inceptSpread} & \geq_4 \\ e.\mathsf{det}. & \end{array}$$

Inequality (1) holds by definition of maxFreezeTime and maxInceptSpread, inequality (2) by definition of freezeTime for C; inequality (3) holds because of **(E1)**, requiring event e' being detected within its freezing time $e.\mathsf{classfreezingTime}$. Inequality (4) holds, since conditions **(E3)** and **(E4)** guarantee $|f.\mathsf{inceptOcc} - e.\mathsf{inceptOcc}| \leq c'.\mathsf{class.inceptSpread}$, as **(E4)** limits $|f.\mathsf{occ} - e.\mathsf{occ}|$ and **(E3)** limits $|e.\mathsf{inceptOcc} - e.\mathsf{occ}|$. 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. We use the same argumentation as above for announcing events that all constituent events of c are still alive when the deletion of e is detected, with the difference that with $c.\mathsf{class} = c'.\mathsf{class}$ and that inequality (3) holds because of **(E2)**.

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 $c.\mathsf{key} = c'.\mathsf{key}$. We use the same argumentation as above for announcing events that all constituent events of c' are still alive when e' is detected, with the difference that inequality (3) holds because of **(E2)**.

Unchanging events. If an event c remains unchanged with $c \in D_{i-1}$ and $c \in D_i$, we first show that either one of the following two cases holds.

— For $c \in D_{i-1}^P$, we have $c \in D_i^P$: Then $f.\mathsf{expiration} \geq t_{i-1}$ for all constituent events $f \subset c$. Since we compute O_i^P with $O_i^P = \mathsf{purged}(O_i, t_{i-1})$, we obtain $f \in O_i^P$, implying $c \in D_i^P$.

— For $c \notin D_{i-1}^P$, we have $c' \notin D_{i-1}^P$ and $c' \notin D_i^P$ for any c' with $c'.\mathsf{key} = c.\mathsf{key}$: 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 $c \notin D_i^P$.

In the first case the same actions are triggered for `ONTIME` and `LATE` as in the buffering semantics. In the second case no action is triggered, as c is missing in both sets D_{i-1}^P

and D_i^P. It remains to be shown that the second case does not occur before, due to advancement of time, a complex event is fired ONTIME or LATE in case the event processor was inactive at that time.

We obtain for all constituent events $f \subset^+ c$:

$$\begin{array}{ll} f.\mathsf{expiration} = f.\mathsf{inceptOcc} + f.\mathsf{class.maxFreezeTime} + f.\mathsf{class.maxInceptSpread} & \geq_1 \\ f.\mathsf{inceptOcc} + c.\mathsf{class.freezingTime} + c.\mathsf{class.inceptSpread} & \geq_2 \\ c.\mathsf{inceptOcc} + c.\mathsf{class.freezingTime} & \geq_3 \\ c.\mathsf{occ} & \geq_4 \\ t_{i-1}. & \end{array}$$

Inequality (1) holds by definition of maxFreezeTime and maxInceptSpread, inequality (2) holds by the construction of inceptSpread, inequality holds (3) by **(E3)** and the complex event being defined relative to some subscribed constituent event. Inequality (4) holds as the last processing cycle has been at time $t_{i-1} < c.\mathsf{occ} = \mathtt{NOW}$. Complex event c is fired ONTIME if the processing time of the current cycle $t = \mathtt{NOW}$, it is fired LATE, if NOW $< t$.

Thus, $\mathsf{triggered}(D_i, D_{i-1}) \subset \mathsf{triggered}(D_i^P, D_{i-1}^P)$ such that $[\![\mathbb{E}]\!] \subseteq [\![\mathbb{E}]\!]_{\mathsf{purged}}$. □

We now define a revised sliding window semantics supporting also non-monotone programs, by purging "spurious" actions potentially arising out of non-monotonicity.

Definition 4.10 (*Sliding Window Semantics for Non-Monotone Programs)*). Given a sequence of subscribed events $O_0, O_1, \ldots$ for time instants $t_0, t_1, \ldots$, we define the *sliding window semantics* for a non-monotone program at instant t_i

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

where $\mathsf{actionsPurged}(X) = \{a(t, c, c') \in X \mid c = c' \vee (\max(c.\mathsf{occ}, c'.\mathsf{occ}) + 2\ c.\mathsf{class.freezingTime} \geq t)\}$, with $\max(c.\mathsf{occ}, c'.\mathsf{occ}) = c.\mathsf{occ}$ for $c' = \bot$.

THEOREM 4.11 (SEMANTIC EQUIVALENCE FOR NON-MONOTONE PROGRAMS). *If program* $\mathbb{E}$ *and events* $O_0, O_1, \ldots$ *satisfy* ***(E1-4)****, then* $[\![\mathbb{E}]\!]^{\mathsf{nm}}_{\mathsf{purged}}(O_i, t_i, O_{i-1}, t_{i-1}) = [\![\mathbb{E}]\!](O_i, t_i, O_{i-1}, t_{i-1})$

PROOF. We show **(I)** $[\![\mathbb{E}]\!]^{\mathsf{nm}}_{\mathsf{purged}} \subseteq [\![\mathbb{E}]\!]$ and **(II)** $[\![\mathbb{E}]\!] \subseteq [\![\mathbb{E}]\!]^{\mathsf{nm}}_{\mathsf{purged}}$.

(I): Given $a(t, c, c')$, we show, each event f that is potentially used in a query for c or c' is alive at time instant t:

$$\begin{array}{ll} t & \leq_1 \\ c.\mathsf{occ} + 2\ c.\mathsf{class.freezingTime} & \leq_2 \\ f.\mathsf{occ} + 2\ c.\mathsf{class.freezingTime} + c.\mathsf{spread} & =_3 \\ f.\mathsf{occ} + c.\mathsf{inceptSpread} & \leq_4 \\ f.\mathsf{occ} + f.\mathsf{maxInceptSpread} & \leq_5 \\ f.\mathsf{inceptOcc} + f.\mathsf{freezingTime} + f.\mathsf{maxInceptSpread} & \leq_6 \\ f.\mathsf{inceptOcc} + f.\mathsf{maxFreezeTime} + f.\mathsf{maxInceptSpread} = f.\mathsf{expiration} & \end{array}$$

Inequality (1) holds by definition of actionsPurged and (2) by definition of spread. Equation (3) is the definition of inceptSpread, inequalities (4, 5, 6) hold because of definitions of maxInceptSpread, conditions **(E1, E2, E3)**, and definition of maxFreezeTime.

Thus, although $\mathbb{E}$ may not be monotone in general, it is so with respect to the querying of c and c' since no event relevant for querying c or c' has been purged from O_i or O_{i-1} and we have $\mathsf{actionsPurged}(\mathsf{triggered}(D_i^P, D_{i-1}^P)) \subset \mathsf{triggered}(D_i, D_{i-1})$

(II): Because of **(E1, E2, E3)**, for $a(t, c, c')$, $t \leq c.\mathsf{inceptOcc} + c.\mathsf{class.freezingTime} \leq c.\mathsf{occ} + 2\ c.\mathsf{class.freezingTime}$ (if $c \neq \bot$; and likewise for c'). Thus, by Definition 4.10,

$\mathsf{triggered}(D, D') = \mathsf{actionsPurged}(\mathsf{triggered}(D, D'))$. The result $\mathsf{triggered}(D_i, D_{i-1}) \subset \mathsf{triggered}(D_i^P, D_{i-1}^P) = \mathsf{actionsPurged}(\mathsf{triggered}(D_i^P, D_{i-1}^P))$ follows from Theorem 4.9. □

5. PEACE'S IMPLEMENTATION AND DEPLOYMENT

PEACE's flexible architecture serves simple as well as complex, performance critical cases. In the simplest setup, a PEACE system consists of three components: one *event detector,* one *event processor,* and one *action executor.* More complex cases employ several instances of each component. When subscribed events are extracted from multiple sources, one event detector per source must be deployed. Likewise, for each type of action to be executed, one action executor must be deployed. These extractors and executors can be distributed over multiple machines to scale the system. Finally, different complex event classes can be processed by individual event processors to scale and distribute the processing load.

