Runtime Semantic Query Optimization for Event Stream Processing

Lueling Ding¹, Songting Chen², Elke A. Rundensteiner¹, Junichi Tatemura²
Wang-Pin Hsiung², K. Selçuk Candan²

¹Worcester Polytechnic Institute
{lisingd, rundenst}@cs.wpi.edu
²NEC Laboratories America
(songting, tatemura, whsiung, candan)@sv.neclabs.com

Abstract—Detecting complex patterns in event streams, i.e., complex event processing (CEP), has become increasingly important for modern enterprises to react quickly to critical situations. In many practical cases business events are generated based on pre-defined business logics. Hence constraints, such as occurrence and order constraints, often hold among events. Reasoning using these known constraints enables us to predict the non-occurrences of certain future events, thereby helping us to identify and then terminate the long running query processes that are guaranteed to not lead to successful matches.

In this work, we focus on exploiting event constraints to optimize CEP over large volumes of business transaction streams. Since the optimization opportunities arise at runtime, we develop a runtime query unsatisfiability (RunSAT) checking technique that detects optimal points for terminating query evaluation. To assure efficiency of RunSAT checking, we propose mechanisms to precompute the query failure conditions to be checked at runtime. This guarantees a constant-time RunSAT reasoning cost, making our technique highly scalable. We realize our optimal query termination strategies by augmenting the query with Event-Condition-Action rules encoding the pre-computed failure conditions. This results in an event processing solution compatible with state-of-the-art CEP architectures. Extensive experimental results demonstrate that significant performance gains are achieved, while the optimization overhead is small.

I. INTRODUCTION

A. Constraint-Aware Event Stream Processing

As automated business processes, such as Web services and online transactions [11], [19], [24], become ubiquitous, unprecedented volumes of business events are continuously generated and recorded as event streams. Complex Event Processing (CEP), which aims to detect interesting event patterns in event streams, is gaining adoption by enterprises for quick detection and reaction to critical business situations. Common CEP applications include business activity monitoring, supply chain management, and anomaly detection. Major database vendors have recently taken significant efforts in building event-driven architectures [3], [9].

The event patterns in CEP specify complex temporal and logical relationships among events. Consider the example event pattern EPI below, in which “SEQ” represents the temporal relationship between two events and [totalPrice > 200] is the predicate on the GenerateQuote event. This pattern monitors the cancelled orders that involve the participation of both suppliers and remote stocks, with quote’s price > $200. Frequent occurrences of such patterns may indicate, e.g., the need for an immediate inventory management.

Event Pattern EPI:
SEQ[(OrderFromSupplier, GenerateQuote[totalPrice > 200])
AND SEQ[(UseRemoteStock, GenerateInvoice), CancelOrder)]

State-of-the-art CEP systems employ automata for event pattern matching [10], [31]. When there are large numbers of concurrent business processes, many partial query matches may be kept in automata states. Events arriving later need to be evaluated against all these partial matches to produce query results. Also, event streams tend to be high-speed and potentially infinite. To provide real-time responses, as often required by applications to take prompt actions, serious challenges in CPU/memory utilizations are faced by CEP.

In this paper, we target an important class of event queries, namely alert queries [31]. Alert queries correspond to key tasks in business activity monitoring, including detection of shoplifting, or large/suspicious financial transactions, or other undue business actions like orders cancelled for certain reasons (see example above). These queries detect exceptional cases to the normal business flows and are thus expected to be highly selective. Keeping large numbers of partial matches that do not lead to any query results can cause a major drain on available system resources.

We observe that in practice, many business events do not occur randomly. Instead they follow pre-defined business logic or rules, such as a workflow model [19]. Below we list a number of such CEP applications.

1) Business activity monitoring: an online retailer may want to detect the anomalies from its order processing transactions. In this case, the events are generated from a BPEL workflow engine [4], a business rule engine [5] or simply a customized program.
2) Manufacturing monitoring: a manufacturer may want to monitor its stream-line production process [21]. The process events correspond to pre-defined procedures.
3) ClickStream analysis: a shopping website may want to monitor the click stream [11] to discover the user navigation pattern. Here the user click events depend on how the website is structured.

As consequence, various constraints may exist among events in these CEP applications. In particular, occurrence con-
straints, such as mutually exclusive events, and order constraints, such as one event must occur prior to the other event, can be observed in all the applications listed above. A recent survey [13] shows that the majority of the software design patterns exhibit such constraints as well.

The availability of these constraints enables us to predict the non-occurrences of future events from the observed events. Such predictions would help identify which partial query matches will not lead to final results. Further efforts in maintaining and evaluating these partial matches can be prevented. Example 1 below illustrates such optimization opportunities that remain unexplored in the literature.

Example 1: Assume the event stream is generated by the online order transactions [24], [29] that follow the workflow in Figure 1. We assume each task in the workflow, if performed, will submit an event to the event stream. We can see that both occurrence and order constraints can be inferred from this workflow. For example, the UseLocalStock and the UseRemoteStock events are mutually exclusive. Also, any GenerateQuote event, if it occurs, must be before the SendQuote event in a transaction.

