SCEPter: Semantic Complex Event Processing over End-to-End Data Flows

Qunzhi Zhou[1], Yogesh Simmhan[2] and Viktor Prasanna[1,2]

[1]Department of Computer Science,

[2]Ming Hsieh Department of Electrical Engineering,

University of Southern California

Acknowledgement

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000192.

Disclaimer

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF SOUTHERN CALIFORNIA

SCEPter: Semantic Complex Event Processing over End-to-End Data Flows

Qunzhi Zhou[1], Yogesh Simmhan[2] and Viktor Prasanna[1,2]

[1]Department of Computer Science,
[2]Ming Hsieh Department of Electrical Engineering,
University of Southern California

TECHNICAL REPORT USC-XXXX
Completed April 2012

Abstract

Emerging Complex Event Processing (CEP) applications in cyber physical systems like Smart Power Grids present novel challenges for end-to-end analysis over events, flowing from heterogeneous information sources to persistent knowledge repositories. CEP for these applications must support two distinctive features – easy specification patterns over diverse information streams, and integrated pattern detection over realtime and historical events. Existing work on CEP has been limited to relational query patterns, and engines that match events arriving after the query has been registered. We propose *SCEPter*, a semantic complex event processing framework which uniformly processes queries over continuous and archived events. *SCEPter*is built around an existing CEP engine with innovative support for semantic event pattern specification and allows their seamless detection over past, present and future events. Specifically, we describe a unified semantic query model that can operate over data flowing through event streams to event repositories. Compile-time and runtime semantic patterns are distinguished and addressed separately for efficiency. Query rewriting is examined and analyzed in the context of temporal boundaries that exist between event streams and their repository to avoid duplicate or missing results. The design and prototype implementation of *SCEPter*are analyzed using latency and throughput metrics for scenarios from the Smart Grid domain.

1 Introduction

The pervasive deployment of sensing instruments is enabling us to monitor environments and infrastructures in realtime. These range from sensors monitoring traffic flow [13] to smart meters recording realtime power usage in buildings [4]. Event streams generated from such sensors need to be integrated with supplemental information sources to support analysis. Such analysis offer deeper insight into system behavior and dependencies to help with continuous planning, and for making prompt operational decisions. These emerging class of *Cyber Physical Systems* (CPS) [17] base the operation and optimization of complex physical systems on information analysis capability present in cyberspace.

The continuous nature of events generated from numerous sensors in CPS along with the need for low latency analysis makes them well suited for Complex Event Processing (CEP). For example, consider the Smart Power Grid [24] CPS where a utility is responsible for monitoring and shaping power demand using incentives to ensure that there is no mismatch between supply and demand[4]. Such energy management requires analysis over realtime events streaming from thousands of power meters, building sensors, facility schedules, mobile consumer apps, and weather stations. Besides utility operators, this information may also be used by power consumers to manage their energy use profile and by building managers to control facility scheduling. CEP can be used to analyze the realtime operational behavior of the power grid and help detect patterns that indicate peaking power usage, changing pricing incentives, and opportunities for demand curtailment.

However, these novel CPS applications motivate certain distinctive features to be supported by CEP engines. CPS applications tend to be cross-disciplinary, combining engineering, social behavior, and public policy. This means that the information space can be complex just as the consumers of these information diverse. As a result, not every user has a holistic understanding of all information sources and their structures. This requires easy specification of event patterns over information streams, which abstract away the individual domain complexities. Often, semantic ontologies for individual domains are used to achieve this abstraction and make knowledge more accessible to the end user[10, 25].

Another feature that is commonly sought is to analyze both continuous and archived information. Event streams are often archived to comply with regulations[18], perform cause-effect analysis and mine for novel patterns of energy use. This provides the opportunity to perform queries over end-to-end data that flows from sensors to event repository. Seamless pattern detection over realtime and historical events would make it easier for users specify a uniform query – irrespective of past, present or future occurrence of a pattern, and ensure ordered and consistent result streams – without duplicates or missing events, to be provided. Existing work on CEP have been limited in these directions. Most CEP engines process structural events and pattern queries are specified at the relational level, requiring users to be aware of the underlying structural heterogeneity[6]. Active and temporal databases leverage relational query engines for time varying data that is persisted [8], using triggers and incremental queries to match newly arriving data. Some even build CEP engines on top of a relational database kernels[15]. These however sacrifice the expressivity of CEP patterns and introduce latency into results that are returned. Contemporary work on recency-based pattern correlation queries (PCQ) [16] support a happen-before relation linking live streams with patterns present in archived streams. However, this is more a data correlation feature that materializes archived events as streams, rather than provide seamless pattern matching across live and archived events.

In this report, we propose *SCEPter*, a semantic complex event processing framework which uniformly processes queries over continuous and archived events. *SCEPter*is built around an existing CEP engine with innovative support for semantic event pattern specification and allows their seamless detection over past, present and future events. *SCEPter*is motivated by the Smart Power Grids domain, as part of the US Department of Energy sponsored Los Angeles Smart Grid project, and applied to the USC Campus Microgrid testbed.

Specifically, our contributions here are as follows:

- We propose a unified semantic query model that can operate over end-to-end event data in the past, present and future.
- We present a prototype implementation of *SCEPter*, which supports the proposed model and operates on continuous event streams and archived event repository.
- We introduce compile-time pattern rewriting, runtime semantic annotation and filtering of semantic patterns, and resilient streaming of results in the presence of temporal gaps between streams and the repository.
- We analyze the processing latency and throughput provided by *SCEPter*for diverse queries from the Smart Grid domain.

The rest of the report is organized as follows: Section 2 introduces semantic CEP applications in the Smart Grid Domain, Section 3 summarizes the solution approach of *SCEPter*, Section 4 describes the semantic event model and query model, Section 5 describes *SCEPter*semantic CEP engine for stream processing, Section 6 describes semantic CEP on archived data, Section 7 describes integrated semantic CEP query over end-to-end data flows which consist of both streaming and archive data, Section 8 reviews the architecture of *SCEPter*and describes system module implementations, evaluation experiments are presented in Section 9, we review related work in Section 10 and finally discuss future work and conclude the report in Section 11.

2 Background

2.1 Demand-Response in Smart Grids

Smart Power Grids are an exemplar of cyber physical systems and form an emerging application domain for complex event processing. Specifically, they refer to the modernization of the electric grid by integrating sensors and communication devices – at the generation, transmission and distribution to end users, and software for data analytics and automated operations.

Demand response optimization (DR) is one of the cornerstone Smart Grid applications that uses these improved capabilities to prevent a mismatch between power generation and consumer demand. It helps a utility predict when a peak load will occur and identifies demand curtailment strategies to shape or shift the load. This improves operational reliability, avoids blackout/brownouts and reduces the maximum power generation capacity required. CEP can help DR applications to migrate from a static strategy, based on historical load averages and *a priori* load reduction commitment by consumers, to a dynamic strategy that uses realtime event patterns to determine peaking trends and offers targeted curtailment incentives such as variable rate pricing.

2.2 Smart Grid CEP Characteristics

Smart Grid event processing present novel characteristics that existing CEP systems are not well designed for. Specifically, they require two distinctive features – specification of semantic patterns over diverse information streams, and integrated pattern detection over realtime and historical events.

