Application of a Temporal Database Framework for Processing Event Queries

By

Foruhar Ali Shiva

A Proposal for Research in Partial Fulfillment

Of the Requirements for the Degree

Doctor of Philosophy

Arizona State University

July 2011

ABSTRACT

Event Stream Processing enables online querying of streams of events to extract relevant data in a timely manner. Due to semantic ambiguity and lack of expressiveness, event stream processing languages have been moving away from point-based semantics, where an event is associated with a single time of occurrence, towards increasing adoption of interval-based semantics, where an event occurs over a period of time with a starting and ending timestamp. Transforming point-based operators to interval-based counterparts is not sufficient to capture the complex relationships possible between interval-based events. Temporal databases and temporal query languages have been a subject of research for more than 30 years and are a natural fit for expressing queries that involve a temporal dimension. This proposal presents a research plan for the investigation of an event query language that incorporates temporal relational operators to provide a higher degree of expressivity for event queries. The approach is based on extending a preexisting relational framework for event stream processing to support temporal queries. The language features and formal semantic extensions required to extend the relational framework are identified. The research includes the development of a prototype that supports the integrated event and temporal query processing framework, with support for incremental evaluation and materialization of intermediate results.

Table of Contents

1. Introduction

The increasing availability of network infrastructure and bandwidth has increased the prevalence of system designs in which autonomous entities share data through the use of events. An event represents a data notification pertaining to a specific instant in time. For example, in a smart home application, temperature readings or the entrance or exit of people from a room can be modeled as events. Homeland security, supply chain management, and any environment where sensor readings and event occurrences are monitored are examples of the many applications that require online processing of streaming data.

In event-driven applications, the direct communication of event data is fast, push-based, and, in general, happens when an entity decides it has new data to share. Based on the input, any recipient of the event might extract information that can be used to invoke actions or to generate new events. Extracting relevant information from these continuous streams of events is an important and relatively new challenge. The detection of events, and in particular, the detection of complex relationships among several different events, needs to happen in real-time. The simplest example of a temporal relationship between two events is that of a sequence, where one event occurs in time before the occurrence of another event.

The process of detecting complex events is similar to query processing over data streams. In contrast with traditional database queries, event stream queries add a temporal axis to existing data schema, which creates the need for languages that are enabled to query temporal data. Two distinct but related approaches are being pursued for querying this streaming data: Complex Event Processing (CEP) (Barga & Caituiro-Monge 2006; M. Eckert 2008; Eugene Wu et al. 2006; Cugola & Margara 2010; Gyllstrom et al. 2006) and Data Stream Processing (Patroumpas & Sellis 2006; Arasu et al. 2006). CEP typically focuses on detecting patterns in the order of occurrence of the incoming events. DSP, in contrast, focuses on filtering and aggregation of stream data. Another term that is found in the literature is Event Stream Processing (ESP). ESP is often used analogously to Complex Event Processing in the context of streams of events arriving over the network.

Most of the earlier work in event processing considers events to be instantaneous and subsequently assigns a single timestamp value to each event. However, it has been demonstrated that point-based semantics cannot always correctly capture desired semantics in different applications. Additionally point-based events do not always lend themselves well to intuitive operator semantics.

These issues have led to the increasing adoption of an interval-based model in the research community. The interval-based model assigns a begin time and end time to each event. The interval-based approach does not suffer from the same semantic issues as the point-based approach, and it also supports sequence semantics in a way that upholds transitivity, which is intuitively expected.

The concept of temporal databases as databases that are used for storing historic data has been a subject of research for several decades but has not yet been commercialized (Date et al. 2002; Etzion et al. 1998; Snodgrass 1995). Interest in temporal databases stems from the desire to assign a validity interval to relational data. While there are simple ways for a relational database to assign a time interval to tuples, querying this temporal data is not always straightforward. The field of temporal databases has emerged primarily to deal with these challenges. In essence, temporal databases are databases that operate on temporal data. Their expressivity is analogous to that of relational databases with support for correlation of temporal data. Unlike ESP, temporal database queries are not specifically concerned with continuous processing of events that arrive over a stream of data. Temporal databases are typically used to query data that has a temporal dimension, sometimes referred to as historic data, such as periods of time during which a supplier supplied some part in a specific city.

Current work in stream and event processing languages impose limitations on the expressivity of the query language in different ways. Stream processing systems typically support an SQL-like language to query the data. To cope with unbounded input streams, stream processing languages require the extensive use of windowing to avoid running into unbounded wait times and memory requirements. An example of a classic stream processing language is Continuous Query Language (CQL) (Arasu et al. 2006). Several other languages employ the same concepts introduced in CQL (Purich 2010; Morrell & Stevan D. 2007; Esper). Event processing languages typically follow a composition operator-based model. Such languages offer a set of event operators, such as sequence or conjunction, which detect how the operand events are temporally related. The output of each operator is also an event.

A more recent approach to querying event streams can be found in XChange^{EQ} (F. Bry & M. Eckert 2007). In contrast to composition operator-based languages, the XChange^{EQ} approach treats detection of temporal relationships and event composition as two separate query dimensions. This separation of concerns results in more expressivity and clearer semantics. But the semantics of what constitutes an event places strict schema and temporal restrictions on the query input and output.

In contrast, temporal databases don't impose any temporal conditions in the use of querying constructs. In fact, temporal databases can be viewed as a generalization of relational databases (Date et al. 2002) in which temporal versions of traditional relational database operators are supported. Traditional database operators can be viewed as a specialization of this model in which the temporal dimension is ignored.

The approach in (F. Bry & M. Eckert 2007) performs complex event processing using operators that are implemented using a restricted version of relational algebra with constructs added for event composition. This is significant because it demonstrates a fundamental compatibility between relational database queries and complex event processing. However, the restrictions imposed do not allow the same level of temporal expressibility as that found in temporal databases.

The proposed research aims to bridge the gap between complex event processing and temporal databases by developing a framework that would allow the use of temporal relational operators for querying event data. The proposed research involves extending a relational framework with language features and architectural requirements for on-line processing of temporal event queries.

Different Architectures have been used for efficient evaluation of event queries. Much of the work in the area of Active Databases CEP systems use a Finite State Automata (FSA) or a similar structure for evaluating event queries (Gyllstrom et al. 2006; Brenna et al. 2007; Gatziu et al. 1995). In these approaches, the FSA is an internal representation of the composite event definition. The FSA model enables sharing of intermediate results between different queries but cannot directly handle concurrent occurrences of events or negation. The FSA approach is also rigid in terms of the order of operations performed. The use of Petri-Nets has also been proposed in a similar manner (Gatziu et al. 1995).

Some of the more recent approaches in this area such as XChange^{EQ}. (F. Bry & M. Eckert 2007) and ZStream (Mei & Madden 2009), utilize a query processing approach. The query processing model, aside from being well understood, has the benefit of building on already established query optimization techniques. It also has the flexibility of reordering operations for optimization purposes. ZStream propagates events from event buffers through an internal tree structure when an evaluation cycle is triggered. Language-wise it is a composition operator-based language with a mandatory window specification. The perceived limitation of data under consideration has led to the design decision of keeping track of all concerned events in all nodes of the evaluation tree.

In contrast, in the XChange^{EQ} model no mandatory temporal limitation is placed on the constituent events. The framework supports materialization of intermediate results which can be used for generating results more efficiently. Moreover the XChange^{EQ} model allows the flexibility of choosing which intermediate results should be materialized.

Adoption of interval semantics in event processing languages has adequately dealt with semantic problems with the point-based approach. However event operators originally defined in the point-based settings are not expressive enough to capture the complex queries which are possible in an interval-based setting. This research investigates the premise that operators developed in the context of temporal database query languages are a natural fit for querying interval-based event data and enable a new dimension for expressing event queries. The proposed research will develop a language that allows for querying of event streams using temporal database constructs while also supporting event composition. This research will include the development of a framework to support incremental evaluation of the language, which will also support materialization of intermediate results.

The rest of this proposal is structured as follows. A detailed overview of the work done in the related areas of stream processing, event processing, and temporal databases is provided in Section 2. The research objectives of this proposal are then introduced in Section 3, followed by a more detailed discussion of the research plan in Section 4. Section 5 presents the dissertation outline and timetable. Finally, Section 6 presents a summary of the proposed work together with expected contributions.

2. Related Work

This section provides an overview of related work. Section 2.1 outlines event processing work that has originated from the area of active database systems. Section 2.2 addresses the work done in the area of distributed event systems. Section 2.3 outlines issues with approaches that utilize composition operators. Section 2.4 provides an outline on work done in stream processing languages, by presenting a detailed look at CQL (Arasu et al. 2006) which provides the basis for many commercial stream processing languages. Section 2.5 presents the more recent work done in event stream processing. Section 2.6 addresses research on the specification of event processing languages. Section 2.7 describes the work done in the realm of temporal databases. Finally, section 2.8 summarizes and provides a discussion of relevant issues with existing approaches as presented in the related work section.

2.1. Composite Events in Active Database Management Systems

Complex event languages find their roots in active database research (Chakravarthy & Mishra 1994). The necessity to react to different kinds of events occurring within a DBMS gave rise to active databases that use the Event-Condition-Action (ECA) format for rules. In active databases, the event component is used to describe what has occurred. The condition component describes any conditions that need to be fulfilled, in order for the specified action to be taken. The condition is only checked upon occurrence of the event. The event itself can be a composition of different event types, in which case it is called a composite event.

SNOOP (Chakravarthy & Mishra 1994) is an example of a language that supports composite events for active databases. SNOOP provides the following operators for composing events:

Sequence: Detects the sequential occurrence of events.

Conjunction: Detects the occurrence of several events, regardless of their relative order.

Disjunction: Detects the occurrence of one out of several specified events.

A-Periodic: Detects the occurrence of an event, between the occurrences of two other specified events. Some languages implement an n-ary sequence operator which can provide a similar functionality.

ANY m out of n: As the name suggests this operator is similar to a disjunction operator, but it requires that a specified value 'm' out of the 'n' specified event types occur.

Non-Occurrence or Negation: This operator requires that the specified event does not occur. Negation needs to be bounded by a specified event or an amount of time.

Chakravarthy et. al. (Chakravarthy & Mishra 1994) argue that composing all possible combinations of input events would lead to a large number of composite events being generated that are not of interest to the programmer. On the other hand, these unwanted events cause a lot of overhead on the system by requiring a complete history of all involved events to be retained and matched against incoming events at all times. As a result, *consumption modes* (also called parameter contexts in some work) were defined to modify the standard behavior of the complex event statement to which they are applied. However, the operators themselves are only formalized in the unrestricted context in which all combinations of events available in the event history are considered.

As described in (Chakravarthy & Mishra 1994), the initiator and terminator of a complex event are defined as follows:

- The initiator of a complex event is the event that begins the process of detecting a composite event
- The terminator of a complex event is the event that ends the process of detecting a composite event and possibly generating an occurrence of the complex event.

The consumption modes introduced consist of the *recent*, *chronicle*, *continuous*, and *cumulative* modes. Consumption modes rely on the terminator of a composite event to decide which instances of constituent events are selected for generating the composite event. The recent and chronicle mode, cause the latest and earliest instances of the constituent events to be selected, respectively. With the cumulative mode, the parameters of all instances are accumulated to support aggregation of event parameters. The continuous mode will detect a complex event over all possible combinations of event instances.

SNOOP combines the notions of event selection and event consumption into a single dimension. This limits the language to specific predefined combinations of selection and consumption policies. Additionally, in SNOOP, consumption modes are associated with composite event statements and not the constituent event types. As a result, a sub-expression of a composite event is a different kind of entity than the composite event statement since a consumption mode cannot be associated with a sub-expression. This creates a non-uniform language composition and destroys the benefits of equivalence between sub-expressions and named composite events.

ODE (Gehani et al. 1992) is an active object-oriented database system that supports triggers and constraints. ODE combines the Event and Condition parts of an ECA rule, presenting an Event-Action model. The basic events are events that are found on a traditional database setting such as start and end of a transaction. Timer events including relative timers are also supported in ODE. ODE also supports specification of composite events utilizing operators such as conjunction, disjunction, sequence, negation and selecting the n^{th} event in a sequence of events. The event model assumes instantaneous events. The event specification follows a regular expression approach and it is implemented using an automata model.

SAMOS (Swiss Active Mechanism-Based Object-Oriented Database System) (Gatziu et al. 1995) is an active object-oriented database. In SAMOS, the primitive events can originate based on the database or be user-defined timer events. SAMOS supports a rule definition language to specify ECA rules. SAMOS attempts to simplify event specification by supporting a limited number of orthogonal event operators. These include the usual conjunction, disjunction, and sequence operators. In addition, a * and a last operator are used to signify the first time and last time an event type occurs. A history operator, TIMES(n,E), is used to detect that E occurs n times. The *, last, negation, and TIMES operators all need to be specified within a time window. The time window is specified using a begin and end time which can be the occurrence of an event, or relative or absolute event times. The * and last operator are simpler versions of what was later developed into consumption modes.

2.2 Events in Distributed Systems

The roots of complex event processing can also be traced back to distributed event based systems (Mühl et al. 2006). This is actually the field where the term “complex event processing” originates, in contrast to database systems where the term “composite event detection” was originally used. The main thrust of this line of research has been to provide a software layer that facilitates communication between the application and network layers in distributed applications for developers. This new layer, called the middleware, is utilized in large distributed systems (Luckham 2007) and implements a publish/subscribe model to support communication between different components. In this model, the publisher generates and submits event notifications. The subscribers can subscribe to events that interest them. The middleware is in charge of filtering the generated events and delivering them to appropriate subscribers.

Much of the focus of this area of research has been on efficient filtering of events with different filtering algorithms being developed targeting different distribution of components and different topologies (Bittner & Hinze 2004). A significant amount of the work in the field of distributed event-based systems focuses on subscriptions that specify topics in the form hierarchies or allow specification of simple value filters on the notifications (Mühl et al. 2006). However, most of the work in this area does not address the correlation of events or specifications of temporal relations between events, which is a topic of primary interest to this research. These issues are primarily addressed in some of the more recent work (P. R. Pietzuch et al. 2004; Sánchez et al. 2003).

Similar to languages found in the database area, the Event Correlation Language (ECL) described in (Sánchez et al. 2003) is based on composition operators and follows an automata based model. A couple of languages in this area also support negation (Dong Zhu & Sethi 2001; Hinze & Voisard 2002). In these languages negation needs to be specified together with a time window. The work in SEL (Dong Zhu & Sethi 2001) supports several kinds of fixed and sliding windows, with the option to have one end of the window fixed and the other sliding.

In general the work in this area is more concerned with efficient routing of subscriptions to the subscribers and reducing network load, and besides a few notable exceptions, does not focus on composite events or temporal conditions. As seen above, when these issues are addressed, the approaches are similar to work found in active databases.

2.3 Issues with Composition Operator-Based Approaches

As mentioned in the previous sections, using operators that compose simpler events into more complex events has been a very popular approach for languages that are used to process events in different settings. These languages are sometimes collectivity referred to as event algebras, which reflects the fact that they produce new events by combining different events using the provided operators. However the term algebra is not very precise for several reasons, one of which is that the operators supported are not always binary operators. In this proposal, this group of languages is referred to as composition operator-based languages.

The composition operator-based approach for specifying patterns of events seems intuitive at first glance. However, existing implementations of this approach suffer from several problems, among them disagreeing and often ambiguous semantics across different languages (Zimmer & Unland 1999). Semantic issues with the composition operator-based approach have been presented in previous work, for example, in (W. White et al. 2007). A good overview can be found in the literature review and chapter 17 of (M. Eckert 2008).

