Predictive Publish/Subscribe Matching

Vinod Muthusamy, Haifeng Liu, Hans-Arno Jacobsen
University of Toronto

ABSTRACT

A new publish/subscribe capability is presented: the ability to *predict* the likelihood that a subscription will be matched at some point in the future. Composite subscriptions consisting of temporal and logical operators are efficiently represented by a set of finite state machines and rules. The algorithm trains a Markov model to an application's event workload, and predicts the probability that a given subscription will match within a window in the future event stream. Evaluations demonstrate that the memory and processing costs of the algorithm scales well with the number of subscriptions, and the prediction precision is high, especially when the workload characteristics do not change rapidly. A comparison with a hand-crafted Markov model using real data traces shows that the algorithm consumes much less memory and processing power, and still delivers prediction precision that approaches the hand-crafted model's. This is especially impressive since the algorithms lack any of the domain expertise embedded in the hand-crafted model.

1. INTRODUCTION

A wide variety of scenarios require the detection, correlation, or aggregation of events. For example, a credit card fraud detection system may correlate a user's purchases to find any suspicious usage patterns. Current complex event processing engines and publish/subscribe systems support the real-time detection of such complex patterns [4, 6, 20, 19, 12]. In many applications, however, it is too late to raise an alert of a malicious activity after it has occurred. What is needed is a way to *predict* the likelihood of a given pattern matching at some point in the future. This paper proposes a novel publish/subscribe matching engine that computes the probability that a subscription, or pattern, will match based on the application workload and partial match of the subscription.

Many applications would benefit from predictive subscription matching capabilities. For example, in a network monitoring system [11], a particular pattern of login attempts and port scan operations may not, constitute a successful attack, *per se*, but experience may indicate it is a precursor to an intrusion. The system may then proactively block the intruder or take other defensive measures. Other scenarios that may exploit predictive matching include algorithmic trading [16], business activity monitoring [11], and epidemic warning [22].

One approach to realize predictive capabilities is to define additional subscriptions whose match indicates some likelihood that an activity is about to occur. For example, the increased sales of a cough medication followed by long hospital wait times may signify an 80% probability of a flu outbreak. The problem with this approach, however, is that defining these additional subscriptions and rules requires domain expertise, can be complex and time consuming, and may need to be revised as the workloads change.

The predictive publish/subscribe matching engine in this paper, on the other hand, is easy to use, and can predict when a subscription is about to be matched without requiring any domain expertise or additional rules to be defined. The prediction is dynamically tuned to the current application workload based on the history of events. Furthermore, the user is able to control the quality of the prediction, in terms of the number of false positives, by specifying the minimum prediction threshold and how far into the future to look ahead.

This work complements probabilistic databases event processing engines [7, 22, 3] that work with uncertain data. Whereas these systems are designed for scenarios where there is imprecision in the event stream and event values, this paper assumes that the events themselves are precise; the only uncertainty is in the future event stream. This work also complements existing publish/subscribe research [1, 10, 5, 9, 18] that determine matching subscriptions based on published events, but can not predict future matches based on events observed in the past.

This paper makes three key contributions: (1) A set of algorithms are developed to represent arbitrarily complex subscriptions consisting of temporal and Boolean operators. Temporal subexpressions of a subscription are matched by a finite state machine and Boolean subexpressions by a rule-based engine. This partitioning avoids the state explosion of FSMs with Boolean expressions. As well some common subexpressions among a set of subscriptions are merged so to reduce computation and memory costs. (2) An efficient engine is proposed to predict matches of subscriptions. A Markov model is used to perform the prediction, and the model is continually tuned to the application workload. The engine can be configured to report only those predictions above a given threshold and within a given time in the future. (3) A thorough quantitative evaluation of the costs of the algorithms is presented. The computation and memory consumption scalability of the algorithms are presented, the quality of the predictions under a variety of situations is analyzed, and a comparison of the prediction engine compared to a hand-crafted Markov model for a real-world scenario is made.

Technical Report, Middleware Systems Research Group, University of Toronto. March, 2009, Toronto, Canada.

2. RELATED WORK

This paper complements ideas developed in probabilistic databases, CEP systems that operate on uncertain data, and publish/subscribe.

Probabilistic Database: Probabilistic databases are currently an active area of research. A probabilistic database is an uncertain database in which the possible worlds have associated probabilities. Fuhr introduced a probabilistic relational algebra to represent imprecise attribute values and integrate vague queries in database systems [13]. While there are currently no commercial probabilistic database systems, several research prototypes exist including Trio [3], Orion [21], MystiQ [8] and MayBMS [2].

Trio [3] is a database management system which integrates data, uncertainty of the data, and data lineage together. Trio is based on an extended relational model called ULDBs, and it supports a SQL-based query language called TriQL.

The ORION database system [21], is an uncertain database management system with built-in support for probabilistic data as first class data types. In contrast to other uncertain databases, Orion supports both attribute and tuple uncertainty with arbitrary correlations.

MystiQ [8] is a system that uses a probabilistic data model to find answers in large numbers of data sources exhibiting various kinds of imprecisions. Moreover, users sometimes want to ask complex, structurally rich queries, using query constructs typically found in SQL queries: joins, subqueries, existential/universal quantifiers, aggregate and group-by queries.

MayBMS [2] uses probabilistic versions of conditional tables as the representation system, but in a form engineered for admitting the efficient evaluation and automatic optimization of most operations of a language using robust and mature relational database technology.

Event uncertainty management: In complex event processing systems, events from the environment are correlated and aggregated to form higher level events. Uncertainty in the events may be due to a variety of factors including imprecision in the event sensors or generators, and corruption of the communication channel possibly dropping events.

Wasserkrug *et al.* [22] present an event processing engine that supports matching events with degrees of uncertainty. The uncertainty of events propagates through to the event materialized as a result of a pattern matching, thus transmitting the uncertainty across the event causality chain. An algorithm based on a Bayesian network and more efficient approximation algorithm are developed.

PEEX [15] manages ambiguous and unreliable events from RFID tags by annotating events with a probability distribution on their values. Evaluations show that the system improves recall at the expense of precision.

Publish/Subscribe Systems: Publish/Subscribe systems are an active area of research [1,10,5,9,18]. Subscriptions expressing subscribers' interest in events are continually evaluated against publications representing events. The approaches are distinguished by the data formats they process and by the algorithmic design. Common among the approaches is the determination of a match based on the publication processed. None of the approaches predicts matches based on publications processed in the past.

Unlike probabilistic databases and uncertain event processing systems, this paper assumes that the events are precise and have no ambiguity. Instead, it is the future event stream that is unknown and the matching of a pattern at some point in the future is predicted with some probability.

Table 1: Event stream of login history

EID	TIME	STATUS	IP
e_0	2007/02/14/12:38:10	denied	128.100.2.15
e_1	2007/02/14/12:42:10	denied	128.100.2.15
e_2	2007/02/14/12:42:10	denied	128.100.2.15
e_3	2007/02/14/12:43:28	success	128.100.2.15
e_4	2007/02/14/12:43:56	logoff	128.100.2.15
e_5	2007/02/14/12:45:28	success	128.100.5.10

3. SYSTEM MODEL

In this section we describe the language and data model underlying our predictive publish/subscribe approach. The objectives are to allow subscribers to express interests in complex constellations of events over event streams allowing the subscriber to join individual events through Boolean operators and constrain interests via explicit and implicit temporal conditions. The language is the basis for algorithms to match subscriber interests and to predict matching results from partially detected events.

3.1 Publication Data Model

Publications describe real-world events and are defined by a set of attribute value pairs: $e_i = \{(a_1, v_1) \cdots (a_n, v_n)\}$.

