On Supporting Kleene Closure over Event Streams

Daniel Gyllstrom, **Jagrati Agrawal**, **Yanlei Diao** and **Neil Immerman**
Department of Computer Science, University of Massachusetts Amherst
{*dpg, jagrati, yanlei, immerman*}*@cs.umass.edu*

***Abstract*— Complex event patterns involving Kleene closure are finding application in a variety of stream environments for tracking and monitoring purposes. In this paper, we propose a compact language, SASE+, that can be used to define a wide variety of Kleene closure patterns, analyze the expressive power of the language, and outline an automata-based implementation for efficient Kleene closure evaluation over event streams.**

I. INTRODUCTION

Complex event processing is a new stream processing paradigm where continuously arriving events are matched against complex patterns and the events used to match each pattern are transformed into new events for output. Of particular interest are *Kleene closure patterns* that can be used to extract from the input stream a finite yet unbounded number of events with a particular property. Such patterns are finding application in a growing number of areas including financial services, RFID-based inventory management, click stream analysis, and electronic health systems.

While Kleene closure has been proposed and well studied for regular expression matching, Kleene closure over event streams has the following features that fundamentally distinguish it from conventional regular expression matching.

Relevant Event Definition: Sophisticated predicates define what events are relevant to Kleene closure. Such predicates may specify constraints on a value of an individual event, on how this value compares to that of a previous event, or how this value compares to an aggregate value computed from a series of earlier events.

Event Selection Strategy: Kleene closure over streams also requires flexibility in deciding how to select relevant events from a stream mixing relevant and irrelevant events. Some queries only intend to select relevant events contiguous in the input, while others want to sift out relevant events from interleaving irrelevant ones.

Termination Criteria: Since the input is an infinite stream and queries can proceed as far as possible by skipping irrelevant events, the termination criteria in this problem also differ from regular expression matching where the input is a finite string and the finite automata for matching have limited flexibility in skipping characters.

Despite recent interest in event processing, Kleene closure over streams remains insufficiently addressed. Event languages for active databases [3], [10] offer temporal operators including variants of Kleene closure, but do not support complex predicates to compare events. As we shall show, such predicates are crucial to Kleene closure patterns. Relational stream systems [2], [4] use windowed joins to specify event patterns. Joins, however, are inherently unable to express Kleene closure, as the number of inputs that may be involved is *a priori* unknown.

In this paper, we present the design, analysis, and implementation of SASE+, an event language that supports Kleene closure over streams. Our first contribution is the design of a compact, rich language that allows patterns to be fully defined regarding the relevant event definition, event selection strategy, and termination criteria. The language further allows complete pattern matches to be output to end applications. Our second contribution is an analysis of the expressive power of SASE+ and its relationship with recent relevant event languages. The results of this analysis are summarized in this paper. Our third contribution is an automata-based approach to efficient Kleene closure evaluation over event streams, which is only briefly outlined in this paper due to space constraints.

II. THE SASE+ EVENT LANGUAGE

In this section, we present the underlying event stream model and introduce the reader to SASE+.

Our proposed language uses the following event model. An event stream is an infinite sequence of events, and each event represents an occurrence of interest at a point in time. An event contains the name of its event type and a set of attribute values. Attributes can take complex data types, which are categorized along two dimensions: the first makes a distinction between *atomic* types, whose values are indivisible, and *sequence* types, whose values consist of a sequence of values; the second distinguishes between *simple* types, which are not defined in terms of other data types, and *composite* types, which are. The event model allows combinations along both dimensions (examples are presented shortly). Each event also has a special time attribute that is set by the event provider to capture the occurrence time of the event. In this work, we assume that events arrive in order of the occurrence time.

A. Overview of the Language

SASE+ is a declarative language for specifying complex event patterns over streams. It extends the SASE language that we proposed previously [9] with Kleene closure. The overall structure of SASE+ is shown in Figure 1(a).

The FROM clause specifies the input stream to a query, which can be the default input stream to the system or the result stream of another query. We survey other constructs of the language through examples below. For ease of composition, these constructs are described by first considering events of simple-atomic data types and then adding additional constructs to handle more complex data types.

(a) Language Structure of SASE+:

```
FROM <input stream>
   PATTERN <pattern structure>
   [WHERE   <pattern matching condition>]
   [WITHIN   <sliding window>]
   [HAVING   <pattern filtering condition>]
   RETURN   <output specification>
```

(b) Query 1:

```
FROM  InputStream
   PATTERN SEQ(Start a, RFID+ b[ ], End c)
   WHERE    skip_till_next_match(a, b[ ] !, c) {
                [loading_dock]
         and    a.session_id = c.session_id
         and    b[i].packaging_level = 'pallet' }
   RETURN  a.session_id, count(b[ ]),
                b[ ].(tag_id, reader_id)
```

(c) Query 2:

```
FROM  InputStream
   PATTERN SEQ(Alert a, Shipment+ b[ ])
   WHERE    skip_till_any_match(a, b[ ]) {
                a.type = 'contaminated'
         and    b[1].from = a.site
         and    b[i].from = b[i-1].to }
   WITHIN    3 hours
   RETURN  a.type, a.site, b[ ].to
```

Fig. 1. Example queries expressed in SASE+.