As described in (M. Eckert 2008), the operators commonly found in these languages include sequence (;), conjunction (^), disjunction (V) and negation. Several languages add more operators, but the added operators are not always orthogonal to these common operators. Additionally, some languages support sequence, conjunction, and disjunction in binary format, while other introduce n-ary versions of them which can take more than two events as operands. In the remainder of this section an overview of issues with existing languages is presented:

<u>**The sequence operator:**</u> The concept behind the sequence operator is straightforward; it is supposed to signify when two events of specified types happen after each other. However existing implementations offer different semantics for the sequence operator.

Some languages require that for a sequence operator to be detected no other event should occur between the two events participating in the sequence. This requirement is sometimes referred to as an event immediately following the other one. Other languages, however, do not have an immediacy requirement and trigger the sequence regardless of whether or not any other event happened in between. A third group of languages will fire the sequence as long as no other event present in the event pattern occurs between the two. For example if the pattern being detected is C^(A;B) if an instance of B follows an instance of A, the sequence operator will be fired as long as an instance of C does not occur between them.

Another issue with the Sequence operator more specifically applies to languages that do not support interval based semantics. Intuitively, it would seem that the sequence operator should be associative. For example, the event pattern (A;B);C should be equivalent to A;(B;C). In practice however this is not the case (W. White et al. 2007). Keeping in mind that the timestamp of the second operand is assigned to the sequence operator, it can be observed that the first pattern requires that A happens before B and B happens before C. The second pattern however requires that A and B both happen before C. It places no restriction on the relationship between A and B. As so, if the events b, a and c happen in that order, the second statement will detect this as a valid pattern while the first one will not. This issue can and has been addressed in interval based languages which assign an interval to the sequence operator.

Finally, it should be noted that when interval semantics are employed, the relationship between two events is no longer restricted to one happening after the other. Allen (Allen 1983) identifies 13 different relationships which are possible between two intervals: before, contains, overlaps, meets, starts, finishes, equals, and their inverses after, during, overlapped by, met by, started by, finished by. Equals does not have an inverse. The work in (Zaidi 1999) further extends these relationships by distinguishing between time points and intervals and the relationships they can have with each other. As is noted in (M. Eckert 2008), there is an important difference between these relationships and composition operators. A relationship merely maps to a true or false value based on the intervals that are submitted to it. However an operator takes two events as input and generates an output stream of events. The language in (Roncancio 1999) proposes operators based on these relationships. Besides the sequence operator, which maps to the before relationship, there is usually no n-ary extension to binary operators based on Allen's relationships. As will be discussed later, specifying temporal distance can be problematic with binary operators.

The conjunction operator: The conjunction of two events occurs when an instance of each one occurs, regardless of which one happened first. While the symbol for conjunction in event processing is borrowed from logic, it is important to note that the interpretation of conjunction is different in event processing than its counterpart in logic. Consider the following example:

(A^B);C

The logical interpretation dictates that A and B both need to occur before C. However this statement cannot be rewritten as:

(A;C)^(B;C)

While logically the two statements seem to be saying the same thing, the second statement allows for two different instances of C to be used to trigger the event pattern. For instance the pattern of events a, c1 , b, c2 causes the first statement to be fired once using c2, while the first statement will fire twice, first using c1 in conjunction with a and then c2.

The conjunction operator is called temporal conjunction in (M. Eckert 2008) to avoid the confusion. In general reusing the same event type more than once in an event pattern can lead to confusion. One work around could be to enable the user to specify aliases for the event types being used, thus signifying whether an event type that is referenced more than once needs to always refer to the same instance of the event or is bound to instances independently of other references.

The disjunction operator: The disjunction operator fires when one or the other of its operands occurs. In general, the semantics of disjunction are straightforward. However, there are different options on what can happen when more than a single operand occurs. Consider the following example from (M. Eckert 2008):

A; (BvC);D

If both b and c occur between a and d, the entire event sequence might be triggered once or twice depending on how the semantics of the language is defined.

Negation: Negation signifies that a specified event has not occurred. Since the incoming stream of events is unbounded, a temporal bound needs to be specified, after which the non-occurrence of an event can be signaled. (Chakravarthy & Mishra 1994) disagree with treating Negation as an operator. Since it's semantics do not conform with the basic requirement of something happening at a specific time. Alternatively, negation is sometimes proposed as a separate condition instead of an operator to avoid this theoretical objection.

Specification of Temporal Distance: Finally, many languages couple composition operators with the specification of a temporal distance. This allows the user to specify not only the order of events participating in an operator but also how far they are allowed to be from each other. For example $(A;B)_{1h}$ specifies that an instance of B needs to happen after an instance of A but within 1 hour from it. This approach seems straightforward, but has some unexpected restrictions as demonstrated in (M. Eckert 2008). For example, assuming interval based semantics, if we want to detect a pattern in which A happens and then B happens within an hour of A. If C must happen within an hour of that same B, this query cannot be expressed in most existing languages. The two candidate expressions for specifying this query are:

$((A;B)_{1h};C)_{1h}$

$(A; (B;C)_{1h})_{1h}$

It should be noted that both of the above queries require that C occur within an hour of A, a requirement which was not present in the original query.

This concludes the overview of issues in composition operator-based languages. Essentially it can be argued that the composition operators mix two separate query dimensions together. For instance, the sequence operator simultaneously is composes two different events together while imposing a temporal condition over them. The approach of this proposal is similar to that of XChange^{EQ} in which event composition and temporal conditions need to be specified separately. This approach also avoids issues discussed above. The proposed language design is discussed in Section 4.1. The following section addresses the related work in the areas of stream processing languages.

2.4. Stream Processing Languages

Stream Processing Languages aim to query data that is arriving over incoming streams. The term stream is used to typically signify data that is continuously arriving over a network medium. In this setting, the input streams have a very high throughput and so it is undesirable or even infeasible to store the incoming data before processing it. Stream processing languages themselves tend to resemble standard SQL found in relational database systems. Instead of tables, streams of data are considered. Some of the more prominent stream processing research projects include (Abadi et al. 2003; Arasu et al. 2006).

The requirement to process queries without the ability to indefinitely store the data, coupled with the fact that the input streams are theoretically unbounded, raises the necessity to limit the number of elements under consideration. This necessity is met by specifying windows, which are sets of recent elements that are currently under consideration. The number of items in the window is limited by either a specific number of events, or based on a specified duration of time. There are also other kinds of windows that determine whether a tuple is allowed in the window or not, based on its data value. These kinds of windows are referred to as semantic windows (Qingchun Jiang et al. 2007).

CQL (Continuous Query Language), developed at Stanford University (Arasu et al. 2006), formalizes some of the important concepts in Stream Processing Languages and is the basis for several query languages used in commercial products (Purich 2010; Morrell & Stevan D. 2007; Esper). The following subsection elaborates on CQL as a representative example of stream processing languages and then outlines different forms of window semantics as described in (Patroumpas & Sellis 2006).

2.4.1 CQL

CQL considers windows as time-varying relations. A time-varying relation, R(t), at any point in time t, will include a set of tuples that conform to the window specification associated with it. The tuples in a time-varying relation do not persist in the relational sense. New tuples are added to the relation, and older ones are removed from the relation based on the window specification, and without requiring a data manipulation operation. CQL provides three classes of operators: Stream-to-Relation, Relation-to-Relation, and Relation-to-Stream. The original semantics for CQL lack Stream-to-Stream operators and the bulk of data manipulation is performed using the Relation-to-Relation operations, which are equivalent to SQL operations.

Stream-to-Relation Operators:

Stream-to-Relation operators are window specifications. Three main kinds of sliding windows are supported: time-based, count-based, and partitioned windows. Sliding windows, in principle, add the newest items appearing in the input stream to the window, and remove from the oldest items. A time-based sliding window always contains the tuples that have appeared in a specified length of time since the current time. Count-based windows always contain the last n items that have appeared on the input stream. Partitioned windows first form sub-streams from the input stream based on the values of a specified set of attributes, similar to a Group By clause in SQL. Then the last n tuples from each group are retained in the window. There are also a couple of special kinds of windows that are useful for formulating different queries. The Now window, always contain tuples that have arrived within the duration indicated by the current value of the timestamp. The Range-Unbounded window contains all items that have ever appeared on the stream.

Relation-to-Relation Operators:

These consist of standard SQL data manipulation operations, with the difference that they operate on time-varying relations instead of standard ones. The following example taken from (Arasu et al. 2006) demonstrates a stream-to-relation operation, and relation-to-relation operations.

```
select distinct vehicleId
from posSpeedStr [range 30 seconds]
```

A time-based window with the length of 30 seconds is first defined over the input stream posSpeedStr. The result is a time-varying relation. SQL-like operations can be performed over this time-varying relation. In the above example, vehicles with distinct identifiers are selected from the time-varying relation. This makes the result of the query another time-varying relation.

Relation-to-Stream Operators:

Relation-to-Stream operators take a time-varying relation as input and produce a stream as output. There are three Relation-to-Stream Operators:

Istream, or Insert Stream - returns the stream of tuples as they are inserted into a relation.

Dstream, or Delete Stream - returns the stream of tuples as they are removed from a relation.

Rstream, or Relation Stream - if any item exists in R(t) it will also be in the stream with timestamp t.

The following example from (Arasu et al. 2006) demonstrates the use of the Insert Stream operator in combination with a range unbounded window:

```
select istream(*)
from posSpeedStr [range unbounded]
where speed > 65
```

First an unbounded window is defined over the input stream, which results in a time-varying relation that will include every tuple that appears in the stream. The filtering operation declared in the where clause omits tuples with speed less than 65 from the relation. Finally, the istream operator outputs every new tuple that is added to the time-varying-relation as a new stream.

As mentioned before, CQL in its original form lacks Stream-to-Stream operators. This implies that even for simple filtering of a stream, a window definition and subsequent transformation into a Stream is necessary. Operators similar to event composition operators were not originally supported. Some of the later work in languages based on CQL, such as Oracle CQL (Purich 2010), support a limited number of direct stream-to-stream operations. Most significantly the MATCH_RECOGNIZE clause in Oracle CQL allows patterns of values, such as a W-pattern[1] in the value of some attribute to be detected.

CQL cannot be extended to support a continuous time model because time-varying relations and Relation-to-Stream operators require discrete time points. CQL does not support semantic or value-based windows which can be useful to group more meaningful data together. Tuples, while within a window, follow the relational standard of belonging to a set/bag. As members of a set/bag, tuples are no longer ordered. This implies that the relation-to-relation operators which are designed to undertake the bulk of data manipulation in CQL, cannot make use of any ordering information or detect a temporal pattern.

Istream and Dstream act purely on data values. If a new tuple with the same data values of a tuple that is about to be removed from the window arrives, the insertion of the new tuple and the deletion of the old one will be missed by Istream and Dstream, respectively. This could be rectified by defining other versions of Istream and Dstream that are sensitive to timestamps.

2.4.2 Window Semantics

Trying to directly apply data manipulation operations from the database world to stream data is problematic. Given the high volume of stream data, it is usually not possible to have to store the entire stream history, and even if it were possible it may not be practical to have to examine the entire history of incoming streams to produce a new result. Even if we limit ourselves to monotonic queries that only append new results, there are various problems that can arise. Blocking operators such as aggregation are unable to produce a single result without access to the entire event history. Other operators that are stateful such as join or intersection also cause problems due to the fact that every newly arriving tuple needs to be matched against the entire event history. Patroumpas and Sellis formalize window semantics for continuous queries over data streams in (Patroumpas & Sellis 2006). The importance of windows in Stream Processing systems is due to the ever increasing volume of incoming data. Windows, in effect, limit the number of items that are considered for evaluating a query at any moment. This also makes evaluation of blocking operators feasible, because they will only need to consider items that are currently in the window, instead of waiting on an input stream that never ends. On the other hand, even for non-blocking operators, the number of items that need to be considered at any given instant is bounded by the window. This becomes even more significant when the query requires a join-like operation that considers multiple input streams.

Patroumpas and Sellis offer a categorization of different kind of windows possible. First windows can be categorized into physical (or time-based), and logical (or count-based) windows. Additionally a window can be specified either by marking its two edges or by using an edge along with the desired size which can be either in logical or physical units. The window bounds can be fixed or variable, in which case a progression step can also be specified.

Different combinations of these conditions lead to different window types that are formalized using algebraic expressions in (Patroumpas & Sellis 2006). The main categorization of the windows is as follows:

Physical Windows: Physical windows are sliding by default and have the following sub-types:

Count-based Windows: A count-based window contains a fixed maximum number of the most recent tuples.

Partitioned Windows: Similar to a GROUP BY clause, a partitioned window divides an incoming stream into partitions that have the same value for a specified set of attributes. A count-based window is then applied to each substream using a specified value. The contents of the partition at any given instance, is the union of the contents of these partitioned windows.

Logical Windows: Inclusion of events in a logical window relies on the timestamp of the incoming tuple. The following main logical window sub-types are defined:

- Landmark Windows**:** A landmark window encompasses different kinds of windows that have one bound fixed, but the other bound is allowed to progress with time.
- Fixed Band Windows**:** A fixed-band window has both bounds fixed to specific points in time.

[1] A W-pattern means the following pattern of values in an attribute: decrease, increase, decrease and a final increase

- Time Based Sliding Windows**:** In its basic form, a time-based sliding window contains the most recent items that occur within a specified amount of time.
- Time Based Tumbling Windows**:** Time-based tumbling windows rely on a hop unit, where the hop unit is equal to the window length. As a result no tuple is repeated in two consecutive states of the window. This can be useful when the tuples need to be aggregated in groups.

Patroumpas and Sellis additionally provide definitions for relational operations such as join, union and aggregation over these windows. Windows that may be defined over semantic conditions are not considered. An equivalent mechanism to consumption modes is also not offered.

2.5. Event Stream Processing

Event stream processing languages are used to detect temporal patterns of incoming events. The events themselves can be thought of as time-stamped tuples, with different event types having different headings. Events can be categorized into two kinds: primitive and complex events. Primitive events are events that are generated by the operational environment. Examples of these might be an operation in a database system, or a sensor or RFID reading. Complex events, on the other hand, are patterns of events detected by the event processing system. A complex event relies on several events occurring in a specified pattern. For example, a complex event may require that two primitive events happen concurrently or one after the other.

Event stream processing languages, while conceptually similar to event specification of ECA rules in active database systems, go beyond basic event patterns. ESP languages typically add features for condition filtering that are traditionally found in the condition part of an ECA rule. ESP languages also offer composition operators that include sequence, conjunction, disjunction, repetition, and the a-periodic operator as defined in (Chakravarthy & Mishra 1994).

The result of each operator is then a composite or complex event, which can then be input to other event operators to form even more complex events. Event processing languages typically do not support a window structure similar to that found in DSP Languages.

In terms of their architecture, composition operator-based languages usually use an automata or a similar structure internally to evaluate event patterns. While these kinds of structures are helpful for optimizations across multiple queries, unlike query plans, they are rigid in terms of their evaluation order. Also, it is not always straightforward to handle simultaneous occurrence of events using an automata structure.

The languages in (François Bry & Michael Eckert 2007; Mei & Madden 2009; Barga & Caituiro-Monge 2006), use a query plan based approach, which offers flexibility in terms of order of query operations similar to optimization of database queries. This leads to optimization opportunities in terms of space utilization for intermediate results that are materialized.

In the following sub-sections, an overview of some of the prominent event stream processing languages is provided.

2.5.1 SASE

SASE (Gyllstrom et al. 2006) is one of the first projects to utilize a query-plan based approach for event processing. The objective of SASE is filtering, correlation, and detection of complex event patterns over RFID streams of data. Similar to relational databases, SASE builds a query plan in the form of a tree structure, with the different query operators forming the nodes of the tree.