Event detectors, event processors, and action executors run in parallel, where event detectors and event processors run in configurable intervals. Event processors must run even if no new events have been detected since the result of their queries may depend on the wall-clock time. Event sources can forward their events to one or more event processors, and each processor can subscribe to several sources. In contrast, action executors wait for actions tuples originating from one or several sources and execute the actions described therein sequentially in arrival order.

Running event detectors, event processors, and action executors in parallel usually achieves a significant amount of performance improvement. As our performance evaluation has shown (see next section), the typical bottleneck is not the event processor but the detection of subscribed events based on web-event extraction and the execution of actions. Event detection can be significantly sped up by running different event detectors, one for each class of subscribed events, in parallel. In our bi-temporal setting, events will be processed as they arrive according to the conditional actions set for each detected simple or complex event. Event processing works with the knowledge at hand at a particular point in time and if an action has to be taken at some point in time, it is taken based on the knowledge available then. Speeding up event detection by using parallel event detectors improves the quality of event processing in so far as it increases the likelyhood of processing events properly on time according to the perfectness assumption.

The complex events produced by a previous processing step do not directly influence the next processing step. Only new subscribed events do. There may be indirect effects though, but with some external delay. Actions triggered by detected events may lead to processing or human actions resulting in other subscribed web-events.

EXAMPLE 5.1 (IMPLEMENTATION OF RUNNING EXAMPLE). *In Figure 8, we present the implementation and deployment of the running example. For each subscribed event class, we employ one event detector, namely extracting* `FlightArrival` *(Example 3.1) and* `FlightDeparture` *events from websites, while obtaining the remaining* `BusinessMeeting`*,* `FlightBooking`*, and* `Flight` *events from internal information systems. For the purpose of this showcase only, we distribute the event processing:* `MissedConnectingFlight` *(Example 4.2) events are detected in one event processor and forwarded to the other event processor which detects* `ArrivedAtDestination` *complex events (Example 4.3). Further, our running example necessitates three action executors – one action executor per action type: The* `publishOnTwitter` *action is performed by an* OXPATH *action wrapper (Example 3.2), the* `bookNewFlights` *action is also an* OXPATH *action (not shown in this paper) whereas* `cancelMeeting` *is a remote procedure call on our internal server.*

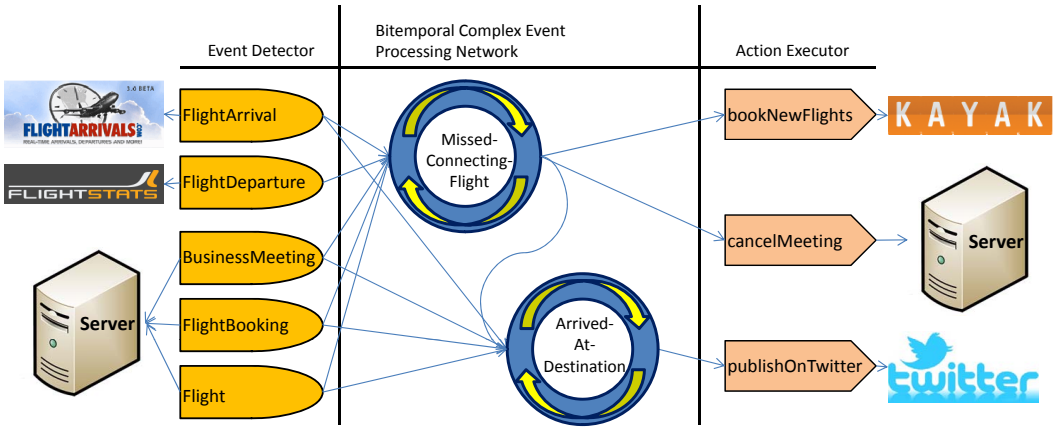


Fig. 8: Implementation of our Running Example with PEACE.

Mobile Deployment. PEACE not only supports server systems but also applications on small mobile devices and has been tested successfully on Android and Ubuntu. Our implementation is designed to be lightweight and portable, implemented with Java and SQLite, an in-memory database. PEACE features a very flexible architecture and can be deployed either in a one-tier manner where all event detectors, event processors, and action executors run on the same device, or as a 2-tier system where event detectors and/or action executors can run on remote servers and are called as web services. The 2-tier architecture is tailored for mobile devices, typically offering less processing power, since event detection from multiple web sites can consume significant processing resources.

Visual Editor. The Development of a PEACE program requires not much effort due to our visual editor for Eclipse (Figure 9). In the editor, the user can arrange event detectors, complex event processors, and action executors, and define the event flow via connections. We offer a set of different predefined event detectors, a generic one, e.g., parameterized with an OXPATH event detection wrapper, as well as specific ones, e.g., for detecting flight arrivals parameterized with the arrival airport and the flight number. An event detector must specify its unique name, the interval between individual detection runs, a subscribed event class definition, and in case of an OXPATH wrapper, the wrapper and its parametrization. A complex event processor must be declared with a unique name, chronon size, and complex event definitions. For subscribed and complex classes alike, one can choose between different buffer implementations, currently offering an SQLite and H2 implementation. As for action executors we offer different generic and specific implementations, similarly to event detectors. An action executor is specified with a unique name, and specific parameters for the OXPATH action wrapper. The architecture of the editor is designed to be easily extended with new components, e.g., new specific event detectors or action executors.

Simulation and Visualization. PEACE offers a simulation environment to trace events through their extraction to arising complex events and the actions thus taken. Within the simulation environment the entire system may be run at different speeds, skimming over uneventful phases and carefully analyzing more turbulent phases. Figure 10 shows a screenshot of our simulation environment. The simulation start time and simulation speed are set in the control panel. Play, pause, and stop buttons allow

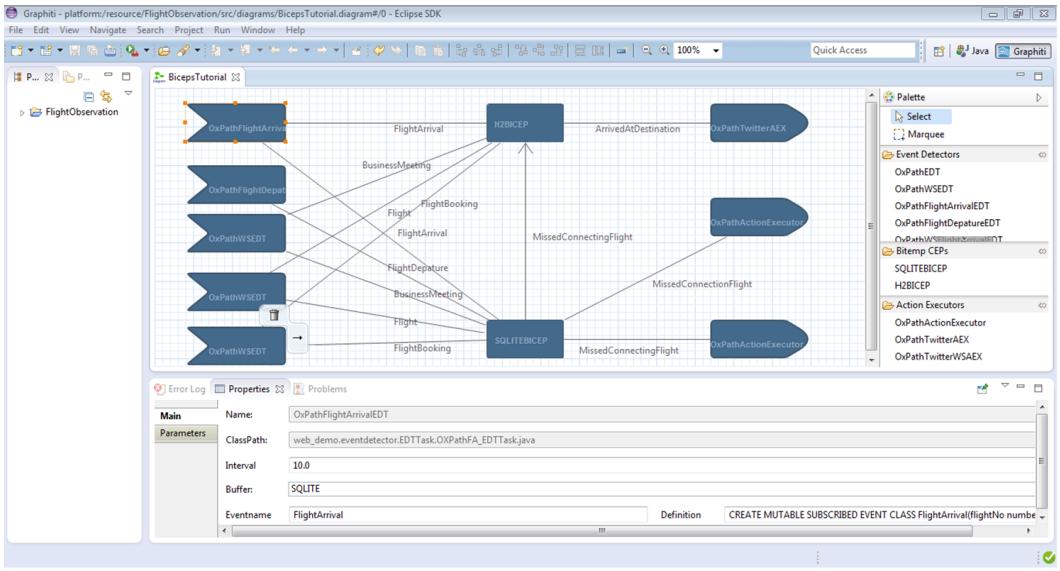


Fig. 9: PEACE Editor.

