SASE+: An Agile Language for Kleene Closure over Event Streams

Yanlei Diao Neil Immerman Daniel Gyllstrom

University of Massachusetts, Amherst
{yanlei, immerman, dpg}@cs.umass.edu

Abstract

In this paper, we present SASE+, a complex event language that supports Kleene closure over event streams, and provide a formal analysis of the expressibility of this language. Complex event patterns involving Kleene closure are finding application in a growing number of stream applications including financial services, RFID-based inventory management, monitoring in healthcare, etc. While Kleene closure has been well studied for regular expression matching, Kleene closure patterns over streams have unique features regarding the event definition, event selection, and termination criteria, which fundamentally distinguish them from patterns studied in conventional problems. This paper addresses Kleene closure in this new context. In particular, we propose a compact language that can be used to define a wide variety of Kleene closure patterns, develop a formal model that precisely describes the semantics and expressive power of the language, and characterize its relationships to standard languages in the literature as well as recent proposals for stream-based event languages.

1 Introduction

Complex event processing over streams is a new processing paradigm where continuously arriving events are matched against complex event 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 and to transform them into output events. Such event patterns are crucial to a variety of emerging stream applications including financial services [8], RFID-based inventory management [27], click stream analysis [24], electronic health record systems [12], etc. In financial services, for example, a brokerage customer might want to retrieve a sequence of stock transaction events representing a period where the price of a particular stock steadily increased and this increase resulted in a dramatic increase in the trading volume of this stock. In an RFID-equipped retail store, the retail manager might be interested in a sequence of shelf reading events that capture unusually fast depletion of a valuable product, possibly resulting from a shoplifting activity.

While Kleene closure was originally proposed for and has been well studied for regular expression matching [14], Kleene closure patterns on event streams have several features that fundamentally distinguish them from patterns used in conventional problems.

Relevant Event Definition. Sophisticated predicates define what events are relevant to Kleene closure. Such predicates might specify constraints on a value in an individual event, on how this value compares to that of a previous event, or how this value compares to an aggregate value of 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 that are contiguous in the input stream, while others may want to sift out relevant events from the interleaving irrelevant events. The latter requires event processing to be able to skip irrelevant events and select non-contiguous relevant events.

Termination Criteria. A third issue is when Kleene closure computation terminates. Given that the input is an infinite event stream and some queries may want to skip irrelevant events to continue as far as possible, the termination criteria in this new problem can also differ significantly from the traditional settings where the input is a finite string and no character in the input is allowed to be skipped.

There has been significant recent interest in event processing in both active databases and relational stream systems. Kleene closure over streams, however, remains insufficiently addressed. Event processing in active databases [7, 11, 10, 4, 21, 19, 30] focuses on temporal operators, including some variants of Kleene closure, that can be used to define event patterns. It does not support predicates that compare different events. As we shall show in this paper, such predicates are crucial to all three dimensions of Kleene closure definition listed above. Relational stream systems [5, 6, 22] offer windowed joins for specifying certain event patterns. However, joins are inherently unable to express Kleene closure, as the number of inputs that may be involved is *a priori* unknown. Several recent studies on event streams support Kleene closure patterns. In particu-

lar, we analyze the expressive power of [24] and [8], and show that they are strictly less expressive than SASE+.

In this paper, we present the design and analysis of SASE+, a complex event language that supports Kleene closure over streams. Our contributions include:

- **A Compact, Rich Language**: SASE+ is designed to be declarative and compact. Despite that, it is a rich language in that it allows patterns to be fully defined along three dimensions, namely, the relevant event definition, event selection strategy, and termination criteria.
- **A Formal Semantic Model**: We offer a formal definition of the semantics of SASE+, which addresses all possible semantic variations along each of the three dimensions. This semantic model is crucial for understanding the meanings of queries and for later producing query plans that faithfully implement those queries.
- **A Formal Analysis of Expressibility**. Based on the formal model, we perform an analysis of the expressive power of SASE+ and characterize its relationships to standard languages, including temporal logic [16] and modal μ-calculus [18], as well as recent proposals for stream-based event languages [24, 8]. In particular, we show that SASE+ and some of its natural sublanguages, as well as the recent stream languages listed above, fit tightly into the standard complexity classes NC^1, DSPACE[$\log n$] and NSPACE[$\log n$].

The remainder of the paper is organized as follows. Section 2 presents the SASE+ language. Section 3 describes its semantic model. Section 4 analyzes the expressive power of SASE+ and related languages. Section 5 discusses an extension of the language. Section 6 covers related work and Section 7 concludes the paper. An appendix includes additional sample queries and a more formal treatment of the syntax and semantics of SASE+.

2 The SASE+ Event Language

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

2.1 Event Stream Models

Input event stream. The input to an event processing system is an infinite sequence of events, referred to as an event stream. Each event in a stream represents an atomic occurrence of interest at an instant in time. Similar to the distinction between types and instances in database systems, our model includes event types that describe a bounded set of attributes that a class of events must contain. Each event consists of the name of its type and a set of values corresponding to the defined attributes.

Each event is assigned a timestamp from a discrete ordered time domain. We assume that such timestamps are assigned by a separate mechanism before events enter a processing system. We also assume that timestamps of events are monotonically increasing, giving rise to a natural total order of events.[1] Timestamps are treated as an implicit attribute of events and can be retrieved (but not modified) in queries.

Output event stream. The output of an event processing system is also a stream of events and each event again contains a bounded set of attributes. The output model is an extension of the input model in that it allows attributes to take complex data types. These data types can be categorized along two dimensions. The first dimension makes a distinction between *atomic* data types, whose values are indivisible, and *sequence* data types, whose values consist of a sequence of values. The second dimension distinguishes between *simple* data types, which are not defined in terms of other data types, and *composite* data types, which are defined by combining other data types. The output model allows all combinations along the two dimensions, namely, atomic-simple, atomic-composite, sequence-simple, sequence-composite (in comparison, the input model currently only supports atomic-simple). Examples of these data types are presented in the next subsection.

It is important to note that our ultimate goal is to make the two models identical, especially, to relax the input model to the more general output model. As such, the query language will satisfy the closure property and queries written in the language can be arbitrarily composed. In this work, we restrict input streams to the simpler model in order to focus on several fundamental issues pertaining to Kleene closure over streams. In scenarios where queries need to be composed, even if the output model is restricted to the case of atomic-simple for the time being, the query language is already quite expressive. A more detailed discussion of this issue is postponed until Section 5 after we present our main techniques and results.