To leverage the fact that the data arrives over time and the queries are looking for sequences of data, SASE uses a special operation called Sequence Scan and Construction (SSC). SASE first extracts the specific sequence of events that are required from the query, ignoring negations. Checking for this sequence over incoming streams is the job of the SSC operator, which forms the basis for all query plans. SSC itself is implemented using an automata mechanism. SASE further optimizes the detection of event sequences by pushing predicates and time window conditions down in the query plan.

When composing events, SASE includes the entire heading of the contributing event. When only a subset of data is necessary for the resulting event, this causes an excessive amount of overhead. The situation can become worse if events with large headings or a correlation of numerous events types are being considered. Ultimately, SASE relies on composition operators; and shares some of the same semantic issues as described in Section 2.3. SASE does not support the use of complex events when specifying other complex events. In SASE, events are instantaneous. Interval-based semantics and aggregation operators are not supported.

2.5.2 MavEStream

MavESstream (Qingchun Jiang et al. 2007) aims to integrate the event and stream processing models. The proposed model uses a three layer approach. At the lowest layer, a stream processing engine handles incoming streams and performs filtering and aggregation operations. At the next layer event processing occurs over filtered streams that can be treated as input events. Finally, the third layer is a rule processing layer that can trigger rules based on incoming events. Event specification takes the following form in MavEStream:

```
CREATE EVENT Ename
SELECT A1, A2, ..., An
MASK Conditions
FROM ES | EX
```

Where Ename is a new event being defined and A1, A2, ..., An are attributes. A mask is an attribute-based constraint that can be pushed down to sources. ES and EX in the from clause represent the data sources that can be continuous stream queries or event pattern specifications.

A notable concept introduced in MavEStream is that of a semantic window. A semantic window is distinct from a count-based or time-based windows in that the window items are determined by a user-specified condition. The condition itself can involve the parameter values of the events involved. While this is conceptually a powerful tool, MavEstream's approach to implementing it does not seem very intuitive: the user-specified condition is iteratively checked by removing older items in the window. No checking is done on entry of new items into the window. This mechanism is rather inflexible: it enables simulation of time-based and count-based windows, but lack of control over events entering the window and strict, in-order, removal of existing items make the mechanism rigid and only appropriate for very specific applications.

2.5.3 CEDR

CEDR's (Complex Event Detection and Response) stated goal is to attempt to unify three different approaches to temporal data arriving over streams (Barga et al. 2006), namely: event processing, stream processing and publish/subscribe systems. The authors argue that the main difference between the different models is the workload that is expected of the target systems. In order to unify the three different models, different consistency levels are supported. A consistency level is based on a target system's tolerance to out of order processing of events. In order to achieve unification, the consistency level can be directly specified by the user. The entire stream is viewed as a time-varying relation, with each event playing the role of a tuple in the relation. All tuples also need to have a unique ID.

Three different timestamps are supported for achieving the diverse requirements of the underlying systems. The first dimension is the *validity interval* assigned by the provider of the event. The ID allows an event provider to reference an event which it has 'inserted' earlier and to change its validity interval or other attributes. The second time interval, called *occurrence time,* indicates when modifications are made by the provider. Conceptually, this second interval is comparable to transaction time in temporal databases (see section 2.7). Tuples with expired occurrence times are no longer valid from the point of view of the event provider and have been replaced with a tuple with the same ID and a currently valid occurrence time. This can be viewed as a modification operation from the relational database perspective.

The following CEDR event example is from (Barga et al. 2006):

```
EVENT CIDR07_Example
WHEN UNLESS(SEQUENCE(INSTALL x,SHUTDOWN AS y, 12 hours),RESTART AS z, 5 minutes)
WHERE {x.Machine_Id = y.Machine_Id} AND {x.Machine_Id = z.Machine_Id}
```

The language of CEDR uses a composition operator-based language. The WHEN clause is used to specify the event operators. Multi-ary operators to detect sequence and m out of n input events are supported. Negation is supported using multiple operators. In the example above, the UNLESS operator is an example of negation. The UNLESS operator, and by consequence the entire event, is triggered if the sequence is not followed by a 'Restart' event within 5 minutes. As can be seen, the sequence operator is similarly bound by a 12 hour time window. A mandatory time window is required for all event operators that correlate different input events.

In (Barga et al. 2006), it is said that event selection and consumption are supported, and the observation is made that selection and consumption need to be decoupled from operators and associated with input streams. This is a welcome observation and is likely to reduce the possibility of confused semantics sometimes found in other work. However, selection and consumption semantics are not presented nor are any examples provided.

The query evaluation is performed using pipelined operators that form a query plan. The third dimension, called CEDR time is the clock of the server and is used for handling out of order events. CEDR time cannot be queried by the language. By looking at CEDR time and the timestamp of the provider, it can be determined whether an event has arrived out of order or not. CEDR time is additionally used to correct incorrect outputs produced based on out of order events. The CEDR time of an output that is determined to be incorrect is terminated, and the correct output is produced with valid CEDR time assigned to it.

CEDR falls into the family of composition operator-based languages. CEDR allows the query language to define windows over the desired validity intervals or occurrence times for any given query. But manipulating timestamps is otherwise limited to the use of composition operators, which are limited in their expressivity. Materialization of different intermediate results and its effect on performance are not explored.

2.5.4 Cayuga

The Cayuga project (A. Demers et al. 2005) aims to develop an expressive stream processing language by extending event processing languages. The emphasis is on achieving high performance and scalability. In order to achieve this, an event algebra named CESAR (Composite Event Stream AlgebRa) is developed. CESAR is evaluated using an automata approach, which enables cross query optimizations, similar to those found in some publish/subscribe systems, e.g. (Y. Diao et al. 2002). As the name implies, CESAR uses event composition operators. The extensions to publish/subscribe event languages include support for value correlation and aggregation of events.

CESAR supports time intervals. The basic binary operators supported by CESAR are Union and Sequence. There is also an Iteration operator that can be used for checking for monotonic increase in a parameter value by repeated application of the sequence operator.

2.5.6 XChange^{EQ}

XChange^{EQ} is a more recent language that was designed for querying events. It is significant because it introduces a new style for querying events which is different from both composition-operator based event languages and data stream languages. The style introduced is similar to logical formulas, but the language is more user friendly and targeted specifically for events (M. Eckert 2008; François Bry & Michael Eckert 2007). In this section, we present an overview of the language design in XChange^{EQ} and discuss some of the areas in which it is deficient.

Bry and Eckert (François Bry & Michael Eckert 2007) argue that different querying dimensions have traditionally been mixed in Event Processing Languages. They identify four separate dimensions for an event query language. They are:

- **Data Extraction:** Refers to the data that needs to be extracted from input events. This is done through variable bindings. The data is then used for testing query conditions, comparing and combining with persistent data, constructing new events, or for triggering actions.
- **Event Composition:** This dimension refers to the constructs that allow different events to be juxtaposed and create complex events. The composition constructs are sensitive to data, allowing the correlation to be limited to equivalent data across the events.
- **Temporal relationships:** Refers to different query conditions that involve time. XChange^{EQ} breaks temporal relationships into two kinds: qualitative and quantitative. Qualitative relationships are concerned with the ordering of different events while quantitative ones are concerned with the amount of time elapsed between their occurrences.

- **Event Accumulation:** Aggregating and checking for non-occurrence of events over an infinite input stream require setting a bound on the number of events being considered. Event accumulation is the dimension that deals with this issue.

According to these criteria, composition operators, which can be found in previous work, such as the sequence operator involves two different dimensions. Since two (or more) events are used to produce a new one, a sequence operator involves the event composition dimension. Furthermore, since the two events need to be ordered in a certain way, the sequence operator involves the temporal relationship. Eckert and Bry argue that this is undesirable because it causes mixing of inherently separate concerns. As a result, they omit the sequence operator. The only composition operators supported in XChange^{EQ} are conjunction and disjunction. The qualitative, and quantitative temporal relationships are expressed in the where clause of an event query statement, with constraints on event time stamps.

The Temporal Model used in XChange^{EQ} uses time intervals to designate the occurrence time of events. For simple events, the start time and end time of the interval are assumed to be the same. For complex events, the time interval consists of the time period that subsumes all constituent events. XChange^{EQ} supports the thirteen different kinds of temporal relationships specified in Allen's interval temporal logic (Allen 1983).

XChange^{EQ} allows for the definition of two kinds of rules: Deductive and Reactive Rules. Deductive rules cause a new event to be generated based on the composite event statement, while reactive rules cause some other action to be performed within the system. XChange^{EQ} also aims to support declarative semantics. To achieve this goal, XChange^{EQ} avoids employing selection and consumption modes available in other event languages, which cause operators to behave differently based on the context in which they are being used.

XChange^{EQ}'s semantics work directly on streams. This is in contrast with the CQL approach, which utilizes conversion operators to transform streams to temporal relations to perform data manipulation operations, before transforming the relations back to a stream. This roundtrip conversion does not occur in XChange^{EQ}. XChange^{EQ} was also designed with interactions between Web-based systems in mind and thus events are expressed in the XML format. The language Xcerpt (Francois Bry & Schaffert 2003) is used both to specify classes of relevant events and to extract data through variable bindings. The following example is from (François Bry & Michael Eckert 2007), showing a deductive rule based on a conjunction operator with temporal conditions.

```
DETECT earlyResellWithLoss { customer { var C } ,
                             stock { var S } }
```

```
ON and {
        event b : buy {{ customer { var C } ,
                stock { var S } ,
                price { var P1 } }} ,
        event s : sell {{ customer { var C } ,
                stock { var S } ,
                price { var P2 } }}
  } where { b before s , timeDiff (b , s )<1hour , var P1>var P2 }
END
```

The composite event detects when a stock is bought and then sold by the same customer at a lower price. The reselling of the stock needs to happen in less than an hour after the original purchase. The buy event and the sell event need to agree on the value of the variables C and S, which signify the customer and stock, respectively. The conjunction of the two forms a composite event titled earlyResellWithLoss, which includes the customer and stock values. The interval associated with the output event spans the intervals of input events that are composed together. As can be seen in the example, temporal conditions are expressed in the where clause right next to value constraints.

As mentioned earlier, a mechanism is needed to limit the number of events that are being accumulated for negation or aggregation. Since events themselves happen over a time interval, and in the interest of having a simple language with fewer constructs, XChange^{EQ}'s accumulation construct consists of accumulating events while a certain event holds true. The event that specifies the accumulation period can be specified separately as a complex event or can be a relative or absolute timer event. XChange^{EQ} offers several operators that can stretch or shrink the interval of an input event in different directions. The following example from (François Bry & Michael Eckert 2007) displays checking for a negation which is bound by a relative timer event:

```
DETECT buyOrderOverdue { orderId { var I } }
ON and {
        event o : order {{orderId { var I }
                buy {{ }} }} ,
        event t : extend [ o , 1 min ] ,
        while t : not buy {orderId { var I } }
  }
END
```

The above query checks when a buy order is overdue. This is defined as an order event *not* being followed by a buy event within 1minute. The event t is a relative timer event. It is created by extending the o event by one minute. It is then used in the while clause to specify the window for the negation. It is important to note that unlike most other languages the while clause is not a separate clause, it is actually specified within and is participating in the conjunction operator. The while t: not q clause is successful if during the period specified by t, the query specified by q is not satisfied. The period associated with the clause is the same as the t event.

Comparing the XChange^{EQ} model to the CQL model, it could be said that two kinds of windows are supported: an unbounded window, when no limits are placed on the input, and a fixed window when an event interval is specified for accumulation. Sliding windows and all its different variants are not supported. Similarly cumulative-mode like behavior in which arrival of a new event spawns a new window is not supported.

XChange^{EQ} can possibly be augmented by addition of operators that simulate selection mode behaviour: First and Last selection modes can be simulated over a window as aggregate functions without undermining the 'declarative'-ness of the languages.

XChange^{EQ} does not allow direct access to the timestamp values. This is done in the interest of keeping the language simple but can be restrictive on the kind of conditions that can be placed on timestamp values. This issue will be further discussed in section 4.

2.5.7 Tesla

Tesla (Trio-based Event Specification LAnguage) (Cugola & Margara 2010) is a complex event processing language. Telsa supports value and temporal filters, timers, negation, aggregates, and specification of separate and customizable event selection and consumption. Many existing event processing languages only support event selection and consumption in a rigid predefined manner, sometimes combining the two together. The temporal model uses instantaneous timestamps. Whether the timestamps are issued by the event source or the event processing system and the prospect of out of order events are considered a separate issue from the language and are discussed in (Srivastava & Jennifer Widom 2004).

An example Tesla rule taken from (Cugola & Margara 2010) is:

```
define Fire(Val)
from Smoke() and each Temp(Val > 45)
within 5min from Smoke
```

```
where Val = Temp.Value
consuming Temp
```

The define clause defines the name and heading for the output event. The from clause is where the event pattern can be specified. In the above example, the keyword ‘each’, coupled with the ‘within’ condition specifies a selection policy which signifies that each occurrence of Temp that satisfied the value condition in a 5 minute time window will be used for generating a new event. The where clause assigns values to the output parameters. These values may come from the event pattern. The consuming clause is optional and specified the consumption policy. In the above example the Smoke event is being consumed, which means that no smoke event can be used twice for generating an output. Telsa aims to have precise semantics and is formalized using a temporal first order logic called TRIO (Ghezzi et al. 1990).

Telsa’s focus on flexible and precisely defined selection and consumption policies is a welcome one. However lack of support for interval semantics is a serious drawback. The approach of this proposal is to treat temporal conditions similar to any regular value conditions. Tesla moves away from this approach by specifying them in separate clauses. The implementation is based on finite state machines which do not offer the flexibility of query plans. Temporal relational operators are not supported as they require support for intervals.

2.5.8 ZStream

ZStream (Mei & Madden 2009) which is a more recent event processing language, like other event processing languages, supports direct specification of temporal patterns using event operators. The queries are evaluated using a query plan-based approach which supports dynamic reordering of the plan to achieve optimized evaluation. The plan is modeled using a tree structure with leaf nodes corresponding to primitive events and internal nodes representing event operators. Each node has a corresponding buffer, with leaf node buffers corresponding to incoming event streams and internal nodes storing intermediate query results. ZStream is designed so that all buffered events are stored in End-Time order. This facilitates discerning events that are within the time window formed by the final event and the specified length of time. While ZStream has a time window option, it does not offer any of the more complex window behavior found in stream processing languages, such as sliding windows. The language constructs offered in ZStream lack selection and consumption capabilities. ZStream does not support the more complex temporal relationships that are outlined in Allen (Allen 1983).

2.6 Specification and Formalization of Event Languages

Many existing event languages suffer from a lack of proper formalization. This has resulted in differing and ambiguous semantics. However some of the previous work in the event processing area do offer formalizations or attempt to take a formalized look at semantics of other languages. This section takes a brief look at some of the notable efforts in this area.

2.6.1 Zimmer and Unland's Meta-Model

Zimmer and Unland present a generic meta-model (Zimmer & Unland 1999) for describing event languages used in active database management systems. This meta-model is used to model features present in different event languages.

To achieve this model semantics, events are broken into three independent dimensions: event patterns, event selection, and event consumption, which operate on event histories. An event history for any given event type consists of all instances of that event.

The event pattern is concerned with defining the type and order of the events. It also includes the ability to specify a repetition count of a specific event.

Event instance selection determines, in case of the existence of multiple eligible instances of a constituent event, which instance[s] should be selected for generating the output event. The options include the selection of the first or last instances. A third option is using all instances of the input events that do not contradict the event pattern.