Consider the example event pattern EPI again. By exploiting the event constraints, whenever a UseLocalStock event occurs, this transaction is guaranteed to not match the query because the UseRemoteStock event will never occur in this transaction. Also, once a SendQuote event is seen in a transaction, and no GenerateQuote event with totalPrice > 200 has been observed so far, the transaction will not match the query because no GenerateQuote event will happen after the SendQuote event. In either case, any partial matches by these transactions need not be maintained and evaluated further as they are guaranteed to never lead to a final result. If the query processing of large numbers of transactions could be terminated early, a significant amount of CPU and memory resources would be saved.

Several observations can be made from the above example. First, although the event constraints are known at query compilation time, the real optimization opportunities only emerge at runtime, based on the partial workflow executed so far (i.e., what events have been observed). For example, although the UseLocalStock and the UseRemoteStock events are known to be exclusive, only when one of them occurs, can we infer that the other one will not be seen in the same transaction. Second, both occurrence and order constraints can be exploited to short-cut query execution.

B. Our Approach

Several key challenges must be tackled to exploit constraints for CEP. One critical question is how to identify unsatisfiable partial query matches at runtime. In addition, there may be thousands or even millions of concurrent business processes. To assure the efficiency and scalability, the runtime reasoning for each individual transaction must be lightweight. Otherwise, the overhead of constraint reasoning may outweigh its benefits. In this paper, we propose the first general framework to address the above challenges for constraint-aware CEP (C-CEP). The main contributions are summarized below:

1. We propose a polynomial time, sound and complete runtime query unsatisfiability (RunSAT) checking algorithm for detecting the unsatisfiable query matches. This algorithm is based on a formal logic reasoning considering the event query, the partial event history and the event constraints such as workflows (Section III).

2. To improve the RunSAT performance, we propose a general pre-processing mechanism (based on abductive inference [14], [15]) to pre-compute query failure conditions. Further, we identify a set of simple yet common event constraints that allow constant time RunSAT (Section IV).

3. We propose to realize the above techniques based on augmenting event queries with pre-computed failure conditions. This facilitates the integration of our techniques into state-of-the-art CEP architectures [10], [31] (Section V).

4. Our experimental study demonstrates that significant performance gains, i.e., memory savings up to a factor of 3.5 and CPU savings at a factor of 2, are achieved through our approach, with a very small almost negligible overhead for optimization itself (Section VI).

II. PRELIMINARIES

Event Model. An event (or event instance), denoted as the lower-case letter \( e_i \), is defined to be an instantaneous, atomic (happens completely or not at all) occurrence of interest. An event type, denoted as the corresponding upper-case letter \( E_i \), defines the properties that all the event instances \( e_i \) must have.
The properties of an event instance \( e_i \) include a set of attributes \( e_i.A_1, ..., e_i.A_n \), and a timestamp \( e_i.t \) of its occurrence.

The input to the CEP system is a stream of events ("event history") ordered by their timestamps \(^1\). We assume that the event history can be partitioned into multiple sub-sequences based on certain criteria, such as transactions ids, session ids, RFIDs, etc. In the rest of this paper, we call each partition of the event history a trace \( h \).

**Event Constraints.** Software and workflow models exhibit certain order and occurrence constraints (Section I), CEP queries also need to capture these occurrence and order between events (defined later). These constraints can be expressed using a subset of a general event language \( \mathcal{L} \).

**Definition 2.1:** An event language \( \mathcal{L} \) contains a set of event types \( E_i \), denoted as \( \mathcal{E} \), a variable \( h \) denoting the event history, a binary function \( < \), logic connectives (\( \land, \lor, \neg, \rightarrow \)), quantifiers (\( \exists \) and \( \forall \)). A formula of \( \mathcal{L} \) is either:

1. \( E_i[h] \) iff an event instance \( e_i \in h \) of type \( E_i \);
2. \( E_i[h] < E_j[h] \) iff event instances \( e_i, e_j \in h \) of type \( E_i \) and \( E_j \), respectively, with \( e_i.t < e_j.t \);
3. Any formula built upon the above two atomic formulas by means of the logical connectives and \( \exists h \) and \( \forall h \).

\( \mathcal{L} \) and its derivatives have been used in the literature to describe the semantics of various applications. Since \( \mathcal{L} \) is very general, in many practical scenarios, only subsets of \( \mathcal{L} \) are considered. In this paper, we focus on the following two types of constraints that allow polynomial time reasoning under both static and runtime case (Section III). These constraints may be explicitly given by the business rules or they can be extracted from a given workflow model \([19]\). We denote \( \mathcal{C} \) as a conjunction of a set of event constraints, which contains order constraints \( \mathcal{C}^o \) and occurrence constraints \( \mathcal{C}^o \).

- \( \forall h, \neg (E_i[h] < E_i[h]) \), called order constraints, denoted as \( \mathcal{C}^o \);
- Horn clauses built upon \( E_i[h] \) and \( \forall h \), called occurrence constraints, denoted as \( \mathcal{C}^o \).

Here \( h \) denotes the entire trace, indicating that the constraint must hold w.r.t. the scope of the entire trace. Such global semantics is common \([13]\).