Unlike conventional publish/subscribe approaches, the model in this paper operates over event streams, where an *event stream* is interpreted as infinite sequence of events, and each event represents an occurrence of interest at a designated point in time. As is common in the publish/subscribe literature, we are using the terms *event* and *publication* synonymously. Each event contains a field that records when it occurred. Events are assumed to be processed in the order they occured.

For example, Table 1 shows the events logged by a typical operating system monitoring facility. Whenever a login attempt is registered by the system, the event is recorded with a timestamp, a status message, and the IP address from where the attempt was made. We add an event ID, as a unique identifier for each event.

3.2 Subscription Language

A subscription expresses a subscriber's interest. A *primitive subscription* is matched by a single event, whereas a *composite subscription* is matched by a set of events.

A primitive subscription is a conjunction of predicates, each predicate defining a constraint over an attribute: $s = p_1 \wedge p_2 \wedge \cdots \wedge p_k$, where the p_i are Boolean predicates.

For example, a primitive subscription that monitors a failed login from a specific computer is expressed as s_{failed_login} : $(STATUS = denied) \wedge (IP = 128.100.2.15)$. Events e_1 and e_2 would each match primitive subscription s_{failed_login}.

A composite subscription expresses interest in *composite events*, which represent constellations of events. More formally, a composite subscription is defined as an expression over primitive subscriptions that are composed with temporal or Boolean operators. The temporal operators supported include *contiguous sequence* (whose symbol is ","), *non-contiguous sequence* (symbol ";"), and *explicit temporal* (symbol "@"); and the supported Boolean operators are conjunctions (symbol "∧") and disjunctions (symbol "∨").

The contiguous sequence operator, ",", requires its operand subscriptions to be matched by contiguous events in the event stream. The non-contiguous sequence operator, ";", is similar, but allows other events to occur between the events in the event stream matching the operand subscriptions, as long as the matched events occur in order. Since events in the event stream occur in order and we assume no events occur at the same time, there is an implicit temporal condition associated with both "," and ";" operators. That is the time interval between the events matching the operand subscriptions has to be greater than zero. One event cannot match two operands of the same composite subscription. In contrast, "@" is an explicit temporal operator; it adds an explicit temporal condition that requires the matching events to occur in the specified time interval.

In our intrusion detection scenario a possible intrusion can be modeled by a composite subscription as follows:

$$\begin{aligned} s_1, s_2, s_3 &: \quad IP = \$x \wedge STATUS = denied \\ s_4 &: \quad IP = \$x \wedge STATUS = success \\ cs_{intrusion} &: \quad s_1; s_2; s_3@(t(s_3) - t(s_1) < 5min); s_4 \end{aligned}$$

$cs_{intrusion}$ defines an intrusion as at least three failed login attempts within a 5 minute interval followed by one successful login, with intermitting events allowed before the successful login captured by s_4. The variable, x, in the IP predicate is used to represent a join condition that all login attempts originate from the same IP address. The non-contiguous sequence operator used allows for other events to occur in between.

A more general example:

$$\begin{aligned} s_1, s_2 &: \quad IP = \$x \wedge STATUS = denied \\ s_3, s_4 &: \quad IP = \$x \wedge STATUS = success \\ s_5 &: \quad IP = \$x \wedge STATUS = passwd \\ s_6 &: \quad IP = \$x \wedge STATUS = logoff \\ cs_{compromised} &= \quad s_1; ((s_2; s_3@(t(s_3) - t(s_1) < d)) \vee \\ &\quad\;\; (s_4, s_5)); s_6 \end{aligned}$$

$cs_{compromised}$ defines an intrusion pattern bracketed by a failed login attempt and a logoff action; in between these events, either there are at least one failed login followed by one successful login, or one successful login followed by a password reset action. This composite subscription illustrates the combination of both temporal and Boolean operators.

3.3 The Matching and Prediction Problems

The matching problem is similar to the standard publish/subscribe matching problem: *Given a set of composite subscriptions CS and an event stream E, find all $cs \in CS$ such that the events in E satisfy the primitive subscriptions in cs as well as any temporal and Boolean operators.*

For example, $cs_{intrusion}$ is matched by the event sequence e_0, e_1, e_2, e_3 from Table 1.

Next, we describe the prediction problem. At some time, t_{now}, events in the event stream may have matched some of the primitive subscriptions registered with the system. A composite subscription for which some of its primitive subscriptions are matched is said to be in a *partially matched* state. This is the case, for example, when s_1 and s_2 in $cs_{intrusion}$ are matched.

Our objective is to be able to predict that the probability a subscription, cs, will match is greater than a threshold, θ_{cs}, after processing N further events.

Say, based on past observations, we know that an intrusion occured half of the time after the first failed login and 80% of the time after two failed logins. Based on this, we conclude that the composite subscription, $cs_{instrusion}$, is matched with a probability of 0.5 after observing the first login failure and with a probability of 0.8 after the second failure.

That is given an event stream, we aim to calculate the probability that a composite subscription, cs, is matched some time in the future given its current partial matching state. The challenge is to efficiently calculate this conditional probability for all composite subscriptions. Thus, the prediction problem is as follows: *Given a set of composite subscriptions, CS, and an event stream, find all partially matched composite subscriptions $cs \in CS$ such that the probability of cs transitioning from a partial match to a full match, after processing N events, is greater than the threshold, θ_{cs}.*

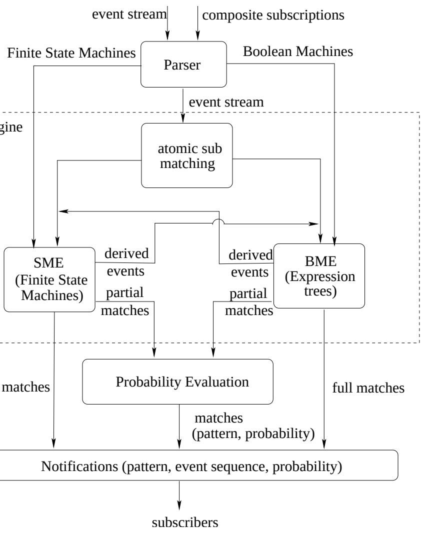


Figure 1: System architecture

4. SUBSCRIPTION PROCESSING

The publish/subscribe system presented in this paper performs four principal matching tasks. These are the matching of primitive subscriptions, the matching of sequence and temporal operators, the matching of Boolean expressions underlying composite subscriptions, and the prediction of subscription matches based on partial matching state evolving over time. Figure 1 shows the matching engine architecture and hints at the processing flow of events through the system.

Before discussing the event processing flow in more detail, we describe how composite subscriptions are decomposed and represented in the system. A composite subscription is an expression tree. Intermediate tree nodes represent the operators and leaf nodes the expressions of the primitive subscriptions.

The Boolean expressions defining the composite subscription are represented as Boolean trees and are managed by the *Boolean Matching Engine* (BME) in the architecture. The temporal subexpressions of composite subscriptions are represented as finite state machines (FSM) and are managed by the *State Machine Engine* (SME). Individual primitive subscriptions, which are at the leaves of the tree, are managed by the *Atomic Subscription Matcher* (ASM). The decomposition of a composite subscription into FSMs and Boolean trees is described in Sec. 4.2.

Input events are first evaluated against all primitive subscriptions stored in the ASM. The resulting matches drive the state transitions of the SME and the matching of the Boolean operators of composite subscriptions in the BME. SME and BME produce three kinds of outputs. The first kind are the fully matched composite subscriptions. The second kind are referred to as *derived events*. These are events that are fed back from the SME to the BME and from the BME to the SME to trigger further state transitions and Boolean matches, respectively. The third kind are partial matches passed on to the *Prediction Engine* (PE).