Event instance consumption determines whether or not the detection of a composite event, renders events used to detect that event ineligible for further consideration. Two options are possible: *shared* and *exclusive* consumption. Shared consumption means that constituent events may be used for generating further instances of that composite event. In contrast, exclusive consumption indicates that once a constituent event is used to generate an event, it is deleted from the event history of the composite event.

2.6.2 SNOOP and SNOOP-IB

The language SNOOP (Chakravarthy & Mishra 1994) is one of the most widely referenced works in the area of event processing. It also later adopted interval semantics in the form of SNOOP-IB (Adaikkalavan & Chakravarthy 2005). In (Galton & Augusto 2002), the authors analyze the semantics of the original SNOOP in order to make an argument for support of interval-based semantics. They introduce event *occurrence* and event *detection* as two separate concepts. Events may happen over a period of time. This period of time is referred to as the occurrence time of an event. However the system does not become aware of an event until that period is over. The point in time in which the occurrence of an event ends is when it is detected. The notation D(E, t) is introduced to indicate that an event of type E is detected at time t. Likewise, O(E, [t, t′]) indicates that event of type E occurs over the interval [t, t′].

Galton & Augusto continue to define the SNOOP operators using the introduced notations. For instance, the sequence operator is defined as:

$$D(E_1;E_2, t) = \exists t' < t\ (D(E_1, t') \wedge D(E_2, t))$$

Galton & Augusto illustrate several issues with the detection based semantics of SNOOP; for instance the equivalence of E_1; (E_2;E_3) and E_2; (E_1;E_3) using the above definition for the sequence operator. Consequently interval based or, as it is called here "occurrence-based", semantics are suggested for SNOOP. Under the occurrence-based semantics, the sequence operator is redefined as:

$$O(E_1;E_2, [t_1, t_2]) = \exists t, t'\ (t_1 \le t < t' \le t_2 \wedge O(E_1, [t_1, t]) \wedge O(E_2, [t', t_2]))$$

Definitions for the rest of the SNOOP operators are also presented in (Galton & Augusto 2002). The ideas in this paper lead to development of SNOOP-IB (Adaikkalavan & Chakravarthy 2005), which incorporates interval-based semantics.

Given the issues with point/detection based semantics, the move towards interval based semantics is a welcome one. However, incorporating intervals entails dealing with the more complex relationships that are possible between them. The set of operators available in SNOOP is simply not expressive enough to capture these relationships.

2.6.3 SASE

SASE assumes a total order of incoming events over a discrete time model. The formalization of event operators is then done similar to those of SNOOP. In particular, the occurrence of an event is defined as a mapping from the time domain to a Boolean value. The occurrence of an event at a specific time is true if that event occurs at that instant and false otherwise. For example, the ANY operator, which is triggered if any of its contributing events are triggered, is formalized as follows (Gyllstrom et al. 2006):

$$\mathrm{ANY}(A_1, A_2, \ldots, A_n)\ (t) \equiv \exists\ 1 \le i \le n\ A_i(t)$$

Which means that the truth value of the mapping for the ANY operator, at time t, is equivalent to the truth value of the occurrence of any one of the specified events at time t.

As discussed before operationally the operators are placed in a pipelined query plan. However, the Sequence and Scan operator is detected using an NFA mechanism. SASE does not support interval-based semantics.

2.6.4 Tesla

The Trio-based Event Specification LAnguage or Tesla (Cugola & Margara 2010) is one of the languages that does provide a formal definition, however this formal definition is based on Temporal Logic. The language and its evaluation were discussed in the previous section. Its formalization is discussed here. TRIO (Ghezzi et al. 1990) is a metric, first-order temporal logic, which is used to formally specify Tesla. The meaning of TRIO formulas depend on the current moment of time. TRIO includes two temporal operators called *Futr* and *Past*. The formula Past(A,t) is true if A was true t time units in the past, and Futr is similarly defined. There are also a few other temporal operators which are defined based on Past and Futr. For instance:

$$\mathrm{Alw}(A) = A \wedge \forall t\,(t > 0 \rightarrow \mathrm{Futr}(A, t)) \wedge \forall t\,(t > 0 \rightarrow \mathrm{Past}(A, t))$$

$$\mathrm{WithinP}\,(A, t_1, t_2) = \exists x\,(t_1 \leq x \leq t_1 + t_2 \wedge \mathrm{Past}(A, x))$$

For example consider the following Tesla rule from (Cugola & Margara 2010):

```
define Fire(Val)
from        Smoke() and
            each Temp(Val > 45) within 5min from Smoke
            where Val = Temp.Val
```

This rule raises a Fire event based on Smoke and Temp. For each Temp with value greater than 45 within 5 minutes from smoke, the fire rule is triggered. Below a simplified version of the TRIO formulation for the conjunction operator with the ‘each’ selection policy is presented:

$$\begin{array}{l}\text{define CE from A and each B within x from A =}\\ \mathrm{Occurs(CE)} \leftrightarrow\\ (\mathrm{Occurs(A)} \wedge \mathrm{WithinP}\,(\mathrm{Occurs(B)}, \mathrm{Time(A)}, x))\end{array}$$

CE is the composite event, A and B are the input events. Occurs is a predicate that is used to map the truth of a TRIO formula to the detection of an event. Further details of the formalization of Tesla are beyond the scope of this proposal and can be found in (Cugola & Margara 2010). Tesla does not support interval based semantics.

2.6.5 XChange^{EQ} and CERA

The work in XChange^{EQ} uses an algebra closely based on relational algebra to describe the operational semantics of the language. This algebra is called Composite Event Relational Algebra (CERA). XChange^{EQ} works with XML data. But for the sake of its operational semantics, it is assumed that the XML data is stored as relational attribute values. The XChange^{EQ} operators are then formalized based on relational operators and concepts.

In CERA, event histories are treated as relations. CERA imposes several restrictions on relational algebra and also introduces several short-hands for ease of writing frequently used constructs.

In CERA, begin and end timestamps of events are treated as attributes in a relation. Projections that result in the omission of the timestamps are not allowed. Direct modification of the timestamp values is also not allowed. As mentioned before, since XChange^{EQ} deals with events that are expressed in XML, CERA adds *Matching* and *Construction* operators that enable mapping of XChange^{EQ} queries to a relational algebra-based model. To simplify discussion of issues, a relational version of XChange^{EQ} called Rel^{EQ} is introduced in (M. Eckert 2008). In Rel^{EQ} incoming events are assumed to be relational tuples instead of the XML format which is expected in XChange^{EQ}. The following is a simple example of a Rel^{EQ} query from (M. Eckert 2008).

comp(id, p) ⟵ o : order(id, p, q), s : shipped(id, t), d : delivered(t) o before s, s before d, {o, s, d} within 48

The above event signals a completed order which is defined as an order event followed by a shipped event which itself is followed by a delivered event. Additionally this sequence of events needs to be completed within 48 hours. The relational algebra expression for the above query can be expressed as:

$$
\begin{aligned}
&\sigma[\max\{o.e, s.e, d.e\} - \min\{o.s, s.s, d.s\} \leq 48]\\
&\qquad (\sigma[s.e < d.s](\\
&\qquad\qquad \sigma[o.e < s.s](\\
&\qquad\qquad\qquad (R_o \bowtie S_s) \bowtie T_d)))
\end{aligned}
$$

For ease of reading, instead of the traditional subscript notation, brackets are used for specifying the parameters of relational operators. The variables s and e are the beginning and ending timestamps of events, respectively. The above expression assumes full knowledge of every event that has happened in the past or will happen in the future. In reality, we have access to past events but future events are yet to come, and the relational expression needs to be evaluated in a continuous, step-wise manner. A more detailed look at CERA along with Step-wise evaluation of relational algebra expressions in XChange^{EQ} are presented in Section 4.

The model presented in (M. Eckert 2008; François Bry & Michael Eckert 2007) has the following limitations:

- Strict requirements on the timestamps: The output timestamp can only be produced using the merging operator which is required to appear in every rule head which specifies the heading for the output. Since extended projection, which allows for modification of heading attribute values, is not allowed, output timestamps cannot be modified. For example, to modify the starting timestamp or postpone the ending timestamp of a produced event is not allowed.

- No interaction between event data and timestamp data: Since the timestamps can only be accessed in very specific ways, comparison or modification of timestamps based on event data is not possible. This can be a significant shortcoming if the event itself includes data that is temporally relevant.

Further, because specification of temporal conditions is only possible through the use of specific operators, the kinds of conditions that can be expressed on them even without the use of event data is limited. More recent work has tried to ally this problem by providing a longer list of operators, which are more appropriately restricted or open (Walzer et al. 2008). Such extensions might fine tune the language for a specific application. However, they do not address the root cause of expressivity issues which are a result of hiding timestamps. The benefits of providing direct access to timestamps will be further discussed in Section 4.1.

2.7 Temporal Databases

Temporal Databases (Etzion et al. 1998) are databases that maintain historical data, which is data with an added temporal axis. Conceptually the temporal axis associated with the data might reflect an interval in the past, in the future, or currently ongoing. There are multiple different proposals for modeling of temporal databases. Two of the most prominent proposals are the approach outlined in (Date et al. 2002) and the approaches based on TSQL2 (Snodgrass 1995).

Both of the aforementioned proposals model temporal databases by associating an interval with each tuple in a table that is to hold historical data. In general more than a single interval attribute may be used in a relation. A common approach is to use two interval attributes, indicating *valid time* and *transaction time*. Valid time reflects the interval during which the predicate associate with a tuple is assumed to be true. For example, if a tuple reflects the fact that a supplier supplied a part in a city, the associated valid time interval reflects the period during which this was true. In contrast, transaction time is used to signify the state of the data in the database and can be used, for instance, to model deletion operations in a non-temporal database, without actually deleting the underlying data. In this case, only data with transaction time values that include the current moment in time are considered to reflect the current state of the database. Conversely, tuples with transaction times in the past reflect the historical state of the database.

2.7.1 TSQL2-Based Approaches

An approach to temporal databases that has received considerable attention is the TSQL2 approach (Snodgrass 1995) developed by a number of database researchers. Also, a number of different proposals have been offered based on the TSQL2 approach, which adopt the same key concepts. In this section, an overview of the central concepts of the TSQL2 approach is offered.

In the TSQL2 approach, tables that are used for storing temporal data can have one or two temporal attributes. These are used to store the valid time and transaction time explained above. In the TSQL2 approach, these attributes are hidden from the user and their values cannot be directly accessed. Since both of these attributes are optional, TSQL2 ends up supporting 4 different kinds of tables:

- A *bi-temporal table*: This kind of table includes both the valid time and transaction time timestamp attributes
- A *valid-time table*: This kind of table includes the valid time attribute but not the transaction time attribute.
- A *transaction-time table*: This kind of table only includes the transaction time attribute but not the valid time attribute
- A *regular table*: This kind of table doesn't include temporal attributes.

TSQL2 categorizes operations into 3 different kinds:

- *Current*: Current operations are only applied to the most recent data.
- *Sequenced*: Sequenced operations are applied to all data, or data at all points of time.
- *Non-Sequenced*: Non-sequenced operations are applied to a specified subset of data.

These different kinds of operations are specified by using what are called *Statement Modifiers*. Consider the sample set of data in Figure 1 from (H. Darwen & Date 2005).

Table S:

S#	
S1	[d01:d01]
S1	[d05:d06]
S2	[d02:d04]
S2	[d06:d99]
S3	[d05:d99]
S4	[d03:d99]
S6	[d02:d03]
S6	[d06:d09]

Figure 1. Sample TSQL2 Table

The first column is the identifier of S called S#. The second column is a valid time interval in days. The two figures indicate the day the interval begins and ends.

Now consider the following simple query:

```
Select * from S
```

If the current day is d10 and the above query is run in the Current mode, the result will include only tuples that include the current day in their valid time. So the result would be: S2, S3, S4

The same query if run in the Sequenced mode would return all distinct S# values. The output would be S1, S2, S3, S4, S6. The result would also include the hidden valid time attribute.

The non-sequenced version of the query would return all S tuples that have ever existed in that table but would not include the hidden valid time attribute.

2.7.2 Date, Darwen and Lorentzos' Approach

As described in (Date et al. 2002), in any relation that is to store temporal data, an interval attribute is associated with each tuple. The interval is indicated by a start time and an end time. A discrete model of time is assumed and picking the granularity of the time attribute is left as a design decision to the database designer.

Two basic operators which are used in the processing of temporal database queries are: pack and unpack. There are different terms employed in the literature, but the basic functionality remains the same. Both operators take a relation which has an interval attribute as input and produce a relation with the same heading and representing the same semantic information. But their presentation is slightly different. Consider the example in Figure 2 of the supplier table from (Date 2003), where S# is the identifier of the relation and the DURING attribute displays the validity duration each tuple.

S#	DURING
S2	[d02:d04]
S2	[d03:d05]
S4	[d02:d05]
S4	[d04:d06]
S4	[d09:d10]

Figure 2. The Supplier Table

Figure 3 displays the effect of applying pack and unpack to the above relation:

PACK		UNPACK	
S#	DURING	S#	DURING
S2	[d02:d05]	S2	[d02:d02]
S4	[d02:d06]	S2	[d03:d03]
S4	[d09:d10]	S2	[d04:d04]
		S2	[d05:d05]
		S4	[d02:d02]
		S4	[d03:d03]
		S4	[d04:d04]
		S4	[d05:d05]
		S4	[d06:d06]
		S4	[d09:d09]
		S4	[d10:d10]

Figure 3. Pack and Unpack

Informally, pack returns the relation by merging all intervals that either meet or overlap for the same supplier. unpack on the other hand expands the interval attribute in each tuple so that in the resulting relation, all tuples have a "unit interval" associated with them.

The pack and unpack operators are used to define temporal versions of regular relational operators. Temporal versions of relational operators are defined similarly. Given a unary relational operator its temporal version is defined as:

PACK (op (UNPACK r))

For binary relational operators their temporal version is defined as:

PACK ((UNPACK r1) op (UNPACK r2))

For instance temporal join is defined as:

PACK ((UNPACK r1) JOIN (UNPACK r2))

Another example from (Date 2003) is as follows; having the following relations:

S_CITY_DURING, with attributes S#, CITY and DURING, which indicates the existence of a supplier in the city during the specified period,

And SP_DURING, with attributes S#, P# and DURING, indicating that a supplier was able to provide a specific part in the specified period

Consider the query "Get S#-CITY-P#-DURING" tuples such that supplier S# was located in city CITY and was able to supply part P# throughout interval DURING".

The result of this query can be obtained by simply performing a temporal join between S_CITY_DURING and SP_DURING.

In general, the approach presented in Date allows for more than a single interval attribute to be present in a relation, but for the sake of simplicity in the above discussion a single interval attribute has been assumed. The relational operators JOIN, SELECT, PROJECT, UNION, INTERSECT, MINUS all have temporal counterparts and are defined in a similar fashion. Temporal relational operators give us a powerful tool to reason about temporal data. They enable us to extract relevant temporal information that would otherwise be hard or impossible to extract. However, the application of temporal relational operators to event data is not trivial and will be discussed in more detail in Section 4.

2.7.3 Integration of Temporal Database Operators with Event Processing

Temporal databases operators are a natural candidate for querying of event data, but to the best of our knowledge, utilizing temporal database operators for querying event streams has not been explored in the literature. Among the two proposals mentioned above, the work in (Date et al. 2002) is a more natural candidate for the proposed research for the following reasons;

Relative ease of adoption: While the work in (Snodgrass 1995) is well developed, it diverges from the relational model and for this reason it was not adopted as an industry standard. Hence its prevalence in the future is in question. In contrast, the work in (Date et al. 2002) is strictly based on relational concepts, and implementing it in an existing database framework is possible by extending the supported types and introducing linguistics short-hands.