As part of the Los Angeles Smart Grid Project in the largest public utility in the US, the University of Southern California (USC) campus serves as a testbed to experiment and evaluate DR technologies. This microgrid environment encompasses diverse data and concepts adopted by its multi-disciplinary participants that include device vendors, end-use customers, facility managers and the utility. Continuous, time-series data from sensors and information sources can be abstracted as events. These sources and event types include equipment and appliances (*ThermostatChange* event, *TemperatureChange* event), smart meters (*MeterUpdate* event), weather services (*HeatWave* event) and consumer activities (*ClassSchedule* event, *RoomOccupancy* Event). Even event sources that produce the same type of events may vary in terms of data schema and terminologies. On the other hand, users of CEP systems such as USC's Facility Management Service (FMS) operators, building managers or even individual student/staff/faculty customers may define energy use pattern queries based on their own domain knowledge. The heterogeneity of data sources and concepts in an evolving power grid makes it unreasonable for users to be aware of fine grained event details. A CEP system should capture the semantics of events and their attributes, such as the types and relations of domain entities, to ease the specification of CEP pattern queries at a higher level of abstraction that suits their application.

Secondly, data generated in the Smart Grid flows from sensors and meters, through communication networks, and onto data repositories for archival. The latter is required both by data mining tools [9] and for regulatory compliance of utility operations. Given the different consumers of this information, CEP queries may not have been specified *a priori* to monitor the data streams. For example, a query to detect curtailment opportunities for DR may only be activated after receiving a DR signal from the utility, or new queries may be created and added to the system. This introduces the problem of a user's need to detect patterns that occurred after the query was submitted as well as their historical occurrences for decision making. This motivates the need for a unified query framework over the end-to-end flow from streams to event repositories.

Consider the following pattern to illustrate the need for these two CEP features in the USC campus microgrid.

> An operator in the USC FMS control center receives a DR event notification from the utility at noon requesting load curtailment. The operator then wants to detect *Office rooms* where the *airflow rate* of the Heating/Ventilation/Air Conditioning (HVAC) unit exceeds *500 cfm*, but wants this pattern to be

initiated from 9AM since this morning. Upon detecting these situations, the operator can reduce the fan speed and duty cycle of the HVAC unit for those rooms to limit power usage.

In this scenario, "Office" is a semantic concept, not presented in the event schema itself but available as part of the domain knowledge-base and associated with the location of the HVAC unit generating the airflow measurement events. Second, buildings with different HVAC unit manufacturers and building control systems may adopt terminologies such as "flowrate", "airvolume" or "airflow" to describe the same event. Finally, the operator is interested in historical occurrences of this pattern since this morning as it indicates the rooms that have been consistently consuming more power today and offer a higher probability of ensuring sustained reduction in energy usage.

3 Approach

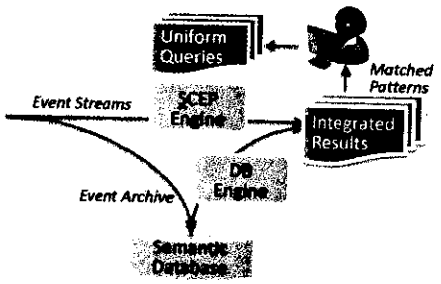


Figure 1: Uniform query on end-to-end data flow

Figure 1 shows our overall approach to support historical and realtime matching of semantic CEP patterns. Event data arriving continuously on streams is forked and passed to both event database for archival and to a semantic CEP engine. CEP queries provided by users are uniformly defined over past, present and future events using semantic concepts for the domain. The semantic CEP engine monitors the event streams to detect these patterns after query submission while a database query engine locates these patterns on the archived event repositories to discover historical matches. Results from both the database and CEP engine are integrated in time order and provided to the user.

There are several alternate solutions for this problem, the obvious one being to use a single query engine to process both stream data and archived data. Using a single *event* processing engine, we can extract the historical data from the repository and generate a virtual event stream that is fed to the CEP engine for pattern matching, as a precursor to current events. The problem with this approach is the need to retrieve and re-orchestrate all archived events for a stream whenever a query that uses this stream is submitted or updated. This event materialization can pose a large overhead. An alternative is to use a single *database* query engine that use triggers to perform standing queries when new data arrives. The drawbacks of this approach are twofold. First, CEP queries are more intuitive for defining patterns over stream than database query languages. Second, active database systems do not scale well to high data rates for realtime query processing.

Our hybrid solution marries the best of both worlds and leverages both CEP and database engines to do things they are best suited for. Users can use CEP queries uniformly over both historical and continuous events irrespective of the event location. These CEP queries support semantic predicates, and we extend an existing CEP system to support semantic event processing. We discern static semantics associated with event queries and process them at compile/query specification time, and dynamic semantics present in events are handled at runtime.

We unify database and stream queries through query rewriting. CEP pattern languages offer extensions to database query models using temporal operators like time windows, sequence, and so on. We discuss rewriting rules to transform temporal CEP operators to database query expressions. In particular, to facilitate semantic predicates in queries, we adopted a semantic data store as our persistent event repository and rewrite CEP queries into SPARQL.

Finally, we need to ensure that processing queries over events flowing through streams to repositories does not change the expected order of the matched results, or cause duplicates or missing results. This is non-trivial when considering that it may not be possible to ensure that an event is atomically present only in the CEP engine

or the database. As shown in Figure 1, there is often a temporal boundary or gap between an event processed by the stream engine and those processed by the database query engine due to the way the event stream is forked for archiving. At the moment a query is submitted, a relevant event may have already passed the stream engine but not have reach the database ("positive" gap), or it is possible that a relevant event that has reached database will also pass through the stream engine subsequently ("negative" gap). A simple union of the results from both the CEP and database queries may cause duplicates when the gap is negative, or missing patterns if the gap is positive. We analyze approaches to ensure that the results are integrated seamlessly and consistently on the end-to-end data flow.

4 Semantic Complex Event Patterns

One of our contributions is to support semantics in Complex Event Processing over streams. Toward this, we propose an event model which captures semantic event attributes, domain entities and their relations [27]. Such a model further allows CEP queries themselves to incorporate semantic predicates within them.

4.1 Semantic Event Model

We assume event data emanating from stream sources is a light-weight data tuple with a schema defined as a list of attribute name and type pairs, i.e.,

`Event Stream Data := {<attribute, type>*}`

In addition to this structural information, two types of semantic information need to be captured – the semantic meaning of event attributes, and the semantics of the domain entities and concepts that relate to an event.

Consider the example from Section 2 where the same concept of "airflow" has multiple structural variants. This is an example of capturing the semantic meaning of an event attribute. We observe other structural variants of the "airflow" concept such as "flowrate", "airvolume" and "airrate" as alternative representations. A traditional CEP system cannot apply a unified query over these event streams unless pattern designers are aware of the underlying structural heterogeneity of events and manually transform queries to suit each stream. This approach does not scale, not to mention that the data providers and CEP application users are decoupled.

We propose an ontology-based approach to capture semantics of event attributes. Domain event types that are captured include "TemperatureMeasurement", "AirflowMeasurement" and "CO2Measurement". One standard concept is used to represent a class of semantically equivalent attributes, and alternative concepts are modeled as its sub-class. For example, "evt:airflow" is the standard concept of airflow measurement readings, while equivalent classes "evt:flowrate" and "evt:airvolume" are its sub-classes. Integration of data with heterogeneous schemas can be automated using this ontology model.