Query 1 in Figure 1 counts the pallets read by an RFID reader at a loading dock. The PATTERN clause specifies a sequence pattern with three components: the first and the third refer to the events sent by the control system to signal the start and the end of a loading session; the second component addresses *one or more* readings in the session using the ***Kleene plus*** operator '+'. Each component declares a variable to refer to the corresponding event(s), in particular, an array variable ("[ ]") for a Kleene plus component.

The WHERE clause further refines the constraints on the addressed events. (For now, focus on the predicates enclosed in { }.) The first predicate, [loading_dock], requires all the events to refer to the same loading dock. It is called an ***equivalence test*** as in SASE. The second predicate requires the start and end events to match on the session id. The third predicate, used with the Kleene plus, selects only the readings of pallets; it uses the $b[i]$ variable (where $i \geq 1$) to specify the requirement that "every" relevant reading refer to a pallet, hence called as an ***iterator predicate***.

Each match of the pattern consists of a unique sequence of events, stored in a, $b[\,]$, and c. For each match, the RETURN clause creates an event with three attributes. In particular, b[].(tag_id, reader_id) selects the tag_id and reader_id from each pallet reading and converts them into a composite type, resulting in an attribute of the sequence-composite type.

Query 2 shows an example in food supply chain monitoring. It captures an alert for a contaminated site and reports a unique series of infected shipments in each pattern match. A Kleene plus is used to compute each series of shipments, with b[1] referring to the shipment from the origin of contamination, and b[i] referring to each subsequent shipment infected via collocation with the previous one. The WITHIN clause constrains the pattern to a 3 hour period.

The structure of the pattern and associated predicates form the first dimension of the Kleene closure definition, which we call **relevant event definition**.

B. Event Selection Strategies

A second dimension of the Kleene closure definition, **event selection strategy**, addresses how to select the relevant events from an input stream mixing relevant and irrelevant events. SASE+ offers significant flexibility in event selection, hence able to support a wide range of applications.

Strict contiguity. In the most stringent event selection strategy, two selected events must be contiguous in the input stream. This requirement is typical in regular expression matching against strings, DNA sequences, etc.

Partition contiguity. A relaxation of the above is that two selected events do not need to be contiguous; however, if the events are conceptually partitioned based on a condition, the next relevant event must be contiguous to the previous one in the same partition. In SASE+, the equivalence tests, e.g. [loading_dock] in Query 1, are commonly used to form partitions, as they amount to partitioning the stream on the specified attribute and matching the pattern in each partition. Partition contiguity, however, is not flexible enough to support Query 1, where readings of pallets can be mixed with readings of cases and items.

Skip till next match. A further relaxation is to remove the contiguity requirements: all irrelevant events will be skipped until the next relevant event is read. As such, Kleene closure will go as far as possible to select the next relevant event until its termination criteria (explained shortly) are met. Using this strategy, Query 1 can conveniently ignore readings of items and cases. This strategy is important in many real-world scenarios where some events in the input are the "semantic noise" to a particular pattern and should be skipped to enable the evaluation to continue.

Skip till any match. Finally, skip till any match relaxes the previous one by further allowing non-deterministic actions on *relevant* events. Query 2 illustrates this use. Suppose that the last shipment selected by the Kleene plus reaches the location X. When a relevant shipment, e.g. from X to Y, is read from the input stream, skip till any match has two actions: (1) it selects the event in one instance of execution to extend the current series, and (2) it ignores the event in another to preserve the current state of Kleene closure, i.e. location X, so that a later shipment, e.g. from X to Z, enables a different series to be instantiated. This strategy essentially computes the transitive closure on a subset of events (e.g. all infected shipments in three hours).

Event selection strategies can be applied beyond Kleene closure to all the events selected in a pattern match. In SASE+, the strategy used in a query is declared as a function in WHERE that includes the pattern variables as arguments and encloses all the predicates in its body. Further, different strategies can be used for different pattern components, expressed as a series of functions, one for each pattern component over its variable. The interested reader is referred to [1] for more details.

C. Termination Criteria

A third dimension in the Kleene closure definition relates to its **termination criteria**. With contiguity requirements,

Kleene closure terminates when the next event in the input or a particular partition fails to satisfy the relevant predicates. However, when event selection is relaxed to skip till next match or further, Kleene closure does not terminate at this point due to its ability to skip irrelevant events. To ensure expected results while avoiding unnecessary work, SASE+ offers two ways to terminate each Kleene plus in the pattern.