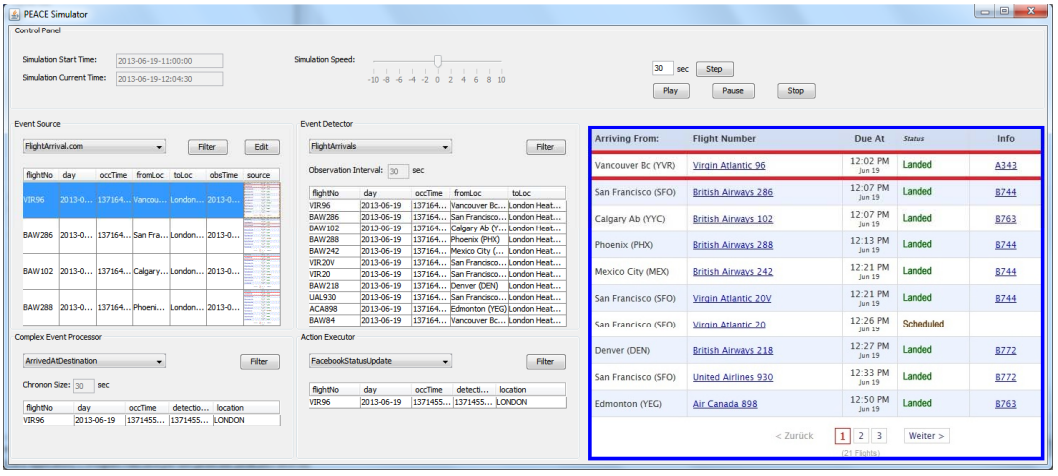


Fig. 10: PEACE Simulator.

to run and control the simulation. The step button runs the system for the number of seconds specified in the text field next to it and pauses it afterwards. When the system is paused or stopped the simulation speed may be changed. The *event source panel* displays *all* observed events originating from the selected event source and shows a screen shot (if extracted from the web) of the selected event at the right side of the window. As a subset of these events, the *event detector panel* shows only those events which are going to be processed in the next run by of one the complex event processors. The *complex event processor panel* shows all events belonging to the selected complex event class after purging. Similarly, the *action executor panel* displays all actions awaiting processing by the selected action executor.

Database Backends. PEACE requires an SQL database backend to store subscribed events and actions in tables and to implement complex events as views. We implemented PEACE with two database management systems, namely SQLite and H2, since they are examples of lightweight database management systems fitting our application scenarios. SQLite is chosen as it is the standard database on Android systems, also supported on various other platforms. Additionally, the complex event processor is implemented atop of H2 database which is promoted to have performance advantages over SQLite.

6. PERFORMANCE EVALUATION

This evaluation comes in two parts, first considering the entire PEACE system with all its components, and second focusing on the event processor and its scalability for various scenarios.

For the first part, we profile the components of PEACE, i.e., event detectors, event processors, and action executors, on varying platforms and with different workloads. (Subsection 6.1). Our evaluation shows that PEACE is dominated by OXPATH which is in turn dominated by the browser overhead to load and render all visited web pages. Thus, PEACE spends only 0.6% of its runtime in the event processor – at least if most subscribed events and actions are web based.

In the second part of the evaluation, we consider the performance of the event processor in stress tests, exceeding typical requirements by an order of magnitude (Subsection 6.2). Our experiments show that the event processor's performance is more sensitive to the event class size, i.e., the number of events belonging to the same event class, than to the event rate, i.e., the number of events added per clock tick. Therefore, the sliding window semantics is beneficial to run PEACE at scale. The overhead for purging expired events in sliding window semantics pays off quickly, leading to constant performance, as opposed to the polynomial performance in case of the buffering semantics. Our implementations based on H2 and SQLite databases show advantages in different situations: H2 is faster in small settings while SQLite performs better in more complex setups.

Lacking comparable systems, we cannot directly compare with other systems. Most existing systems are heavy-weight centralized server solutions, while we aim for a light-weight and decentralized solution. In particular, our event processor runs on mobile devices with limited resources.

6.1. PEACE Components

We evaluate the relative runtime of the individual components of the PEACE system which implements our running example based on an SQLite database. To this end, we process a single cycle of the entire system, where we run the event detectors and action executors either locally or remotely, and where we vary the number of extracted and processed events.

Results. PEACE **(1)** spends approximately 97.4% of its runtime with problem inherent and thus unavoidable web accesses, **(2)** spends at most only 0.6% on event processing, and **(3)** scales well: With a 10 times higher system load, the event extraction only increases at most 7 times, event processing at most 4 times, and local action execution 8.5 times.

Experimental Setup. We perform 20 process cycles on an initially empty database and take the average running times. First, in each cycle we extract 100 `FlightArrival` and `FlightDeparture` events at once, buffer and process these events, and perform a single action execution. Second, we perform the same experiment with 1000 event extractions and 10 action executions. The very first cycle inserts extracted events into

the database while all other cycles update these events. We evaluate PEACE on two platforms, namely a Windows 7 PC with an Intel i7 at 2.7 GHz and 8 GB RAM, and an Android 4.4.3 tablet with a Qualcomm S4 at 1.5 GHz and 2 GB RAM.

Details. As already mentioned afore, the most important result is that PEACE is dominated by OXPATH, spending more than 99.4% with OXPATH evaluations. These evaluations are in turn dominated by page loading and rendering times, consuming 98% of OXPATH's runtime [Furche et al. 2013a]. Therefore, PEACE spends approximately 97.4% of its runtime with problem inherent and thus unavoidable web accesses. In Table III we show the individual runtimes for event extraction, buffering, processing, and action execution, running PEACE either locally or remotely.

Extracting 1000 events instead of 100 takes only a 7 times longer, since all events are extracted with a single wrapper invocation where the initial browser startup proves to be very costly. Also event processing increases only marginally by less than 15%, as the query invocation overhead dominates the actual query processing. Only the action execution takes roughly 10 times longer, because each triggered action is executed with its own wrapper invocation. Switching to remote evaluation of OXPATH wrappers, we observe a consistent behavior: Event extraction is sped up by 40%, since the server maintains a pool of initialized browsers ready for use. In contrast, the speedup for running the action execution on the server is offset by the overhead of the remote invocation.

On the tablet, PEACE runs OXPATH only remotely because of libraries which are required but unavailable on Android. Event extraction and action execution behave similarly to the PC. Only buffering and processing react much more sensitive to the number of extracted events. Curiously, the tablet consistently executes remote OXPATH queries slightly faster than the PC.

The results of our benchmarks that event handling creates low overhead in comparison to extracting events from web sites may not be surprising. The business use cases we have in mind come with a very manageable amount of data (which we have overestimated in a order of magnitude for our benchmarks) and at the event processing consists of checking simple conditions. However, these results cannot be assumed to be given a priori. We have developed a novel approach for bi-temporal event processing that was inspired by and meets the requirements of a web-based setting. A time-centric and event-oriented approach that treats events as mutable best addresses the perspective of handling web announcements, whereas current event processing systems are typically not time-centric and based on non-mutable events. Since our approach is declarative and inspired by bi-temporal database work, we have developed a corresponding prototype using on-the-shelf relational database technology and SQL for bi-temporal complex event processing. The performance tests demonstrate that our theoretical work can be put easily into practice. We had initially investigated different performance optimization strategies, such as using temporal indices per event class on chronons at which events of the class occur and need to be acted upon once time advance to that chronon. The benchmark results have been obtained without such optimizations. We used, however, available index structure to index the key of events and maintained auxiliary tables for complex events detected so far. At the same time we note that their is still room for performance improvement, if this is considered necessary in the future. As long as we have applications for which event handling causes only low overhead, developing further performance improvements is not worth-while.

Furthermore, the results of our benchmarks demonstrate that our approach and complex event processor is also suitable for mobile environments with limited resources, such as a tablet, using light-weight database engines such as H2 or SQLite for implementation.