2.2 Overview of the Language

In this section, we present SASE+, a declarative language for specifying complex event patterns over streams. This language significantly extends our previous proposal [29] by supporting Kleene closure patterns. The overall structure of SASE+ is:

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

By default, a SASE+ query reads from the input stream of an event processing stream. The `FROM` clause

[1] Our language, in fact, is powerful enough to handle events that have sufficiently close timestamps to be considered simultaneous. A detailed discussion of this topic is beyond the scope of this paper.

can be used to reset the query input to another stream, which we illustrate later in this section. We next explain the various constructs using examples drawn from the scenarios of stock market analysis and monitoring in transport networks.

```
Query 1:
PATTERN SEQ(NEWS a, STOCK+ b[])
WHERE   a.type = 'bad' ∧ b[i].symbol = 'GOOG'
WITHIN  4 hours
RETURN  sum(b[].volume)
```

Query 1 retrieves the total trading volume of Google stocks in the 4 hour period after some bad news occurred. The `PATTERN` clause declares the structure of a pattern. It uses the `SEQ` construct to specify a sequence pattern of two components: the first refers to an event whose type is NEWS, and the second refers to a series of events of the STOCK type. The latter uses the *Kleene plus*, denoted by "+", to represent one or more events of a particular type. A variable is declared in each component to refer to the corresponding event(s). A component that uses the Kleene plus declares its variable as an array using the "[ ]" symbols.

The `WHERE` clause, if present, contains value-based predicates to define the events relevant to the pattern. In Query 1, the first predicate requires the type attribute of the NEWS event to be bad. The second predicate requires every relevant STOCK event to have the symbol GOOG; the "every" semantics is expressed by $b[i]$ (where $i \geq 1$). We refer to such predicates as *individual iterator* predicates. The `WITHIN` clause specifies a window over the entire pattern, restricting the events considered to those within a 4 hour period.

`PATTERN`, `WHERE`, and `WITHIN` clauses together completely define a pattern. Their evaluation over an event stream results in a stream of pattern matches. Each pattern match consists of a unique sequence of events used to match the pattern, stored in the a and $b[\ ]$ variables. The `RETURN` clause transforms each pattern match into a result event. In its specification, $b[\ ]$ implies an iterator over the events in the array, the volume attribute is retrieved from each event returned by the iterator, and the aggregate function $sum()$ is applied to all the retrieved values. According to the output model, this aggregate function creates an attribute of the atomic-simple type for inclusion in the result event.

```
Query 2:
PATTERN SEQ(STOCK+ a[ ])
WHERE   skip_till_next_match(a[ ]!)
        { [symbol] ∧
          a[1].price = 10 ∧
          a[i].price > a[i-1].price ∧
          a[a.LEN].price = 20 }
WITHIN  1 hour
HAVING  avg(a[].volume) ≥ a[1].volume
RETURN  a[1].symbol, a[].price
```

Query 2 captures a one-hour period in which the price of a particular stock increased from 10 to 20 and the trading volume of the stock stayed relatively stable. The `PATTERN` clause defines a pattern with a single component that applies the Kleene plus to STOCK events. The `WHERE` clause uses a variety of predicates to define what STOCK events can be used to match the pattern. For now, focus on the predicates embraced in "{ }"; other constructs will be explained shortly.

The first predicate, [*symbol*], requires that all relevant STOCK events have the same symbol. Such a predicate is called an *equivalence test* on the pattern [29] and its effect amounts to partitioning the stream on the specified attribute and matching the pattern in each partition. The next three predicates specify additional constraints on a sequence of events that can match the pattern. The predicate on $a[1]$ specifies the start condition on the sequence, i.e., the stock price equal to 10. The next predicate specifies the relationship between each pair of adjacent events in the sequence, i.e., for each position i (where $i > 1$), the price of $a[i]$ exceeds that of $a[i-1]$. Such predicates that compare each event to the previously selected events are referred to as *correlated iterator* predicates. The last predicate on $a[a.\text{LEN}]$ defines the end condition on the sequence, i.e., the price reaching 20.

After `PATTERN`, `WHERE`, and `WITHIN` generate pattern matches, the `HAVING` clause filters each of them by applying predicates on the constituent events. In Query 2, the predicate in `HAVING` requires the average volume of the events in $a[\ ]$ to be no less than that of $a[1]$. A pattern match that satisfies the `HAVING` predicate is retained for output. The distinction between `WHERE` and `HAVING` in SASE+ is analogous to that in SQL. The only difference is that `HAVING` here is applied to each pattern match, while `HAVING` in SQL is applied to each group created by `Group By`.

The `RETURN` clause again translates each pattern match into a result event. It retrieves $a[1].symbol$ and $a[\ ].price$ as two new attributes for inclusion in the result event. Note that $a[\ ].price$ here creates a new attribute of the sequence-simple type.

```
Query 3:
PATTERN SEQ(STOCK+ a[ ], STOCK b)
WHERE   skip_till_next_match(a[ ]!, b)
        { [symbol] ∧
          a[1].volume > 1000 ∧
          a[i].price > avg(a[..i-1].price) ∧
          b.volume < 80% ∗ a[a.LEN].volume }
WITHIN  1 hour
RETURN  a[1].symbol, a[].(price,volume),
        b.(price,volume)
```

Query 3 captures a more complex trend for each stock: in the last hour, the volume started high, but after a period when the price increased or remained relatively stable, the volume plummeted. This query is similar to Query 2 in structure, but with several differences. The `PATTERN` structure has two components, a Kleene plus on STOCK events, whose result is in $a[\ ]$, and a separate single STOCK event, referred to using

b. In `WHERE`, the predicate involving $a[1]$ defines a start condition using the volume attribute. The correlated iterator predicate, expressed using $a[i]$, requires the price of each event to exceed the average of all previously selected events, denoted by $a[..i-1]$. Aggregates used in correlated iterator predicates are called *running aggregates*. While there is no explicit condition on $a[a.\text{LEN}]$, the last predicate in Query 3 compares b to $a[a.\text{LEN}]$ on the volume attribute.

For each pattern match consisting of $a[\ ]$ and b, the `RETURN` clause transforms it into a result event with three attributes. In particular, $a[\ ].(price, volume)$ means that for each event in $a[\ ]$, select the price and volume and convert them into a composite data type as specified by the "( )" symbols. Repeating this for all the events results in an attribute of the sequence-composite type. In comparison, $b.(price, volume)$ creates a new attribute of the atomic-composite type.

We presented the basic constructs of SASE+ above. Of particular importance are the predicates in the `WHERE` clause that define the events relevant to Kleene closure. They form a first dimension of the definition of Kleene closure, which we call *relevant event definition*.

2.2.1 Event Selection Strategies

A second orthogonal dimension of the Kleene closure definition, called *event selection strategy*, addresses how to select the relevant events from an input stream mixing relevant and irrelevant events.

Revisit Query 2. It looks for a particular period of price increase of a stock, as defined by the predicates in `WHERE`. Note that it uses both the equivalence test, $[symbol]$ (whose canonical yet verbose form is $a[i].symbol = a[i-1].symbol$), and the iterator predicate, $a[i].price > a[i-1].price$, to define the condition on the next relevant event $a[i]$. What remains unclear is the relationship between the two relevant events $a[i]$ and $a[i-1]$ in terms of their positions in the input stream. Possible positional relationships include:

Strict Contiguity. The two relevant events must be contiguous to each other in the input stream; that is, there can be no other event in between in the input stream. This requirement is typical in regular expression matching against strings, DNA sequences, etc.