The second aspect of semantics relates to domain entities in the knowledge-base that are related to events. For example, the source of a temperature or airflow measurement event may be from different rooms or physical spaces. The user is often interested in a specific category of these spaces when defining queries, such as the average airflow of "Office rooms" in a building or temperature change events in "Classrooms". These domain concepts associated associated with events from a specific stream are less dynamic. However, these concepts are not necessarily present as an attribute in the event itself but rather part of the domain knowledge. It is important to link this knowledge-base with the events for intuitive and expressive query design.

We propose domain models to capture specific concepts that related to events, as shown in Figure 2. For example, we have an electrical equipment ontology to capture types of equipment, and a physical space ontology to capture concepts related to locations and buildings[26]. An event can be linked to these semantic concepts using ontology relation such as "ee:hasID" and "ee:hasLocation".

4.2 Semantic Query Model

We use the semantic event model as the basis for defining a *Semantic CEP (SCEP)* query model. Further, this query model is used uniformly both for continuous and archived events, as we will describe later. Our model starts with a traditional CEP query model and incorporates semantic constraints that are based on semantic query languages. The structure of an SCEP query is:

```
SCEP Query ::=
  [PREFIX <namespace>]
```

```
[CEP subpattern]
[semantic subpattern]
[data window]
```

The CEP subpattern specifies the temporal and relational constraints of events based on their attributes. The semantic subpattern places semantic constraints over events and their associated domain entities. We adopt the W3C Semantic Web query language SPARQL [3] to represent semantic subpatterns as triples.

We introduce a new concept of *data window* (DW) as the time range upon which users wish to apply the query. DW is different from the CEP query window (QW). If the DW overlaps with a time in the past, the query should be executed over both historical and realtime events. On the other hand, a CEP QW specifies the time/length range for component events which constitute a pattern.

We generalize common features of the many CEP query languages [6, 5] and use the following structure for CEP subpatterns. Query usually indicates the target input stream, output definitions, event variable declarations, and temporal and content-based constraints. Our abstract CEP query model is,

```
CEP Subpattern ::=
  SELECT <event*, attribute*, aggregation*>
  FROM <event, input stream>*
  (WHERE <relational constraints>)?
  (SEQ <event, event, ...>)?
  (WINDOW <window specifications>)?
```

We illustrate the constructs of our SCEP query model using examples from the Smart Grid domain. These examples will also serve as candidate queries in our empirical evaluation in Section 9. Assume we have an event stream named `aStream` of airflow measurements, with schema {<sensorID, string>, <flowrate, double>, <timestamp, long> }.

Query 1.1. Simple CEP query. This pattern detects an airflow event with the flowrate measuring more than 500 cfm, with no temporal and window specified.

```
SELECT ?e.sensorID, ?e.flowrate
FROM ?e aStream
WHERE ?e.flowrate > 500
```

Query 1.2. Simple SCEP query. This pattern includes a semantic constraint to detect events from an "Office". The CEP subpattern is identical to Example 1.1.

```
PREFIX bd:<http://cei.usc.edu/Building.owl#>
PREFIX evt:<http://cei.usc.edu/SGEvent.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
    SELECT ?e.sensorID, ?e.flowrate
    FROM ?e aStream
    WHERE ?e.flowrate > 500
    {?e evt:hasEventSource ?src} .
    {?src bd:hasLocation ?loc} .
    {?loc rdf:type bd:Office} .
```

Query 2.1. Sequence CEP query. This pattern detects a sequence of two airflow events in a 5 minute *sliding* time window, with the flowrate of the second event greater than that of the first event by 100 cfm.

```
SELECT ?e1.flowrate, ?e2.flowrate
FROM ?e1 aStream, ?e2 aStream
WHERE ?e2.flowrate - ?e1.flowrate > 100
WINDOW(time, 5min, sliding)
```

Query 2.2. Sequence SCEP query. This pattern includes semantic constraints to detect events from an "Office".

The CEP subpattern is identical to Example 2.1.. Ignoring the namespace definitions for brevity, we have,

```
SELECT ?e1.flowrate, ?e2.flowrate
FROM ?e1 aStream, ?e2 aStream
WHERE ?e2.flowrate - ?e1.flowrate > 100
WINDOW(time, 5min, sliding)
{?e1 evt:hasEventSource ?src} .
{?e2 evt:hasEventSource ?src} .
{?src bd:hasLocation ?loc} .
{?loc rdf:type bd:Office} .
```

Query 3.1. Aggregation CEP query. This pattern computes the average flowrate of airflows in a 5 minute *batch* time window, for flowrates greater than 500 cfm.

```
SELECT AVG(?e.flowrate)
FROM ?e aStream
WHERE ?e.flowrate > 500
WINDOW(time, 5min, batch)
```

Query 3.2. Aggregation SCEP query. This pattern includes semantic constraints to detect events from an "Office". The CEP subpattern is identical to Example 3.1.

```
SELECT AVG(?e.flowrate)
FROM ?e aStream
WHERE ?e.flowrate > 500
WINDOW(time, 5min, batch)
{?e evt:hasEventSource ?src} .
{?src bd:hasLocation ?loc} .
{?loc rdf:type bd:Office} .
```

Query 4. SCEP query with Data Window. This semantic pattern introduces data range specification for querying over both historical and continuous events. The query in Example 2.1 with a date window since 9 AM on 16^{th} March, 2012,

```
SELECT ?e1, ?e2
FROM aStream
WHERE ?e2.flowrate - ?e1.flowrate > 100
WINDOW(time, 5min, sliding)
RANGE[2012-03-16T09:00, ]
```

The starting timestamp in the *RANGE* clause indicates the time of the oldest event that should be considered for a match. If the starting time is older than the time when the query was submitted, the query is executed over historical events. If the ending time is unspecified or is a point in the future, it is also executed over realtime streaming events.

Two things should be highlighted. One, when users design SCEP queries, they do not need to know details of the underlying data such as their schema. Queries are defined at a high level abstraction using domain ontology models, and the semantic mismatch between the incoming event and semantic query is addressed by the SCEP engine. Two, as we discuss later, the SCEP queries are applied to the events flowing from streams to the repository. These CEP subpatterns of the SCEP queries should be suitably rewritten in combination with the semantic subpatterns for execution by the semantic event database.

5 SCEP Querying on Streams

Our SCEP system is designed on top of an existing CEP engine using a pipelined architecture. Raw event tuples first enter a *semantic annotation module* where they are materialized into semantic events. These semantic events

pass to a *semantic filter module* which evaluates semantic subpatterns present in SCEP queries. Events that satisfy a semantic subpattern are allowed to pass to the CEP engine to evaluate the corresponding CEP subpattern of the query.

Semantics are processed both at compile time and runtime depending on whether they fall under static of dynamic categories, as mentioned in Section 4. These are discussed next.