**Table 1** Constraints that allow constant-time runtime reasoning

<table>
<thead>
<tr>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. prior(( E_i, E_j, h )) := ( \forall h, \neg (E_i[h] &lt; E_j[h]) )</td>
</tr>
<tr>
<td>2. exclusive(( E_i, E_j, h )) := ( \forall h, E_i[h] \rightarrow \neg E_j[h] )</td>
</tr>
<tr>
<td>3. require(( E_i, E_j, h )) := ( \forall h, E_i[h] \rightarrow E_j[h] )</td>
</tr>
</tbody>
</table>

Even allow constant-time runtime reasoning (Section IV). This assures negligible runtime reasoning overhead and thus significantly improves the CEP performance.

**Event Query.** In this work, we do not provide a new CEP language as is the focus of a number of existing works \([3], [10]\). Instead we focus on how the core common to most CEP languages can be optimized by exploiting commonly available constraints. Similar to a number of existing works \([3], [10], [31]\), an event query is specified as follows:

\[
\begin{align*}
\text{EVENT} & \quad \langle \text{event expression} \rangle \\
\text{WHERE} & \quad \langle \text{predicates} \rangle
\end{align*}
\]

The EVENT clause specifies the event expression. Due to limited space, we only consider conjunctive queries in this paper, which contain the following two operators. The discussion of disjunctive queries can be found in \([12]\).

- \( \mathcal{SEQ}(E_i, E_{i_2}, ..., E_{i_n})(ts, te) := \exists t_1, t_2, ..., t_n, E_i(t_1, t_1) \land E_{i_2}(t_2, t_2) \land ... \land E_{i_n}(t_n, t_n) \land t_1 < t_2 < ... < t_n \)
- \( \mathcal{AND}(E_i, E_{i_2}, ..., E_{i_n})(ts, te) := \exists t_1, t_2, ..., t_n, E_i(t_1, t_1) \land E_{i_2}(t_2, t_2) \land ... \land E_{i_n}(t_n, t_n) \land t_1 < t_2 < ... < t_n \)

We refer to the output of these operators as a composite event. While the event instance (called primitive event) has a point-in-time semantics, \( e_i.t \), the composite event has an interval semantics, where \( t_s \) and \( t_e \) are the timestamp of the first and the last event in the event expression, respectively. The above definitions adopt this interval semantics and support the arbitrary nesting of these operators. As a special case, when \( E_i \) is a primitive event type, \( t_s = t_e \).

The WHERE clause contains an equality condition on some common attributes across multiple event types in the query, which is typical for monitoring applications \([3], [31]\). This equality condition partitions the event history into sub-sequences. Each subsequence correspond to an event sequence \( h_s \) defined previously. The query is then evaluated against each \( h_s \). There might be additional predicates over the other attributes as well. The output of the query contains the concatenation of all matching event instances. While customized output results can be further accomplished \([3], [10]\), this is independent of the work presented here.

For ease of presentation, we use an acyclic directed graph \( G(Q) = (N, V) \) to represent an event query \( Q \). Each node is either an event type or one of the two special types of nodes, namely, the start \( \text{AND}^S \) and end \( \text{AND}^E \) of the AND operator. Each edge represents the ordering relationship between event types in the query. Since query \( Q \) is well nested, the corresponding start and end of AND nodes are paired as well. Figure 2 depicts such an example.

**Unsatisfiability-preserving Translation.** The event query is translated into a formula in \( \mathcal{L} \) that preserves unsatisfiability (proof omitted for space concerns). For any conjunctive event query \( Q \), the corresponding formula in \( \mathcal{L} \) is: \( \exists h, \forall \{E_i[h]\} \land \{E_i[h] < E_k[h]\} \), for any \( E_i \in Q \) and for any \( E_j, E_k \) which have a order relationship in \( Q \) (i.e., closure). Through
this translation, we can reason between $C$ and $Q$ to check its unsatisfiability.

III. QUERY UNSATISFIABILITY CHECKING

A. Overview

As motivated in Example 1, given an event query $Q$, event constraints $C$ and a partial trace $h_p$ observed at runtime, we want to determine whether a query match may exist in the complete trace $h_e$ with $h_p \subseteq h_e$. We refer to this problem as the runtime query unsatisfiability (RunSAT) problem. There is an extreme case of this problem, i.e., given an event query $Q$ and event constraints $C$, does a query match exist in any trace $h_e$. We refer to this extreme case as the static query unsatisfiability (SunSAT) problem. In this section, we will describe these problems in details.

B. Static Query Unsatisfiability

We formalize the static query unsatisfiability (SunSAT) problem in Definition 3.1.

Definition 3.1: Static Query Unsatisfiability (SunSAT)

Given a query $Q$ and event constraints $C$, $Q$ is said to be statically unsatisfiable if there does not exist a trace $h_e$ which is consistent with $C$ and matches $Q$.

Static satisfiability checking is to check whether $C \land Q \vDash \bot$. This involves two parts, namely, the occurrence consistency checking and the temporal consistency checking, based on the constraint-based translation of $Q$.

Occurrence consistency makes sure that all the event instances required in the query can indeed occur together. This is achieved by checking whether the following boolean expression is satisfiable: $\bigwedge \{E_i[h_e]\} \land C^o$, for any $E_i \in Q$. When the query is conjunctive and $C^o$ contains only Horn clauses, the checking can be done in polynomial time [25].