Exposing the Temporal Axis: Hiding the temporal axis of the data as in (Snodgrass 1995) and only allowing the user to query it through provided operators limits the users ability to express desirable conditions. This will be addressed further in the discussion of research objectives in Section 4. A specific benefit of the model presented in (Date et al. 2002) to the proposed research is that it treats the temporal axis as a regular attribute in a relation. In contrast, the model presented in (Snodgrass 1995) suggests that the temporal axis be modeled through hidden attributes.

Section 4 will introduce the problem of mapping CEP to a temporal database in detail.

2.8 Discussion

An overview of the current status of related work has been presented in previous subsections. This section summarizes the issues and problems with current work and serves a motivation for the research objectives of this proposal.

Work in the stream processing area (Arasu et al. 2006; Abadi et al. 2003) aims at filtering of high volume streams of data. A considerable amount of the focus is on the architecture necessary to cope with these high volume requirements. Different approaches have been pursued to handle excess volume when available resources are not sufficient to produce precise results to queries. CQL is the language that forms the basis for much of the work in this area, including commercial products. All of these stream processing languages use the concept of a time-varying relation, which is used to perform the bulk of data manipulation operations in a query. Using time varying relations entails converting an input stream to a relation for processing and then transforming the result back into a stream, which is not very convenient. In stream languages, there is usually little to no support for stream-to-stream operators that can directly detect temporal patterns of events. Additionally much of the work in stream processing assigns a unary timestamp to an input stream and thus interval-based semantics are ignored.

Event processing focuses instead on detection of temporal patterns and extraction of order information between different events. Some of the earlier work on processing of event streams is based on the Event portion of an ECA rule in active databases (Chakravarthy & Mishra 1994). Work found in publish-subscribe systems use finite state structures to encode user requested event patterns or subscriptions (Mühl et al. 2006). The automata structure is well suited for cross query optimization and reducing space requirements for storing and processing of an event query. However, it is inflexible in its order of evaluation of different operators and is unable to formulate alternate evaluation plans. These languages usually do not support value correlation of different events.

SnoopIB expands on Snoop's composite operator-based approach by adding interval semantics to event queries. Lack of interval semantics can lead to ambiguous operator meaning, which has been discussed in (M. Eckert 2008; W. White et al. 2007). Snoop also presents the concepts of consumption modes, which are used to reduce the number of active events under consideration at any given time. Closer inspection reveals that consumption modes as discussed in Snoop is in fact a mix of both selection and consumption modes. The use of consumption modes is associated with an entire event definition instead of individual input events which makes its use rather inflexible. Later work in (Sánchez et al. 2003; Cugola & Margara 2010) decouples selection and consumption modes, which leads to clearer semantics. However according to (M. Eckert 2008) performance benefits of using selection and consumption modes have not been demonstrated yet.

Composite event operators have been the main approach for detecting temporal patterns of events. Composite events operators, however, have consistently been subject to misunderstanding or unclear definitions. Introduction of consumption modes with unclear or confusing semantics has made matters worse. Utilizing interval semantics instead of detection semantics helps clarify some of the inherent issues of this approach, but is still prone to semantic issues as discussed in Section 2.3.

Finally, the work in XChange^{EQ} addresses many of the problems in previous event languages. However, it does not directly support temporal relational operators. Additionally, the values of timestamps are not directly available for querying, which leads to the introduction of a multitude of temporal operators. This makes the language larger and more complex and, at the same time, less flexible.

None of the previous work supports evaluation of temporal relational operators, as presented in (Date et al. 2002), over streams, and thus the technical challenges associated with evaluation of temporal relational operators over streams of data have not been addressed.

3. Statement of Objectives

The purpose of this research is to enable efficient application of temporal database operators to event streams. More expressive semantics have led to increasing adoption of an interval-based model in event processing. Operators traditionally used in event processing are no longer sufficient for querying of interval-based events. This proposal investigates the application of temporal database operators in an event setting. Application of temporal operators requires adoption of a relational framework for event processing. However relational algebra cannot be directly applied for evaluation of event queries due to its allowance of non-monotonic behavior. We have chosen the tailored version of relational algebra, called CERA, and the evaluation framework presented in (M. Eckert 2008) as a basis for this research. In particular, this research aims to achieve the following objectives:

- Develop a Relational Framework for evaluation of temporal queries
    - CERA will be extended to support temporal queries
    - Develop formulae for incremental evaluation of the extended relational framework
- Language Design and specification:
    - Develop an SQL-like Event Query language that also supports event composition, aggregation and negation
    - Present the semantics of the language using the developed relational framework
- Implement and evaluate a proof-of-concept prototype of the language
    - Develop a framework to support incremental evaluation of temporal event queries and materialization of intermediate results.

The following section elaborates on the research issues that will be addressed through these objectives.

4. Discussion of Research Objectives

The objective of this research is to explore the integration of temporal database features and event stream processing. Achieving this objective consists of three fundamental parts: language design, development of the relational framework for specification of the language, and the development of the supporting event processing prototype. For the language, the proposed research will augment a restricted subset of SQL with event stream processing features and add support for temporal relational operators. As was discussed in the related work section, the language in XChange^{EQ} is formalized based on a tailored version of relational algebra that enables incremental evaluation of incoming event streams. This research will extend the work relational framework presented in XChange^{EQ} to support temporal relational operators. For the event processing architecture, the proposed research will augment the plan-based query processing algorithms for efficient application to temporally-enabled event queries. The integrated approach will provide the ability to evaluate temporal relational operators over incoming streams of data, opening up a category of expressibility in specification of event patterns that none of the previous work in event stream processing provides.

The rest of this section discusses the research plan for the objectives of this research. Section 4.1 discusses the issues involved in the design of the language for the specification of events. Section 4.2 describes the specification of language and introduces the extensions necessary to CERA to support temporal relational database operators. Section 4.3 discusses incremental evaluation of the temporal queries. Section 4.4 describes an overview of the architecture for the prototype that will be developed to demonstrate the language.

4.1 Language Design for the Integration of Temporal Database Queries and Event Pattern Specification

As seen in the related work section, there have been many different proposals for event and stream processing languages. It was also demonstrated that many of these proposed approaches suffer from ambiguity or lack of expressiveness. The approach of this proposal is to build upon the strong theoretical foundation of relational and temporal relational databases to avoid the problems found in many other event specification languages. In this section a preliminary version of the proposed language is presented.
The language design builds upon at least three separate bodies of work. The basic event processing constructs are borrowed from the relational version of XChange^{EQ} called Rel^{EQ} (M. Eckert 2008). However access to input and output timestamps is also provided for stronger expressivity. The overall structure and feel of the language follows the SQL language structure. Finally temporal relational operator specifications are based on Date, Darwen and Lorentzos' work (Date et al. 2002).

The proposed language will operate on histories of events that are continuously being updated. The event histories are treated as relations with mandatory timestamp attributes. Timestamps are assigned in the form of intervals, which include a beginning and ending timestamp value. Instantaneous events can be specified by assigning the same value to the begin and end timestamps. Primitive events are assumed to be instantaneous.

Table 1 summarizes the different language features to be supported.

Table 1.Summary of Language Features

The Select-Project-Join class of Queries	This is the basic query structure found in SQL. In general unbounded query operations need to be monotonic and non-blocking in an event-processing environment.
Temporal conditions	Temporal conditions include operations such as sequence and specification of other temporal conditions as specified in (Allen 1983)
Output timestamp specification	This feature allows the output timestamp to be controlled by the user
Bounding Window	This feature allows a bound to be specified on the input events after which an output needs to be produced. A bounding window is needed to support non-monotonic or blocking language features. These include negation and aggregation as well as temporal relational operators.
Temporal Relational Operators	Temporal relational operators were presented in Section 2.7. Temporal versions of relational operators are supported. The fact that the pack operation is non-monotonic makes all temporal relational operators non-monotonic, which indicates that all temporal operators require a bounding window.
Non-monotonic Operations	These operators include Negation and Aggregation which alongside temporal relational operators need to be bound by a window definition to be viable in a streaming environment

The following subsections discuss the different language features and issues related to them in more detail.

4.1.1 Basic Query Structure

The basic structure of queries is based on the SQL structure as follows:

```
Select attribute_list
From event_list
Where condition_list
```

attribute_list: The attribute_list consists of the list of attributes that are associated with the output event. The output attribute should always include an attribute called *duration* which signifies the interval associated with the output of the query. The timestamp for the output needs to be specified using an *interval construction function* called merge. The merge function and how it can be used to modify the value of the output timestamp will be explained in following subsections.

event_list: The event_list is the list of input event types. Additionally, the event_list may include windowed, non-monotonic sources that can use relative and absolute timer events using the style of XChange^{EQ}. Windowed sources will be explained in following subsections.

condition_list: The condition_list includes any value conditions that can be specified over the attributes of the events in the event_list. Temporal relationships as specified in Allen (Allen 1983), such as *before* or *after*, can be specified over the event types included in the event_list. Additionally, the keyword *within* can be used to signify the distance allowed between events. The different kinds of temporal relationships specifiable are similar to XChange^{EQ} and are described below:

- i before j: event i ends before event j starts
- i contains j: event i contains event j
- i overlaps j: event i starts before event j but ends within j
- i meets j: the end time of i is the same as the begin time of j
- i starts j: i and j begin simultaneously but i ends before j does
- i finishes j: i and j end at the same time and i's begin time is after j
- i equals j: the intervals for i and j are the same

The following example scenario and requested queries are taken from (M. Eckert 2008). These are used to present the initial language design and capabilities. The example scenario is an online ordering system, with the input event streams, order, shipped and delivered signifying completion, shipping and delivery of an order. All of these events include the order id, simply called id. The variable p is the product name and q the quantity.

The following query defines a completed order as an order that is shipped and delivered within 48 hours of the order being placed. In this example, the time unit is assumed to be one hour. In practice, the time unit needs to be specified for any application scenario.

```
DEFINE Completed_Order (id, p, duration) AS
SELECT o.id, o.p, merge() as duration
FROM
            order o, shipped s, delivered d
WHERE
            o BEFORE s,
            s BEFORE d,
            (o,s,d) WITHIN 48
```

The select clause indicates the product id and name as attributes that are needed in the output event. The merge function will be discussed shortly, but for now suffice it to say that it is used to generate the value for the duration attribute, which is the interval associated with the output event. The FROM clause indicates the input event streams. The WHERE clause is used for specification of temporal conditions or any other value filtering conditions. Here, the before keyword is used to indicate that the order event needs to occur before the shipping event, and the shipping event before the delivery. Finally, the within keyword is used to specify that the order, shipping, and delivery event need to occur within 48 hours of each other.

Similar to Rel^{EQ}, all of Allen's temporal relationships (Allen 1983) can be specified in the WHERE clause. The specification of temporal conditions in the WHERE clause is straightforward. Considering that input event histories are viewed as relations, and begin and end timestamps as relational attributes, a temporal condition such as o BEFORE s is a simple value comparison: o.e < s.b, where e and b signify the begin and end timestamps in the duration attribute of each event history. The proposed language allows for all of these operations to be performed directly on timestamps in the where clause. However, it is desirable to have short-hand notations to easily specify event queries. In practice, the timestamps e and b are themselves components of the complex type attribute, duration, but in this document they are referenced directly for the sake of brevity.

Similarly (o,s,d) WITHIN 48 is equivalent to the relational statement, max(o.e,s.e,d.e) - min(o.b,s.b,d.b)<48

The following section discusses the relational operators that are supported in the language and the bounding window mechanism.

4.1.2 Relational Operators and Bounding Windows

In general, not all relational operations can be directly applied in an event environment. Monotonic, non-blocking relational operators can be directly supported in an event-stream processing environment. This includes the Select-Project-Join class of queries, which produce a steady stream of output as new input arrives. In contrast, any aggregation operator needs to be performed on a specific set of items. New events continuously arrive in an event processing environment. Aggregation operations are classified as blocking operations in database terminology, and since the underlying set is theoretically unbounded, an aggregate operation may block indefinitely in an event environment. Similarly, with a negation operation, the lack of occurrence of an event can only be meaningful over a specified period. These operators demonstrate the necessity for using bounding windows as was previously discussed in Section 2.3.2.

As in Rel^{EQ}, in the proposed language, windows are based on the duration intervals associated with events. The logic behind this is that desirable intervals can be constructed using the query language, and thus an event interval itself is a flexible way for specification of intervals. In Rel^{EQ} several different operators are introduced in order to allow the manipulation of the source event interval in different directions. A more flexible and succinct syntax is introduced for this purpose in this proposal called the WINDOWED_SOURCE operator. While WINDOWED_SOURCE is used to specify bounds on other operations, it simultaneously acts as an event source to the rest of the query. The window source specification operator has four operands:

```
WINDOWED_SOURCE((interval_source_alias, [b_offset, o_offset]), {window attributes}, [temporal_
query| [[COLLECT|NOT] event_source]])
```

The first operand is an input event stream, which forms the basis for the window. All attributes of the source event are accessible within the window, but most importantly the duration of the source event forms the bounds of the window. This interval can be modified using the second operand which can be used to specify offsets to the begin and end timestamps of the input event stream. The third operand specifies the attributes that need to be projected from the window statement and forms the attributes for the window output. In the case of an aggregation operation the aggregate function needs to be specified here. Besides these attributes, each window has a mandatory 'Duration' attribute, which takes its value from the offsets applied to the input event stream. The fourth and final operand is the statement that is temporally bound by the window. This can be a temporal query or a negation or aggregation operation. Temporal queries will be discussed shortly. Examples of negation and aggregation follow.

The following example is a query which indicates overdue orders. An overdue order is defined an item that has not shipped in 6 hours since the order event:

```
DEFINE Overdue_Order (id, duration) AS
SELECT o.id, merge() as duration
FROM
        order o, WINDOWED_SOURCE(o, [0: 6], {o.id},  NOT shipped(o.id,o.t,duration)) w
WHERE
        o.q <10, o.id=w.id
```

The output event consists of the item id and duration, which is calculated using the merge function that will be discussed shortly. The order stream is used as input and is also the basis for the window specification. The window itself, does not modify the start time of the order, but extends its end time by 6 hours. A shipped event for the corresponding event id should not have happened during this period. Only one attribute is projected: the id of the order. Shipped has two attributes: id which is the order id and t which is the tracking number. The windowed source specification performs an anti-semi-join operation to pair the negated event (shipped) with the base event (order). Finally, this event is only raised when the quantity of the order is less than 10 items which is modeled as a simple value condition.

The following is an example of an aggregation query that utilizes the window specification mechanism to perform a count function. The query asks for the number of orders that were shipped in the 24 hour window preceding an overdue event.

```
DEFINE Load (c, duration) AS
SELECT c, merge() as duration
FROM
        Overdue_Order o, WINDOWED_SOURCE(o, [-24: 0]), {o.id, count(id) as c}, COLLECT
        shipped(id,t,duration)) w
WHERE
        o.id=w.id
```

The Overdue_Order event from the previous example is used as input for this query and is the basis for the window specification. The window shifts the start time 24 hours into the past, but does not modify the end time of the window. Aggregation is specified using the keyword collect, and the type of aggregate function being used is specified in the select list, in this case, a count. The item being counted is the number of shipped items.

The following section discusses specification of temporal relational operators.