Partition Contiguity. The two relevant events need not be contiguous to each other in the input stream. However, if the events are conceptually partitioned based on a certain condition, the next relevant event must be contiguous to the previous one in the same partition, thus referred to as partition contiguity. For example, to apply this requirement to Query 2, we can use the equivalence test to form a partition for each stock symbol. The query then captures the trend of "monotonic" increase in price in each partition.

The condition used to form partitions can be any boolean combination of predicates that do not use aggregation functions. This is slightly richer than equivalence tests (which are similar to `GROUP BY` in SQL and `PARTITION BY` in CQL [2]).

Skip till next match. In this category, the two relevant events need not be contiguous in a relevant partition. In the absence of contiguity requirements, the selection of the next relevant event can completely ignore irrelevant events and only compare the current event with the previously selected ones. In this way, Kleene closure can go as far as possible until explicit termination criteria (explained shortly) are met.

With this requirement, Query 2 will acquire a different meaning: it looks for an overall trend of price increase from 10 to 20 while ignoring intermediate temporary fluctuating values. The ability to ignore such local fluctuating values proves important to many real-world applications where the emphasis is on broad trends, such as the change in price in the stock market and the rate of depletion of valuable products in retail, as opposed to monotonic behaviors.

Skip till any match. The last category further relaxes the previous requirement with more flexibility in selecting the next relevant event. For each relevant event encountered, it allows non-deterministic decisions between including the event into the Kleene closure and simply skipping it. Such behaviors could be useful in scenarios where skipping some relevant events would prolong the Kleene closure computation, resulting in a longer sequence of events selected. For example, given a sequence of stock prices "1, 2, 7, 3, 4, 5, 6", the longest sequence of events with increasing price is "1, 2, 3, 4, 5, 6". The detection of this result would require the value 7 to be skipped, despite that the value itself is relevant to the Kleene closure.

The above discussion also applies when we consider the relationship between two events selected for adjacent components, e.g. $a[a.\text{LEN}]$ and b in Query 3. A detailed discussion is omitted in the interest of space.

For sufficient support of Kleene closure, SASE+ offers an additional construct in `WHERE` to specify the above requirements as event selection strategies. As shown in Query 2 and Query 3, the strategy in use is specified as a function over the variables declared for the pattern (the "!" symbol is explained shortly); the body of the function includes the original predicates. The rationale behind this design is that different components of a pattern can be associated with different strategies; this design allows it to be expressed as a series of functions, one over each variable referring to the respective component. Queries showing such uses are omitted due to space constraints.

The default event selection strategy in SASE+ is skip till next match. It is assigned to queries without explicit specification, e.g. Query 1. When partition continuity is used, by default the equivalence test(s) present in a query are used to define partitions. A query can customize the partition definition by underlining any predicates in the `WHERE` clause. Again, examples are omitted in the interest of space.

2.2.2 Termination Criteria

A third dimension in the definition of Kleene closure relates to its termination criteria. Given that queries take an infinite stream as input and may allow irrelevant events to be skipped in the Kleene closure computation, the termination of such computation needs to be carefully addressed in order to obtain expected results while avoiding unnecessary work.

Condition on the Last Event Selected. Query 2 illustrates a first type of termination. The predicate on $a[a.\text{LEN}]$ in its `WHERE` clause specifies a constraint on the last event selected by the Kleene closure for each pattern match. The predicate itself does not specify what should be done to the Kleene closure upon detection of such an event, e.g., if it should terminate or continue to produce more matches. Consider the example of stock price change in Figure 1(a). The sequence from Point 1 (where the price is 10) to Point 2 (where the price is 20) produces a pattern match for this query. Since there can be future points whose price equals 20, e.g. Point 3, the Kleene closure can continue to generate more matches. For those queries that are not interested in such additional results, they can force the predicate on $a[a.\text{LEN}]$ to be used as the termination criterion. This is expressed by appending the "!" symbol to the Kleene plus variable in the function for event selection strategy declaration (see Query 2), similar to the "!" cut operator in Prolog.

Next Component of the Pattern. Query 3 demonstrates a second type of termination. This query does not have a predicate constraining $a[a.\text{LEN}]$. However, it has a subsequent b component. Consider the price and volume changes in a series of stock events depicted in Figure 1(a) and 1(b), respectively. The Kleene closure starts at Point 1 (based on the condition $a[1].volume > 1000$) and runs successfully up to Point 2 (by checking $a[i].price > avg(a[..i-1].price)$). The event corresponding to Point 3 satisfies the predicate on b ($b.volume < 80\% * a[a.\text{LEN}].volume$), resulting in a pattern match. The same event, however, can be used to continue the Kleene closure, as it also satisfies the iterator predicate on $a[i].price$; such continuation can lead to future pattern matches, e.g., produced at Point 4. In SASE+, a query can force the Kleene closure to terminate once the next pattern component is matched, by applying "!" as shown in Query 3, or allow it to continue and produce additional matches by omitting "!".[2]

Window Constraints. Another type of termination comes from the window constraint, if present. In addition to any of the above termination criteria used, the window constraint forces the Kleene closure to terminate when a specified amount of time has elapsed.

The relevant event definition, event selection strategy, and termination criteria together completely define Kleene closure patterns over streams.

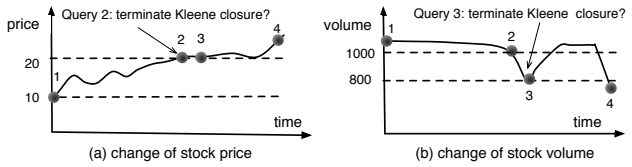


Figure 1: Examples of price and volume change in a sequence of stock events.

2.2.3 Negation on Kleene Closure

SASE+ also allows negation to be applied to Kleene closure. Query 4 shows such an example in the scenario of object tracking in a transport network. Objects were scheduled to be shipped from New York to Amherst and to be scanned at each point in the transport network. The query aims to detect anomalies in the transport process. More specifically, it captures the scenario that for a particular object, there was an initial scan at New York and a final scan at Amherst, but there did not exist (expressed using $\sim$) a series of scans in between that correspond to a normal route of at most 3 hops. In the `WHERE` clause, the equivalence test $[object\_id]$ is applied to all events addressed by the pattern, and the predicates on $b[\ ]$ describe a normal route. Negation evaluates to true if any of the predicates constraining a normal route is violated, e.g. the route not forming a connected path or consisting of more than 3 hops.