5.1 Compile-time Semantic Processing

Semantic mismatches present in the event metadata, particularly the event schemas, and in the specification of the user queries can be addressed offline when a query is compiled since they do not vary often. Assume we have three airflow streams registered with the SCEP system with different attribute names for the same concept,

```
stream_a:=
 {<sensorID,long>, <airflow,double>, <timestamp,long>}
stream_b:=
 {<sensorID,long>, <flowrate,double>, <timestamp,long>}
stream_b:=
 {<sensorID,long>, <airvolume,double>, <timestamp,long>}
```

A CEP subpattern defined on these streams uses a different attribute name for the same concept,

Query 5. CEP query with static semantics

```
SELECT ?e
FROM stream_a, stream_b, stream_c
WHERE ?e.airrate > 500
```

Existing CEP engines require that queries and streams use syntactically identical event schemas. To process CEP queries as above, the semantic mismatches between event schemas and user query has to first be addressed. Here, "flowrate" is defined as the equivalent class of "airflow", "airvolume" and "airrate" in our domain ontology. This allows straightforward static semantic inferencing. When new streams are registered with our SCEP system, it queries the domain ontologies to map the attributes in its schema to its equivalent standard concept. So the schema for the above three streams are all normalized to,

```
{<sensorID,long>, <flowrate,double>, <timestamp,long>}
```

Similarly, when a new user query is registered with the SCEP system, we use the ontologies to normalize concepts used in the user query. Thus in the above query, the "airrate" will be statically rewritten to "flowrate" when the query is registered.

5.2 Runtime Semantic Filtering

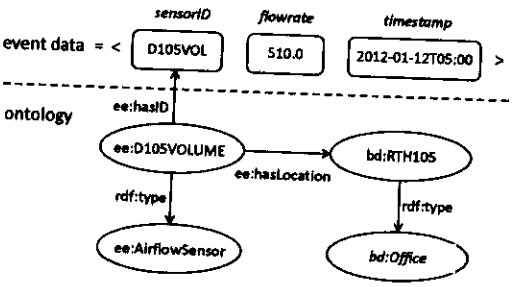


Figure 2: Dynamic Semantics

Dynamic event semantics are associated with event instances. For example, Figure 2 shows the semantic type of the location (`bd:RTH105`) from where an event originates. Though this information is not presented in event

tuple, it is related through the "sensorID" value of that particular event instance. User queries such as Query 1.2, 2.2 and 3.2 which involve dynamic event semantics are processed at runtime.

The baseline approach of runtime semantic event processing is to annotate and materialize semantic events from arriving event tuples and then perform semantic query over each new semantic event and the domain ontologies. A caching strategy for further optimization is discussed in our implementation.

6 SCEP Querying on Archive

Here, we describe our approach to transform the uniform SCEP query model for both streams and archives into Semantic Web SPARQL queries to detect patterns on archived data. The unified query model contains the semantic and the CEP subpatterns. The semantic subpattern already conforms to SPARQL and does not require further processing. However the CEP subpatterns have a different construct and need to be mapped to SPARQL. We use rule-based rewriting for this transformation, and present rewriting rules for different CEP query operators into the target SPARQL expressions.

6.1 Simple CEP Query

Simple CEP queries such as Query 1.1 in Section 4 have no temporal operators such as sequence and window. We use the following rewriting rules for each of the "SELECT", "FROM" and "WHERE" clauses, as illustrated for Query 1.1 in Figure 3.

R1:SELECT. CEP "SELECT" clause maps to the "SELECT" clause of the SPARQL query with triple patterns in the SPARQL "WHERE" clause representing the relations between selected event attributes and events.

R2:FROM. CEP "FROM" clause maps to the triple patterns in the SPARQL "WHERE" clause which represent the relations between events their source streams.

R3:WHERE. CEP "WHERE" clause maps to filter patterns in SPARQL "WHERE" clause which evaluate attributes of events.

In addition, a CEP query outputs the result set in time order by default. Hence, we have the following additional rule to guarantee ordering of SPARQL query results,

R4:ORDER. By default, create a triple pattern in the SPARQL "WHERE" clause which selects the logical timestamp of the matching event, and create an "ORDER BY" clause to order the results on this timestamp.

6.2 Sequence Query with Sliding Window

Compared to simple CEP queries, sequence queries with sliding time window introduce two new operators, "SEQ" and "WINDOW". The rules to rewrite "SELECT", "FROM" and "WHERE" clauses remain the same. For the "SEQ" clause, we have rewriting rule,

R5:SEQ. CEP "SEQ" clause maps to triple patterns in the SPARQL "WHERE" clause which selects the timestamp of the events in addition to SPARQL filters to order the timestamps sequentially.

The sliding time window on database queries is enforced using the rule,

R6:Sliding Window. A sliding time window operator for CEP sequence query maps to a SPARQL filter that constrains the timestamp of the last event in the sequence minus the timestamp of the first event be less than the window length.

In addition, CEP sequence queries detect patterns once the last component event arrives. To guarantee the output order of the corresponding archived queries, we have the rule,

R7:Order. By default, create an "ORDER BY" clause to the SPARQL query which orders the results using the logical timestamp of the last component event of the sequence.

Figure 4 shows the query rewriting for Query 2.1.

6.3 Sequence Query with Batch Window

A sequence query can be also applied over a batch time window. The rewriting rule for a batch window operator is,

R8:Batch Window. A batch window for sequence query maps to a SPARQL filter which evaluates,

$$floor(?e_{last}.timestamp/w) - floor(?e_{first}.timestamp/w) = 0$$

where $?e_{last}$ is the last component event in the sequence, $?e_{first}$ is the first component event in the sequence and w is the window length. Figure 5 shows rewriting Query 2.1 with a batch window.

6.4 Aggregation Query with Sliding Window

An aggregation operator can simply map to its corresponding SPARQL aggregation function. To place a sliding window on the database query, we group the aggregation function by each distinct event $?e_{dist}$ in the database, and calculate the aggregation function within the window start with $?e_{dist}$. The rewriting rule is,

R9:Sliding Window. The sliding window specification for a CEP aggregation query maps to a SPARQL filter that evaluates,

$?e.timestamp - ?e_{dist}.timestamp < w$

and group the aggregation function by $?e_{dist}$

where $?e$ is the aggregated event variable, and $?e_{dist}$ is a variable of events from the same stream. Figure 6 shows rewriting Query 3.1 using above rule.

6.5 Aggregation Query with Batch Window

The rewriting rule for batch window operator in aggregation queries are different from sliding window. The rewriting rule is,

R10:Batch Window. The batch time window operator for aggregation query map to the following SPARQL assignment expression ,

$LET(?e\_group := floor(?e.timestamp/w))$

and groups the aggregation function by $?e\_group$, where w is the window length. Figure 7 shows rewriting Query 3.1 with a batch window specification.

The rule-based approach is not limited to CEP-to-SPARQL transform. If the backend database has a language other than SPARQL, such as SQL, a similar set of rules apply.