Primitive subscription matching and Boolean expression matching has been extensively studied [17, 5, 10, 18] and will not be further discussed here. Here, we focus our discussion on the algorithms underlying the State Machine Engine (SME), which leverages FSMs [14] for the processing of sequence operators. Subscription expressions are graphs that represent FSMs derived from the sequence operators. In the graph a node represents a state, an

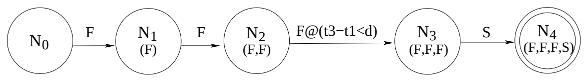


Figure 2: FSM for $F, F, F@(t_{N_3} - t_{N_1} < d), S$

edge represents a state transition labelled by a primitive subscription whose match triggers the transition. State transitions are triggerd by events. Below, we describe the FSM construction and event processing algorithms for our composite subscription expressions.

4.1 Sequence Operators

Constructing an FSM and processing it for expressions using the contiguous sequence operator is simple. It constitutes a building block for supporting the other operators. In our presentation, we assume the processing of one expression, deferring optimizations for the merging and joint processing of multiple FSMs to below.

Given an expression with k primitive subscriptions $cs = s_1, s_2, \cdots, s_k$, the FSM has $k+1$ states capturing the intermediate states of matching the individual primitive subscriptions. N_0 is the initial state (no matches), N_1 represents the state that s_1 matches, N_{k-1} represents the state that the previous $k-1$ contiguous events match $s_1, s_2 \cdots s_{k-1}$, and the last state, N_k, represents the state that the whole expression is matched. For each state N_i, where $0 \le i \le k-1$, we add a transition from state N_i to state N_{i+1} upon the match of the primitive subscription s_{i+1}.

During matching, the initial state is always checked for each new event to see whether a new partially matching instance of the FSM must be initialized. If nothing matches the incoming event, all current FSM instances are simply discarded, exploiting the property that no intermittent events are permitted in matching the contiguous sequence operator.

We treat the temporal condition defined by @ as an additional predicate of the associated primitive subscription. The same algorithm can be applied to build the FSM for expressions containing @ operators as for building the FSM for contiguous operators.

To support the explicit temporal operator in event processing, each state has an associated field that records time of the most recent transition into the state. Furthermore, when a new instance of a state machine is created, separate transition times are maintained for the states in each instance. These transition times are used when the time condition is evaluated to determine whether to take the transition.

Consider the example subscription and associated state machine in Figure 2. This composite subscription is similar to $cs_{intrusion}$ from the earlier intrusion detection scenario, and expresses an interest in three failed login attempts, followed by one successful attempt with the additional constraint that the three failures occur within a time window of duration d. In the figure, F and S represent the primitive subscriptions that define a failed or successful login attempt, respectively. That is, we abstract in the presentation from the underlying primitive subscription matching, where F and S are the results of evaluating the underlying primitive subscription.

Continuing the example, suppose an event stream contains a sequence of three failed login attempts occurring at times 1, 2, and 3. At this point there would be three instances of the state machine in Figure 2. In one instance, the current state is N_1 with a transition time of 3; in another instance, the current state is N_2 and state N_1 has a transition time of 2; and in the last instance, the current state is N_3, state N_2 has a transition time of 2, and state N_1 has a transition time of 1.

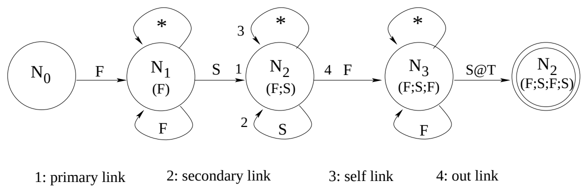


Figure 3: FSM for $F; S_1; F; S_2@(t_{S_2} - t_{S_1} < T)$

There are two differences when constructing an FSM for expressions with non-contiguous sequence operators. First, since events not contributing to matching an expression are allowed to occur during matching, the FSM must remain at the current state, even if the primitive subscription that triggers the transition to the next state is not matched. Second, if the next primitive subscription is matched, we need to take two transitions in order to track all matching possibilities. One transition leads to the next state, the other transition leads back to the state itself to allow future matches of the next primitive subscription to trigger the transition to the next state.

To correctly support the explicit temporal operator "@", it is necessary to differentiate the transitions by which a state is reached. Taking Figure 3 as an example, there are three incoming links to state N_2. From state N_1, there is a link to state N_2 upon the match of S. This is the first time that N_2 is reached upon a match, and we call this link the *primary* link. From state N_2, there are two links back to itself. One is also triggered upon a match of S, and we call this link the *secondary link*. The *self* link which is labeled by $\star$, is triggered for every event except those that cause a transition of the primary or secondary links. Only *primary* and *secondary* links are triggered upon a match and the transition times associated with a state should be recorded on these two transitions for future use when evaluating relevant time conditions.

When both contiguous and non-contiguous sequence operators are used in one composite subscription, we decompose the composite subscription into multiple *terms* separated by the non-contiguous sequence operators where each *term* contains only contiguous sequence operators. We first build a state machine for each term and then add transitions between terms. Algorithm *BuildStateMachine* describes the detailed process.

```
Algorithm BuildStateMachine($cs$)
(* construct an FSM for a CS containing temporal operators *)
1.  Decompose $cs$ into $n$ terms separated by non-contiguous sequence operators: $cs = T_1; T_2; \cdots; T_n$; where each term $T_i = S_{i1}, S_{i2}, \cdots, S_{ik_i}$ contains only contiguous sequence operators
2.  $m = k_1 + k_2 + \cdots + k_n$
3.  Construct $m$ states $N_0$, $N_1 = s_1, \cdots, N_m = s_{11}, s_{12}, \cdots, s_{1k_1}; \cdots; s_{n1}, s_{n2}, \cdots, s_{nk_n}$, where each state represents the partial match state of one more primitive subscription
4.  for each state $N_i = s_1, s_2, \cdots, s_i$ where $(0 \le i \le m)$
5.      add an edge $s_{i+1}$ from $N_i$ to $N_{i+1}$
6.  for each state $N_j = \cdots; s_{i1}, s_{i2}, \cdots, s_{ik_i} (= s_j)$
7.      add an edge $s_{j+1}$ from $N_j$ to $N_{j+1}$
8.      add an edge $s_j$ from $N_j$ to $N_j$
9.      add an edge $\star$ from $N_j$ to $N_j$
```

In the above algorithm, "$\star$" represents any input. Figure 4 shows an example for a composite subscription composed of three terms: (F, F, S), (F, S) and (F, S).

Complexity Analysis: In Algorithm *BuildStateMachine*, there are two steps to build a state machine: first build the graph for each term, then add transitions between terms. Consider a subscription with m sequence operators of which n are non-contiguous opera-

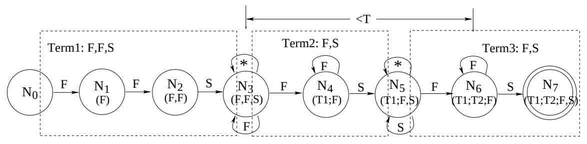
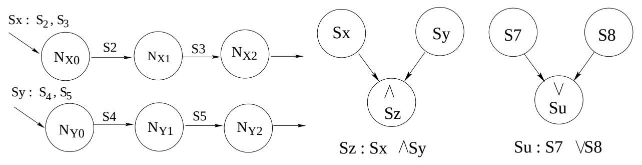
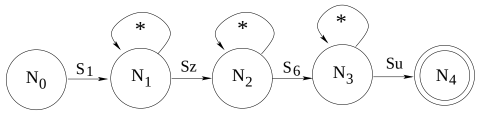


Figure 4: FSM for $F, F, S_1; F, S; F_2@(t_{F_2} - t_{S_1} < T), S$



(a) State machines and Boolean machines



(b) Master state machine for $cs = s_1; ((s_2, s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$

Figure 5: General composite subscription

tors, thereby having $n+1$ terms and m states in total. Suppose the time to create a new state is t_s and the time to add an edge is t_e. When building the state machine for term T_i, we need to create k_i states and add k_i edges. Thus, the time to build a state machine for a term is $(t_s+t_e)*k_i$. In the second step, three edges are added between two sequential terms, and so the second step takes $3nt_e$ time. Therefore, the total time to build a state machine for a subscription is $\sum_{i=1}^{n+1}(t_s+t_e)*k_i + 3t_e*(n-1) = m(t_s+t_e)+3nt_e$. Since $n < m$, the time complexity to build a state machine becomes linear in the number of states: $T_{build\_state\_machine} = O(m)$.