For each anomaly detected, the query returns the object id and courier id of the last scan in the result event. In addition, it uses the AS construct to name the type of the result event and each of its attributes, and the IN STREAM construct to name the output stream (for use of the next example query).

```
Query 4:
PATTERN SEQ(SCAN a, ∼(SCAN+ b[]), SCAN c)
WHERE   partition_contiguity(a, b[], c)
        { [object_id] ∧
          a.location = 'New York' ∧
          c.location = 'Amherst' ∧
          b[1].location = a.next ∧
          b[i].location = b[i-1].next ∧
          c.location = b[b.LEN].next ∧
          b.LEN ≤ 3}
RETURN  c.object_id, c.courier_id
        AS ANOMALY(object_id, courier_id)
        IN STREAM Q4-OUTPUT
```

2.2.4 Other Language Features

We remark on other important features of SASE+.

Aggregates. SASE+ supports all usual aggregation functions including count, sum, avg, max, min, and some slighlty less standard ones including highest k and lowest k for any fixed constant, k.[3] All these aggregates

[2]Caution should be taken when applying "!" to the Kleene plus. If a pattern has a few subsequent components whose predicates use the Kleene plus variable, stopping Kleene closure early may fail to produce any match that satisfies those components. The appropriate decision is left to the discretion of the user.

[3]The precise set of aggregation operators included is the set of

can be used in WHERE to select events, in HAVING to filter pattern matches, and in RETURN to return aggregate values.

Union. Similar to SQL, SASE+ also supports UNION to connect multiple query blocks into one query.

Query Composition. Finally, SASE+ queries can be composed by feeding the output of one query as input to another. As noted previously, for the time being, we restrict the output of the first query to the atomic-simple type, exemplified by Query 4. Query 5 below takes the output of Query 4, as specified in the FROM clause, and identifies each courier that is responsible for at least 10 anomalies within a week. It returns the courier id and the complete list of affected objects in each result event.

```
Query 5:
FROM    Q4-OUTPUT
PATTERN SEQ(ANOMALY+ a[])
WHERE   partition_contiguity(a[])
        { [courier_id] }
WITHIN  1 week
HAVING  count(a[]) ≥ 10
RETURN  a[1].courier_id, a[].object_id
```

3 A Formal Semantic Model

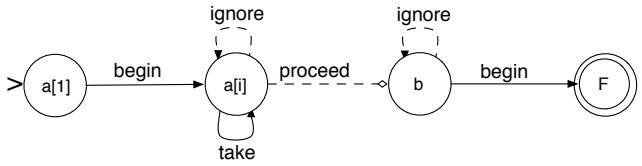
We presented the SASE+ language in the previous section. The semantics of the language is rich, spanning three dimensions in the Kleene closure definition as well as involving negation and composition. In this section, we present a formal model that precisely defines the semantics of SASE+. This formal semantics is crucial for understanding the meanings of queries, and for later producing efficient query plans that faithfully implement the queries.

The semantic model consists of a nondeterministic finite automaton (NFA) combined with a ***match buffer***.[4] We call the combined machine, which is significantly more powerful than a standard NFA, an NFA^b. In this section we explain this model somewhat intuitively, with a more formal presentation in Section A of the appendix.

3.1 NFA^b: A Linear, Modular Structure

Each simple SASE+ query, i.e. one without negation or composition, determines an NFA^b consisting of a linear sequence of states. The NFA^b for Query 3 is illustrated in Figure 2(a).

In this example, the start state, $a[1]$, is where the matching process states. Here, the process awaits input to start the Kleene plus computation and select an event into the $a[1]$ unit of the buffer. At the next $a[i]$ state, the process awaits input to select another event into $a[i]$ (where $i > 1$) in the buffer. The subsequent b state denotes that the matching process has fulfilled



(a) NFA structure

relevant event definition		
θa[1]_begin = a[1].volume>1000	**θa[i]_take** = ¬θa[i]_forced-stop ∧ a[i].symbol=a[i-1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_begin** = b.symbol=a[a.LEN].symbol ∧ b.volume<80%*a[a.LEN].volume
event selection strategy		
	θa[i]_ignore = ¬θa[i]_forced-stop ∧ ¬(a[i].symbol=a[i-1].symbol ∧ a[i].price>avg(a[..i-1].price)	**θb_ignore** = ¬(b.symbol=a[a.LEN].symbol ∧ b.volume<80%*a[a.LEN].volume)
termination criteria		
θa[i]_forced-stop = a[i].time-a[1].time≥W ∨ θb_begin		
θa[i]_auto-stop = ¬θa[i]_take ∧ ¬θa[i]_ignore		
θa[i]_proceed = θb_begin ∨ θa[i]_auto-stop		

(b) Conditions on edges

Figure 2: The NFA^b for Query 3.

the Kleene plus (for a particular match) and is ready to process the next component of the pattern. The final state, F, represents the successful completion of the process, resulting in the creation of a pattern match.

In general, each Kleene plus component of a pattern has a pair of states, the first for the initial selection of an event (i.e., the *select-first* state), and the second for the iterative selection of subsequent events (i.e., the *select-more* state). In comparison, a non-Kleene plus component has a singleton, select-first state.

Each state is associated with a number of edges, representing the possible actions that can be taken at the state. There are four types of edges in total: *begin*, *take*, *ignore*, and *proceed*. As shown in Figure 2, a select-first state usually has two edges: a begin edge that selects the current event into the buffer and moves to the next state, and an ignore edge that skips the event and loops back to the same state. As an exception, the start state does not have an ignore edge because a match must begin with a selected event. A select-more state has three edges: a take edge that selects the current event into the buffer and loops back to the same state, an ignore edge as discussed above, and a proceed edge that transitions to the next state, thus declaring the Kleene plus computation completed.

Each edge can be precisely described as a triplet: a condition on traversing the edge, an operation (i.e., consume or not) on the input event stream, an operation (i.e., write or not) on the match buffer. In Figure 2(a), we use solid lines to represent begin and take edges that consume an event from the input and write it to the buffer, and dashed lines for ignore edges that consume an event but do not write it to the buffer. In this example, the proceed edge does not consume an event from

binary operations that are associative, have an identity element and an NC^1 iterated multiplication algorithm.

[4] A similar but not identical NFA-based model is used in [8].

the input and thus cannot write to the buffer. We distinguish the proceed edge from ignore edges in the style of arrow, denoting that it does not consume any event. The meanings of these edges will become more clear after we explain the conditions on them next.

3.2 Edges: The Logic of Control

In our model, all specifications in the `WHERE` and `WITHIN` clauses, including the relevant event definition, event selection strategy, and termination criteria, are realized by the conditions on edges. Thus, the conditions on edges implement the logic of control.