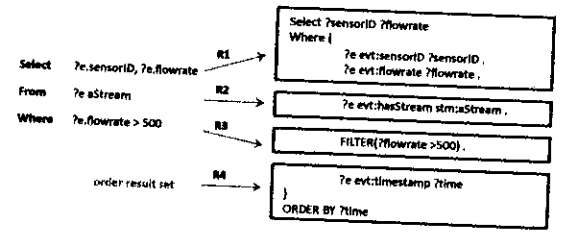


Figure 3: Rewrite Simple CEP Query

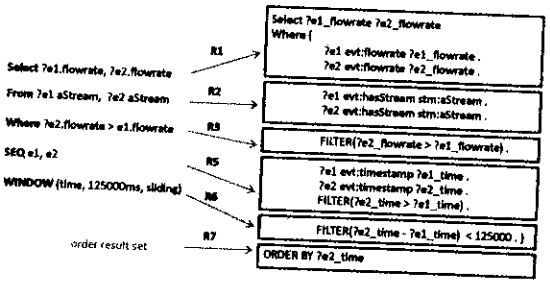


Figure 4: Rewrite Sequence Query with Sliding Window

7 Integrated End-to-End Query

Our earlier Sections 5 and 6 discussed our decoupled approaches for the SCEP queries over streams and databases separately. here we integrate these two strategies to provide seamless querying over end-to-end event data flows

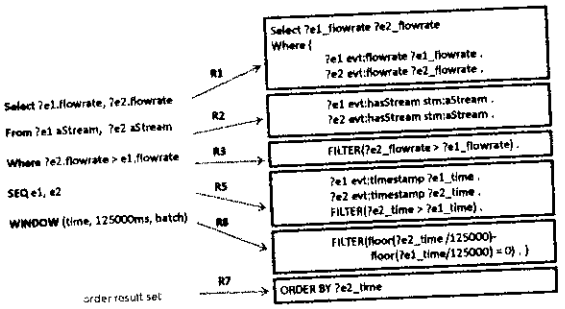


Figure 5: Rewrite Sequence Query with Batch Window

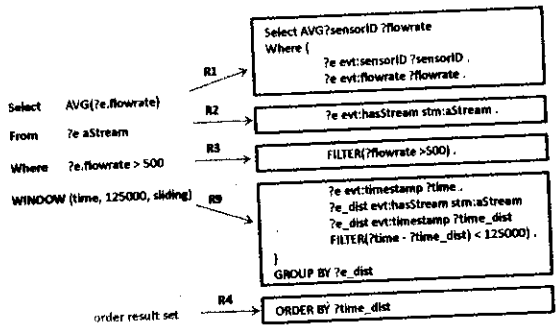


Figure 6: Rewrite Aggregation Query with Sliding Window

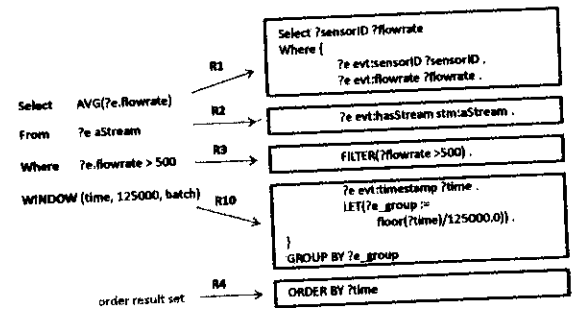


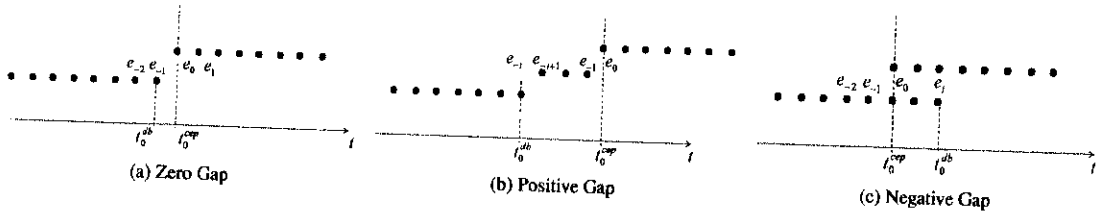
Figure 7: Rewrite Aggregation Query with Batch Window

while ensuring in order patterns detection, without duplicates or missing results.

7.1 Integrated Query Formulation

Firstly, we discuss the integrated query plan over the event flow assuming that events that pass atomically through the CEP engine enter the backend database immediately after without a gap i.e. every event must be visible to either the CEP engine or to the database engine, but not both. Without loss of generality, consider a SCEP query Q submitted to the event flow F which consists of a single logical stream S and a semantic repository D. Figure 8a shows the time when we submit Q on F as t_0^{cep}, the timestamp of the latest event in D as t_0^{db}, the first event observed by the CEP engine as e_0, the second event as e_1 and so on; the events before e_0 are denoted as $e_{-1}, e_{-2}, \ldots$, which are stored in the repository D.

In the following, we discuss the integrate query plan for different types of SCEP queries.



(a) Zero Gap (b) Positive Gap (c) Negative Gap

Figure 8: Different Gap Widths.

7.1.1 Simple Query with No Window

Consider an SCEP query Q with no window specified, such as Query 1.2. At time t_0^{cep} we submit Q to both stream S and database D. Let the subset of patterns detected on S be R_{cep} and patterns detected in D be R_{db}. The integrated result is $R = R_{cep} \cup R_{db}$. Given the no-gap scenario, there will be no duplicate or missing patterns in R.

7.1.2 Simple Query with Window

For an SCEP query Q with a window specified as W_{query}, denote the length of the window as W_{query}^{length}. By submitting Q on S and D separately, we can retrieve patterns R_{cep} and R_{db} which only contain events observed on S or D respectively. However, as shown in Figure 9, some patterns may be missed because a window can span the boundary between the stream and the database. Denote the missing pattern set as $R_{bounary}$, and define the boundary windows as,

$$W_{boundary}^{db} = (t_0^{cep} - W_{query}^{length}, t_0^{cep})$$
$$W_{boundary}^{cep} = [t_0^{cep}, t_0^{cep} + W_{query}^{length})$$

If the query window W_{query} is a batch window and t_0^{cep} was selected as a start point of one window, then obviously $R_{bounary} = \phi$. In the following we discuss $R_{boundary}$ only for sliding windows.

Denote $r_{bounary} \in R_{bounary}$ as one of the missing patterns. Assume $C = \{c_i | i = 0, ...n, n > 0\}$ as the component events of pattern $r_{bounary}$ in time order. Denote $c_i^{timestamp}$ as the timestamp of event c_i. $r_{bounary}$ need and only need to satisfy the constraints,

$c_0^{timestamp} \in W_{boundary}^{db}$, and
$c_n^{timestamp} \in W_{boundary}^{cep}$

To retrieve the missed pattern set $R_{boundary}$, we can wait for time $W_{boundary}^{cep}$ after which all the data in the boundary would have arrived in the database. We then extend query Q to Q' adding the necessary and sufficient constraints for missing patterns, and submit Q' to the database for retrieving $R_{boundary}$. The integrated query result set is $R = R_{cep} \cup R_{db} \cup R_{bounary}$.

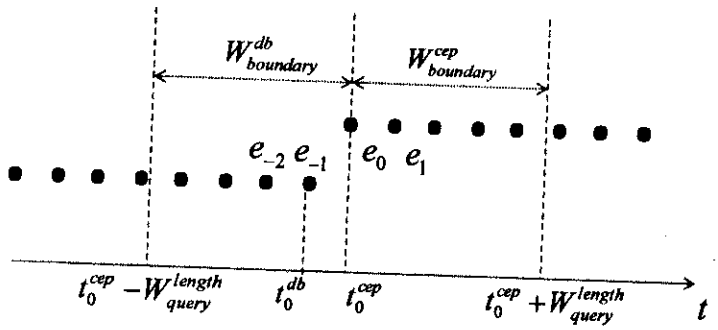


Figure 9: Zero Gap with Window

7.2 Impact of Gap Width

When we relax the assumption of a zero time gap between the CEP engine and database, two possibilities arise of either negative or positive gap on the data flow.