Temporal consistency means that each event instance required in the query could occur in the desired order. This is to check $\bigwedge \{E_j[h_e] < E_k[h_e]\} \land C^t$, for any $E_j$, $E_k$ that have order relationship in $Q$. The expression is not satisfiable iff at least one $-(E_j[h_e] < E_k[h_e])$ can be inferred from $C^t$. This involves the computation of the closure on $Q$ and $C^t$, which can also be done in polynomial time.

C. Runtime Query Unsatisfiability

As stated before, RunSAT checking differs from SunSAT checking in that RunSAT checking considers a partial trace observed so far. In this sense, SunSAT checking can be considered as a special case of RunSAT checking, i.e., with empty partial trace. Since event data becomes available to the CEP engine in the order of occurrences, the partial trace $h_p$ is always a prefix of the entire trace $h_e$. Definition 3.2 formalizes the RunSAT problem.

Definition 3.2: Runtime Query Unsatisfiability (RunSAT)

Given a query $Q$, event constraints $C$ and a partial trace $h_p$, $Q$ is said to be runtime unsatisfiable iff there does not exist a trace $h_e$ that is consistent with $C$ and contains a match to $Q$, where $h_p$ is prefix of $h_e$.

Matching and Remaining Sub-Query. Given a partial trace $h_p$, the matching sub-query $Q_m$ can be defined as follows. A query node $E_i$ is contained in $Q_m$ iff the sub-graph that contains $E_i$ and all the nodes that can reach $E_i$ in $G(Q)$ has a match over $h_p$. The remaining query $\overline{Q_m}$ contains all the unmatched query nodes $E_i$. The AND nodes are included in $\overline{Q_m}$ if not all of its branches are matched. Figure 3 depicts a query $Q$, partial trace $h_p$, matching and remaining sub-query $Q_m$, $\overline{Q_m}$.

Lemma 1: Given a partial trace $h_p$ and event constraints $C$, if there does not exist a remaining trace $\overline{h_p} = h_e - h_p$ that contains a match to $\overline{Q_m}$, then $Q$ is runtime unsatisfiable.

Our goal is then to check the unsatisfiability of $\overline{Q_m}$, which will lead to the unsatisfiability of $Q$. This naturally leads to the next issue to find the constraints that must hold true for the remaining trace $\overline{h_p}$, referred to as dynamic constraints. To distinguish, the initially given event constraints (Section II) are called static constraints. The dynamic constraints are derived from the static constraints and hold true for the future data.

Dynamic Constraints. The constraints that the remaining trace $\overline{h_p}$ must satisfy evolve as the partial trace $h_p$ grows. Intuitively, the event instances in $h_p$ serve as facts. New constraints can be inferred based on these additional facts and the static constraints. The facts provided by $h_p$, denoted as $F_{h_p}$, include:

- $\bigwedge \{E_i[h_p]\}$, for any $e_i \in h_p$ of type $E_i$
- $\bigwedge \{\neg E_i[h_p]\}$, for any $e - \{E_i\}$ above

The dynamic constraints $C_d(\overline{h_p})$ can be evaluated as follows.

$$C_d(\overline{h_p}) = C \land F_{h_p} = C \land \{E_i[h_p]\} \land \{\neg E_i[h_p]\}$$

The evaluation of Exp.(1) differs from the traditional propositional logic resolution, which basically removes two opposite
literals from two clauses [25], in that first C also contains order constraints C', and second each constraint has its own scope.

\text{First of all, } E_1 \text{ could occur, otherwise } C \land F_{h_p} \land \overline{Q_m} \vdash \neg E_1[h_e] \land E_1[h_e] \text{ and is thus not satisfiable. Next, } E_1 \text{ could occur after } h_p, \text{ otherwise by rule } O_3 \text{ in Figure 4, } C \land F_{h_p} \land \overline{Q_m} \text{ is not satisfiable. Lastly, as long as the prior relationship graph in } C' \text{ does not contain cycle, we can find a position in } h_{Q_m}, \text{ for the occurrence of } E_1 \text{ without violating the order constraints in } C'. \text{ By repeatedly adding all the required events into } h_{Q_m}, \text{ we obtain an event history } h_e \text{ such that it contains a match to } Q \text{ and } C \land F_{h_p} \text{ is satisfiable.}

Hence, RunSAT checking for a given prefix trace } h_p \text{ involves two tasks. First, we derive the dynamic constraints } C_d(h_p) \text{ that hold true for the remaining trace } h_p, \text{ as shown in Exp.(1). Then RunSAT reasoning checks whether the remaining query } \overline{Q_m} \text{ is unsatisfiable by } C_d(h_p) \land \overline{Q_m}. \text{ Note that if } Q \text{ is statically satisfiable, then only occurrence consistency needs to be checked. There is no need to re-check the temporal consistency for remaining query.}

\begin{equation}
C_d(h_p) \land \overline{Q_m} = C_d(h_p) \land \{E_j[h_p]\}, E_j \in \overline{Q_m}
\end{equation}

The evaluation of Exp.(1) and (2) both utilizes the resolution rules in Figure 4. Since these rules add a constant scope checking cost to the classic resolution rules, it can be done in polynomial time for Horn clauses.