Relevant Event Definition. Recall that the predicates in the `WHERE` clause define relevant events. They are used to set conditions on begin and take edges. We identify predicates relevant to these edges and set them on the corresponding edges. This is a straightforward process. Consider Query 3. The predicates on the begin and take edges are shown in the first row in Figure 2(b). At the $a[1]$ state, the condition on the begin edge contains the (only) predicate on $a[1]$. We denote the condition as $\theta_{a[1]\_begin}$. At the $a[i]$ state, the condition on the take edge, $\theta_{a[i]\_take}$, contains two iterator predicates on $a[i]$: one for the equivalence test on symbol, and the other for the required change in price. Note that $\theta_{a[i]\_take}$ also has a factor of $\neg\theta_{a[i]\_forced-stop}$. It is used to control the termination of the Kleene plus. For now, treat this factor as a placeholder; its condition is set using the termination criteria, which we discuss shortly. Finally, θ_{b_begin} at the b state is set similarly, using the two relevant predicates, one for the equivalence test, and the other for the change in volume.

Event Selection Strategy. Recall that event selection strategies address how to select relevant events from an input stream mixing relevant and irrelevant events. SASE+ offers four strategies for uses in queries. The strategy specified in each query is used to set all its ignore edges. Our discussion below focuses on select-more states, as they represent a more complex case.

Consider Query 3 and allow its event selection strategy to be varied among the four choices. Table 1 shows how to set the ignore edge at the $a[i]$ state in relation to the take edge. From top-down, it shows $\theta_{a[i]\_take}$ set on the take edge and $\theta_{a[i]\_ignore}$ to be set on the ignore edge. Both the take and ignore edges have the $\neg\theta_{a[i]\_forced-stop}$ factor, which controls the termination of the Kleene plus but otherwise does not affect the discussion below. The remaining factor of the take edge, denoted as $\theta_{a[i]\_select}$, contains all the predicates that define relevant events. The remaining factor of the ignore edge, $\theta_{a[i]\_skip}$, is what remains to be determined.

As the table shows, when strict contiguity is applied, $\theta_{a[i]\_skip}$ is simply set to False. This disallows any event to be skipped, thus ensuring that all relevant events selected are contiguous in input. If partition contiguity is used, one or multiple predicates are used to define each partition. $\theta_{a[i]\_skip}$ is set to the negation of these predicates. In Query 3, $a[i].symbol = a[i-1].symbol$ defines a partition for each symbol. By setting $\theta_{a[i]\_skip}$ to the negation of this predicate, only events irrelevant to a partition can be skipped. To support skip till next match, $\theta_{a[i]\_skip}$ is set to the negation of $\theta_{a[i]\_select}$. This amounts to skipping any event that does not satisfy $\theta_{a[i]\_select}$ and thus cannot be selected. Finally, to support skip till any match, $\theta_{a[i]\_skip}$ is simply set to True, allowing any (including relevant) event to be skipped.

Table 1: Conditions on Ignore Edges for Query 3 with Varied Event Selection Strategies

Take $\theta_{a[i]\_take}$	$\neg\theta_{a[i]\_forced-stop} \wedge \theta_{a[i]\_select}$
	$\theta_{a[i]\_select} =$
	$a[i].symbol = a[i-1].symbol \wedge$ $a[i].price > avg(a[..i-1].price)$
Ignore $\theta_{a[i]\_ignore}$	$\neg\theta_{a[i]\_forced-stop} \wedge \theta_{a[i]\_skip}$
	$\theta_{a[i]\_skip} =$
Group Contiguity:	False
Partition Contiguity:	$\neg(a[i].symbol = a[i-1].symbol)$
Skip till next match:	$\neg(a[i].symbol = a[i-1].symbol \wedge$ $a[i].price > avg(a[..i-1].price)$
Skip till any match:	True

Similar to the $a[i]$ state, the ignore edge at a select-one (e.g. b) state, is set in relation to the begin edge based on the strategy used. For Query 3 using skip till next match, the conditions on ignore edges are shown in the second row in Figure 2(b).

Termination Criteria. We next consider issues related to the termination of the Kleene plus.

A first task is to use query-specified termination criteria to set $\theta_{forced-stop}$ at the corresponding state. Recall that there can be three types of criteria. A simple type is the window constraint. Accordingly, $\theta_{forced-stop}$ has a term that requires the Kleene plus to terminate if the specified amount of time has elapsed since the first selected event. When the condition on the last event selected for each pattern match is used as a stopping criterion (e.g. Query 2), this condition is included in $\theta_{forced-stop}$ as an additional term. In the absence of such a condition, if a query decides to terminate the Kleene plus when an event matches the next pattern component (e.g. Query 3), the condition on the next component should be added to $\theta_{forced-stop}$.

A second task is to set the condition on a proceed edge that departs from a select-more state. A transition along this edge can occur before as well as when the Kleene plus terminates, discussed as follows. In a simpler case, the query imposes a condition on the last event selected by the Kleene plus in each pattern match, e.g. Query 2. Then, no matter what termination criteria are used, the proceed edge is set using this condition. A more complex case relates to queries that do not have such a condition, e.g. Query 3. Then, the proceed edge can have two terms in its formula.

The first term contains the predicates on the next pattern component, if present. It means that if the cur-

rent event can match the next component, the select-more state may have collected enough events for a potential pattern match, so it is worth an attempt to move to the next state and make progress there. This transition can occur before the Kleene plus terminates.

The idea behind the second term is that if the Kleene plus has come to a stop, we can also make an opportunistic move to the next state and let that state decide what can be done next. For a complete description of termination, however, we need to introduce another concept called auto-stop, defined as $\theta_{auto-stop} = \neg\theta_{take} \wedge \neg\theta_{ignore}$. While $\theta_{forced-stop}$ encodes the query-specified stopping condition, $\theta_{auto-stop}$ captures a complete set of stopping conditions in execution. As both θ_{take} and θ_{ignore} contain $\neg\theta_{forced-stop}$ as a factor, $\theta_{auto-stop}$ obviously subsumes $\theta_{forced-stop}$. It may include additional conditions, however. Consider Query 3 using partition contiguity. If an event belongs to a partition but cannot be selected, the evaluation of $\theta_{a[i]\_take}$ and $\theta_{a[i]\_ignore}$ will both fail, causing the Kleene plus to come to a stop. This shows that true stopping conditions are determined by both the query specified termination criteria and the event selection strategy. $\theta_{auto-stop}$ captures both such conditions.

The definition of the proceed edge for Query 3 is shown in the last row of Figure 2(b). It is worth noting that this edge does not consume any event from the input (i.e. it is an ϵ edge). In its evaluation, however, it needs to peek at the current event but keep it in the input for the next state. We introduce this non-traditional ϵ edge to achieve a modular design: the proceed edge performs a single function—to depart from the select-more state—and leaves any decision about the current event regarding the next component (i.e., to take or to ignore) to the next state.

In summary, the discussion above reflects the compilation rules that we offer for automatically translating a simple SASE+ query, i.e. one without negation or composition, to a presentation in the formal semantic model. This representation can be later used as a query plan for execution. For ease of exposition, we omitted the discussion about how to place the window constraint onto all take and begin edges; this can be added in a straightforward way.