4.1.3 Specification of Temporal Relational Operators

All temporal relational operators utilize the pack and unpack operators. Given a relation that includes an interval timestamp value, the unpack operator creates a new relation with an identical heading. In the new relation, each tuple is expanded into multiple tuples for each unit of time of the interval. The value of non-interval attributes is the same as before, but each interval is broken down into its unit values. It follows that specification of the time unit is necessary in any environment. The following example is used to illustrate pack and unpack operations to demonstrate a temporal projection. The time unit is assumed to be in days. The sequence of operations is as follows: First the relation is unpacked, then a projection operation is performed, and finally the result is packed.

Consider the example relation E in Figure 4, and the temporal projection of the attribute Section.

Id	Section	Duration
E1	1	[11:13]
E2	1	[12:14]

Figure 4. Example Temporal Relation E

The unpacked version of E is shown in Figure 5.

Id	Section	Duration
E1	1	[11:11]
E1	1	[12:12]
E1	1	[13:13]
E2	1	[12:12]
E2	1	[13:13]
E2	1	[14:14]

Figure 5. E Unpacked

The projection operation would be applied normally to the Section and Duration attributes as shown in Figure 6.

Section	Duration
1	[11:11]
1	[12:12]
1	[13:13]
1	[14:14]

Figure 6. Projection of attributes Section and Duration

The pack operator merges continuous intervals of time together. Since in the above example all the intervals meet with each other, the pack operator merges each tuple based on the duration as shown in Figure 7.

Section	Duration
1	[11:14]

Figure 7. Packing the result

In general all temporal relational operators are defined similarly. If op is a regular relational operator, its temporal version is defined by unpacking the operands, applying the op regularly to the unpacked version and finally applying the pack operator. A few of the temporal relational operators to be supported in the language are specified below. The syntax provided is in the algebraic style of Tutorial D (Date & Hugh Darwen 2000) for ease of illustration.

- u_restrict: This is the temporal select or restrict operator, and is specified below:

  PACK ((UNPACK R) WHERE *p*)

  Unpack the operand, perform the restrict operation, pack the result. *p* is the restrict condition.

- u_select: This is the temporal projection operator, and is specified below:

  PACK ((UNPACK R) PROJECT {BCL})

  Unpacks the operand, performs the projection and packs the result. BCL is the list of attributes to be projected and must include the timestamp attribute in order for the pack operation to be performed correctly.

- u_join: This is the temporal join operation and is specified below:

  PACK ((UNPACK R1) JOIN (UNPACK R2))

  Unpacks each of the two operands, performs the join operation and does a final pack operation. Notice that the join operation here refers to a natural join and will include the interval attribute that the operands share.

It should be noted that pack is not a monotonic operation. Considering the previous example, if the table that contained the result of the projection were to be made available to the pack operator over time, applying the operator would produce a result that would only be correct for that instant in time. Figure 8 shows the result of applying pack before all the inputs tuples have arrived.

Input:

Section	Duration
1	[11:11]
1	[12:12]
1	[13:13]

Result of pack:

Section	Duration
1	[11:13]

Figure 8. Non-monotonicity of pack

As was seen previously, as more tuples arrive in the input, the previous output of pack, as shown in Figure 8 is no longer correct and needs to be replaced with the current result. This makes the pack operation non-monotonic. By extension, all temporal relational operators are also non-monotonic. As such temporal relational operators need to be defined within a window if they are to be used in an event processing environment.

Moving to the event processing environment, consider as an example, a smart office setting, where different sensors provide the status of lights and movement in different rooms. Also assume that complex events have been defined over these primitive streams that produce interval based events indicating the periods during which a room has been occupied and when the lights in a room are turned on. The following complex event streams are already available:

```
room_status(id, status, dept, duration)
light_status(room_id, power, duration)
```

The room_status stream sends events regarding whether different rooms in the office building are occupied or not. The id is the room identifier, status can be either 'empty' or 'occupied', dept indicates the department name which the room belongs to, and duration is the associated timestamp.

The light_status stream sends events regarding the status of the lights in a certain room and whether they are powered or not. The room_id is the room identifier, power can be either 'on' or 'off' and duration is the associated timestamp.

In this example it is also assumed that we have access to an absolute timer event called the day_timer. The day_timer event occurs every 24 hours. The begin time of the interval is the first minute of the day, and the end time is the last minute of the day.

Consider the following query: Return all department names that have had an occupied room and the maximal durations during which the room was occupied, every 24 hours. To simplify presentation, the query is performed in two steps. First occupied rooms are redefined as an event stream:

```
DEFINE occupied_room (id, status, dept, duration) AS
select *
from room_status
where status='occupied'
```

Now the temporal selection is performed over the derived event stream:

```
select dept, orDuration,merge()
from day_timer d, windowed_source(d, [0:0], {dept, occupied_room.Duration as orDuration}
u_select dept
from occupied_room) w
```

In the above example, day_timer is an absolute timer event that has a duration of 24 hours. First a temporal projection is done over the dept attribute using the u_select operator over occupied_room. This results in maximal durations in which a dept had an occupied room. Finally this information is collected in the window called w. Because windows have a duration attribute, when specifying temporal queries, any underlying duration attribute needs to be renamed.

As another example, consider the following query: Return the rooms that were empty while the power was on, and the associated maximal periods, every 24 hours. The query is illustrated in a multi-step approach:

```
DEFINE empty_room (id, status, dept, duration) AS
select *
from room_status
where status='empty'

DEFINE on_light (room_id, power, duration) AS
select *
from light_status
where power='on'

select id, joinDuration,merge()
from day_timer w, windowed_source(d, [0:0], {empty_room.id, Duration as joinDuration}
     while w:
   select * from
            empty_room
         u_join
            on_light
   on empty_room.id=on_light.room_id
```

In the above example, first the rooms that have their lights on, and the rooms that are empty, are selected in the first two queries. In the third query a temporal join is performed between the two, which is signified by the u_join operator. Temporal operators, including temporal join, always produce a single interval attribute in their output called Duration. This results in the selection of time intervals in which both conditions are true. Finally, this information is collected over a 24 hour period using the day_timer window.

4.1.4 Specification of the Output Timestamp using the Merge Function

By default the merge function calculates the output interval based on the semantics of the *merge operator* as defined in XChange^{EQ}. The merge operator takes the intervals of all input event types as specified in the event_list and assigns the interval that spans the smallest start time and the greatest end time as the output interval.

This proposal aims to give the user a more direct control to the output timestamp. This is achieved through using the operands of the merge function. Similar to the window specification, the user can specify offsets on the values of the output duration. These offsets are applied to the default output of the merge operation.

```
merge(b_offset, e_offset)
```

The b_offset can be any integer value, but the e_offset can only be non-negative values. The reason for this is that, as in XChange^{EQ}, in the proposed approach, current results should always be calculated using current events. If the end time of an event is allowed to be reduced, that would mean that an event currently being generated actually belonged in the past, and could have effected the output of other events that reference this event. As such, shifting the occurrence time of an event at the current instant of time to a value less than the current instant of time should not be allowed. This concept is directly related to the concept of Temporal Preservation which is introduced in XChange^{EQ} and will be elaborated on in Section 4.2.

The following subsection elaborates on the benefits of treating timestamps as relational attributes.

4.1.5 Timestamps as Relational Attributes

In XChange^{EQ} and other event languages the user is not able to directly access and manipulate the timestamp values. Treating timestamps as relational data has the following advantages:

Expressive condition specification: Most event languages, including XChange^{EQ}, do not expose timestamps for querying. To compensate for lack of direct access to timestamp values, a plethora of different operators are provided to specify temporal conditions. The problem with this approach is that, in general, when the desired conditions do not directly match the provided operators, the corresponding query becomes increasingly complicated or even impossible to express. For example, XChange^{EQ} provides equivalents of Allen's temporal operators for comparing intervals. The language also provides a WITHIN clause to specify absolute bounds on the co-occurrence of events in a query. While these different operators aim to simplify specification of temporal conditions, it seems that this is only achieved for cases when the query conditions match exactly one of the provided operators. For example, a simply expressible temporal condition may require a disjunction of a multiple of Allen's relationships. Specifying the conditions directly on the timestamps leads to a much shorter expression. Consider the case when the begin time of event A needs to be greater than or equal to the begin time of event B. Relationally, it is simple to express this condition: a.b >= b.b However, if we are restricted to using Allen’s operators the expression becomes:

```
(a after b) or (a during b) or (a overlaps b) or (b meets a) or (b starts a) or (a starts b) or (b finishes a)
or (a finishes b)or (a equals b)
```

Besides being extremely tedious to formulate, the above expression is also readily prone to errors.

There are also issues with specification of boundaries when using the WITHIN clause or when specifying windows in XChange^{EQ}. In XChange^{EQ}, specifying bounds with inclusive limits is not possible. Without direct access to timestamps, supporting different kinds of comparisons can only be done by providing different versions of operators, some with the equality condition and some without.

A recent paper (Walzer et al. 2008) tries to ally the expressibility problem associated with the use of Allen's operators by adding even more operators. Any such extension does not solve the underlying problem. Additional operators may expand the set of queries that are easily expressible for a specific application, but they also make the language more complicated without giving the programmer the necessary control. Ultimately, it should be up to the programmer to decide how to best express the desired conditions.

Ease of manipulation: Treating timestamps as relational attributes has the added benefit that it also gives the programmer the ability to directly access timestamp values. This can be useful for extending and shrinking of intervals to specify relative time intervals.

Additionally, it is useful when the data payload of the event contains temporal information. For example, consider a sensor reading that also advertises the amount of time that it will go to sleep as an attribute value. Clearly, it is desirable to be able to perform arithmetic operations including such data and timestamp values. XChange^{EQ} and similar languages do not allow any interaction between event data and the timestamp values.

However manipulating timestamps should happen in a way as to not have an adverse effect on query evaluation. How this can be achieved is one of the questions this research will address. This is directly related to the notion of temporal relevance that will be further discussed in section 4.2.

4.1.6 Summary of language features

In summary, the operations that are to be supported in the proposed language are as follows:

- Projection of event attributes
- If more than one event stream is mentioned in the from clause, their Cartesian product is considered within any temporal bounds specified, similar to SQL.
- Join of input events
- Filter conditions on temporal attributes in the where clause
- Filter conditions on event payload attributes in the where clause
- Specification of output timestamp:
    - If a single input event is specified the timestamp of the output would be the same as the input event
    - If more than one input event stream is specified, they are composed in the output. The output schema consists of any attributes specified in the select clause. The output timestamp interval is calculated using the merge operator which is defined in following subsections and is equal to the interval that spans the smallest start time and largest timestamps of the input intervals.
- Non-monotonic operators within windows:
    - Negation and Aggregation are supported within a bounded window.
    - Temporal relational operators are also non-monotonic and need to be specified within a window.

This proposal has mapped the Rel^{EQ} structure to an SQL like syntax and the basic query structure has been introduced. A monotonic subset of SQL that can be used for querying events has been identified. Additionally, a preliminary syntax structure has been introduced that allows specification of windowing and temporal relational operators. As part of this research, the language syntax will be fully defined, and issues of access to timestamps will be addressed.

4.2 Language Specification and Formalization

CERA is the basis for formalizing the semantics of XChange^{EQ}. CERA treats all incoming event histories as relations. Timestamps are maintained in interval format, and are considered similar to relational attributes. Primitive events are considered to be instantaneous, having equal start and end times. The maximum end time of any tuple is considered the occurrence time of that tuple.

Even though XChange^{EQ} is based on an XML format, CERA is based on relational algebra. This discrepancy is resolved by utilizing a pair of operators called *Matching* and *Construction* that are utilized to transform XML data into relational format and vice, versa respectively. Matching and Construction will not be covered here in the interest of space; the interested reader may refer to (M. Eckert 2008).

Besides the traditional operators found in relational algebra, CERA includes a grouping operator, called γ, which is common extension to relational algebra for aggregation. CERA also introduces a new merging operator called μ which, given several time intervals in the input tuple, merges them together into a single interval which spans the minimum start time and maximum end time in the input timestamps, and discards all the other timestamp values.

CERA places several restrictions on relational algebra, which are listed below. The aim of these restrictions is to achieve a quality called *temporal preservation.* Temporal preservation enables incremental, step-wise evaluation of CERA expressions and will be explained shortly. Under temporal preservation:

- All relations must include at least one interval attribute
- End timestamps should always be greater than or equal to begin timestamps
- Projection π cannot drop timestamp attributes
- The γ grouping operator is supported with the following restriction: Any timestamp attribute present in the operand must be included in the list of grouping attributes.
- Difference \ and union U are not supported in CERA.
- A rule head must include a merging operator

To place limits on negation and aggregation, temporal join and temporal anti semi-join are used for specification of windows. ‘Temporal’ here should not be confused with the temporal relational operator which utilizes pack and unpack. Definitions of temporal join and temporal anti semi-join are presented below.

A temporal θ join is used for specifying windows over aggregation operations and is defined as follows:

$$R \bowtie_{i \sqsupseteq j} S = \sigma\ [i.s \leq j.s \wedge j.e \leq i.e](R \bowtie S)$$

Where i and j are occurrence durations for R and S respectively, and i.s, j.s and i.e, j.e refer to their start and timestamps, respectively. The notation $i \sqsupseteq j$ is used to signify the falling of one interval (j) within the other (i) as expanded in the selection condition above. The result includes tuples of R joined with S but only includes those tuples where S's interval fall within the interval of a corresponding R. This creates a windowing behavior over S with the interval of R serving as the bounds of the window.

A temporal anti-semi-join is used for specifying windows over negation operation and is defined as follows:

$$R \,\overline{\ltimes}_{i \sqsupseteq j}\, S = R \setminus \pi_{\ sch(r)}\ (\sigma_{\ i \sqsupseteq j}\ (R \bowtie S))$$

Where i is an interval in R and j is the only interval in S. First, the same operation as the temporal θ join is performed. Then the schema of R is projected from the result of the first step. Finally, the result of this projection is subtracted from R itself. The result is tuples of R that did not have a corresponding S such that S's interval fell within R's interval. Again R serves as the bounds of the window, but since a negation operation is being considered, 'empty' windows are of interest and are returned.

Collectively the conditions above make CERA expressive enough to formalize XChange^{EQ}'s functionality while ensuring the temporal preservation property. CERA expressions are expressed over conceptual relations that include all events that have occurred in the past and all events that will occur in the future. In reality, such relations do not exist. To calculate current results, we only have access to events that arrive in the current instant of time or have arrived previously. Temporal preservation ensures that current results can always be calculated using already available data.

To support temporal relational operators, CERA needs to be extended with the unpack and pack operators. Since pack can have an effect on previously generated results, any temporal relational operator specification needs to be bounded within a window. Additionally, as it currently stands, the CERA restrictions only allow a single timestamp attribute in the output which overlaps internal timestamps. This CERA property would cause any temporal information obtained by temporal relational operations to be discarded in favor of the bounding window. CERA properties need to be reevaluated to allow temporal data to be extracted more freely while not violating the temporal preservation property.

This proposal has identified CERA as a suitable basis for formalizing the event query language, which needs to be evaluated in an incremental method. The extensions necessary to CERA in order to support temporal relational operators have been identified. These include the pack and unpack operators and windowing over temporal relational operators. As a part of this research, these extensions will be formalized in a way that supports temporal preservation.

4.3 Support for Incremental Evaluation

Utilizing a query plan based approach for evaluation of event queries has benefits in terms of added flexibility but it also presents some challenges for efficient query processing. The event pattern evaluation structure used in automata-based approaches clearly reflects the sequencing of the incoming events. In the relational approach, however, query plans are applied to relations that do not natively represent the ordering information. Furthermore, the relational approach to query processing assumes that all the tuples that are required for processing of the query are currently residing in the underlying relations, or, in our case, in the event histories. This assumption obviously does not hold in the event processing area; in reality the input event streams are continuously generating events and the corresponding event histories get updated accordingly.