**Effective Dynamic Constraints.** Assume that the original conjunctive query } Q \text{ is statically satisfiable. Based on Exp.(2), the only dynamic constraints that can fail } \overline{Q_m} \text{ must be in the form of a disjunction of negated atomic literals, such as } \neg E_i[h_e] \lor \neg E_j[h_e] \text{ or } \neg E_k[h_p]. \text{ We refer to these constraints as effective dynamic constraints, } C_{\overline{Q_m}}(h_p), \text{ where } C_d(h_p) \models C_{\overline{Q_m}}(h_p). \text{ This leads us to goal derived derivation of these specific dynamic constraints (Section IV).}

IV. TOWARDS EFFICIENT RUNSAT

To achieve earliest possible detection of the runtime query unsatisfiability, RunSAT checking should be conducted each time when } h_p \text{ grows, i.e., whenever a new event instance is received. In other words, the dynamic constraints derivation, Exp.(1), and RunSAT reasoning, Exp.(2), have to be performed for each event instance. Unfortunately, on first sight this appears to be much more expensive than simply processing the original query. In this section, we will address this performance issue for RunSAT.

A. Abductive Inference

As } h_p \text{ grows from } h_{p_1} \text{ to } h_{p_2}, \text{ even an incremental method for deriving } C_d(h_{p_2}) \text{ from } C_d(h_{p_1}) \text{ may not be satisfactory. The reason is that first we may have to store some constraints in } C_d(h_{p_2}) \text{ in order for incremental reasoning, and second we may derive many dynamic constraints that are not useful to fail the query at all.}

Fortunately, given the fact that only the effective dynamic constraints could fail the query, we thus propose an abduction-based [14], [15] method to pre-compute the conditions when those effective dynamic constraints will become true. If any
of the conditions are met at runtime, which presumably are cheap to monitor, we know some effective dynamic constraints begin to hold. Abductive inference can be formally defined as follows [14], [15]. For a given effective dynamic constraint \( f_d \), \( p \) is called an explanation of \( f_d \) if \( C \) and \( p \) are consistent with each other and together entail \( f_d \).

1) \( C \land p \models f_d \).

2) \( C \land p \) is satisfiable.

Here \( p \) has to be a conjunction of \( E_1[hp] \) and/or \( \neg E_2[hp] \), since these are the only facts we can draw from the prefix trace \( hp \). Our goal is to find all such explanations \( \forall \{p\} \).

To infer the non-occurrence of \( E_1 \) in the remaining trace, the following three expressions compute its possible explanations.

\[
C^o \land p_1 \models \neg E_1[hp] \quad (3)
\]

\[
C^o \land p_2 \models \neg E_2[hp] \quad (4)
\]

\[
C^o \land C^t \land p_3 \models \neg E_1[hp] \quad (5)
\]

First, by using order constraints \( C^o \) alone, we can only derive \( \neg E_1[hp] \) from Rule 1 in Figure 4. Hence, \( p_1 = E_2[hp] \) if \( C^t = \neg E_1[hp] < E_2[hp] \).

Next, from rules 01-03 in Figure 4, we know that there are two alternative ways that \( \neg E_1[hp] \) can be inferred, namely, from occurrence constraints \( C^o \) only or from both occurrence \( C^o \) and order constraints \( C^t \). Solving Exp.(4) is the classic propositional abductive inference problem [14], [15].

Lastly, solving Exp.(5) needs aid from Rule 03 in Figure 4. For any order constraint \( \neg (E_1[hp] < E_2[hp]) \), given the fact that \( \neg (E_1[hp] < E_2[hp]) \land E_2[hp] \land \neg E_1[hp] \rightarrow \neg E_2[hp] \), we rewrite Exp.(5) into (6) below, which replaces the order constraint by the occurrence constraints it can possibly imply. Then \( p_3 = E_1[hp] \land \neg E_2[hp] \land p' \).

\[
C^o \land E_2[hp] \land \neg E_1[hp] \land p' \models \neg E_1[hp] \quad (6)
\]

Although abductive inference for Exp. (5) and (6) is NP-Complete in general (details in [14], [15]), since it is a one-time cost compared to the long-running event query, the abduction cost may be still acceptable. However, note that the explanations can contain multiple positive events, such as \( E_1[hp] \land E_2[hp] \land E_3[hp] \) or \( E_4[hp] \land E_5[hp] \). In fact, monitoring all such complex explanations could be more expensive than just executing the event query itself and thus becomes infeasible. Hence, a cost-based approach, i.e., monitoring only those explanations that will provide the best cost benefit, is necessary. This remains our future work. In this paper, instead we show that when the explanations contain a single positive event for the common yet simple constraints in Table 1, they can be monitored in constant time.

B. Incremental RunSAT Reasoning

The second performance issue with RunSAT is that we still have to perform the RunSAT reasoning Exp.(2) for \( C^E_d(hp_1) \) and \( C^E_d(hp_2) \), respectively. In other words, we need to store the constraints \( C^E_d(hp_1) \) in order to check whether they would fail the new remaining query. In fact, we find that for monotonic queries, this is not necessary.

Definition 4.1: Monotonic Query. Assume two prefix traces \( hp_1 \) and \( hp_2 \) where \( hp_1 \) is a prefix of \( hp_2 \). The matching sub-queries for a given query \( Q \) under these two prefix traces are \( Q_{m1} \) and \( Q_{m2} \), respectively. Query \( Q \) is monotonic if and only if \( Q_{m1} \) is a subquery of \( Q_{m2} \).