4 Expressibility of SASE+

In this section, we restrict ourselves to SASE+ queries that return atomic-simple values. We study the expressive power of SASE+ and related languages. As we will see, the SASE+ query language can be naturally mapped into large subsets of the complexity classes NC^1, DSPACE[$\log n$], and NSPACE[$\log n$], when we restrict queries to uses of only strict or partition contiguity, only skip_till_next_match, or the full language, respectively. For background on complexity results and classes see [26].

An extensive intuitive introduction to SASE+ has been given, and a more formal treatment of SASE+ and the related NFA^b model can be found in Section A of the appendix. We prove there that a simple SASE+ query, i.e, without composition or negation, can be compiled into an NFA^b automaton, and thus any SASE+ query is in cl($\mathrm{NFA}^b, \circ, \sim$), i.e., the closure of NFA^b languages under composition and negation (Lemma A.1).

The subtlety of characterizing the expressive power of SASE+ has to do with the interaction of Kleene + and aggregation. To get started, in our first theorem we simply remove aggregation from consideration.

Let SASE+(w.o. aggregation) and NFA^b(w.o. aggregation) be the restriction of these two models to have no occurrences of aggregation operators. In this case we can think of the input alphabet, $\Sigma = D_1 \times \cdots D_k$ as the product of the domains of possible attribute values in the event stream. It is not surprising that without aggregation we are limited to the regular sets:

Theorem 4.1 *Let $A \subseteq \Sigma^\star$. The following conditions are equivalent:*

1. *A is regular*
2. *A is recognizable by a SASE+(w.o. aggregation,$\sim$) query*
3. *A is recognizable by a SASE+(w.o. aggregation) query*
4. *A is in cl(NFA^b(w.o. aggregation),$\circ$)*
5. *A is in cl(NFA^b(w.o. aggregation),,$\circ, \sim$)*

Proof: Since SASE+(w.o. aggregation,$\sim$) contains single letter alphabets and is closed under concatenation, union, and Kleene +, it contains the regular languages. We further know that $2 \to 3 \to 5$ and $2 \to 4 \to 5$. Finally, since an NFA^b(w.o. aggregation) automaton is a special kind of NFA, $5 \to 1$. □

We note that temporal logic is equivalent to first-order logic and thus the star-free regular languages on words [16, 20]. Thus, SASE+, even without aggregation or negation is more powerful than temporal logic.

In the presence of aggegation, we can express non-regular properties, e.g., a simple, strictly_contiguous SASE+ query can read a consecutive sequence of letters having the property that it contains more a's than b's. In fact, we show in Theorem 4.3 that the SASE+ with only contiguous queries expresses a rich subset of the complexity class NC^1. First we show,

Lemma 4.2 *The word problem for S_5 – an NC^1-complete problem – is expressible in a simple SASE+ query of the form strict_contiguity.*

Proof: The word problem for S_5 can be represented as a simple strict-contiguity a+ query: we define an aggregate that keeps track of a value, v, from 1 to 5 and combines that with the input π, an element of the fixed, finite alphabet, S_5 and computes the next value, $\pi(v)$. The beginning and ending condition of the SASE+ query is that $v = 1$. □

Theorem 4.3 *SASE+ with only strict contiguity or with only strict and partition contiguity expresses a subset of* NC^1 *that includes complete problems for* NC^1.

Proof: The NC^1 completeness comes from Lemma 4.2. For containment in NC^1: the SASE+ with partition contiguity query can be simulated in NC^1 as follows: first replace any input from an event not in the partition by the identity element for the aggregation operation in question. Then do a partial-prefix computation of the aggregation operation. □

The language SQL-TS described in [24] provides a stream-processing addition to SQL. Just looking at that stream processing facility, it is not hard to see that the expressive power – assuming the same set of aggregate functions – is the same as SASE+ without negation and restricted to uses of strict or partition contiguity. It thus follows that this stream language is restricted to at most the same subset of NC^1.

The ordered graph reachability problem, oREACH, consists of the set of directed graphs on vertices numbered 1 to n such that there is a path from 1 to n and all edges (i, j) are increasing, i.e., $i < j$. It is well known that oREACH is complete for NSPACE[$\log n$]. Similarly, oREACH_d, the restriction of oREACH in which there is at most one edge from each vertex is complete for DSPACE[$\log n$].

It is not hard to see that

Lemma 4.4 *$oREACH_d$ is expressible in a simple SASE+ query of the form skip_till_next_match. Similarly, $REACH_d$ is expressible in a simple SASE+ query of the form skip_till_any_match.*

Proof: In both cases the input stream consists of a sequence of edge events with attributes head and tail. A simple a+ query checking that a[i].tail = a[$i-1$].head finds the path. In the deterministic case this is of the form skip_till_next_match because there is at most one edge with a given tail, but in the general case this is a skip_till_any_match because nondeterminism is involved in finding the right path. □

Theorem 4.5 *SASE+(without skip-till-any-match) expresses a subset of the* DSPACE[$\log n$] *queries including some that are complete for* DSPACE[$\log n$].

Proof: The DSPACE[$\log n$] completeness comes from Lemma 4.4. The only subtlety about containment in DSPACE[$\log n$] comes with the possible nondeterminism between (Ignore or Take) versus Proceed. Since there are only a bounded number of places where this nondeterminism can occur in any SASE+ query, we remain in logspace by sequentialy trying each possible choice. This involves adding a $\log n$-bit counter for each of the states of the NFA where such a non-deterministic move could occur. □

The Cayuga system described in [8] is built from an algebraic stream processing language. A least-fixed-point operator is described to express Kleene +, and the semantics of simple, i.e., not composed, queries is given via an automaton model quite similar to our NFA^b. It is not hard to see that in fact that the expressive power – assuming the same set of aggregate functions – is the same as SASE+ without negation and restricted to skip_til_next_match queries. It thus follows Cayuga is restricted to at most the same subset of DSPACE[$\log n$].

Finally, for full SASE+ we have,

Theorem 4.6 *SASE+ expresses a subset of* NSPACE[$\log n$] *including some queries that are complete for* NSPACE[$\log n$].

Proof: The NSPACE[$\log n$] completeness comes from Lemma 4.4. Containment is obvious. □

Theorem 4.6 gives an upper bound on the expressive power of SASE+. It is contained in FO(TC) – first-order logic with a transitive closure operator – and it is not as rich as the modal μ-calculus which can express polynmial-time complete languages [15, 18].

The above results give an interesting view of the expressiblity of sublanguges of SASE+ as well as of SQL-TS and Cayuga. We feel that the above theorems give a good initial picture of the expressiveness of SASE+. However, many questions remain concerning the complexity of evaluating SASE+ queries. On the theoretical side, there is a strong connection to the complexity of branching programs that may only read their input a bounded number of times, cf. [28]. In summary, understanding stream languages means understanding the interaction between Kleene plus and aggregation.