7.2.1 Negative Gap

Figure 8c shows a negative time gap between events in stream and the database. This means that events observed by the CEP engine after t_0^{cep} have already been stored in the database. This can lead to duplicate matching patterns to be returned if not accounted for. We handle this by adding a filter clause to the database query to only consider events with timestamp less than t_0^{cep}, as shown in Figure 10, so that we can otherwise use the same integrated query plan as the zero gap case.

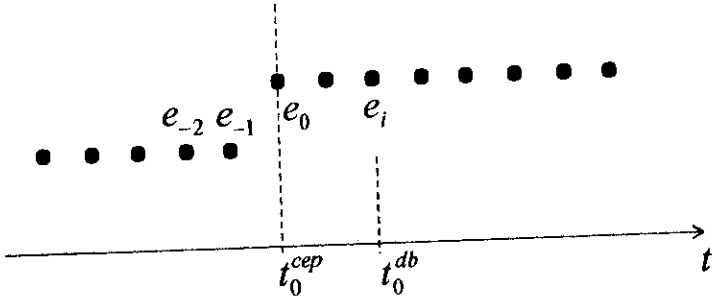


Figure 10: DB Query with Negative Gap

7.2.2 Positive Gap

With a positive time gap between events in the stream and the database, as shown in Figure 8b, at time t_0^{cep}, we have the event set $M = \{e_i | i = -i + 1 to - 1\}$ that have not arrived at the database. When submitting query Q at t_0^{cep}, these events are not visible to either the CEP or the database engine, causing patterns to be missed.

This is avoided as follows. For queries with no window specification, at time t_0^{cep}, we execute Q on stream S with the detected pattern set denoted as R_{cep}. Further, we wait for a time duration of $t_0^{cep} - t_0^{db}$ and execute Q on the database with a resulting pattern set R_{db}. The integrated result we get is $R = R_{cep} \cup R_{db}$.

For queries with window specifications, the query plan is similar, with the exception that the missed patterns in the boundary of the stream and the database is also considered.

8 *SCEPter* Architecture

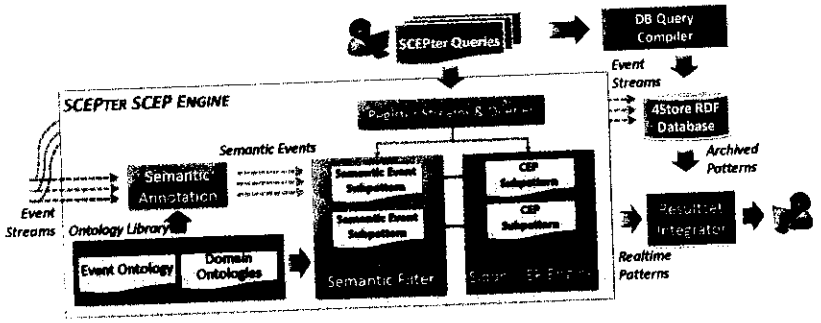


Figure 11: *SCEPter*Architecture Overview

*SCEPter*is our implementation of a semantic complex event processing system over event flowing from streams to archive. *SCEPter*is implemented fully in Java and follows the models and approaches introduced in the previous section to validate our design (Figure 11). The system can be view as two parts – the SCEP stream processing engine and the semantic database subsystem.

8.1 Semantic Stream Processing

The SCEP stream processing engine manages semantic event queries over data streams, performing semantic and CEP pattern matching. The SCEP engine is built around an existing CEP kernel, *Siddhi*[2]. Siddhi uses an event model based on data tuples and supports simple filtering, sequence and aggregation patterns over event streams. The current implementation of Siddhi supports sliding time windows but not batch windows. Siddhi is an exemplar of a traditional CEP engine that supports common CEP query operators and we leverage.

*SCEPter*provides additional streaming processing features around the Siddhi module to support the semantic querying models that we introduce. These key modules includes the ontology model, stream and query register, annotation module and semantic filter module.

The *domain ontology models*, represented in OWL, form a knowledge-base of concepts and their relations to support semantic query rewriting and processing. We organize ontologies in a modular and layered architecture for easier extension[26]. Connections between the event ontology and domain ontologies are made using properties like "hasEventSource" whose domain is an event and value is a domain entity.

The *stream and query register module* performs semantic rewriting of queries provided by the user. When a new event stream is registered with the SCEP engine for querying, this module will process the original event schema, referencing the domain ontologies to replace attribute fields with standard concepts. This allows semantically equivalent streams to be identified and queried by the same set of queries. On the other hand, users define queries over high level ontology models. When such a user query was registered, the module parses the query into a collection of clause objects and normalize concepts used in each type of clauses to conform with stream descriptions.

The *semantic annotation module* materializes semantic events from raw event tuples arriving on a registered input event stream. An annotation file describes the mappings from stream schema to semantic event properties for each stream. Several types of mappings are implemented, including direct mapping and query based mapping. In direct mapping, an event tuple attribute is directly used to create the URI of the corresponding semantic event property. In query based mapping, a query template is associated with the raw data attribute. When a new event tuple arrives, the annotation module populates the query template using event data, queries domain ontologies and retrieves the corresponding semantic property. Though both approaches were implemented, we chose to use the direct mapping in our experiments.

The *semantic filter module* processes dynamic event semantics. Domain ontologies are loaded in a JENA [1] memory model when SCEPter engine starts. At runtime, the system initiates a thread for each semantic subpattern of SCEPter queries and evaluates it over incoming events and ontologies. If an event satisfies the query, it will be passed onto the input stream feeding the Siddhi CEP engine; otherwise, the event is dropped. To improve the semantic pattern matching performance, we implemented caching optimization to reduce the frequencies of semantic queries. This is described next.

Finally, the Siddhi CEP kernel processes CEP subpatterns which are typical CEP event queries. The kernel is implemented in a plug-and-play manner so that other CEP engines can be easily integrated with our system.

8.2 Archive Data Processing

The database subsystem is used to archive a fork of the incoming events streams in a semantic database. The key modules in it are the semantic database and a database query compiler. We use the 4Store RDF database as our semantic repository due to its scalability properties [14]. 4Store offers a SPARQL query endpoint service that can be accessed to insert events into the database and query it for patterns. The query compiler create native SPARQL queries from the unified SCEPter query object. Rewriting rules discussed in Section 6 are implemented. At runtime, when a SCEPter query is submitted to the system, the corresponding database queries will be executed at a time depending on the gap width between stream and archive data. The gap width is currently implemented as a configuration parameter.

8.3 User Queries and Resultset

Users create a Java query object that contains fields for different SCEP subpatterns. This is registered with the *SCEPter*engine, and tied to incoming streams. We support streams that arrive over the network or in-memory streams. Results from the *SCEPter*come in an output stream. This output stream has query results from the stream processing and database subsystems. A *resultset integration module* is responsible for combining and returning the matched patterns in expected order.

8.4 Semantic Caching

We implement a cache optimization strategy to improve the performance of runtime semantic event processing. Semantic querying can be time consuming, and requires inferencing and self-joins to be performed over the triple database. We model semantic queries using a query graph, such as Figure 12 for the semantic subpattern of Query 1.2. We define the cache key for a query as the event tuple attributes used to materialize the semantic event properties present in the query graph. For example, the cache key for Query 1.2 is "sensorID" because this attribute is used to materialize the "evt:hasEventSource" property of the semantic event, while the "evt:hasEventSource" property presents in the only path of the query graph. If an event from the sensor with a particular sensorID satisfied the query, we can infer other events from the same sensor will satisfy this query. A semantic cache is implemented as a Java `HashMap`,where the key of the hashmap is the cache key and the value is a boolean varible indicates whether events with the cache key satisfy the query. When a new event arrives, the semantic filter first looks at the cache map for each query. If there is no hit, the filter performs the query over ontologies and caches the result, otherwise the cached value is retrieved as the query result directly.