There are different approaches possible to resolve this discrepancy and augment the query processing framework so it is also applicable to continuously arriving events. Let the arrival of each new event mark a new evaluation step. A naïve approach is to reevaluate the entire query processing tree in each evaluation step. This approach will produce continuous outputs based on new input arriving in each step, but it is far from efficient (F. Bry & M. Eckert 2008). Essentially in each step, what is being computed is not only the output of the query based on the newly added event, but also all the query results corresponding to previous steps. The result of the query tree reevaluation needs to be filtered so that it only contains events that were generated in the step corresponding to the current instant in time. It is obvious that the amount of computation required can be significantly reduced if recalculation of previously generated output and intermediate results can be avoided.

The work in XChange^{EQ} (F. Bry & M. Eckert 2008) addresses the problem of re-computation of previously computed results through a technique called *finite differencing*. In this approach, for each evaluation step, the underlying data is categorized into two distinct sets. The historical data that existed in the relevant histories prior to the current instant of time, and the delta, or the new data in the relations based on the arrival of new events. The delta would consist of the newly arrived primitive events, for the primitive event histories, and the resulting tuples in the relational expressions in the intermediate results.

The XChange^{EQ} framework allows any step in the query tree-based on the CERA expression to be specified as a materialization point to store intermediate results.

The CERA expressions associated with each query, and materialized intermediate results, are rewritten based on the above categorization of the data. In the converted form, queries will no longer make unqualified references to event histories; rather each query takes as input the preexisting data and the deltas.

The aim of this proposal is to calculate query results in a step-wise incremental manner in a manner similar to XChange^{EQ}. Formalization of event queries using relational algebra provides us with the precise semantic meaning of the queries. However, relational algebra statements can only be applied to relations that already include all the data. In an event processing environment events are continuously arriving and new tuples are added to the event histories. What is required is, given a query of the from $Q_n := E_n$, where Q_n indicates a query or a materialization point name, and E_n the corresponding relational algebra expression, the aim is to calculate ΔQ_n using the following input items (M. Eckert 2008):

1. *The relations that contain the newly arrived primitive events* $\Delta B_1 ,..., \Delta B_m$. If B_i is the conceptual relation that includes all events that occur, be it in the past, present or future, ΔB_i would be those events whose maximum end time stamp, or occurrence time is equal to *now*. Formally: $\Delta B_i = \sigma[M_{Bi}=now](B_i)$ where M_{Bi} is the maximum end timestamp value in B_i

2. *Base relations that contain previously arrived primitive events (primitive event histories)* $oB_1 ,..., oB_m$. Formally, $oB_i = \sigma[M_{Bi}<now](B_i)$

3. *Relations containing event histories of materialization points* $oQ_1 ,..., oQ_n$. Formally: $oQ_i = \sigma[M_{Qi}<now](Q_i)$. Again, Q_i is the conceptual version of the materialization point that includes all events that have ever happened, in the past, preset or the future.

After calculating the outputs, the evaluation step also needs to $oB_1 ,…, oB_m$ and $oQ_1 ,…, oQ_n$ need to be updated (to $oB'_1 ,…, oB'_m$, $oQ'_1 ,…, oQ'_n$) so they can be used for the next evaluation step. $oQ'_I = oQ_i \cup \Delta Q_i$.

Finite differencing finds its roots in incremental evaluation of materialized views (Griffin & Libkin 1995). Finite differencing provides the relational algebra formulation for calculating the new output items or output deltas, based on newly arrived items, or input deltas, and items that have arrived previously and already residing in the event histories.

The following example from (M. Eckert 2008) demonstrates how the delta of a join operation is calculated based on the underlying deltas and histories:

$$\Delta(E1 \bowtie E2) = \Delta E1 \bowtie \circ E2 \cup \Delta E1 \bowtie \Delta E2 \cup \circ E1 \bowtie \Delta E2$$

In which ο represents the items preexisting in the event history, Δ the new items, and the rest of the symbols are standard relational operators. Finite differencing for rest of the relational operators supported in XChange^{EQ} are similarly specified. It is important to note that current results can be calculated using events that already exist in the histories or have arrived at the current moment. However current results never rely on future events. If this were the case, then the event processing system would no longer be append only and would need to support retracting of already generated results also.

The correctness of finite differencing relies heavily on the temporal preservation property of CERA. Any new operators added to CERA also need to support temporal preservation so that the history and delta operators can be correctly applied to them. However, support for temporal preservation cannot simply be assumed for new operators. The proposed language extends CERA with temporal relational functionality including the following three operators: unpack, pack and windowing over temporal relational operators. Consider the input event stream in Figure 9, in which a temporal projection of the attribute dept is being performed. The unpack and pack steps are also shown. The relation on the left side of Figure 9 shows the original table, and the relation on the right shows it after application of the unpack operator.

input event stream:

id	dept	Interval
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

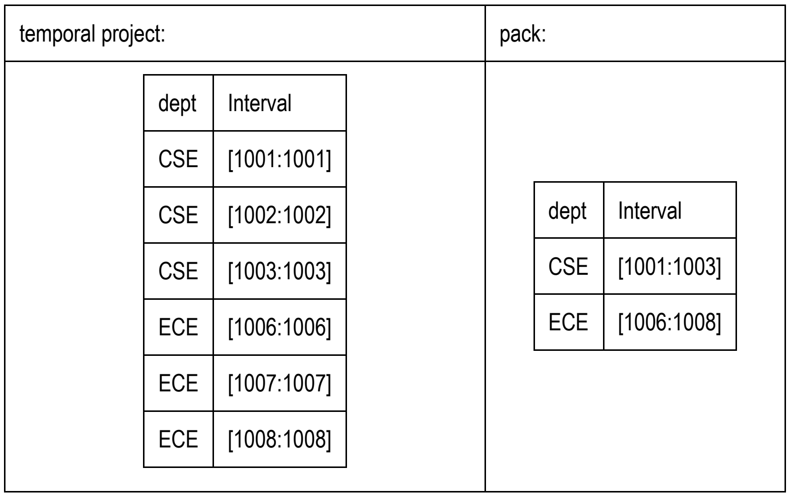
unpack:

id	dept	Interval
e1	CSE	[1001:1001]
e1	CSE	[1002:1002]
e1	CSE	[1003:1003]
e2	CSE	[1002:1002]
e2	CSE	[1003:1003]
e3	ECE	[1006:1006]
e3	ECE	[1007:1007]
e3	ECE	[1008:1008]

Figure 9. Temporal Projection of attribute Dept - Step 1 Unpack

As can be viewed in the above example, unpack does not adhere to the temporal preservation as it loses the original timestamp values. At any instant if we apply the delta operator it will only return the tuple that corresponds to that instant, but the other tuples that should belong in the same delta are discarded.

Figure 10 illustrates the result of project on the unpacked relation and the final pack step.


temporal project:

dept	Interval
CSE	[1001:1001]
CSE	[1002:1002]
CSE	[1003:1003]
ECE	[1006:1006]
ECE	[1007:1007]
ECE	[1008:1008]

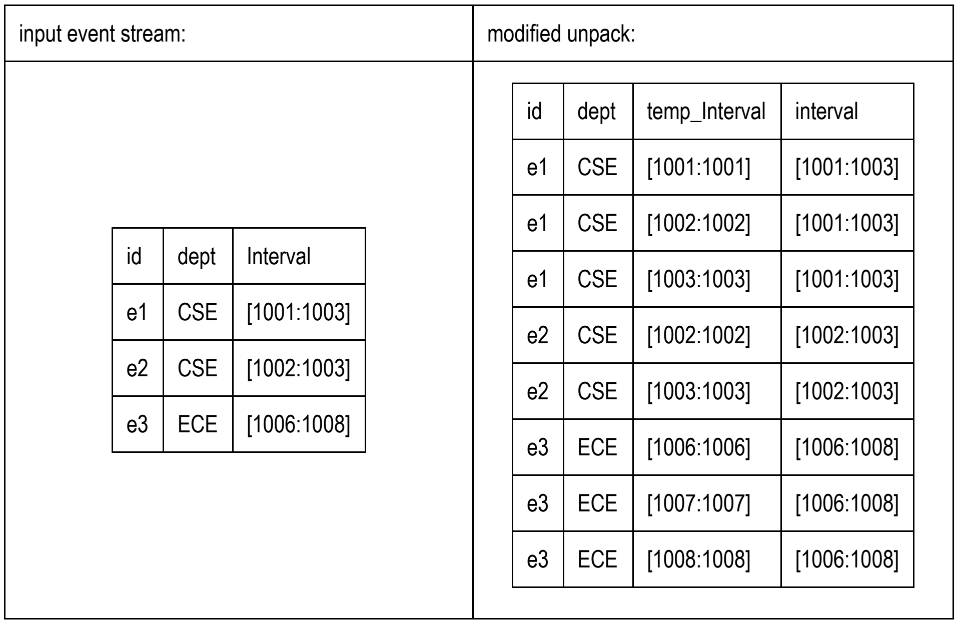
pack:

dept	Interval
CSE	[1001:1003]
ECE	[1006:1008]

Figure 10. Temporal Projection of attribute Dept – Steps 2 & 3 Project and Pack

Again, since the project is operating on the unpacked version, deltas cannot be correctly applied to it.

One of the challenges of the proposed research is to define temporal operators so that they adhere to temporal preservation. Figure 11 shows a possible workaround to the above problem by modifying the behavior of the unpack and temporal project. The *modified unpack* operator tags each tuple it outputs with the original interval timestamp, and so adheres to temporal preservation:


input event stream:

id	dept	Interval
e1	CSE	[1001:1003]
e2	CSE	[1002:1003]
e3	ECE	[1006:1008]

modified unpack:

id	dept	temp_Interval	interval
e1	CSE	[1001:1001]	[1001:1003]
e1	CSE	[1002:1002]	[1001:1003]
e1	CSE	[1003:1003]	[1001:1003]
e2	CSE	[1002:1002]	[1002:1003]
e2	CSE	[1003:1003]	[1002:1003]
e3	ECE	[1006:1006]	[1006:1008]
e3	ECE	[1007:1007]	[1006:1008]
e3	ECE	[1008:1008]	[1006:1008]

Figure 11. Modified Unpack

The original interval is called interval, and the result of unpack is called temp_interval. Figure 12 shows the modified temporal projection and subsequent unpack:

modified temporal project:

dept	temp_Interval	interval
CSE	[1001:1001]	[1001:1003]
CSE	[1002:1002]	[1001:1003]
CSE	[1003:1003]	[1001:1003]
ECE	[1006:1006]	[1006:1008]
ECE	[1007:1007]	[1006:1008]
ECE	[1008:1008]	[1006:1008]

pack:

dept	Interval
CSE	[1001:1003]
ECE	[1006:1008]

Figure 12. Modified Temporal Projection

The result of temporal projection result is equivalent to the original definition of temporal projection, but also includes the added interval attribute. The interval attribute for each tuple is a merge of the time intervals that were used to produce that tuple. The added attribute preserves the original timestamp values and, as a result, deltas can be correctly applied. The final pack result is the same as before.

The above example illustrates that by tagging tuples with original timestamp values and preserving them using merging operations, temporal preservation becomes possible.

In summary, while the finite differencing approach presented in XChange^{EQ} adequately addresses the problem of incremental evaluation, since temporal relational operators were not originally supported in the language their incremental evaluation is also not explored. Potential temporal relational extensions to CERA were identified above and a potential solution for achieving temporal preservation was presented. As part of this research, operator definitions will be finalized and formal definitions using a temporally-extended relational algebra will be developed for final operator definitions. Finite differencing of newly added operators will be performed to enable incremental evaluation. Proofs of correctness for finite differencing formulas will also be provided.

4.4 System Architecture

As part of this research a prototype will be developed to demonstrate the proposed language. This prototype will serve as a proof of concept for evaluation of temporal queries in an incremental manner. This section discusses some of the issues that need to be addressed in the architecture of the prototype system.

The system needs to satisfy the following requirements:

After specification of an event query the system should:

- Translate temporal queries into an internal representation corresponding to required relational operations.
- Perform finite differencing on the relational query plan to produce an incremental plan.
- Enable the storage of primitive events and any intermediate materialized results in their corresponding "histories".

In each step of the evaluation the system should:

- Store newly arrived events in corresponding histories
- Recalculate any materialized result and query results in a bottom-up order based on newly arrived events
- Update histories accordingly

Due to the dependency of the proposed research on relational operators extending an existing relational DBMS for evaluation of temporal event queries is being considered. The rest of the section discusses how the DBMS needs to be extended for the purposes of this research.

Primitive Event histories are treated as relations. The relation stores all events that arrive over the event stream. The time-stamp of a primitive event is assigned on arrival by the system based on the current system time. A primitive event will be assigned a unit interval. The arrival of an event consists of assigning it a timestamp and its insertion into the relevant event history relation.

Continuous queries are defined as views, or more precisely as materialized views defined over the event history relations.

An overview of the basic system architecture is illustrated in Figure 13.

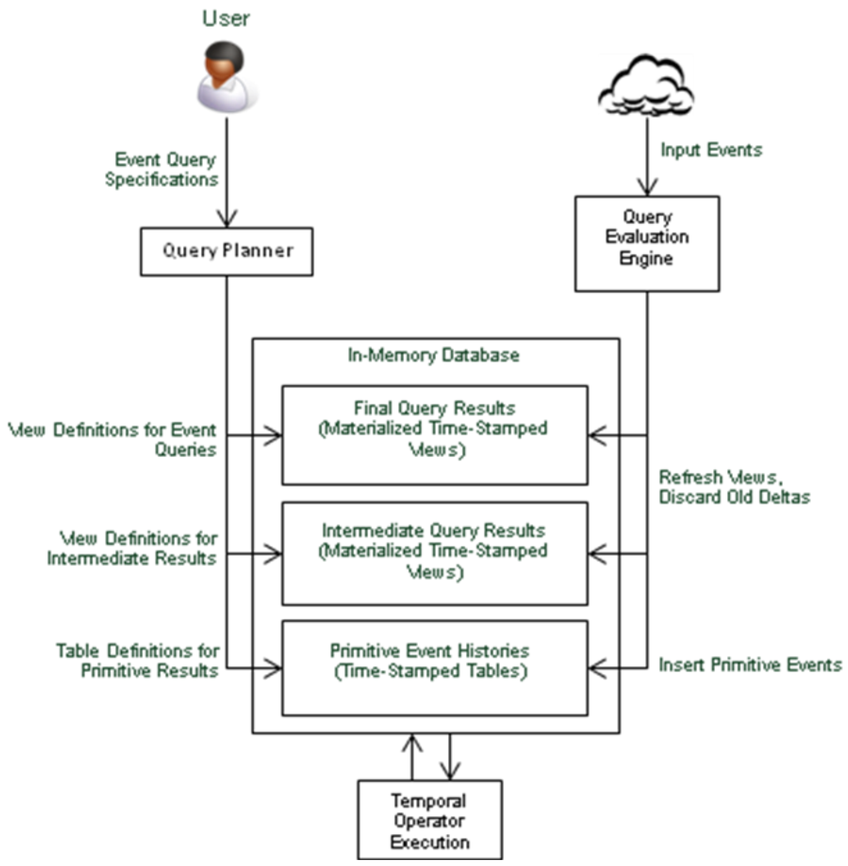

User
Event Query Specifications
Query Planner
Input Events
Query Evaluation Engine
In-Memory Database
View Definitions for Event Queries
Final Query Results (Materialized Time-Stamped Views)
Refresh Views, Discard Old Deltas
View Definitions for Intermediate Results
Intermediate Query Results (Materialized Time-Stamped Views)
Table Definitions for Primitive Results
Primitive Event Histories (Time-Stamped Tables)
Insert Primitive Events
Temporal Operator Execution

Figure 13. Overview of Basic System Architecture

Briefly, the following functionality needs to be implemented:

Language: The proposed language has been described in the preceding sections. The language needs to be translated into definition of relations and materialized views for primitive events and event queries. Since existing DBMSs do not natively support perform temporal operations, these operations will need to be performed through extensions.