5 Discussion on Query Composition

In this section, we revisit the event stream models for query input and output and discuss their impact on query composition. Recall from Section 2.1 that the difference between the current input and output models is that an input event can only contain attributes of the atomic-simple type, while an output event can take complex types such as atomic-composite, sequence-simple, and sequence-composite. The discrepancy between the two models is a potential barrier to query composition. In our previous discussion, in order to compose two queries, we restrict the output of the first query to the case of atomic-simple. Below, we discuss a general solution to query composition.

A generic solution is to relax the input query model to the more general output model. To this end, several issues need to be examined. The first issue is event encoding. As attributes of an event can now be complex objects such as a sequence of atomic or composite values, XML appears to be a natural solution to encoding events comprised of such complex objects.

Given an XML-encoded event stream, a second issue is how to extend SASE+ to operate on events in the new

format. We expect the majority of the language constructs to remain unchanged, as they address relationships between events. It seems a promising approach to obtain the necessary changes by leveraging standard XML query languages such as XQuery [3]. One such change, for example, would be to replace the current "." operator for retrieving the value of an attribute (e.g. a.price) with the path operators (e.g. "/" and "//") in XQuery. Another change would be to overload comparators, e.g., "=" and "<", with existential qualification, when they operate on sequence-valued attributes. As can be seen, an important benefit of leveraging XQuery is that the necessary extension to SASE+ will have well-defined semantics.

We also want to be cautious when adapting constructs from XQuery, as our goal is to keep SASE+ as compact as possible, yet expressive enough to handle events containing complex objects as defined in the output model. It will be an interesting study to exploit known results pertaining to XQuery in the literature [17] to identify a minimal subset of XQuery necessary for the extension of SASE+.

6 Related Work

Much related work has been covered in previous sections. We discuss broader areas of related work below.

Traditional publish/subscribe systems [1, 9] offer predicate-based filtering of individual events. Our system significantly extends them with the ability to process complex event patterns across multiple events. A mentioned in Section 4, Cayuga [8] offers a complex algebra (but no query language) for expressing Kleene closure patterns that amount to those in SASE+ using partition contiguity or skip till next match, but not negation.

Research on sequence databases [25, 24] offers SQL extensions and efficient implementations for sequence data processing. Like relational stream systems, SEQIN [25] uses joins to specify sequence operations and thus cannot express Kleene closure. SQL-TS [24] adds new constructs to SQL to support Kleene closure over partitioned sequence data, but can only match such patterns with contiguous tuples in each partition.

The recent event systems [13, 27, 23, 12] offer simple event languages and consider stream-based processing. None of these systems support Kleene closure or offer a former definition of their language semantics. In contrast, SASE+ offers both a rich language and a formal model for describing its semantics and characterizing its expressibility.

7 Conclusions

In this paper, we described SASE+, a compact yet expressive language that can be used to define a wide variety of Kleene closure patterns, and we rigorously defined its semantics. We analyzed the expressive power of SASE+ and its natural sublanguages as well as some related languages including SQL-TS and Cayuga. We showed that these languages fit neatly into the standard complexity classes NC^1, DSPACE[$\log n$] and NSPACE[$\log n$].

We are currently developing an efficient implementation of SASE+ as well as working on various extensions, including the addition of the sequence-composite data type to the query input model.

We are interested in the theoretical tradeoff between the expressive power of SASE+ and the complexity of evaluating SASE+ queries. More practically, we believe that many optimizations are available to evaluate typical SASE+ queries very quickly.

There are many deep theoretical issues concerning stream languages, which are quite different from those in tranditional languages. One approach to addressing these issues is to more closely relate this new problem to that of read k times branching programs.

In implementation, many issues need to be considered, including the design and analysis of algorithms to efficiently implement match buffers and eliminate unnecessary work. To address these issues, significant research effort is underway.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[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. Boag, D. Chamberlin, M. F. Fernndez, D. Florescu, J. Robie, and J. Simon. XQuery 1.0: An XML query language, 2006.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, 1994.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[6] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[7] U. Dayal, B. T. Blaustein, A. P. Buchmann, U. S. Chakravarthy, M. Hsu, R. Ledin, D. R. McCarthy, A. Rosenthal, S. K. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HiPAC project: Combining active databases and timing constraints. *SIGMOD Record*, 17(1):51–70, 1988.

[8] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[9] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.

[10] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Rules in Database Systems*, pages 23–39, 1993.

[11] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *VLDB*, pages 327–338, 1992.

[12] L. Harada and Y. Hotta. Order checking in a cpoe using event analyzer. In *CIKM*, pages 549–555, 2005.

[13] A. Hinze. Efficient filtering of composite events. In *BNCOD*, pages 207–225, 2003.

[14] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[15] N. Immerman. *Descriptive complexity*. Springer, 1999.

[16] J. A. Kamp. *Tense logic and the theory of linear order*. PhD thesis, University of California, Los Angeles, 1968.

[17] C. Koch. On the complexity of nonrecursive xquery and functional query languages on complex values. In *PODS*, pages 84–97, 2005.

[18] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[19] D. F. Lieuwen, N. H. Gehani, and R. M. Arlein. The Ode active database: Trigger semantics and implementation. In S. Y. W. Su, editor, *ICDE*, pages 412–420, 1996.

[20] R. McNaughton and S. A. Papert. *Counter-free automata*. MIT Press, Cambridge, MA, 1971.

[21] R. Meo, G. Psaila, and S. Ceri. Composite events in chimera. In *EDBT*, pages 56–76, 1996.

[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[23] S. Rizvi, S. R. Jeffery, S. Krishnamurthy, M. J. Franklin, N. Burkhart, A. Edakkunni, and L. Liang. Events on the edge. In *SIGMOD*, pages 885–887, 2005.

[24] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. Database Syst.*, 29(2):282–318, 2004.

[25] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *VLDB*, pages 99–110, 1996.

[26] H. Vollmer. *Introduction to Circuit Complexity*. Springer, Berlin, 1999.

[27] F. Wang and P. Liu. Temporal management of rfid data. In *VLDB*, pages 1128–1139, 2005.

[28] I. Wegener. *Branching programs and binary decision diagrams: theory and applications*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[29] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, 2006.

[30] D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE*, pages 392–399, 1999.

A Formal Syntax and Semantics of SASE+

In this section as in Section 4, we restrict ourselves to SASE+ queries that return atomic-simple values. In the main text we gave a detailed intuitve treatment of SASE+. All that needs to be added here are some details in order to explain precisely

1. Exactly which formulas may occur in SASE+.
2. Exactly what constitutes an NFA^b automaton.
3. A formal definition for the strings accepted by an NFA^b automaton, and finally,
4. The semantics of a SASE+ query via the automaton model.

A.1 SASE+ Syntax

Recall that the overall structure of SASE+ is:

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

The pattern structure of a simple SASE+ query is of the form

`SEQ(([~]<qitem>)`$^+$`)` where

`<qitem> :  <ename> <let> | <ename>+<let>[]`

A SASE+ query is a composition of simple SASE+ queries – where we have to name a new stream as the output of one query to use in the next one, as shown by Query 4 and Query 5 in Section **??**.