Time constraint. A common type of termination uses a time constraint. For a pattern like SEQ(A+ a[]), a query can use the predicate $a[a.\text{LEN}]$.time - $a[1]$.time $< T$ to specify a time window on the Kleene plus, where $a[a.\text{LEN}]$ refers to the last event selected for each match. If the query has the WITHIN clause, the time window for the entire pattern is also applied to every Kleene plus in the pattern.

Minimal effort. A second type of termination is to allow Kleene closure to perform minimal computation and then break. It is often used as an optimization to save work. For instance, the Kleene plus in Query 1 can stop as soon as the event signaling the end of the session is read; continuing the Kleene plus further will not produce any result. To do so, the forced break operator, '!', is appended to the Kleene plus variable in the event selection strategy declaration. However, caution should be taken when using '!' as it may produce only a subset of the desired results. For example, applying '!' to Query 2 will produce only one series of infected shipments, while the query intends to return all unique series. The semantic correctness of '!' is query-specific and the appropriate decision is left to the discretion of the user.

We briefly remark on other features of SASE+. SASE+ offers a **HAVING** clause that filters each pattern match by applying predicates on the constituent events. The distinction between WHERE and HAVING in SASE+ is analogous to that in SQL. The difference is that HAVING is applied to each pattern match in SASE+ whereas it is applied to each group created by GROUP BY in SQL. SASE+ also allows **negation** to be applied to Kleene closure. Aggregation functions can be applied to WHERE, HAVING, and RETURN clauses. Finally, SASE+ supports two output formats: the default format returns all matches of a pattern, while NON-OVERLAP outputs only one among those that satisfy the same partition condition and overlap in time. Examples of these features are omitted here and are available in [1].

D. Extension for Complex Data Types

SASE+ queries can be composed to detect more complex patterns. To do so, the output stream of one query is named and fed as input to another query. The language presented thus far supports composition if the output events of the first query contain only simple-atomic attributes. As the above queries show, the output events may also contain complex data types such as sequence-composite. To handle such events, SASE+ leverages XML as the event encoding scheme and adopts several features of XQuery for more advanced event processing: The '.' operator for retrieving an attribute (e.g. a.type) from a tuple-based event is replaced by the path operators (e.g. '/' and '//') for hierarchical data. Further, the comparison operators are overloaded with those in XQuery, including general comparison operators (e.g. '=') for existentially qualified comparisons between sequences.

III. Expressibility of SASE+

We summarize results about the expressibility of SASE+ below. We can show that SASE+ has a close correspondence with certain low-level complexity classes. Namely, when we restrict queries to use only (1) strict or partition contiguity, (2) these or skip-till-next-match, or (3) the full language, respectively, we obtain robust subsets of the complexity classes (1) NC^1, (2), DSPACE$[\log n]$, and (3) NSPACE$[\log n]$ [8]. In each case, the subsets include complete problems for the relevant classes.

These results also allow us to show that SASE+ is much more expressive than standard languages such as the regular languages and temporal logic [5]. Further, existing stream languages such as SQL-TS [7] and Cayuga [6] can be mapped to strict sublanguages of SASE+. SQL-TS adds a pattern matching part to SQL to handle Kleene closure. The expressive power of this extension can be shown to be the same as SASE+ without negation and restricted to strict or partition contiguity. We can also show that Cayuga amounts to SASE+ using partition contiguity or skip till next match but without a negation operator.

IV. An Automata-based Approach

We propose a new automaton model, NFA^b , that consists of a nondeterministic finite automaton and a match buffer, to formally define the semantics of SASE+ including all its semantic variations. This model is crucial for understanding the meanings of queries and for producing efficient query plans that faithfully implement them. To generate NFA^b based query plans, we propose compilation techniques that translate SASE+ queries into these plans and optimizations that improve these plans for more efficient evaluation. We further design a runtime system that evaluates these plans over event streams. Our runtime optimizations exploit sharing in both storage and processing of NFA^b evaluation.

Acknowledgments. This work has been supported in part by a gift from Cisco.

References

[1] J. Agrawal, Y. Diao, et al. On supporting kleene closure over event streams. Technical Report 07-03, UMass Amherst, 2007. http://www.cs.umass.edu/~yanlei/sase-plus.pdf.
[2] A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.
[3] S. Chakravarthy, V. Krishnaprasad, et al. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, 606–617, 1994.
[4] S. Chandrasekaran, O. Cooper, A. Deshpande, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
[6] A. J. Demers, J. Gehrke, M. Hong, et al. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.
[7] R. Sadri, C. Zaniolo, et al. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.
[8] H. Vollmer. *Introduction to Circuit Complexity*. Springer, Berlin, 1999.
[9] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.
[10] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE*, pages 392–399, 1999.