System OXPATH	#Events	Event Extraction *local*	Event Extraction *remote*	Buffering	Processing	Action Execution *local*	Action Execution *remote*
PC	100	103.7	59.1 sec	14 ms	85 ms	19.6	17.3 sec
PC	1,000	740.8	400.3 sec	44 ms	96 ms	165.9	184.6 sec
Tablet	100	-	56.3 sec	64 ms	367 ms	-	17.1 sec
Tablet	1,000	-	383.6 sec	321 ms	1,495 ms	-	182.3 sec

Table III: Average time spent in different PEACE components over 20 executions

6.2. Evaluation of the Bitemporal Complex Event Processor

We evaluate the performance of BICEPL's event processor in a stress test with event rates and event class sizes that are an order of magnitude higher than in typical settings. More specifically, we performed experiments in two modes: In both modes the complex event processor starts a new processing cycle, consisting of (a) subscribed event registration, (b) complex event detection and condition-action evaluation, and (c) action notification every 3 seconds, or immediately after the processor has finished a processing cycle if it has lasted more than 3 seconds. In **(1)** *round-based experiments* at the begin of each processing cycle (round), all available incoming subscribed events are collected from even detectors and loaded into the event repository. We plot processing time against the number of events in the event repository with these experiments. Round-based experiments are natural to consider for studying the effect of the size of the repository (buffer) on performance when using the buffering semantics in event processing. In **(2)** *time-triggered experiments,* we feed a fixed amount of subscribed events to the event repository at each chronon, regardless whether the BICEPL processor is idle (it has finished its previous processing round) or not. We plot processing time against running time of the event processor with these experiments. Time-triggered experiments are more natural to consider when benchmarking the performance according to the sliding window semantics in which events are purged after having reached their expiration time. We also use time-triggered experiments to relate the performance of the buffering semantics and the sliding window semantics and to determine the break-even point when event processing according to the sliding window semantics outperforms event processing according to the buffering semantics. We present four different evaluations:

— *Buffering Semantics Scalability.* In a round-based experiment, we evaluate the performance of processing subscribed events. In the buffering semantics, events are kept forever and are never purged. We are interested here in the change of performance based on the size of the buffer (event repository) relative to the event rate, number of event classes, and number of condition-action statements.
— *Buffering versus Sliding Window Semantics.* We compare buffering semantics against sliding window semantics in a time-triggered experiment and evaluate the break-even point when the sliding window semantics outperforms the buffering semantics. We are interested here in the change of performance based on the accumulated running time of the system relative to event rate, number of event classes, and number of condition-action statements.
— *SQLite Implementation versus H2.* We compare our database backends with a round based experiment running buffering and sliding window semantics.
— *Mobile Performance.* We evaluate the performance of the buffering and sliding window semantics on a tablet with a time-triggered and a round-based experiment.

These evaluations are done with SQLite, except for comparing our SQLite and H2 implementations.

Results. Aside platform differences, we have identified two main factors that affect the performance of the BICEPL event processor. Interestingly, *the event rate,* i.e., the number of events handled each clock tick, has only marginal influence on the processing time, given event classes of similar size and structure.

— *Event class size.* The class size, i.e., the number of events belonging to the same class, is the main performance driver. The more events belong already to a class, the lower is its performance.
— *Condition-action statements.* For each of these statements, PEACE needs to evaluate its own query during each processing round.

The sliding window semantics outperforms the buffering semantics after reaching a certain event class size. The point, when the overhead of identifying and purging stale events is amortized by the smaller event class size, depends on the BICEPL program and the employed technical infrastructure. H2 performs better than SQLite on smaller instances with up to 150,000-200,000 events per event class, while SQLite outperforms H2 on larger instances. On mobile devices with less processing power the event processor is much more sensitive to the event class size, rendering the buffering semantics infeasible, while the sliding window semantics is perfectly working.

We did run our stress tests up to event repositories of several hundred thousand events, which is magnitudes beyond events to be encountered in web-based business applications, or event detectors could reasonable provide (we simulated higher rates in benchmarks) or the owner of a mobile device would collect for him- or herself. Many applications frequently have simple events to be immediately reacted upon (such that a direct flight, shipment or appointment is late), next to a few complex events with a couple of associated condition-action statements. Complex events are usually centered around a main event (such as a flight, a shipment, or an appointment) connected to other events by a one-to-one or one-to-many combination. We have not encountered so far a meaningful business example with many-to-many event combinations. Our benchmarks reflect realistic scenarios, apart from the number of events generated. The number of events maintained is unrealistically high, but we repeatedly and intentionally increased this number to see what our non-optimized implementation could already reasonably handled within 5-30 seconds. Reducing the unrealistical high number of events gives enough room to handle situations having a higher number of condition-action statements with a performance that is still a fraction of the time required for web event extraction and action execution.

Our architecture provides for a network of event detectors and event processors (see Section 5) such that event processing can be partitioned between multiple, decentralized devices reflecting event and data distribution among members of an organization. Such as a single person usually needs to handle a couple of hundred of personal emails per day supported by some dozen email handling rules (if any), we expect the personal mobile device will need to handle at most several thousand personal events per day (some expiring within a few days, others kept a week or a month) with one or two dozens of condition-action rules.

Experimental Setup. Table IV summarizes our workloads **(W1-3)** defining the number of event classes (#EC), number of condition-action statements (#CA-Stmt), event rate (#Events), and the distribution of subscribed events triggering certain condition-action statements (Distribution). The lifespan defines the minimum period an event stays in its event class. All workloads use a chronon size and event detection interval of 3 seconds. With OXPATH event detectors, this event detection frequency can only be reached by employing multiple event detector instances running in parallel. Since

we are only interested in the event processing performance, we simulated these event detectors.

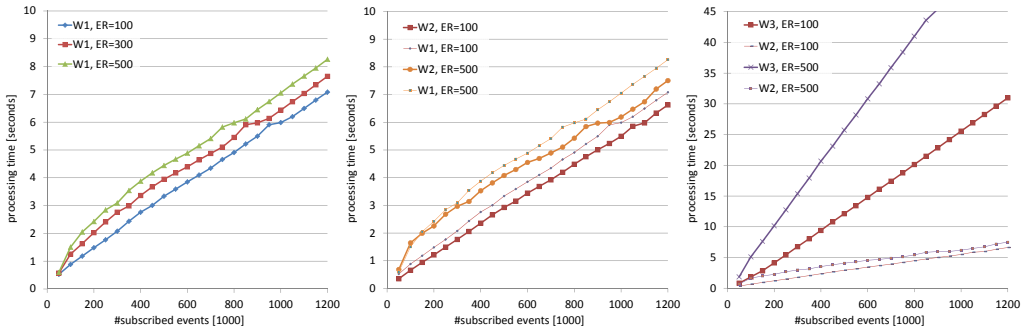
Workload	#EC	#CA-Stmt	#Events	Distribution	Lifespan
(W1)	2	1	100/300/500	1.0 OnTime	-/3600
(W2)	8	4	100/500	1.0 OnTime	-/3600
(W3)	8	28	100/500	0.5 OnTime 0.25 Late 0.25 Future	-/3600

Table IV: Definition of Stress Test's Workloads **(W1-3)**.

In workload **(W1)** we have one subscribed and one complex event class. Each subscribed event gives rise to a complex event with one condition-action statement firing `ONTIME`. Since all complex events fire `ONTIME`, the database contains three times the subscribed events. Running for 2,400 chronons, each time adding 100, 300, or 500 events, we obtain with buffering semantics (never deleting events) up to 3,600,000 events per event class. For the sliding window semantics we fix a window size of 3,600 seconds (1,200 chronons), leading to at most 1,800,000 events. In workload **(W2)** we increase the number of event classes to 4 subscribed and 4 complex event classes, each of the complex event classes defining one `ONTIME` condition-action statement, firing for each complex event. Resulting event class sizes are equal to workload **(W1)**. Workload **(W3)** has 4 complex classes, each with 7 condition-action statements, namely an `ONTIME`, `LATE`, `RETROACTIVECHANGE`, `REVOCATION`, `ANNOUNCEMENT`, `CANCELLATION`, and `CHANGE` condition-action statement. 50% of the subscribed events arrive on time (15% are postponed, retroactively changed, and revoked, respectively, while 55% stay unchanged), 25% late (all unchanged), and 25% are future events occurring one chronon later (15% are changed and canceled, respectively, while 70% stay unchanged). Adding 100 subscribed events in **(W3)** leads to 455 events in the database, whereas **(W1-2)** require only 300 new events. This is case, since specific condition-action statements require additional internal events and each action is internally stored as individual event.