4.2 Combining Boolean/Sequence Operators

A general composite subscription containing both sequence and Boolean operators is decomposed into a set of state machines and Boolean machines. Boolean machines here are simply the expression trees that represent a Boolean expression. The decomposed state or Boolean machines are hierarchically organized and individual machines are referred to as *parent* or *child* machines based on their relationships in the hierarchy. As well, the machine at the root of this hierarchy is called the *master* machine.

As an example, the composite subscription $cs = s_1; ((s_2, s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$ can be represented by three state machines and one Boolean machine as shown in Figure 5. In Figure 5(a), the two state machines labeled S_x and S_y are child machines of the S_z Boolean machine, and the two Boolean machines are in turn children of the master state machine in Figure 5(b). The entire composite subscription is matched when the master machine reaches its matched state.

Special trigger events are used to transition between state and Boolean matching engines: *in-trigger* events are sent by a parent machine to start a child machine, and *out-trigger* events are sent by a child machine to notify the parent of a match. Once a child machine is triggered, it remains active in order to detect more matches.

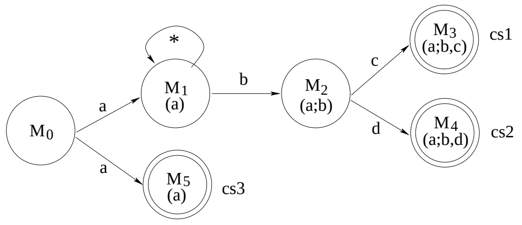


Figure 6: Merging multiple state machines

Matches in the child machine generate derived events that are consumed by the master machine. The child machine only terminates when the master machine terminates, i.e., the whole composite subscription is matched or times out.

For example, in Figure 5, when the master machine transitions to state N_1, it sends an in-trigger to its S_z child Boolean machine to activate it to begin examining the event stream for matching events. Similarly, when the $S_x \wedge S_y$ subexpression matches, the S_z Boolean machine sends an out-trigger that causes the master machine to transition to state N_2.

4.3 Merging Multiple Graphs

When composite subscriptions share common subexpressions, portions of their respective state and Boolean machines can be merged. This saves memory and reduces the matching computation. As merging Boolean expressions has been studied and is not the focus of this paper, we only describe state machine merging here.

Merging state machines should not affect the matching of the associated subscriptions. Two states, N_1 in cs_1 and N_2 in cs_2, can be merged if they are *equivalent*, that is, any time cs_1 arrives at state N_1, cs_2 also arrives at state N_2, and vice versa.
Definition: States N_1 and N_2 are *equivalent* if the following hold: (1) The number of incoming transitions of N_1 and N_2 are equal. (2) The incoming transitions arrive from equivalent states and are triggered by the same set of events. Formally, for every transition $N' \xrightarrow{x} N_1$, there exists a transition $N'' \xrightarrow{x} N_2$, where N' and N'' are equivalent.

Figure 6 shows the result of merging three state machines $cs_1 = a; b, c$, $cs_2 = a; b, d$ and $cs_3 = a$. State M_1 and M_2 are merge states representing the partial matches for both cs_1 and cs_2. Note that states M_1 and M_5 cannot be merged although both are triggered upon an occurrence of event a. M_1 and M_5 are not equivalent because there is another incoming transition (the $\star$ edge) associated with state M_1.

Complexity Analysis: Without merging, the time to build the state machine for a composite subscription with m primitive subscriptions and n non-contiguous operators is $m(t_s + t_e) + 3nt_e$, for a total time of $N[m(t_s + t_e) + 3nt_e]$ when there are N subscriptions. With the merging optimization, subscriptions with common subexpressions can share states. For a set of N subscriptions with a merging degree of $d_m\%$, meaning $d_m\%$ of the states are common, the total time to build the state machines decreases to $N[m(t_s + t_e) + 3nt_e] * d_m\%$.

After merging, all the state machines can be considered as one large graph (perhaps with multiple connected components). The memory required to store the entire graph is the size of the graph, which depends on the total number of states M and the total number of edges L. Therefore, the total space complexity is $O(M+L)$.

5. EVENT PROCESSING

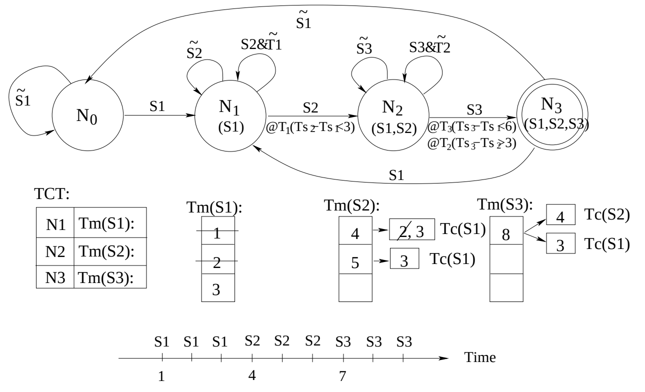


Figure 7: Data structure for subscription $cs = s_1; s_2@(t_{s_2} - t_{s_1} < 3); s_3@(t_{s_3} - t_{s_1} < 6)@(t_{s_3} - t_{s_2} > 3)$

5.1 Matching Algorithm

Composite subscriptions are represented as state and Boolean machines, with each primitive subscription match triggering a transition. During matching, it is important to correctly manage the time information for each matched primitive subscription and to efficiently evaluate the associated time conditions. The discussion below considers cases where the explicit temporal operator references states either within a single FSM or in multiple FSMs.

5.1.1 *Time conditions within an FSM*

Each time condition involves two primitive subscriptions. We refer to the earlier and later matched primitive subscriptions as the *referencing* and *dependent* subscriptions, respectively. In the state machine, the states arrived at when these subscriptions match are called the referencing and dependent states. For example, for a time condition $@(t_{s_2} - t_{s_1}) < d$ between primitive subscriptions s_1 and s_2, s_1 and s_2 are the referencing and the dependent subscriptions, respectively. During matching, all the primary and secondary transitions into the referencing state are recorded so that the time condition can be evaluated when the dependent state is reached.

For each referencing state referred to by a time condition, we associate a time list $T_m(s_i)$ that records when the state is reached (or when the primitive subscription triggering the arrival of this state is matched). New entries will be inserted into the time list when a *primary* transition or a *secondary* transition is taken. Since each state can be referred to by multiple time conditions and each time entry may satisfy a different set of time conditions, we associate a *time compatible set* $T_c(s_i)$ with each entry containing the set of time conditions satisfied by this entry.

Figure 7 shows an example of the data structures to support explicit temporal operators. There are three time conditions in the example involving three subscriptions s_1, s_2, and s_3 (and their associated states N_1, N_2, and N_3), so a time list is maintained for each of these states.