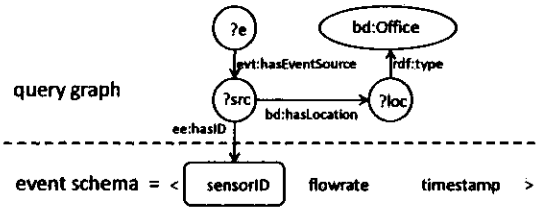


Figure 12: Semantic Query Graph and Cache Key

9 Experimental Evaluation

The *SCEPter*prototype implementation offers the higher level semantic abstractions and seamless querying across event streams and historical data that make it valuable to users in the Smart Grid domain. To ensure that these advances are not mitigated by severe performance overheads, we experimentally evaluate *SCEPter*under application scenarios expected for the USC Campus Microgrid.

For these experiments, we run *SCEPter*on a 12-core AMD Opteron server, with cores rated at 2.8GHz, with 32GB of physical memory and running Windows Server 2008. *SCEPter*is run on this server using 64-bit Java JDK v1.6. The 4Store database runs within a Linux Virtual Machine (due to its OS dependency) that has exclusive access to 1 CPU Core and 8GB of RAM, and is accessed over the local network port by *SCEPter*. All experiments we perform three or more times and the average values reported here.

Sensor data collected from HVAC systems of the RTH building on the USC campus is used as sample data and extrapolated to provide a large dataset for simulating the input streams. This simulation ensures that we can evaluate *SCEPter*for different event rates in a reproducable manner, while not stressing the USC FMS building control system that originally provide the data (and will form the data source for operational use). The sensor stream is simulated by loading the sample data into memory and having multiple threads loop through and insert events, after a specific sleep intervals, into the input event stream Java object that feeds the *SCEPter*system. This allows us to go from stream rates of 10 – 10,000 events/sec. The events in the simulated sensor stream have the schema, {`<sensorID, string>, <flowrate, double>, <timestamp, long>`}.

9.1 SCEP Queries over Streams

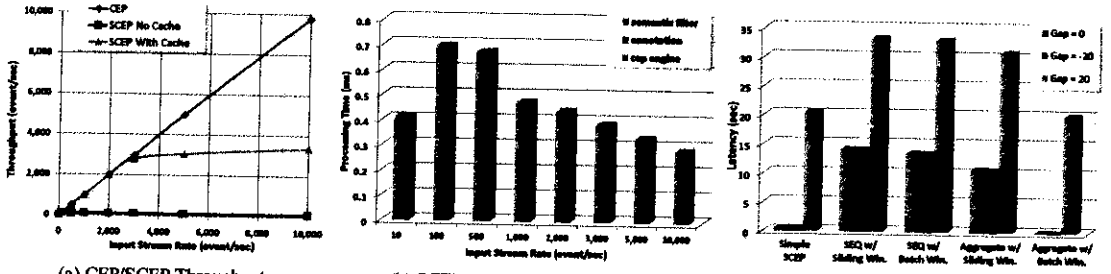
In these experiments, we evaluate the time performance of *SCEPter*'s semantic complex event processing engine. We evaluated the system when running queries relevant to the USC Campus Microgrid discussed in Section 4. The first study evaluates the throughput of the system when running Query 1.1 and 1.2 under three situations, (i) execute Query 1.1 using the CEP engine; (ii) execute the Query 1.2 with both semantic and CEP subpatterns, on *SCEPter*without cache optimization; (iii) execute Query 1.2 with cache optimization enabled. Figure 13a shows the throughput events/sec on the Y Axis for different input event stream rates on the X Axis. Ideally, the throughput should match the input event rate. When executing only CEP subpatterns, the throughput of the system which is essentially the throughput of the Siddhi CEP kernel. As we can see from the figure (blue diamond), Siddhi can scale up to 10,000 events/sec. When we execute the complete semantic CEP query, the throughput without cache optimization on *SCEPter*is very low (red square), achieving a peak of about 115 events/sec. However, with semantic caching enabled, *SCEPter*significantly improves upon this to reach a maximum throughput of around 3000 events/sec (green triangle). This rate is sustained irrespective of any increase in the input event rate past 3000 events/sec, thus indicating a threshold being reached.

We drill down into the processing times spent within each module in the *SCEPter*pipeline. Figure 13b shows the processing time (in milliseconds) of the semantic annotation (with caching enabled), semantic filter and Siddhi CEP engine for different input event rates. We observe that a majority of the time is spent in the semantic filter module, as can be expected since it is evaluates the semantic subpattern in the SCEP query, for every event that arrives, using the domain ontology. The comparative time take by the annotation and CEP kernels are negligible. For larger input event rates, we see that processing time for the semantic filter begins to plateau at about 0.3 ms. The inverse of this corresponds to the 3000 events/sec threshold that we observed in Figure 13a.

We perform these experiments for all SCEP queries listed in Section 4 (figures omitted for brevity) and observe that the results are similar despite the different types of CEP subpatterns.

9.2 Integrated Queries over Data Flows

In these experiments, we evaluate the performance of the *SCEPter*system when applying integrated SCEP queries over streams and database. We use a single event stream rate of 100 events/sec for this experiment, which is comparable to event rates that will be archived. We preload the 4Store RDF database with 20,000 triples ($\sim$ 1,500 events) – it has been observed that as the triple store gets large, certain types of queries slow down. We measure the latency of applying the SCEP query over the database and returning the results to the integrated result set, for different types of queries and for different gap width times between stream and database.



(a) CEP/SCEP Throughput (b) SCEP Module Processing Time (c) SCEP End-to-End Flow Query

Figure 13: Evaluation of *SCEPter*for streaming and historical SCEP queries.

While SCEP queries with different types of CEP subpatterns have similar performance for stream, they show wide variability when rewritten and applied to databases. Consider the zero gap case in Figure 13c (blue). We see that a simple SCEP query or one with batch aggregation subpattern is much faster (1$\sim$500ms) compared to the other three types of queries (10$\sim$15secs). This penalty is because when a CEP query with a sliding time window or sequence operator is rewritten to SPARQL, a large number of self-join is performed on the event timestamps in the database. Increasing the number of triples in the database exacerbates this problem. This motivates the need for further research to improve continuous query performance over semantic databases.

We also evaluated the impact of gap width on the query latency using positive and negative time gaps of 20 secs between events flowing into the SCEP engine and the database. Figure 13c shows that a negative gap does not change the latency much compared to a zero gap. This is because our query plan in this scenario will execute

the database query right after the it is submitted, but use constraints as described earlier to query only a subset of events. For the positive gap, our consistency approach is to wait until the data in the gap arrives at the database before executing the query. Hence, the latency of the system in this case is approximately query time plus the gap width.

10 Related Work

Our novel contributions in this work are toward semantic event pattern detections and seamless queries over real-time and historical event flows. Related research fall into the categories of semantic CEP, stream/event processing and active databases.