These workloads are evaluated on different platforms and on SQLite and H2 databases. The test PCs feature Intel Pentium Dual Core E6700 (3.2 GHz) and 4 GB RAM. The mobile device is a Nexus 7 (2013) running an Android 4.4.3.

The question has been raised how much more expensive it was to move from mono-temporal to bi-temporal event processing. While we did not develop a mono-temporal event processor and run similar benchmarks, the answer to this question can be given in the context of a bi-temporal complex event processor by considering a benchmark case using only condition statements firing `ONTIME`, against a benchmark case using additionally other condition statements that require bi-temporal support. Workload **W2** uses only `ONTIME` condition statements, while workload **W3** uses additional condition statements. Although both workloads come with a different number of condition statements it is meaningful to compare them to answer the question on how much more expensive it was to move from mono-temporal to bi-temporal event processing. Bi-temporal event processing comes with additional functionality, the possibility to cope with imperfectness in a simple and natural way. To address the different kinds of deviations from a perfect world and system, additional condition-action statements are necessary. This additional functionality comes with some cost (see the benchmark results presented in the subsequent paragraphs) but these may be neglected considering the overall PEACE system with its components as presented above. If one wishes



(a) Impact of Event Rate (b) Impact of Event Classes (c) Impact of CA. Stmts.

Fig. 11: Buffering Semantics

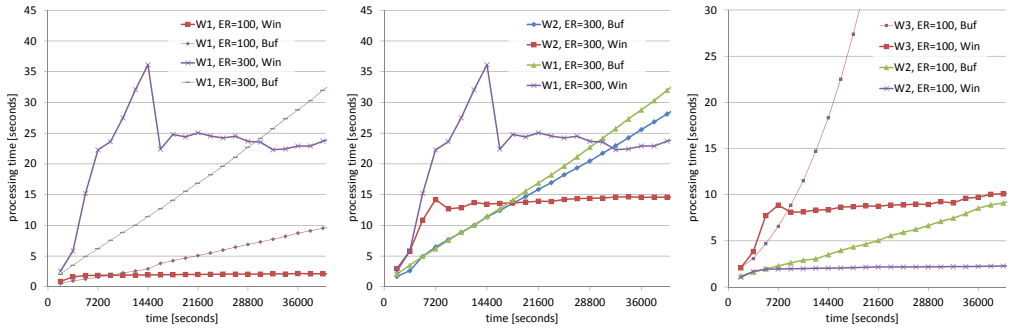
to know the difference in performance of evaluating a mono-temporal condition-action statement versus a bi-temporal condition-action statement, we can state that we have encountered hardly any difference when performing workload **W2** with different timing primitives. This is easy to see, as in order to check whether a condition-action statement triggers an action, a query is to be performed on the database (for query-rewriting see Subsection 4.3).

Buffering Semantics. Figure 11 presents the results of our stress tests performed on our test PC with SQLite. It shows charts for workloads **(W1-3)** at different event rates (ER), determining the number of newly arriving events per chronon. Each chart plots the average processing time needed for dealing with the arriving events against the number of subscribed events, i.e., the summed sizes of all subscribed event classes at the given moment. We plot against the number of subscribed events, since different workloads cause the event classes to grow at different rates. Figure 11a on running workload **(W1)** shows that the event rate influences the processing time only slightly: Adding 500 events takes nearly independently of the event class size only about 1 second longer than adding 100 events to an event class of similar size and structure. Figure 11b on comparing workload **(W2)** with **(W1)** shows that smaller event classes are processed more efficiently, since **(W2)** disperses its subscribed events over 8 classes, while **(W1)** splits them between 2 classes. This is not surprising, as the event class size directly translates into database table sizes. In Figure 11c, we compare workloads **(W3)** and **(W2)**. In **(W3)** the number of internally maintained events increases much faster than in **(W2)**, which leads together with the 28 condition-action statements and the higher number of actions to a much higher workload.

Sliding Window versus Buffering Semantics. Figure 12 compares sliding window with buffering semantics performing workloads **(W1-3)** at event rates of 100 and 300 within a time-triggered experiment.

The charts visualize how long it takes to add 100 or 300 subscribed events, after running for a certain time period while subscribed events have been created continuously every 3 seconds. Figure 12a illustrates for workload **(W1)** that the overhead for purging expired events increases with higher event rates, yielding larger event classes, and the break-even point for the window semantics shifts backwards. The reason for this is the higher sensitivity of the sliding window semantics to the event class size due to expiration time calculations for purging events. Figure 12b on workload **(W2)** shows that the event class size has a much more pronounced impact on sliding window se-

mantics than buffering semantics: With sliding window semantics, the processing time is not only lower, but also the break-even point is reached much earlier with smaller event class sizes. In Figure 12c we compare workload **(W2)** with **(W3)** running with an event rate of 100. The higher number of condition-action statements of workload **(W3)** affects the performance significantly, for both, the sliding window and the buffering semantics. Though, while **(W2)** runs with the buffering semantics at least for 12 hours (36,000 seconds) with reasonable performance, **(W3)** can only run reasonable with the sliding window semantics. The spikes of the curves for the workloads executing under windowing semantics, e.g., W1 ER=300, Win in Figures 12a and 12b, indicate the point in time when purging of subscribed events from the event history sets in for the first time. Processing time increases as long as the event buffer grows. At the time when event purging sets in, the event buffer shrinks, with processing times remaining basically constant as well.

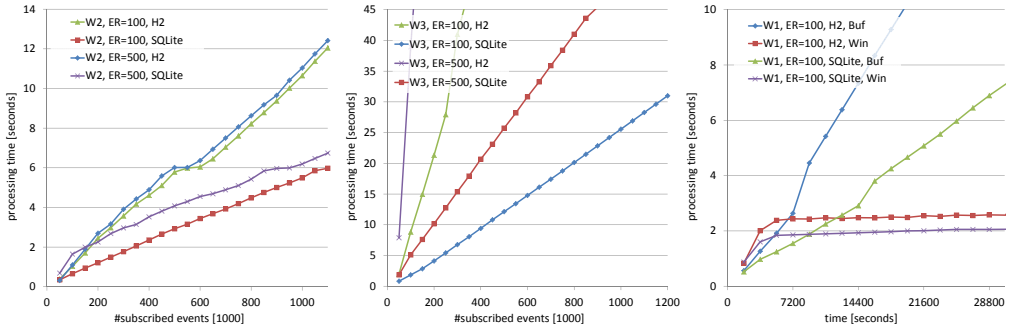


(a) Impact of Event Rate (b) Impact of Event Classes (c) Impact of CA. Stmts.

Fig. 12: Sliding Window versus Buffering Semantics

SQLite versus H2 Implementation. We evaluate the H2 against the SQLite implementation with a round-based and time-triggered experiment for workloads **(W1-3)** (Figure 13). In the round-based experiment, the H2 implementation is quicker up to 180,000 events per event class for workloads **(W1)** and **(W2)** (Figure 13a). In contrast, SQLite outperforms H2 with larger event classes, and in heavier settings with more condition-action statements, such as **(W3)** (Figure 13b). This is the case, since H2's performance on our queries is much more sensitive to the event class size, i.e., table size, than SQLite. As the table size dominates H2's performance, it is only slightly influenced by the event rate: The performance in adding 100 and 500 events differs only marginally, and is lower than for SQLite (Figure 13a). Finally, we compare the performance of sliding window semantics and buffering semantics. H2 is slower than SQLite in all cases. Because of its weaker performance, the break-even point, when the window semantics outperforms the buffering semantics, comes earlier than for SQLite (Figures 13c).