Queries with SEQ, AND operators are monotonic.

Lemma 2: Incremental RunSAT Reasoning. Assume that the prefix trace grows from \( hp_1 \) to \( hp_2 \). For a conjunctive query \( Q \), we assume that the remaining queries are \( Q_{m1} \) and \( Q_{m2} \), and the effective dynamic constraints are \( C^E_d(hp_1) \) and \( C^E_d(hp_2) \), respectively. If \( Q \) is a monotonic query, then \( C^E_d(hp_1) \land Q_{m1} \) is satisfiable \( \rightarrow C^E_d(hp_2) \land Q_{m2} \) is satisfiable.

To summarize, to improve the RunSAT performance, first, the derivation of Effective dynamic constraints can be precomputed through abduction. Second, when the query is monotonic, there is no need to reconsider the previously derived dynamic constraints. These two techniques pave the way for integrating RunSAT into the event query engine.

V. INTEGRATING RUNSAT INTO CEP ENGINE

In this section, we describe how we apply the theoretical results of RunSAT checking as efficient optimization techniques for event query processing.

Our C-CEP engine employs the commonly-used automata model (i.e., NFA) since it has been shown to be a natural fit for event pattern matching [10], [18], [31]. When registering an event query into the C-CEP engine, the engine first checks whether this query is statically satisfiable w.r.t. event constraints \( C \). Then it uses the abductive inference to precompute the failure conditions. The original event query is augmented with these failure conditions as Event-Condition-Action rules. During query execution, these failure conditions are efficiently monitored. If any of these failure conditions are met, the current trace is unsatisfiable to the query and any partial matches are removed.

A. NFA Query Execution Model

For query execution, we adopt and extend the commonly-used NFA model [10], [18], [31] to also support the AND operator. Using this common execution model assures that our work can be easily integrated into existing CEP systems as a semantic query optimization module.

Our NFA model includes two types of states, namely, regular states and logical states, and it can be easily generated from the query graph in Figure 2. Each node \( E_i \) in the query corresponds to a regular state in the NFA. At runtime, the event instances that match these states are kept in the memory in order to generate the final output. The \( ANDE \) corresponds to logical state, which is activated only when all the input transitions have been triggered. There is a self-loop of + transition over those nodes which have non-ε output transitions in order to capture the temporal following semantics. For example, the query in Figure 2 is translated into the automaton in Figure 5.
B. Augment Query with Fail Conditions

Our query engine exploits the constraints in Table 1 for optimizing the event query. We will show that supporting these constraints does not require a cost-based optimization since the extra overhead is small. While developing a cost-based optimization framework for the more complex constraints remains our future work, our performance evaluation for these simple constraints also indicates when such optimization is beneficial, which provides the basis for cost estimation. The effective dynamic constraints that could fail the query are \( \neg E_i[h_p] \) and \( \neg E_i[h_e] \). \( \neg E_i[h_e] \) is called global since it holds for the entire trace and is independent of the query matching status. \( \neg E_i[h_p] \) is called local since it only holds for the remaining trace. Hence whether \( \neg E_i[h_p] \) can be used to fail the query depends on whether the remaining query contains \( E_i \) or not.

1) Managing Global Failing Conditions: We first discuss how to augment the query with global failing conditions. For each \( E_i \) in the query, we derive all failing conditions for \( \neg E_i[h_p] \). By solving \( \text{Exp.(4)} \), we have the failing conditions \( p_2 = E_j[h_p] \) if \( \text{C}^o \models (E_j[h_e] \rightarrow \neg E_i[h_e]) \). By solving \( \text{Exp.(5)} \), which is rewritten into \( \text{Exp.(6)} \), we have the failing conditions \( p_3 = E_j[h_p] \land \neg E_k[h_p] \) if \( \text{C}^o \models (E_k[h_e] \rightarrow E_i[h_e]) \) and \( \text{C}^t \models \neg (E_k[h_e] < E_j[h_e]) \).

These failing conditions can be organized into a simple data structure depicted in Figure 6. We use an array with the size equal to the number of distinct event types. The ‘+’ symbol at \( E_i \) means that \( E_i[h_p] \) is a failing condition of the query. For each entry \( E_j \) marked as ‘+’, we associate a bit array. For any \( E_k \) with the bit being 1 in that bit array, \( E_j[h_p] \land \neg E_k[h_p] \) is a failing condition of the query.

At runtime, given an event instance of \( E_i \), we check if the corresponding entry in the global failing condition is marked as ‘+’. If so, we terminate the processing of this trace. Any partial results or active states for this trace can be removed. If the entry is marked as ‘−’ and there is a bit array associated with it, we perform a bit-AND with a runtime bit array whose entries indicate the occurrence of \( E_i \) in \( h_p \) (1 denotes non-occurrence). If the output of this bit operation is not zero, we can fail the matching for this trace.

2) Managing Local Failing Conditions: Since the local failing conditions are tightly coupled with the particulars of the current query matching status, we build them into the NFA by introducing a special state labeled “I” (for “Failed”). All transitions triggered by local failing conditions are directed to this “Failed” state.