Our work falls in the space of stream and complex event processing systems [7, 11, 6]. These systems create intuitive languages to query data streams and react to continuous data with low latency. For example, the Aurora [7] stream processing system supports realtime monitoring using a "boxes-and-arrows" tuple flow query model. Several event processing systems allow expressive representation of temporal data patterns such as windows and sequences with and efficient processing. Event processing languages typically follow a SQL-like syntax with native temporal operators and pipeline sequences subsequent to the relational operators. for example, Cayuga [6] is designed to efficiently detect a large number of concurrent complex event subscriptions using a Nondeterministic infinite automata (NFA) algorithm to capture the event sequences.There is limited support in existing stream/event processing systems for semantics and our work offers a superset of their query features since we extend an existing CEP system . Historical data is usually not considered if not specified *a priori* as part of an existing query.

Some recent work introduce semantics into CEP. [21] proposed a modular ontology model and architectural vision for semantic complex event processing. In their architecture, the semantic knowledge base includes ontologies and inferencing rules. Syntactic sequence checks and semantic queries are processed in two steps. Our system architecture resembles the architecture vision in some sense. However we integrate existing CEP engines, categorize semantics presented in event processing and processing them at both compile-time and runtime. ETALIS [5, 23] is a rule-based deductive system that acts as a semantic event processing engine. Domain knowledge is modeled using OWL ontologies and event queries defined in SPARQL with extensions for temporal operators. OWL ontologies and user queries are transformed to Prolog rules and executed in a Prolog inference engine. This is a bespoke solution that is independent of traditional CEP engines, while we leverage the strengths of existing CEP and database systems for their expressivity and performance.

Active databases share certain concepts and goals of our work, and reuse a passive DBMS to query time-varying data. ECA rules and trigger mechanisms [22, 19] were defined for active databases to support standing queries with efforts to optimize such "continuous queries"[20]. Tapestry [20] converts a continuous query in active database, into an incremental query that efficiently finds new matches to the original query as new data is added to the database. By limiting the query to the portion of the database that might newly match the query, these queries avoid duplicates. We use similar strategies for SCEP query rewriting for databases, but also seemlessly integrate it with the stream processing engine with support for semantic. In addition, event traditional stream query languages are more expressive than triggers used in active databases that are restricted to data or schema object changes in the database

Recent work has explored the use of database query engines to process continuous queries that also reference static data. DataCell [15] exploits existing relational database algorithms and techniques for efficient stream processing. Incoming data tuples are stored into baskets/tables, then are queried in batch and carefully removed from these temporary tables by queries. While it allows a unified query on streams that also includes historical data, it is distinct from our query model where we see the database as a logical extension of the stream back in time rather than a static data source to perform a query join. Thus we also consider issues of querying boundary data between streams and databases with possible gaps.

The need for processing continuous queries across streams and history repositories, correlated using time windows, has been recognized [12, 16]. Recently, [16] defined the semantics of recency-based pattern correlation queries (PCQ) over live streams and archived streams. The recency clause in PCQ is essentially a "happen-before" relation which specifies the temporal distance between patterns in live streams and in archived streams. This work however limits the size of historical data that is considered in a database query and allows it to materialize historical events out of the database for processing within the CEP engine. We do not impose any correlation requirements and length of history in our integrated queries, and also consider gaps between the two data sources in an integrated data flow.

11 Conclusions

In this report, we have addressed two novel problems of supporting semantic CEP queries and integrated querying over end-to-end data flowing through streams to event archive. Both these allow diverse users to more easily specify their event patterns over heterogeneous information sources. We use compile-time and runtime query rewriting to execute these SCEP queries, and offer approaches to handle data flows with temporal gaps between the SCEP engine and the event repository. We have illustrated the value of these contributions using sample queries and scenarios from the Smart Grid domain, and provided a prototype implementation of the *SCEPter*system to validate the feasibility of our design.

Our empirical analysis of the *SCEPter*system offer insights for future research. While the cache optimization reduced the overhead of SCEP queries, the semantic filter limits the throughput to ~3000 events/sec. Further, the viability of seamless querying over the semantic archive will be challenging when large event rates grow the database size. Optimizations to handle these performance bottlenecks need to be investigated.

References

[1] Jena Framework. `http://jena.sourceforge.net`.

[2] Siddhi. `http://siddhi.sourceforge.net`.

[3] SPARQL. `http://www.w3.org/TR/rdf-sparql-query`.

[4] Ferc assessment of demand response and advanced metering. Staff Report, December 2008.

[5] Stream reasoning and complex event processing in etalis. *Semantic Web Journal*, 2012.

[6] A. Demers et al. Cayuga: A general purpose event monitoring system. In *CIDR*, 2007.

[7] D. Abadi and D. C. et al. Aurora: a data stream management system. In *SIGMOD*, 2003.

[8] R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. *Data and Knowledge Engineering Journal*, 2006.

[9] S. Aman, Y. Simmhan, and V. K. Prasanna. Improving energy use forecast for campus micro-grids using indirect indicators. *International Workshop on Domain Driven Data Mining (DDDM)*, 2011.

[10] J. L. Andrew Crapo, Xiaofeng Wang and R. Larson. The semantically enabled smart grid. Technical report.

[11] Arasu, S. Babu, and J. Widom. The CQL continuousquery language: Semantic foundations and query execution. In *The VLDB Journal*, 2006.

[12] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee. Moirae:history-enhanced monitoring. In *CIDR*, 2007.

[13] A. Biem, E. Bouillet, and et al. IBM infosphere streams for scalable, real-time intelligent transportation services. In *SIGMOD*, 2010.

[14] S. Harris and et al. 4store: The design and implementation of a clustered rdf store. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, 2009.

[15] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the power of relationaldatabases for efficient stream processing. In *EDBT*, 2009.

[16] N.Dindar, M. Fischer, M.Soner, and N.Tatbul. Efficiently correlating complex eventsover live and archived data streams. In *DEBS*, 2011.

[17] R. Poovendran and et al. Special issue on cyber - physical systems. *Proceedings of the IEEE*, 2012.

[18] E. L. Quinn. Smart metering and privacy: Existing laws and competing policies. In *SSRN eLibrary*, 2009.

[19] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming apassivedbms into an active dbms. In *VLDB*, pages 469–478, 1991.

[20] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 1992.

[21] K. Teymourian and A. Paschke. Enabling knowledge-based complex event processing. In *EDBT/ICDT Workshops*, 2010.

[22] J. Widom and S. Finklestein. Set-oriented production rules in relational database systems. In *SIGMOD*, pages 259–270, 1990.

[23] Y. Xu and et al. Semantic-based complex event processing in the aal domain. In *ISWC*, 2010.

[24] Y. Simmhan et. al. An informatics approach to demand response optimization in Smart Grids. Technical report, USC, 2011.

[25] Y. Simmhan, Q. Zhou and V. K. Prasanna. Chapter: Semantic Information Integration for Smart Grid Applications. In *Green IT: Technologies and Applications*, 2011.

[26] Q. Zhou, S. Natarajan, Y. Simmhan, and V. Prasanna. Semantic information modeling for emerging applications in smart grid. In *ITNG*, 2012.

[27] Q. Zhou, Y. Simmhan, and V. Prasanna. Towards an inexact semantic complex event processing framework. In *DEBS*, 2011.