Query Processing: The timestamp for output events needs to be correctly calculated using the merge function. Modifying the output timestamp can also effect 'when' a result is produced. As such, the output of certain events may need to happen in a delayed manner.

Garbage collection: Many event processing language force a time window condition on all event queries by including a mandatory 'within' clause. This places a limit on the number of input events that can be considered by event queries. In the proposed approach, non-monotonic operators need to be bound by a temporal window but there are no mandatory time windows applied to queries, in general. This results in the prospect of indefinitely storing many unnecessary events. In XChange^{EQ} a garbage collection strategy is proposed to handle this problem. After the completion of each step, events stored in primitive event buffers or as intermediate results are examined to determine whether they could be useful in generating future results. Events that aren't considered necessary any longer are removed from the relevant histories. Relevance of existing events based on the temporal conditions specified in the query is called temporal relevance and is addressed in (F. Bry & M. Eckert 2008). However XChange^{EQ} does not address the problem of temporal relevance in the presence of temporal relational operators. This proposal aims to investigate this problem in order to allow garbage collection in the presence of temporal relational operators.

Given its maturity, the open source nature and support for in-memory storage of data in MySQL has been identified as a good candidate for the basis of the architecture. In summary, based on the requirements, extending a database management system, which would serve as the storage and evaluation engine, is being investigated as an implementation strategy. The extensions necessary to the DBMS to support the requirements were identified above. The implementation of the architecture needs to happen in roughly in the same order in which the additional features are outlined above.

5. Dissertation Outline and Timetable

Based on discussion of the research issues in previous sections, the dissertation will be structured as follows:

1. Introduction

Present a general overview of the main research issues, the motivation of the research, and a roadmap to the rest of the dissertation.

2. Related Work

Discuss the existing work in related areas, including event processing, stream processing, event-stream processing, and temporal databases.

3. Relational Framework for Evaluating Temporal Queries

Introduce the requirements for a relational framework for processing of temporal event queries, and continue to present extensions to the CERA framework that satisfy those requirements.

4. Language Design for Temporal Event Queries

Present the expressivity issues with the design of a temporal event query language, and how they are dealt with in the language.

5. Language Specification

Specify the different language operators using the developed relational framework as presented in Chapter 3.

6. Incremental Evaluation of the Language

Present the framework for incremental evaluation of temporal relational queries. Develop finite differencing of temporal operators and windowing. Describe extensions to the XChange^{EQ} framework that make incremental evaluation of temporal operators possible.

7. Prototype Implementation

Present the prototype of the language, demonstrating incremental evaluation of temporal queries over input events.

8. Summary and Future Work

Summarize the results of this research, highlight the main contributions of this research, identify the limitations of the current approach, and discuss future directions based on the existing research results.

The timetable of the research is presented in the table 2:

Table 2: Research outline timetable

Task Number	Task	Start Date	End Date
1.	Design Temporal Event Query Language Constructs	July 2011	October 2011
2.	Develop Temporal Relational Framework	September 2011	January 2012
3.	Specify language operators using relational Framework	December 2011	February 2012
4.	Develop finite differencing formulae for language operators to enable incremental evaluation	January 2012	February 2012
5.	Develop language evaluation engine prototype	Ongoing process	
6.	Dissertation writing	August 2011	April 2012
7.	Defense	April 2012	

6. Conclusion and Contributions

This proposal has presented an integrated approach for employing temporal relational operators for querying of event streams. This approach is based on extending a preexisting relational framework for querying of event streams to support temporal queries. The most notable benefit of the proposed approach is the new kind of expressivity that is provided for interval-based event queries. Given the growing trend of adoption of interval-based semantics in the research community, this is an important contribution. Interval based versions of operators that were defined for point based event processing have less semantic issues, but are not sufficiently expressive for querying of interval based events. More expressivity has been added in more recent interval based languages by designing language constructs based on Allen's temporal relationships. However work in the temporal database area generalizes relational operators to support interval based data. Integrating these operators into an event processing language adds a new dimension for querying of event streams. Besides the added expressivity the approach proposed here further offers the following high-level advantages:

1. The language and its semantics are firmly based on the theoretical foundation of the relational model which is formally defined, prevalent and well understood. In contrast, current approaches to formalization of event processing utilize widely divergent models, which are often not properly formalized. Further, as seen in the related work section, many of the approaches that do offer some kind of formalization suffer from various issues in terms of ambiguity in semantics or expressivity. Using the temporal relational model as a basis for event processing also allows for the exploitation of the large body of available work on optimization of query plans. Many of the current approaches utilize different, often unique, data structures for querying of events. As has been discussed before, using rigid data structures leaves no room for optimization based on reordering of operations. Another benefit of using the temporal relational model is that, due to this common foundation with relational databases, extensions and development of relational database languages can easily be transferred over to the event processing realm.

2. The proposed approach will remove the mismatch between database data and event data. Since there is a direct correspondence between events and event types on one hand and tuples and relations on the other, querying data of both kinds simultaneously is straightforward. In many of the previous approaches, this correspondence is either non-existent or not immediately clear.

3. The proposed approach will shift the mindset for treatment of events to a database point of view, where the event data persists. The traditional approach to event stream processing has underlined the assumption that the high rates of input data, coupled with a limited amount of main memory make it necessary to discard data that is not going to be used any further. The problem with this mindset is that it creates an artificial classification of relevant and irrelevant data. In almost all work on event stream processing, it is assumed that the limited amount of data that can be maintained in memory is 'relevant' data. Consequently data that cannot be maintained due to limited resources is automatically deemed irrelevant. This translates into very concrete restrictions on the query language in order to enforce ejection of data that we cannot afford to keep in memory. In contrast, in the database approach all data are equal in the

sense that they must persist. We can still try to provide faster access to data that we might require more often by placing it higher in the storage hierarchy. It is also notable that the emergence of solid state memory has made large and fast persistent storage a reality.

4. Finally, since the event data will eventually need to reside in a relational framework, it brings attention to the necessity of thinking ahead about the modeling of event data and mapping them correctly to a temporal relational framework. Issues such as identity and the relationships of different events to each other need to be sorted out in the initial design of the system.

In summary this research makes the following more specific contributions:

- Development of a relational framework that enables processing of temporal queries in an incremental manner,
- Development of an SQL-like event query language with integrated support for temporal queries, including specification of semantics using the relational framework, and
- Implementation of a prototype of the language that demonstrates incremental evaluation of temporal event queries.

7. References

Abadi, D.J. et al., 2003. Aurora: a new model and architecture for data stream management. *The VLDB Journal The International Journal on Very Large Data Bases*, 12(2), pp.120-139.

Adaikkalavan, R. & Chakravarthy, S., 2005. Formalization and detection of events using interval-based semantics. In *IN PROCEEDINGS, INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*. Citeseer, p. 58--69.

Allen, J.F., 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), pp.832-843.

Arasu, A., Babu, S. & Widom, J., 2006. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal—The International Journal on Very Large Data Bases*, 15(2), pp.121-142.

Barga, R.S. et al., 2006. Consistent Streaming Through Time: A Vision for Event Stream Processing. *cs/0612115*. Available at: http://arxiv.org/abs/cs/0612115 [Accessed October 24, 2010].

Barga, R.S. & Caituiro-Monge, H., 2006. Event Correlation and Pattern Detection in CEDR. In T. Grust et al., eds. *Current Trends in Database Technology – EDBT 2006*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 919-930. Available at: http://www.springerlink.com/content/tk7t177l27p58571/ [Accessed June 20, 2011].

Bittner, S. & Hinze, A., 2004. Classification and Analysis of Distributed Event Filtering Algorithms. In R. Meersman & Z. Tari, eds. *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 301-318. Available at: http://www.springerlink.com/content/g705rqv1e6p5ef8e/ [Accessed June 29, 2011].

Brenna, L. et al., 2007. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. SIGMOD '07. Beijing, China: ACM, pp. 1100–1102.

Bry, F. & Eckert, M., 2008. On static determination of temporal relevance for incremental evaluation of complex event queries. In *Proceedings of the second international conference on Distributed event-based systems*. ACM, pp. 289-300.

Bry, F. & Eckert, M., 2007. Rule-based composite event queries: The language XChange eq and its semantics. *Web Reasoning and Rule Systems*, pp.16-30.

Bry, François & Eckert, Michael, 2007. Rule-based composite event queries: the language XChangeEQ and its semantics. In *Proceedings of the 1st international conference on Web reasoning and rule systems*. RR'07. Innsbruck, Austria: Springer-Verlag, pp. 16–30. Available at: http://portal.acm.org/citation.cfm?id=1768725.1768728 [Accessed June 19, 2011].

Bry, Francois & Schaffert, S., 2003. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In A. B. Chaudhri et al., eds. *Web, Web-Services, and Database Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 295-310. Available at: http://www.springerlink.com/content/80wfu9v7r6uv84kg/ [Accessed June 29, 2011].

Chakravarthy, S. & Mishra, D., 1994. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1), pp.1-26.

Cugola, G. & Margara, A., 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS '10. Cambridge, United Kingdom: ACM, pp. 50–61.

Darwen, H. & Date, C.J., 2005. An Overview and Analysis of Proposals Based on the TSQL2 Approach. Available at: http://www.dcs.warwick.ac.uk/~hugh/TTM/OnTSQL2.pdf.

Date, C.J. & Darwen, Hugh, 2000. *Foundation for Future Database Systems: The Third Manifesto* 2nd ed., Addison-Wesley Professional.

Date, C.J., Darwen, H. & Lorentzos, N.A., 2002. *Temporal Data and the Relational Model*, Morgan Kauffman Publishers.

Date, C.J., 2003. *An Introduction to Database Systems* 8th ed., Addison Wesley.

Demers, A. et al., 2005. A general algebra and implementation for monitoring event streams.

Diao, Y. et al., 2002. YFilter: Efficient and scalable filtering of XML documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE, pp. 341-342.

Dong Zhu & Sethi, A.S., 2001. SEL, a new event pattern specification language for eventcorrelation. In *Tenth International Conference on Computer Communications and Networks, 2001. Proceedings*. Tenth International Conference on Computer Communications and Networks, 2001. Proceedings. IEEE, pp. 586-589.

Eckert, M., 2008. *Complex Event Processing with XChangeEQ: Language Design, Formal Semantics, and Incremental Evaluation for Querying Events*. PhD Dissertation. Fakultat fur Mathematik, Informatik und Statistik: Ludwig-Maximilians-Universitat Munchen.

EsperTech, Inc., *Event Stream Intelligence: Esper & NEsper*, EsperTech Inc. Available at: http://esper.codehaus.org/.

Etzion, O., Jajodia, S. & Sripada, S., 1998. *Temporal databases: research and practice*, Springer Verlag.

Galton, A. & Augusto, J.C., 2002. Two Approaches to Event Definition. In A. Hameurlain, R. Cicchetti, & R. Traunmüller, eds. *Database and Expert Systems Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 547-556. Available at: http://www.springerlink.com/content/46mbb6ajt6t20qvd/ [Accessed June 29, 2011].

Gatziu, S., Geppert, A. & Dittrich, K.R., 1995. The SAMOS active DBMS prototype. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*. SIGMOD ’95. San Jose, California, United States: ACM, p. 480–.

Gehani, N.H., Jagadish, H.V. & Shmueli, O., 1992. Event specification in an active object-oriented database. *SIGMOD Rec.*, 21(2), pp.81–90.

Ghezzi, C., Mandrioli, D. & Morzenti, A., 1990. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2), pp.107-123.

Griffin, T. & Libkin, L., 1995. Incremental maintenance of views with duplicates. *SIGMOD Rec.*, 24(2), pp.328–339.

Gyllstrom, D. et al., 2006. Sase: Complex event processing over streams. *Arxiv preprint cs/0612128*.

Hinze, A. & Voisard, A., 2002. *A flexible parameter-dependent algebra for event notification services*, Freie Universitat Berlin. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.494.

Luckham, D.C., 2007. A short history of Complex Event Processing. part 1: Beginnings. *Online only (http://complexevents. com/wp-content/uploads/2008/02/1-a-short-history-of-cep-part-1. pdf).*

Mei, Y. & Madden, S., 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 35th SIGMOD international conference on Management of data*. ACM, pp. 193-206.

Morrell, J. & Stevan D., V., 2007. *Complex Event Processing with Coral8*, Available at: http://www.coral8.com/system/files/assets/pdf/ Complex_Event_Processing_with_Coral8.pdf.

Mühl, G., Fiege, L. & Pietzuch, P., 2006. *Distributed Event-Based Systems*, Springer-Verlag New York, Inc.

Patroumpas, K. & Sellis, T., 2006. Window specification over data streams. *Current Trends in Database Technology–EDBT 2006*, pp.445-464.

Pietzuch, P.R., Shand, B. & Bacon, J., 2004. Composite event detection as a generic middleware extension. *IEEE Network*, 18(1), pp.44- 55.

Purich, P., 2010. Oracle Complex Event Processing Getting Started 11g Release 1 (11.1. 1) E14476-03.

Qingchun Jiang, Adaikkalavan, R. & Chakravarthy, S., 2007. MavEStream: Synergistic Integration of Stream and Event Processing. In *Digital Telecommunications, 2007. ICDT '07. Second International Conference on*. Digital Telecommunications, 2007. ICDT '07. Second International Conference on. p. 29. Available at: 10.1109/ICDT.2007.21 [Accessed October 24, 2010].

Roncancio, C.L., 1999. Toward Duration-Based, Constrained and Dynamic Event Types. In S. F. Andler & J. Hansson, eds. *Active, Real-Time, and Temporal Database Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 176-193. Available at: http://www.springerlink.com/content/cuxlutwd3u4jy82n/ [Accessed June 20, 2011].

Sánchez, C. et al., 2003. Event Correlation: Language and Semantics. In R. Alur & I. Lee, eds. *Embedded Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 323-339. Available at: http://www.springerlink.com/content/06eutrptglke0nw7/ [Accessed June 19, 2011].

Snodgrass, R.T., 1995. *The TSQL2 Temporal Query Language*, Kluwer Academic Publishers.

Srivastava, U. & Widom, Jennifer, 2004. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. PODS '04. Paris, France: ACM, pp. 263–274.

Walzer, K., Breddin, T. & Groch, M., 2008. Relative temporal constraints in the Rete algorithm for complex event detection. In *Proceedings of the second international conference on Distributed event-based systems*. DEBS '08. Rome, Italy: ACM, pp. 147–155.

White, W. et al., 2007. What is next in event processing? In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, pp. 263-272.

Wu, Eugene, Diao, Yanlei & Rizvi, S., 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. SIGMOD '06. Chicago, IL, USA: ACM, pp. 407–418.

Zaidi, A.K., 1999. On temporal logic programming using Petri nets. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 29(3), pp.245-254.

Zimmer, D. & Unland, R., 1999. On the Semantics of Complex Events in Active Database Management Systems. In *Data Engineering, International Conference on*. Los Alamitos, CA, USA: IEEE Computer Society, p. 392.