For each \( E_i \) in the query graph, by \( \text{Exp.(3)} \), we compute the local failing conditions \( \{p_1\} \) for any \( E_i \) that is reachable from \( E_i \) in the query graph. We implement the failing conditions in NFA as the additional transitions of \( E_i \). These failing conditions are valid only when none of these transitions out of \( E_i \) have been matched yet. Hence there is a special runtime issue, i.e., once the NFA transition from \( E_i \) to the next state is made, the local failing conditions at \( E_i \) need to be deactivated. Intuitively, the query matching status is changed, which breaks the assumption that none of \( E_i \)’s descendant states have been matched. Such NFA state deactivation can be efficiently supported using a flag. Obviously, both global and local failing condition checking can be done in constant time.

Figure 7 depicts the augmented query for event pattern EP1 in Section I. The SendQuote event is the local failing condition.

VI. EXPERIMENTAL EVALUATIONS

We have implemented the techniques presented in this paper in a Java-based CEP system. We developed an event generator that creates event streams based on the workflow in Figure 1 with the following parameters: 1) event attributes: 5 attributes (besides timestamp) per event, including three integer-type and two string-type; 2) number of allowed values of each event attribute, used to control the selectivity of the query predicates. The values conform to uniform distribution; 3) probability distribution of exclusive choice construct, used to control the query selectivity; and 4) number of concurrent traces (1000). The events of concurrent traces are interleaved in the event stream. Lastly, we fix the number of loops on GenerateQuote in the workflow to be 3. The test machine has an Intel(R) Pentium I8G processor and 1GB RAM, running Windows XP and Java 1.5 SDK.

We compare the performance of C-CEP, with regular CEP, denoted as R-CEP. For R-CEP, each time a trace is finished, i.e., whenever a CancelOrder, RejectOrder or FinishOrder event is received, any partial matches and automata states
associated with this trace can be removed. For C-CEP, we augment the query with RunSAT failing conditions. Whenever a RunSAT failing condition is satisfied, C-CEP can remove the data. We run both C-CEP and R-CEP in CPU-limit mode [30], i.e., events arrive to the CEP system at a rate such that the query processing never needs to wait for data. We measure 1) total number of NFA probes (for event matching), 2) total execution time for processing the given event stream, and 3) peak number of events maintained in all NFA states, which reflects the peak memory usage. This number is collected after system warm-up, i.e., after 1000 traces are processed. For C-CEP, the execution time includes the RunSAT checking cost. The input event stream contains 400K events from 20,000 traces for all the experiments below.

A. Results on Sequence Queries

We first compare the performances of C-CEP and R-CEP on sequence queries. We show the experimental results for Query Q1 below, which monitors those expensive orders that uses remote stocks (rare case). The global failing condition for this query is the UseLocalStock event, and the local failing condition for the GenerateInvoice event is the SendInvoice event.

\begin{verbatim}
EVENT SEQ(CheckInventory, UseRemoteStock, GenerateInvoice)
WHERE GenerateInvoice.price>200
\end{verbatim}

In the first experiment, we vary the matching probability of the UseRemoteStock event in the query from 0% to 90%. We achieve this by varying the probability distribution of the exclusive choices on UseLocalStock and UseRemoteStock. We define the fail ratio of an event E in the query to be \((1-\sigma_E)\) with \(\sigma_E\) being the matching probability of \(E\). The results are shown in Figure 8(a).

Two observations are made from the results. First, as the fail ratio increases, both the total number of probes (and hence total execution time) and peak memory usage decrease. For 90% fail ratio, significant savings in memory (60%) and in execution time (32%) compared to R-CEP are achieved. This promising result suggests that C-CEP is especially attractive for those targeted alert queries. Note that the savings in execution time by C-CEP are not precisely proportional to the savings in NFA probes. The reason is that after a trace is determined to be unsatisfiable, for every event in the rest of the trace, a single check is needed to determine whether this event belongs to a failed trace. Second, for zero fail ratio (i.e., all traces have matches to the query), which can be seen as the worst case for C-CEP since no evaluations can be terminated early while extra cost has to be paid for RunSAT checking, the execution time of C-CEP is only negligibly higher than R-CEP. This is also promising, indicating that even in the worst case, C-CEP has comparable performance with R-CEP.

Next, we test how the query fail point affects the C-CEP performance. In the previous experiment, the query fails always due to no match for the UseRemoteStock event. We now test the case in which the query fails always due to no match for the GenerateInvoice event with price>200.

We call this the "fail late" case while the previous case the "fail early" case because the UseRemoteStock event is before the GenerateInvoice event in the event query. We vary the matching probability of the GenerateInvoice event to be from 0% to 90%, while fixing the matching probability of UseRemoteStock to 100%. We achieve this by controlling the value range of the price attribute of the GenerateInvoice event. The results are shown in Figure 8(b).

In the "fail late" case, for 90% fail ratio, the memory saving is 54% and execution time saving is 21%. Since failing late incurs more execution overhead, the gains are less than those achieved in the "fail early" case (Figure 8(a)). However, it still provides significant memory savings for alert queries and is thus useful when the memory is a stringent resource.