Now suppose the incoming events match the subscriptions at the times indicated in the timeline at the bottom of Figure 7. The first three times (where events match s_1) are inserted into time list $T_m(s_1)$ to record the times when events matched s_1. At time 4, s_2 is matched, but before moving to state N_2, time condition T_1 is evaluated based on the matching time of s_2 ($t_{s_2} = 4$) and the entries in $T_m(s_1)$. Only times 2 and 3 in the list satisfy T_1, so time 1 can be pruned from $T_m(s_1)$. Time 4 is then inserted into $T_m(s_2)$ and times 2 and 3 are inserted into $T_c(s_1)$. Similarly, at time 5, s_2 is matched again, time 5 is inserted into $T_m(s_2)$ and time 3 is inserted into $T_c(s_1)$. At time 6, s_2 is matched again, but since it does not satisfy the time condition T_1, based on the time entries in $T_m(s_1)$, it is discarded. For the same reason, at time 7, s_3 is matched, but it is discarded because time condition T_3 is not satisfied. At time 8, subscription s_3 is matched again, and we evaluate the associated time conditions T_2 and T_3. This time, T_2 is satisfied using s_2's matching time of 4, and T_3 is satisfied using s_1's matching time of 3. Since s_1's matching time of 2 is no longer referenced, it is pruned from the $T_m(s_1)$ and $T_c(s_1)$ lists. Similarly time 5 is deleted from $T_m(s_2)$. At this point, the time lists and compatible lists will be updated as shown in the figure. According to these entries in the lists, we know that the composite subscription is matched using the events at times 3, 4, and 8.

As explained in the above example, the time lists and compatible lists are updated as the state machine transitions occur. This procedure is detailed in Algorithm *Match*.

```
Algorithm Match($FSM, e$)
(* Given a finite state machine $FSM$ and an event $e$ *)
(* update the current state and time lists *)
1.  $N_n = \delta(N_c, e, t)$ //Find the next state $N_n$ based on current
    state $N_c$, event input $e$, current time $t$ and transition function
    $\delta$
2.  if $\exists$ time condition $@T$ refered by state $N_n$
3.      insert time $t$ into $T_m(s_n)$
4.  for each time condition $@T$ dependent on state $N_n$
5.      $N_i = Get\_Ref\_State(@T)$
6.      List $l_m = N_i.get\_time\_list()$
7.      Set matched_times = $eval(@T, l_m, t)$
8.      if !matched_times.isEmpty()
9.          entry = $T_m(s_n).insert(t)$
10.         entry.set($N_i$, matched_times)
11. if $N_n \neq N_c$
12.     prune($N_c$), $N_c = N_n$
13. if $N_c$ is matched state
14.     find the final matched time if no time list is empty
15.     return $N_c$.getMatchedPattern()
```

```
Algorithm Prune($N_c$)
(* Recursively update the time lists when leaving a state *)
1.  List $l_c = N_c.get\_time\_list()$
2.  for each time condition $@T$ associated with incoming transi-
    tion to $N_c$
3.      $N_i = Get\_Ref\_State(@T)$
4.      List $l_i = N_i.get\_time\_list()$
5.      $U = \bigcup_k l_c[k].get(N_i)$
6.      $l_i$.Update( $U$)
7.      Prune($N_i$)
8.  for each time condition $@T$ associated with outgoing transi-
    tion from $N_c$
9.      $N_i = Get\_Dependent\_State(@T)$
10.     List $l_i = N_i.get\_time\_list()$
11.     for each $l_i[k].T_c(s_c)$
12.         $l_i[k].T_c(s_c)$.update( $l_i$)
13.         Prune($N_i$)
```

Upon an incoming event e, we find the primitive matched subscriptions and the next state N_n based on the current state N_c. If there is no time condition associated with this transition, we just move to the next state. Otherwise, we insert the time into the time list of the next state if it is referred by a time condition (N_n is the referencing state of a time condition). If there are time conditions dependent on N_n (N_n is the dependent state), we update its compatible sets. When leaving the current state, we recursively prune all time lists and compatible sets to find out the final matches that satisfy all time conditions. If N_n is a final matched state, we get the associated composite subscriptions and report the result.

Complexity Analysis: There are three main steps during matching: find the next state, evaluate the associated time conditions and prune the time lists. The first step requires a constant time

Table 2: Time condition $@(s_i, s_j)$ for example subscription $cs = s_1; ((s_2; s_3) \wedge (s_4, s_5)); s_6; (s_7 \vee s_8)$.

Ref. State Location ($SM(s_i)$)	Dep. State Location ($SM(s_j)$)	Example
$SM_x(= S_X)$	SM_x	$@(s_2, s_3)$
master	$SM_y(= S_Y)$	$@(s_1, s_2)$
SM_x	master	$@(s_4, s_6)$
$SM_x \neq SM_y \neq$ master $BM(SM_x) = BM(SM_y)(= S_Z)$		$@(s_3, s_4)$
$SM_x \neq SM_y \neq$ master $BM(SM_x)(= S_Z) \neq BM(SM_y)(= S_U)$		$@(s_3, s_7)$

lookup if the transitions are stored in a hash map. For the latter two steps, the processing time depends on the number of associated time conditions as well as the evaluation time and pruning time for each time condition. Suppose there are on average k_1 dependent time conditions and k_2 referencing time conditions per state. For each time condition, the evaluation time depends on the length of the time list l_i. Thus, the processing time for the second step is $\sum_{i=1}^{k_2} |l_i|$. The time lists pruning recurses through each time condition. From the current state, suppose on average, the pruning process is called recursively α times on dependent time conditions and β times on referencing time conditions. For each time list, the pruning time depends on the length of the time list. So in total, the pruning process requires $\alpha \sum_{i=1}^{k_1} |l_i| + \beta \sum_{i=1}^{k_2} |l_i|$ time. Therefore, the time complexity of the whole matching algorithm is $Time(matching) = \sum_{i=1}^{k_2} |l_i| + \alpha \sum_{i=1}^{k_1} |l_i| + \beta \sum_{i=1}^{k_2} |l_i|$.

The above formula indicates that event processing time depends on the length of the time lists. One solution to control the size of the time lists is to expire events and delete their associated entries from the time lists. However, this may result in not detecting certain matches.

5.1.2 *Time conditions across FSMs*

A composite subscription with both temporal and Boolean operators will contain both state and Boolean machines. According to the locations of dependent and referencing states, we classify time condition evaluation into five cases as listed in Table 2. The graphical representation of the example composite subscription in Table 2 is shown in Figure 5.

In Table 2, SM_x represents state machine x, and $BM(SM_x)$ represents the Boolean machine where a reference to state machine x appears. In Case 1, the time condition is defined between two subscriptions in the same state machine. In Cases 2 and 3, the time condition is defined across the child and master state machines. In Cases 4 and 5, the time condition is across two different child state machines that are referred to by the same Boolean machine (Case 4) or by different Boolean machines (Case 5).

For time conditions across different state machines, a global time condition table TCT (as shown in Figure 7) stores the mapping between referencing or dependent states and the corresponding time lists. When the referencing state is reached (e.g., s_i is matched), a new entry is inserted into its time list in TCT. When the dependent state is going to be reached (e.g., s_j is matched), the associated time condition will be evaluated and the time list and its compatible list is updated accordingly. If s_j is not referred to by any later time condition, the time list of s_j is cleaned up. This procedure is applied for Cases 1, 2, 3 and 5 as listed in Table 2.