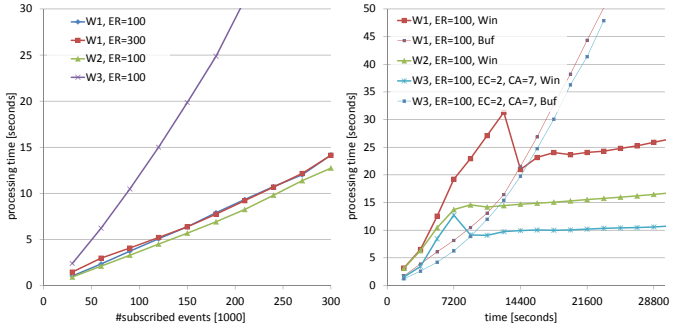
Mobile Performance. On the tablet, we evaluate the buffering semantics with a round-based experiment (Figure 14a) and the sliding window semantics with a time-triggered experiment (Figure 14b). These experiments evidence that our event processor is very light-weight and hence runs with good performance on mobile devices. Unsurprisingly, in contrast to a PC, the tablet can only handle smaller event classes, however, still exceeding typical requirements. So for example, the tablet handles in



(a) Round-based Experiment **(W2)** (b) Round-based Experiment **(W3)** (c) Time-triggered Experiment **(W1)**

Fig. 13: SQLite versus H2 Implementation

buffering semantics event class sizes up to 100,000 subscribed events within 3 seconds at a rate of 100 events per second for workloads **(W1-2)** (Figure 14a). Aside from scale, the characteristics of the plots do not differ from the ones on the PC: The event class size dominates, while the event rate influences the performance only marginally. The buffering semantics becomes infeasible even more quickly than on the PC, but the sliding window semantics runs with good performance (Figure 14b): Running workloads **(W1-3)**, the sliding window semantics reaches quickly the break even, outperforming the buffering semantics thereafter.



(a) Round-based Experiment **(W1-3)** (b) Time-triggered Experiment **(W1-3)**

Fig. 14: Mobile Performance

7. RELATED WORK

To the best of our knowledge, PEACE is the first system that addresses complex event processing for event announcements from the Web. This differs from mining events from Twitter or other sources [Boettcher and Lee 2012; Ilina et al. 2012] but is related to typical complex event processing where many event sources are integrated to detect complex events and react upon them. Therefore, we focus on the difference of PEACE

and its complex event processor and language BICEPL with existing event processing systems.

We classify event processing approaches into four different general approaches: Active Database Management Systems (ADMS), Data Stream Management Systems (DSMS), Event Stream Processing (ESP) and Complex Event Processing (CEP) systems, and Event Calculi (EC). In the following, we examine systems of all four approaches focusing on their time model and ability to handle mutable events, as these features are essential for web events.

7.1. Active Database Management Systems (ADMS)

An ADMS is a database with human-passive database-active interaction pattern in extension of traditional Database Management Systems (DBMS) with human-active database-passive interaction pattern. This extension is typically implemented in terms of event condition action (ECA) rules. Such rules consider internal events, e.g., tuple insertion or updating, and in some systems also external events [Paton and Díaz 1999], e.g., events coming from sensors. Events are defined to be instantaneous, i.e., they occur at a specific point in time and are known instantaneously (in zero time) after they occurred.

To the best of our knowledge, events in ADMS [Dayal et al. 1988; McCarthy and Dayal 1989; Lieuwen et al. 1996; Gehani and Jagadish 1991; Gatziu and Dittrich 1993; Chakravarthy and Mishra 1994; Adaikkalavan and Chakravarthy 2006; Buchmann et al. 1995] are limited to one time dimension, if time is considered at all, and most importantly, non of these approaches considers mutable events – which are pervasive on the web.

Limited to complex event detection, [Galton and Augusto 2002; Adaikkalavan and Chakravarthy 2006] distinguish between detection- and occurrence-based semantics leading to different semantics of complex event operators, such as a sequence operator. The detection-based semantics does not distinguished between an event's occurrence and its detection, whereas the occurrence-based semantics does. But neither mutable events nor comparisons between occurrence and detection times are possible, rendering this approach unsuitable for our ends.

7.2. Data Stream Management Systems (DSMS)

High performance DSMS [Abadi et al. 2003; Arasu et al. 2003; Abadi et al. 2005; Arasu et al. 2003; Bai et al. 2006; Chandrasekaran et al. 2003] are general purpose systems which do not support the concept of events but of general data tuples and therefore do not come with event specific operators. These systems apply a data stream transformation function but do not directly support a publish-on-occurrence function, as required for our event processing purpose. However, in some cases, a DSMS can act as event processing system.

DSMS have been created to overcome the weak performance of ADMS, resulting from the great effort of querying a persistent memory (the database) when working with high data volumes and/or many rules. Most DSMS systems are targeted at high performance real time data analysis, where performance is more significant than functionality. They transform input data streams (subscribed data streams) to output data streams by applying, e.g., join or filter operators. In contrast to an ADMS, a DSMS typically keeps only a set of events over subscribed data streams, i.e., an unbounded sequence of data ordered by their occurrence, that fit into a specified time window in the memory to perform continuous queries [Babu and Widom 2001] on them. There are systems able to manage unordered data streams to a certain extent, or stand out with other time related features.

[Srivastava and Widom 2004] propose an *application timestamp* as an ordering criterion and buffer arriving data (tuples) for a certain amount of time before the stream is ordered and forwarded to the query processor. After the stream is forwarded it is assumed to be append-only, i.e., tuples in the stream are immutable and tuples arriving late are ignored. Further, on arrival of a tuple, a timestamp is assigned to it, the *system timestamp*, which is referred to for windowing. Though this approach supports two time dimensions for tuples (possibly representing events), the semantics of the system is very different from our event processing system. Inherently, deferred event publication at their actual occurrence time is not supported, nor are temporal comparisons between events, e.g., sequences, provided. Also the inherent processing delay because of the entrance buffer is unfitting for our purpose.

Borealis Stream Processing Engine [Abadi et al. 2005; Ryvkina et al. 2006] is another general purpose DSMS system offering a novel feature, namely a revision functionality of query results. It is able to revise tuples in an output stream due to revisions in the input stream which abolishes the append only (immutability) policy [Terry et al. 1992] of other DSMS. While Borealis only distinguishes between update and delete revisions, our system offers a finer granular approach and can quantify the revision by allowing to compare the attributes of the revised and revising event. Due to its general purpose approach, Borealis does not include specific event processing primitives, as needed for our ends, such as the sequence operator to determine the occurrence order of events.

7.3. Complex Event Processing (CEP) and Event Stream Processing (ESP) Systems

There is no CEP or ESP system supporting bitemporal, mutable events suitable for web events. Since research literature and especially industrial reports blur the difference between ESP and CEP systems, and since both approaches are expected to merge [Luckham 2006], we treat them at once. CEPs were created in the 90ies, rooted in event-driven simulations which did not produce event streams, but event clouds, i.e., (unordered) sets of events originating from different sources. To analyze these event clouds, event processing principles were developed, later decoupled and applied to middleware systems where similar characteristics were found, namely multiple event streams from different sources to be integrated into a single event cloud. As PEACE needs to handle sequences of unordered and potentially outdated events produced by various sources, we develop the core event processor of PEACE as CEP system. In contrast, ESPs, also developed in the 90ies, have their sources in DSMS operating with event streams, i.e., time-ordered event sequences. However, ESPs can handle unordered event streams with buffering approaches, as mentioned with DSMS. While in the past ESP systems focused on performance over functionality, nowadays they can detect rich complex event patterns, including temporal and causality relations, like CEP systems do.