B. Results on AND Queries

Next, we compare the performances of C-CEP and R-CEP on AND queries. The query is given below. The global failing conditions for this query are the UseLocalStock and the CancelOrder event, and the local failing condition for the GenerateQuote event is the SendQuote event.

\begin{verbatim}
EVENT SEQ(AND(SEQ(OrderFromSupplier, GenerateQuote),
SEQ(UseRemoteStock, GenerateInvoice)), FinishOrder)
WHERE GenerateQuote.price>200
\end{verbatim}

We conduct two sets of experiments. First, we fix the matching probability of the first AND branch (i.e., SEQ(OrderFromSupplier, GenerateQuote)) (more specifically, the GenerateQuote event) to be 50% and vary the matching probability of the UseRemoteStock event to be from 0% to 90%. The results are shown in Figure 9(a). Second, we fix the matching probability of the second AND branch (i.e., SEQ(UseRemoteStock, GenerateInvoice)) to be 50%, while varying the matching probability of the GenerateQuote event to be from 0% to 90%. Since 3 loops are involved for GenerateQuote event in the workflow, the failure on matching the first AND branch will be detected rather late compared to that for the second AND branch. This may result in performance difference between these two sets of experiments. The results are in Figure 9(b).

Two observations are made from this experiment. First, much more performance gains can be achieved compared to the sequence query Q1. As can be seen in Figure 9(a), for 90% fail ratio, the gains in peak memory usage and in execution time are 72% and 51% respectively. This is because Query Q2 is more complex than Query Q1, thereby rendering bigger partial matches. This causes higher event matching costs and memory overhead in R-CEP. The C-CEP on the other hand, can terminate the query execution as soon as one branch is found to be unsatisfiable. Another important observation is that the performance gains by C-CEP are determined by the AND branch that provides the most performance gains. The second AND branch, by failing early, enables much noticeable performance gains as fail ratio increases (Figure 9(a)). In contrast, the first AND branch, by failing late, enables much less performance gains until the
fail ratio is very high (Figure 9(b)).

**Scalability test.** We also conduct the scalability test for the above sequence, AND queries in which the event stream contains 4M events from 200,000 traces with 10,000 concurrent traces. The results are similar to the ones presented here in terms of percentage-wise performance gains and are thus omitted. This indicates that our C-CEP techniques are also scalable.

**VII. RELATED WORK**

As event processing gains popularity in many applications, an increasing effort has been devoted in developing efficient event processing systems. The existing work include streaming databases such as HiFi [17] that support SQL-style queries, pub/sub systems such as [2], [16] that support simple filtering queries, and CEP systems such as SNOOP [7], Amit [1], CEDR [3], Cayuga [10] and SASE [31], that support event pattern queries expressed by more powerful languages. These works focus on query model/language design and query algebra development. None of these works considers exploiting the common event constraints.

Semantic query optimization (SQO), i.e., using schema knowledge to optimize queries, has been extensively studied for traditional databases [8], [22]. Major techniques focus on optimizing value-based filtering or matching operations, including join and predicate elimination and introduction. They remain applicable in CEP for identifying efficient query plans at compilation time. These existing SQO techniques are mainly designed for static query optimization. They are inappropriate for runtime use. SQO has also been studied for optimizing queries over streaming XML documents [26]. In CEP, we are faced with event data from possibly thousands or millions of concurrent processes interleaved, and thus huge numbers of potential partial matches (one for each process) at runtime. Also, more types of constraints can be observed in business processes than in XML schema. All these pose stringent
requirements on scalability, generality and extensibility on exploiting constraints in CEP. Our work is also related to punctuation [23], [28]. The existing works on punctuation mainly focus on utilizing punctuations to reduce the memory usage of SQL-type of stream query. In this work, we show how to generate punctuations (effective dynamic constraints) from event constraints and how to use them to reduce both CPU and memory cost for CEP queries.

Other related areas include workflow management [19], [27] since the event constraints are extracted from the workflows. The existing work on workflow management focuses on two problems, workflow analysis and workflow verification. Workflow analysis involves the soundness proof of a workflow and the identification of critical activities in a workflow. Workflow verification deals with the following problem. Given a finite set S of dependencies, check whether there is a workflow execution (or all executions) satisfying all the dependencies in S. This conceptually is similar to our SunSAT reasoning. Our exploitation of the order constraints relates to the work on temporal reasoning [20], i.e., to detect whether a cycle exists among the order constraints in query and in event data. However, the existing works on temporal reasoning focus on the language specification and enforcement instead of utilizing temporal constraints to optimize queries.

VIII. CONCLUSION

In this paper, we presented the first work on exploiting constraints to optimize CEP by detecting and terminating the unsatisfiable query processing at the earliest possible time. We abstracted our problem into a query unsatisfiability problem. We formally defined runtime query unsatisfiability (RunSAT) problem and its extreme case, static query unsatisfiability (SunSAT). We then studied the incremental properties of the RunSAT checking procedure, which includes two key operations, dynamic constraint derivation and RunSAT reasoning. Based on the incremental properties, we described a solution to pre-compute the query failure conditions by employing abductive reasoning. We also presented a constraint-aware CEP architecture that integrates our proposed techniques with state-of-the-art CEP techniques. We showed an extensive experimental study based on online order processes. Our experimental results on sequence, AND queries demonstrated that significant performance gains can be achieved through our approach, while the optimization cost is small.

Acknowledgement We thank Divyakant Agrawal for many valuable comments.

REFERENCES