For Case 4, when the time condition spans state machines with the same parent Boolean machine, there is no order constraint between these two primitive subscriptions. In this case, the time condition is treated as an absolute time difference, of the form $|t_{s_i} - t_{s_j}| < T$. For this type of time condition, we can not fix either state to be the dependent state at which the time condition is to be evaluated. Instead, we evaluate the condition $-T \leq t_{s_i} - t_{s_j} < T$ after both s_i and s_j are matched. To reduce the evaluation time, we insert new entries into the time lists upon the match of s_i and s_j, but we do not evaluate the condition, create the compatible list or perform pruning until we transit into the first common descendant node of the two nodes in the Boolean tree referring to the state machines where s_i and s_j appear.

Notice that an absolute time condition is not possible for Case 5. Since SM_X and SM_Y do not belong to one Boolean machine, there must be a sequence operator between SM_X and SM_Y making an absolute time condition impossible.

5.2 Prediction Algorithm

Each FSM records incremental matches of a pattern, with each successive state representing a further partial match status towards the final full match. If the machine is in state N_c, the probability that it moves to state N_n on the next event depends only on the present state. In other words, the present state fully captures all the information that could influence the future evolution of the process. Therefore, an FSM in our model can be considered as a Markov chain.

We propose a mechanism to leverage the properties of Markov chains to predict the probability of a future match of a composite subscription based on the current state and event history. First, we explain how to assign the transition probabilities for a Markov chain (i.e., a state machine) based on the event history. Then we introduce a conditional probability for a single state machine to reach the matched state given the current state. Finally, we define a probability of future match for a general composite subscription represented by multiple state machines and Boolean machines.

5.2.1 *Markov Chain Model Training*

To calculate the probability of reaching a state, we need to determine the long-run transition probability for the FSM, i.e., the Markov chain.

With this objective, we count the number times each transitions in a state machine is traversed, and use these counters to build a Markov chain whose probability distribution captures that of the event stream.

Definition: Given an FSM represented as a digraph $G = (V, E)$, $e_{ij} \in E$ is an edge from node v_i to node v_j, where $v_i \in V, v_j \in V$ are the states. The transition probability from state i to state j is defined as $p_{ij} = \frac{N_{e_{ij}}}{\sum_k N_{e_{ki}}}$ where $N_{e_{ij}}$ is the number of times that the transition from state i to state j has been taken and $\sum_k N_{e_{ki}}$ is the total number of times that all incoming transitions have been taken, which is also the number of times we have arrived at state i.

Given the transition probabilities between the states, the state machine can be considered as a complete Markov chain model. Since the state space is finite, the transition probability distribution can be represented by *transition matrix* P whose (i, j)th element is $p_{ij} = \Pr(X_{n+1} = j \mid X_n = i)$.

5.2.2 *Prediction for Simple Composite Subscription*

We assume the Markov chain is time-homogeneous, so that the transition matrix P remains the same at each step. The following probabilities can be computed:

(a) The probability of reaching the final matched state over the next n events given a current state. The matrix $P^{(n)} = P^n$ is used to determine the transition probabilities over n steps. The probability α_i of reaching the final matching state M (which represents a match of the whole pattern) from the current state i over the next n events is computed as $\alpha_i^{(n)} = P_{iM}^{(n)}$.

(b) The probability of reaching the final matched state *within* the next n events given a current state. The probability β_i of reaching the final matched state M from state i within the next n events is computed as $\beta_i^{(n)} = \sum_{k \le n} \alpha_i^{(k)} = \sum_{k \le n} P_{iM}^{(k)}$.

5.2.3 Prediction for General Composite Subscription

We define a global state to represent the status of composite subscriptions that require multiple state and Boolean machines.

Definition: A *global state* G is a group of sub-states. For master state machine M and child state machines $C_1, \cdots, C_k$, the global state can be represented as $G_i = \{M.N_{i_M}, C_1.N_{i_{c1}}, C_2.N_{i_{c2}}, \cdots, C_k.N_{i_{ck}}\}$.

The prediction problem for a general subscription with multiple state and Boolean machines is then defined as follows.

Definition: Given the current global state G_i, find the probability of reaching fully matched state G_m in n steps: $Pr^{(n)}(G_m|G_i)$, where $G_i = \{M.N_{i_M}, C_1.N_{i_{c1}}, C_2.N_{i_{c2}}, \cdots, C_k.N_{i_{ck}}\}$ and $G_m = \{M.N_{m_M}, C_1.N_{m_{c1}}, \cdots, C_k.N_{m_{ck}}\}$.

When G_i or G_m consist of active states of child state machines, individual conditional probabilities can be computed for each state machine separately, and these probabilities are then combined according to their relationship in the Boolean machine. Based on probability theory, we give the definition of the conditional probability for a simple composite subscription which contains only two child state machines that are combined by one Boolean operator. In this case, there are two states in the master state machine.

Given a composite subscription $s_G = s_{C_1} \wedge s_{C_2}$ where s_{C_1} and s_{C_2} contain only sequence operators and C_1 and C_2 are two child state machines, suppose the current global state is $G_i = \{C_1.N_{i1}, C_2.N_{i2}\}$, and the matching state is $G_m = \{C_1.N_{F1}, C_2.N_{F2}\}$. Then the probability of reaching the final matched state in n steps is computed as $Pr_{\wedge}^{n}(G_m|G_i) = Pr_{C_1}^{n}(N_{F1}|N_{i1}) * Pr_{C_2}^{n}(N_{F2}|N_{i2})$. If, however, $s_G = s_{C_1} \vee s_{C_2}$, the probability would be $Pr_{\vee}^{n}(G_m|G_i) = Pr_{C_1}^{n}(N_{F1}|N_{i1}) + Pr_{C_2}^{n}(N_{F2}|N_{i2})$.

For composite subscriptions with multiple child state machines and Boolean machines, the conditional probability is decomposed into three steps. The first step considers the transition from the current state $M.N_{i_M}$ to the next state $M.N_{i_M+1}$; the second step the transition from $M.N_{i_M+1}$ to the state just before the global state, $M.N_{j_M-1}$; and the third step the transition from $M.N_{j_M-1}$ to the global state $M.N_{j_M}$.

$$\begin{aligned} Pr^n(G_j|G_i) = \Sigma_{p+q+r=n} \quad & Pr^p(G_i \rightarrow M.N_{i_M+1}) \cdot \\ & Pr^q(M.N_{i_M+1} \rightarrow M.N_{j_M-1}) \cdot \\ & Pr^r(M.N_{j_M-1} \rightarrow G_j) \end{aligned}$$

The first and last function will be expanded according to the combination functions if G_i or G_j contains active child state machines. The second function can be computed based only on the master state machine since it does not involve any active child state machine. Taking Figure 5 as an example, suppose currently s_2 and s_4 are matched. The probability that cs is fully matched in n steps is:

$$\begin{aligned} & Pr^n(G_m|G_i) \\ = \; & Pr^n(M.N_4|X.N_{X_1}, Y.N_{Y_1}) \\ = \; & \Sigma_{p+q=n} Pr^p(M.N_2|X.N_{X_1}, Y.N_{Y_1}) \cdot Pr^q(M.N_4|M.N_2) \\ = \; & \Sigma_{p+q=n} (Pr^p(X.N_{X_2}|X.N_{X_1}) \cdot Pr^p(Y.N_{Y_2}|Y.N_{Y_1})) \cdot \\ & Pr^q(M.N_4|M.N_2) \end{aligned}$$

6. EXPERIMENTS

In this section we experimentally evaluate our approach. All the algorithms are implemented in Java and the experiments are run on a 3GHz Linux machine with 4GB of RAM. We are using two

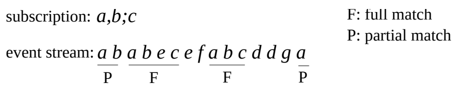


Figure 8: Sample event stream

workloads for experimentation: a synthetic load that lets us independently examine various aspects of our approach by running controlled experiments and a second real-world data set to demonstrate the behavior of our approach under realistic conditions. Our workloads comprise composite subscriptions and events.