Mapping our complex event declarations to existing complex event processing approaches is only partly possible: The *ontime* case, defining the reaction if there are no delays, changes, or errors in the event announcement, is the standard scenario assumed by all approaches. This is the only directly supported case. The *late* case is not supported at all by existing approaches, since the second time dimension is missing completelty or programmatically inaccessible with standard event operators. Thus, while the occurrence time is event inherent and implicitly given, it may not be compared with user defined (explicit) time attributes. Mapping the *retroactive change* and *revoke* case to existing languages is possible, although often requiring the subscription to multiple event types and the introduction of auxiliary event types to signify updates and revocations.

CEPs with Normalized Event Model. There is a mass of CEP systems, hardly covered by a survey article on CEPs [Cugola and Margara 2012]. Most CEP systems [Luckham 1998; Luckham and Vera 1995; Wu et al. 2006; Li and Jacobsen 2005; Jacobsen et al. 2010; Demers et al. 2007; Schultz-Møller et al. 2009; Cugola and Margara 2009; 2010; Sybase 2012] follow the perfect technology assumption, i.e., an event is known by the event processing system instantaneously after it occurred [Wieringa 2003] and remains immutable thereafter. This restrictive event model disables reasoning over events in an imperfect world, as for this task the occurrence time and detection time are essential and mutable events are ubiquitous.

CEPs with Unexposed Noisy Event Models. Some CEP systems deal with noisy events without exposing a multi-dimensional time model to the system user. [Mansouri-Samani and Sloman 1997] highlight the importance of differentiating occurrence and detection time for monitoring distributed systems with inherent communication delays, but only approach the issue with delayed event emissions instead of immediate emissions and subsequent corrective events. However, the detection and occurrence times are inaccessible in complex event specifications. [Li et al. 2007] address out-of-order arriving events and the consequent late detection of complex events within event streams. They only examine the sequence operator and do not distinguish between on time and late detection of a given sequence. [Pietzuch et al. 2003] consider event delays (assuming no unordered arrivals) in their publish-subscribe system and implement strategies where they either ignore delays or wait until a delay is impossible. They do not handle unordered event arrivals nor mutable events,

CEPs with (Partially) Exposed Noisy Event Models. The remaining two systems do not only deal with noisy events but also expose multiple times associated with an event. However, they do not involve all these times into the definition of complex events or fail to support mutable events. CEDR [Barga et al. 2007; Goldstein et al. 2007] proposes a tritemporal time model for events, consisting of an occurrence time interval, i.e., the occurrence (period) of the event (equivalent to our occurrence time), a valid time interval, i.e., the period over which the event is valid for complex event detection (equivalent to the lifespan in BICEPL), and a CEDR time interval, i.e., the time when it arrived at the system (equivalent to our detection time). The valid and occurrence times are provided by the event source, while the CEDR time is set by the event processor. Albeit having a very powerful time model, CEDR does only allow for updating the valid time interval, thus it does not support fully mutable events. Further, the time dimensions are not provided for complex event detection, but only for event retractions due to valid time updates. Note, the successor of CEDR is a commercial product described in [Ali et al. 2009; Ali et al. 2010; Ali et al. 2011] but with only a restricted time model compared to CEDR. To the best of our knowledge, AMIT [Adi and Etzion 2004] is the only CEP system considering two time dimensions similar to ours: Their event's *eventTime* corresponds to our occurrence time, and their *detectionTime* to our detection time. Yet, these time dimensions are not supported for temporal comparisons in complex event processing. Complex events (called situations), are defined by their highly expressive, imperative complex event language, but only involving the event time. Moreover, events in AMIT are immutable.

Commercial Systems. Commercial CEP systems [Sybase 2012; TIBCO 2013b; Oracle 2013; EsperTech 2013; TIBCO 2013a; IBM 2013] mainly focus on realtime processing and emphasize performance criteria but hardly incorporate features to deal with noise events. All these systems are general purpose event processing systems assuming instantaneous event notifications from high volume streams without bitemporality or mutable events.

7.4. Event Calculi (EC)

Besides the above mentioned event processing approaches, knowledge representation also deals with events, as in Event Calculi (EC) [Kowalski and Sergot 1986; Mareco and Bertossi 1999; Sripada 1988], representing knowledge about events for reasoning purposes. Originally unitemporal, though, there exist EC extensions [Mareco and Bertossi 1999; Sripada 1988] 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. While EC provides a general model to reason about events, BICEPL is a practical and succinct yet expressive language tailored for web event processing.

8. CONCLUSION

We have presented PEACE as an integrated framework to extract events, to process these events into complex ones, and to perform actions triggered by these events. PEACE relies on a bitemporal event model which supports BICEPL, a compact and declarative language for complex event processing. This language comes not only with a buffered semantics, not purging any events, but also with an efficiently implementable semantics that purges stale events eventually – while maintaining semantics equivalence. BICEPL is an SQL-based language supporting subscribed and complex event definitions with bitemporal predicates to react upon deviations of the occurrence and detection time of events. Next to BICEPL, we employ highly efficient OXPATH wrappers for event extraction and action execution on the web. Both languages are easy to learn and yield compact and maintainable PEACE applications. We implement our approach with different databases, namely SQLite and H2, in support of different platforms. We also provide an editor and a simulation environment to increase the usability of our framework. At last, we have evaluated our implementation on a PC and a tablet, showing that PEACE requires almost no overhead over the web access itself. Future research concentrates on location-based event processing, especially relevant for mobile devices, and on the Internet of Things which needs to allow its devices to react on a multitude of events.

REFERENCES

Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*. 277–289.

Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal* 12, 2 (2003), 120–139.

Raman Adaikkalavan and Sharma Chakravarthy. 2006. SnoopIB: Interval-based event specification and detection for active databases. *Data & Knowledge Engineering* 59, 1 (2006), 139–165.

Asaf Adi and Opher Etzion. 2004. Amit - the situation manager. *The VLDB Journal* 13, 2 (2004), 177–203.

Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, and Roman Schindlauer. 2011. The extensibility framework in Microsoft StreamInsight. In *Proceedings of the 27th International Conference on Data Engineering (ICDE 2011)*. 1242–1253.

Mohamed H. Ali, Badrish Chandramouli, Balan Sethu Raman, and Ed Katibah. 2010. Spatio-Temporal Stream Processing in Microsoft StreamInsight. *IEEE Data Engineering Bulletin* 33, 2 (2010), 69–74.

Mohamed H. Ali, Ciprian Gerea, Balan Sethu Raman, Beysim Sezgin, Tiho Tarnavski, Tomer Verona, Ping Wang, Peter Zabback, Anton Kirilov, Asvin Ananthanarayan, Ming Lu, Alex Raizman, Ramkumar Krishnan, Roman Schindlauer, Torsten Grabs, Sharon Bjeletich, Badrish Chandramouli, Jonathan Goldstein, Sudin Bhat, Ying Li, Vincenzo Di Nicola, Xianfang Wang, David Maier, Ivo Santos, Olivier Nano, and Stephan Grell. 2009. Microsoft CEP Server and Online Behavioral Targeting. *Proceedings of the VLDB Endowment (PVLDB)* 2, 2 (2009), 1558–1561.

Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. 2003. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin* 26, 1 (2003), 19–26.

Shivnath Babu and Jennifer Widom. 2001. Continuous Queries over Data Streams. *SIGMOD Record* 30, 3 (2001), 109–120.

Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. 2006. A data stream language and system designed for power and extensibility. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management (CIKM 2006)*. 337–346.

Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. 2007. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*. 363–374.

Alexander Boettcher and Dongman Lee. 2012. EventRadar: A Real-Time Local Event Detection Scheme Using Twitter Stream. In *2012 IEEE International Conference on Green Computing and Communications (GreenCom 2012)*. 358–367.

Alejandro P. Buchmann, Jürgen Zimmermann, José A. Blakeley, and David L. Wells. 1995. Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE 1995)*. 117–128.