An **identifier** in a pattern matching or a filtering condition consists of any of the following:

- a, where `<ename> a` is a qitem occuring in the pattern structure
- a[1], a[i], a[i-1], or a[a.len], where `<ename>+ a[]` is a qitem occuring in the pattern structure.

Inductively, a **term** is any of the following:

- c, a constant,
- p.att where p is an identifier and att is an attribute,
- agg(a[].att), or agg(a[..i-1]) where agg is any associative aggregation operation with an identity element and an NC^1 iterated multiplication algorithm.
- $(t_1 \circ t_2)$ where $\circ$ is one of $+, \times, -$, and t_1, t_2 are terms.

An **atomic formula** (predicate) is an expression t_1 `relop` t_2 where `relop` is one of $=, \neq, <, \leq, >, \geq$, and t_1, t_2 are terms. Finally a **formula** is a boolean combination of atomic formulas. While any formula can be compiled into a SASE+ query, in order to avoid misleading queries and unintended consequences, we only allow good formulas in pattern matching conditions. A **good formula** is a formula in CNF such that for each clause, C, the last identifier in the SEQ order, ℓ, occurs in every predicate in C. Thus in a pattern matching condition, C is taken as a selection criterion on ℓ.

A pattern filtering condition is a formula. A pattern matching condition is a good formula preceded by an optional header: one of `strict_contiguity`, `partition_contiguity`, `skip_til_next_match`, or, `skip_til_any_match`.

A.2 NFA^b Automaton

An **NFA^b Automaton**, $A = (Q, E, \theta, q_1, F)$, consists of a set of states, Q, a set of directed edges, E, and a set of formula, θ, labelling those edges. Q is arranged as a linear sequence consisting of any number of occurrences of singleton states, s, or pairs of states, $p[1], p[i]$, plus a rightmost final state, F. See Figure 2 for an example.

Each state, q, has a self-loop labelled with the formula θ_{q_ignore}. Furthermore, each state, q that is a singleton state, s, or the first state, $p[1]$, of a pair has a forward edge labelled with the formula θ_{q_begin}. Finally each second state, $p[i]$, of a pair has a forward edge labelled with the formula $\theta_{p[i]\_\text{proceed}}$. and a self-loop labelled with the formula $\theta_{p[i]\_\text{take}}$. The first state, q_1, has no edges to it, i.e., the corresponding Ignore self-loop is removed. This is because we are only interested in matches that start with the first event they take rather than skipping some initial elements.

The formulas that may occur as edge labels are exactly the formulas as defined in Section A.1, but the only identifiers that may occur on an edge from state q are those that have occurred from q_1 up to and including q, with the exception that a.len and agg(a[].att), may not occur on Ignore or Take edges from state a; they make occur on the Proceed edge.

NFA^b automata may exhibit non-determinism whenever from some state there are two edges whose labels are not-mutually exclusive. For example, if $\theta_{p[i]\_\text{take}}$ and $\theta_{p[i]\_\text{ignore}}$ are not mutually exclusive then we are in a nondeterministic skip-til-any-match situation.

Let $A = (Q, E, \theta, q_1, F)$, be an NFA^b automaton and let $E = e_1, e_2, \ldots, e_n$ be an event stream. A **run** of S on E is a sequence of pairs: $\rho = (j_1, t_1), (j_2, t_2), \ldots (j_k, t_k)$ such that $1 \leq j_1 < j_2 < \cdots < j_k \leq n+1$, and inductively the following conditions apply:

- (**base case:**) If $1 \leq j_1 \leq n$, then (j_1, q_1) is an initial run of A on E.

  (Intuitively, (j_1, q_1) means that we are in the initial state about to look at e_{j_1}.)

- (**begin move**:) If $\rho' = (j_1, t_1), (j_2, t_2), \ldots (j_k - 1, t_{k-1})$, is a run of A on E, t_{k-1} is an identifier, s or $a[1]$, and $\theta_{t_k\_\text{begin}}$ holds, with identifier t_{k-1} instantiated as e_{j_k-1}, and t_k is the successor of t_{k-1}, then ρ is a run of A on E.

  (For example, if $\rho' = (3, a[1])$, and $\theta_{t_k\_\text{begin}}$ holds with $i = 1$ and $a[1]$ instantiated as e_3, then $\rho = (3, a[1]), (4, a[2])$ is a run, in state $a[i]$ about to look at e_4.)

- (**ignore move**:) If $\rho' = (j_1, t_1), (j_2, t_2), \ldots (j_{k-1}, t_{k-1}), (j_k - 1, t_k)$ is a run of A on E and $\theta_{t_k\_\text{ignore}}$ holds, with identifier t_k (temporarily) instantiated as e_{j_k-1}, then ρ is a run of A on E.

  (Inductively we were in state t_k, about to look at e_{j_k-1}, then we looked at it and ignored it, so now we are still in t_k about to look at e_{j_k}.)

- (**take move**:) If $\rho' = (j_1, t_1), \ldots (j_{v_1}, a[1]), \ldots, (j_{v_n}, a[n])$, is a run of A on E, and $\theta_{a[i]\_\text{take}}$ holds, with $a[n]$ instantiated as e_{v_n}, $i = n$, then $\rho = \rho', (j_{v_n} + 1, a[n + 1])$ is a run of A on E.

- (**proceed move**:) If $\rho' = (j_1, t_1), \ldots (j_{v_1}, a[1]), \ldots, (j_{v_n}, a[n])$, is a run of A on E, and $\theta_{a[i]\_\text{proceed}}$ holds, with the successor, t_k of $a[i]$ instantiated as e_{v_n}, and $a.len = n$, then $\rho = (j_1, t_1), \ldots (j_{v_1}, a[1]), \ldots, (j_{v_n}, t_k)$ is a run of A on E.

  (Note that a Proceed edge is a "peeking edge" of the NFA^b: it looks at the next input event, e_{v_n}, but doesn't consume it – it will at the next step.)

An **accepting run** is a run, ρ with $t_k = F$. The **match** occurs from position j_1 through j_{k-1} in E, and the output of the match is the instantiation of the run that we have inductively defined. Finally, the **language accepted** by an NFA^b automaton, A, is defined as,

$$\mathcal{L}(A) = \{E = e_1 \cdots e_n \mid A \text{ has an accepting run, } (1, q_1), \ldots (n{+}1, F), \text{ on } E\}$$

We end by connecting the language SASE+ with the NFA^b model. In Section 3.2 we explained in detail how to compile a simple SASE+ query into an NFA^b which faithfully captures its meaning. Thus we have,

Lemma A.1 *Any simple SASE+ query can be compiled into an equivalent NFA^b automaton. Thus, the languages expressed in SASE+ are contained in the closure under composition and negation of the languages recognizable by NFA^b automata.*