The synthetic subscription workload is generated as follows. N_{cs} composite subscriptions are generated, each containing $Length_{cs}$ primitive subscriptions. Primitive subscriptions in turn are generated to match exactly one of the events in a predefined event pool of size N_{ep}. Of the $Length_{cs} - 1$ operators in a composite subscription, $N_{non\_contiguous\_op}$ are uniformly selected to be the non-contiguous sequence operator and the remainder the contiguous sequence operator. By default, $N_{cs} = 1000$, $Length_{cs} = 10$, $N_{ep} = 50$, and $N_{non\_contiguous\_op} = 2$.

Two sets of subscription workloads are used, characterized by the distribution of the primitive subscriptions. In the *uniform_cs* subscription workload, each primitive subscription matches a random event uniformly drawn from the event pool, whereas in the *gaussian_cs* subscription workload, events are selected following a Gaussian distribution.

Since the number of partial and full matches of subscriptions will affect the prediction algorithm, we control the number of partial and full matches of a composite subscription, N_{pm} and N_{fm}, respectively, when generating the event stream workload. As shown in the example in Figure 8, the event stream contains sequences of events that partially or fully match a given subscription. In addition, a number of irrelevant events that do not match any primitive subscriptions are interspersed among the partial and full match sequences. The event stream is generated as follows. First, we determine with 50% probability whether to generate a set of irrelevant events. If so, a sequence of $Length_{irr}$ irrelevant events are appended to the event stream. Otherwise, one of the composite subscriptions whose quota of partial or full matches (N_{pm} or N_{fm}) has not been met is randomly selected, and a sequence of events that correspond to either a partial or full match is generated. If a full match is required, the necessary set of events is added to the event stream, otherwise a partial match of length $Length_{pm}$ is appended to the stream. By default $N_{pm} = 20$, $N_{fm} = 20$ and $Length_{irr}$ is a uniform distribution in the range $[1, 100]$.

Two sets of event stream workloads are used, characterized by the distribution of the partial match lengths, $Length_{pm}$. In the *uniform_pub* workload, $Length_{pm}$ follows a uniform random distribution in the range $[1, Length_{cs} - 1]$, whereas in the *gaussian_pub* workload it varies with a Gaussian distribution over the same range.

It should be noted that the event stream workload only loosely controls the length and number of partial and full matches of the composite subscriptions. In reality, the events generated to achieve a partial or full match of a particular composite subscription may contribute to matching other composite subscriptions.

In the following experiments, all measurements are performed after the subscriptions have been inserted into the matching engine. To evaluate the matching algorithm, the parameters of interest include the effect of the number of composite subscriptions, their lengths, the number of non-contiguous operators and the size of the event pool (i.e., more or less diverse primitive subscriptions,) For the prediction algorithm, the evaluated factors include the ratio of full matches to partial matches, the number of prediction steps, and the prediction threshold.

6.1 Matching Performance

Number of subscriptions: Figure 9(a) shows the number of states with increasing number of subscriptions. We see that the number of states grows linearly as the number of subscriptions increase. Furthermore, the rate of increase for a small event pool size (i.e., fewer primitive subscriptions) is less than that of a larger event pool size (i.e., more primitive subscriptions). This is because there are more merged common states with a smaller number of distinct primitive subscriptions in the workload. Similarly, for the *gaussian_cs* workload, the workload exhibits more locality and fewer distinct primitive subscriptions are involved and hence there are even more common states than in the *uniform_cs* workload.

Figure 9(b) shows the average time to process one event given a fixed set of subscriptions. As the subscription size increases, so does the matching time. Unlike for the number of states, given a fixed number of subscriptions, the matching time is larger for a workload with fewer distinct primitive subscriptions (i.e., a smaller-sized event pool) or *gaussian_cs* subscriptions where the workload share more common states. This is because with a larger number of common states, a given event may trigger more transitions, thus requiring more processing time.

Number of non-contiguous operators: Figure 9(c) shows that as the number of non-contiguous operators increases, so does the matching time. This is because more subscription instances remain partially matched, waiting for events to trigger their transitions.

Composite subscription length: Figure 9(d) shows the effect of increasing the length of composite subscriptions on merging. We plot the commonality, which is represented by the ratio between the number of shared states with merging and the total number of states without merging. We see that the commonality decreases with increasing subscription length. In our FSM model, each state in the FSM is defined by the prefix of the subscription. For a longer subscription, there are much more combinations to select events to generate a subscription compared to a shorter subscription. This explains why the commonality degree decreases with the increase of the subscription length. This experiment also shows that the merging process favors shorter subscriptions.

Number of distinct primitive subscriptions (size of event pool): The remaining figures show the effect of number of distinct primitive subscriptions on merging and matching. Figure 9(e) shows that the number of states increases as the number of distinct primitive subscriptions increases for both *uniform_cs* and *gaussian_cs* workloads. As the number of distinct subscriptions increases, the number of states for both workloads increases. This is because the number of shared common states among subscriptions decreases with increasing number of distinct subscriptions. The *gaussian_cs* workload results in more locality, that is among the larger number of subscriptions there are fewer distinct subscriptions, relative to the same number of subscriptions in the *uniform_cs* workload, therefore, the *gaussian_cs* workload results in fewer states than the *uniform_cs* workload.

Figure 9(f) shows that the number of transitions decreases with the increasing number of distinct subscriptions for different workloads. Recall that the number of shared states among subscriptions decreases with increasing number of distinct subscriptions. Hence, there are fewer instances during the matching process and so fewer transitions as well. The Gaussian publication workload contains shorter partial matches than the workload generated by the uniform distribution, which also results in fewer transitions.

6.2 Prediction Performance

We evaluate our prediction algorithm on three important metrics: false positives, true positives and precision, which is defined as the ratio between true positives and all predictions. We look at the effect of the number of lookaheads, the threshold, and the ratio between the number of full matches and partial matches, and at the effect of the workload distribution. The prediction results are shown in Figure 10.

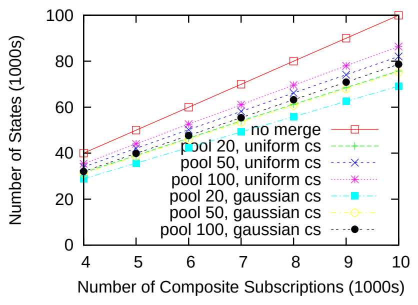
Synthetic workload: First, we compare the precision results when using the same workload to train our model, but test the prediction algorithm on different workloads. The workloads are different in terms of the increased number of partial matches (i.e., decreased ratio of the number of full matches and partial matches) in the event stream. In Figure 10(a), we can see that the precision decreases as the number of lookaheads increases. Also, the precision increases with the increase of prediction threshold but it stabilizes for larger thresholds. The precision decreases when the test file contains more partial matches.

Second, we compare the precision results when using different workloads for training, but test on the same workload. In this experiment, the workloads are different in terms of the event distribution; all the other parameters are the same. Figure 10(b) shows the precision when testing on the same workload as training. Figure 10(c) shows the precision when training with a Gaussian event stream and testing on a uniform event stream. Comparing these two figures, we see that the precision decreases when the training workload and testing workload are not consistent. An additional discovery is that the precision converges for different number of lookaheads in Figure 10(b). We can conclude from this result that if we train and test on the workload with the same distribution, an increase in the number of lookahead steps does not decrease the quality of prediction for larger thresholds.