Sharma Chakravarthy and D. Mishra. 1994. Snoop: An Expressive Event Specification Language for Active Databases. *Data & Knowledge Engineering* 14, 1 (1994), 1–26.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*.

Gianpaolo Cugola and Alessandro Margara. 2009. RACED: an adaptive middleware for complex event detection. In *Proceedings of the 8th Workshop on Adaptive and Reflective Middleware (ARM 2009)*. 5.

Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *DEBS*. 50–61.

Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (2012), 15.

Umeshwar Dayal, Barbara T. Blaustein, Alejandro P. Buchmann, Upen S. Chakravarthy, Meichun Hsu, R. Ledin, Dennis R. McCarthy, Arnon Rosenthal, Sunil K. Sarin, Michael J. Carey, Miron Livny, and Rajiv Jauhari. 1988. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD Record* 17, 1 (1988), 51–70.

Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. 2007. Cayuga: A General Purpose Event Monitoring System. In *Third Biennial Conference on Innovative Data Systems Research (CIDR 2007)*. 412–422.

EsperTech. 2013. Event Processing with Esper and NEsper. (2013). http://esper.codehaus.org/ Last accessed 11/2013.

Tim Furche, Georg Gottlob, Giovanni Grasso, Xiaonan Guo, Giorgio Orsi, Christian Schallhart, and Cheng Wang. 2014. DIADEM: Thousands of Websites to a Single Database. *Proceedings of the VLDB Endowment / International Conference on Very Large Databases (PVLDB'14)* 7, 14 (2014), 1845–1856.

Tim Furche, Georg Gottlob, Giovanni Grasso, Christian Schallhart, and Andrew Jon Sellers. 2013a. OXPath: A language for scalable data extraction, automation, and crawling on the deep web. *The VLDB Journal* 22, 1 (2013), 47–72.

Tim Furche, Giovanni Grasso, Michael Huemer, Christian Schallhart, and Michael Schrefl. 2013b. Bitemporal Complex Event Processing of Web Event Advertisements. In *14th International Conference on Web Information Systems Engineering (WISE 2013)*. 333–346.

Antony Galton and Juan Carlos Augusto. 2002. Two Approaches to Event Definition. In *13th International Conference on Database and Expert Systems Applications (DEXA 2002)*. 547–556.

Stella Gatziu and Klaus R. Dittrich. 1993. Events in an Active Object-Oriented Database System. In *Proceedings of the 1st International Workshop on Rules in Database Systems (RIDS 1993)*. 23–39.

Narain H. Gehani and H. V. Jagadish. 1991. Ode as an Active Database: Constraints and Triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB 1991)*. 327–336.

Jonathan Goldstein, Mingsheng Hong, Mohamed Ali, and Roger Barga. 2007. Consistency Sensitive Streaming Operators in CEDR. (2007). http://research.microsoft.com/pubs/70517/tr-2007-158.pdf Technical Report, MSR-TR-2007-158, Microsoft Research, Dec 2007.

IBM. 2013. InfoSphere Streams. (2013). http://www-03.ibm.com/software/products/en/infosphere-streams/

Elena Ilina, Claudia Hauff, Ilknur Celik, Fabian Abel, and Geert-Jan Houben. 2012. Social Event Detection on Twitter. In *Proceedings of the 12th International Conference on Web Engineering (ICWE 2012)*. 169–176.

Hans-Arno Jacobsen, Alex King Yeung Cheung, Guoli Li, Balasubramaneyam Maniymaran, Vinod Muthusamy, and Reza Sherafat Kazemzadeh. 2010. The PADRES Publish/Subscribe System. In *Principles and Applications of Distributed Event-Based Systems*. 164–205.

Robert A. Kowalski and Marek J. Sergot. 1986. A Logic-based Calculus of Events. *New Generation Computing* 4, 1 (1986), 67–95.

Jochen Kranzdorf, Andrew Jon Sellers, Giovanni Grasso, Christian Schallhart, and Tim Furche. 2012. Visual OXPath: robust wrapping by example. In *Proceedings of the 21st World Wide Web Conference (WWW 2012)*. 369–372.

Guoli Li and Hans-Arno Jacobsen. 2005. Composite Subscriptions in Content-Based Publish/Subscribe Systems. In *Proceedings of the 6th International Middleware Conference (Middleware 2006)*. 249–269.

Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. 2007. Event Stream Processing with Out-of-Order Data Arrival. In *27th International Conference on Distributed Computing Systems Workshops (ICDCS 2007 Workshops)*. 67.

Daniel F. Lieuwen, Narain H. Gehani, and Robert M. Arlein. 1996. The Ode Active Database: Trigger Semantics and Implementation. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE 1996)*. 412–420.

David C. Luckham. 1998. Rapide: A Language and Toolset for Causal Event Modeling of Distributed System Architectures. In *Proceedings of the Second International Conference on Worldwide Computing and Its Applications (WWCA 1998)*. 88–96.

David C. Luckham. 2006. What's the Difference Between ESP and CEP? (2006). http://www.complexevents.com/2006/08/01/whatE28099s-the-difference-between-esp-and-cep/

David C. Luckham and James Vera. 1995. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering* 21, 9 (1995), 717–734.

Masoud Mansouri-Samani and Morris Sloman. 1997. GEM: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering* 4, 2 (1997), 96–108.

Carlos A. Mareco and Leopoldo E. Bertossi. 1999. Specification and Implementation of Temporal Databases in a Bitemporal Event Calculus. In *Proceedings of the First International Workshop on Evolution and Change in Data Management (ECDM 1999)*. 74–85.

Dennis R. McCarthy and Umeshwar Dayal. 1989. The Architecture Of An Active Data Base Management System. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data (ACM SIGMOD Conference 1989)*. 215–224.

Oracle. 2013. Oracle CEP. (2013). http://docs.oracle.com/cd/E16764_01/doc.1111/e14476/overview.htm

Norman W. Paton and Oscar Díaz. 1999. Active Database Systems. *Comput. Surveys* 31, 1 (1999), 63–103.

Peter R. Pietzuch, Brian Shand, and Jean Bacon. 2003. A Framework for Event Composition in Distributed Systems. In *Proceedings of the International Middleware Conference (Middleware 2003)*. 62–82.

Esther Ryvkina, Anurag Maskey, Mitch Cherniack, and Stanley B. Zdonik. 2006. Revision Processing in a Stream Processing Engine: A High-Level Design. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE 2006)*. 141.

Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter R. Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS 2009)*.

Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. *IEEE Computer* 19, 9 (1986), 35–42. DOI: http://dx.doi.org/10.1109/MC.1986.1663327

Suryanarayana M. Sripada. 1988. A logical framework for temporal deductive databases. In *Proceedings of the Fourteenth International Conference on Very Large Data Bases (VLDB 1988)*. 171–182.

Utkarsh Srivastava and Jennifer Widom. 2004. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2004)*. 263–274.

Sybase. 2012. Sybase Event Stream Processor 5.0. (2012). http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc01612.0500/doc/pdf/ccl_programmers.pdf

Douglas B. Terry, David Goldberg, David A. Nichols, and Brian M. Oki. 1992. Continuous Queries over Append-Only Databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data (ACM SIGMOD Conference 1992)*. 321–330.

TIBCO. 2013a. BusinessEvents. (2013). http://www.tibco.com/products/event-processing/complex-event-processing/businessevents/default.jsp

TIBCO. 2013b. Tibco StreamBase. (2013). http://www.streambase.com

Walker M. White, Mirek Riedewald, Johannes Gehrke, and Alan J. Demers. 2007. What is "next" in event processing?. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 11-13, 2007, Beijing, China*. 263–272.

Roel Wieringa. 2003. *Design methods for reactive systems - Yourdon, Statemate, and the UML*. Morgan Kaufmann. I–XXV, 1–456 pages.

Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (ACM SIGMOD Conference 2006)*. 407–418.