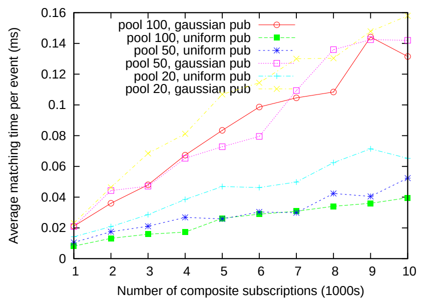
Realistic workload: In order to validate the practicality of our approach, we evaluated with a real-world data set, where the daily temperatures are monitored and an alert is announced when the temperature is high for several consecutive days, perhaps indicating a heat wave. We use real average daily temperature data for 157 U.S and 167 international cities.[1] We define the subscription as $s = (T > 30)$, $cs = s, s, s, s, s$, where it represents the case the temperature is higher than 30 degree for five consecutive days. The generated model is built based only on the composite subscription, without any knowledge of the event data. In the generated model, there are only 5 states and each state represents one more hot day than the previous state. For comparison, we divide the temperature into 4 categories separated by $30, 20, 10$ degrees and manually build a *full* Markov model consisting of all temperature combinations from one day to 5 days. There are 1365 states in total. We ran our predicting algorithm for these two models and the results are shown in 11.

Comparing Figure 11(a) with 11(d), and Figure 11(b) with 11(e), the generated model makes more predictions than the full model, including both false and true positives. However with longer lookahead, both false and true positives drop to zero faster in the full model when increasing the prediction threshold. This is because the additional states in the full model capture more information about the event history, allowing it to better distinguish how a partial match state was reached. For prediction precision, Figures 11(c) and 11(f) show that when increasing the lookahead, the generated model performs much better than the full model. Furthermore, the

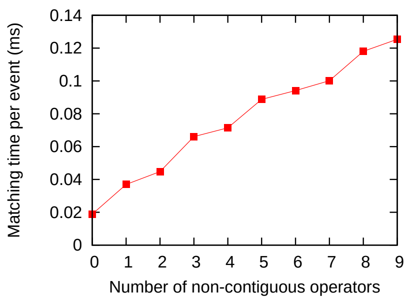
[1]Source data from the Global Summary of the Day database is available at `http://www.engr.udayton.edu/weather/source.htm`.



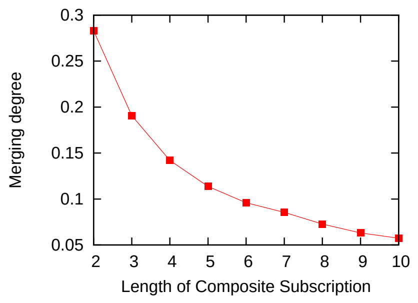
(a) #States vs. #subscriptions



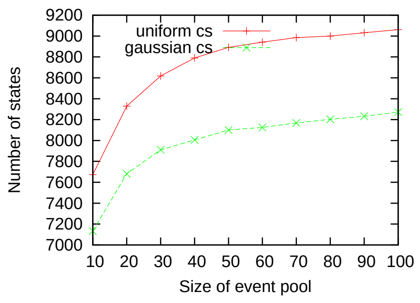
(b) Matching time vs. #subscriptions



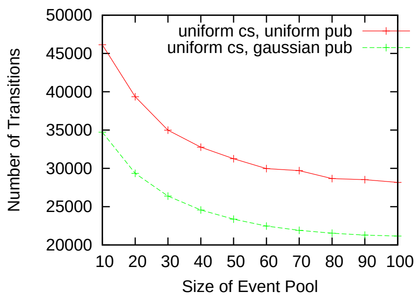
(c) Matching time vs. #operator



(d) Merging degree vs. sub length



(e) #States vs. pool_size



(f) #Transitions vs. pool size

Figure 9: Matching performance

number of states in our generated model is significantly fewer than that in the full model, and so the prediction algorithm runs much faster and consumes less memory with the generated model than with the full model. As well, the memory and computation cost to maintain the transition probability matrix is also much less in the generated model.

7. CONCLUSIONS

This paper presents a new publish/subscribe model that, in addition to reporting when a subscription has matched, predicts the likelihood a subscription will match in the future. The system requires no additional subscriptions to be defined, and dynamically adapts its prediction to the application workload.

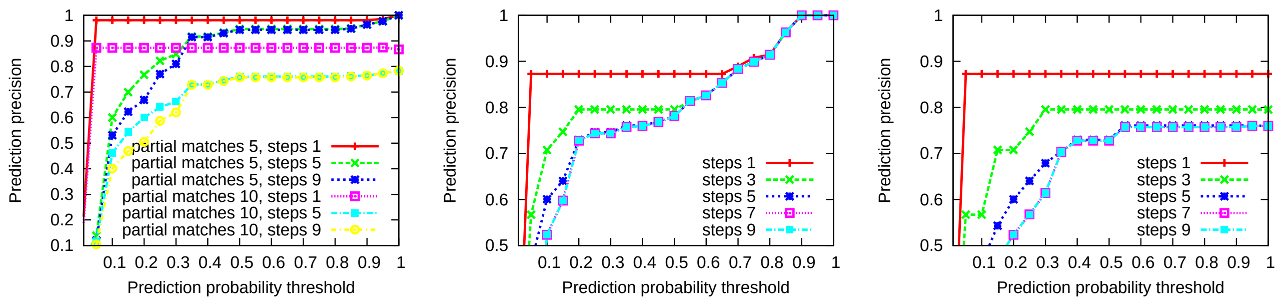
Content-based publish/subscribe semantics are supported including both temporal and Boolean operators. The temporal subexpressions of a subscription are stored as finite state machines, and the Boolean parts as Boolean trees, thereby avoiding state explosion that would result from representing Boolean operators in the state machine. Furthermore, some common subexpressions are merged to reduce memory consumption and matching efficiency.

The finite state machines, which are also Markov chains, are trained using an application's event history, and model the transition probabilities among the partial match states of the subscription. The Markov model is then used to predict the probability of a subscription matching within some number of transitions in the future given its current partial match state.

Experiments show that the memory and computation performance of the prediction algorithms scale with the number of subscriptions, and that the merging optimizations can significantly reduce the state machine sizes especially when subscriptions exhibit locality. Moreover, as expected, the quality of the predictions, in terms of its precision, improves when only higher probability predictions are considered, and as the lookahead into the future is decreased. Also, interestingly, when the system is trained and tested using the same distribution of events, the prediction precision for different lookahead distances improves and converges. When the system is trained using a different set of data, however, while the precision may still be high, increasing the lookahead does impair the precision. Therefore, if an application event distribution does not fluctuate rapidly and a user is only interested in highly precise predictions, there is no harm in asking the system to make predictions further into the future.

The prediction algorithms are compared to a hand-crafted Markov model for a real application workload. The predictions in the hand-crafted model are expected to be better, but requires expertise to construct and is much more expensive computationally and memory-wise. Results show that despite being much cheaper to evaluate, the prediction precision of the model in this paper is not much worse than the hand-crafted one. In fact, when the lookahead is increased, our model performs much better.

As this is the first paper to propose predictive matching capabilities in a publish/subscribe system, there are many opportunities to extend the work. One avenue we plan to explore is to support matching and prediction where the subscription may not be precisely defined. For example, in an intrusion detection system, there may be many ways to perform an intrusion that are similar to, but not exactly like, the intrusion signature defined by an administrator.



(a) Precision vs. #partial matches (b) Precision for uniform workload (c) Precision for gaussian workload

Figure 10: Experimental results for prediction

8. REFERENCES

[1] M. K. Aguilera et al. Matching events in a content-based subscription system. In *PODC*, 1999.

[2] L. Antova et al. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.

[3] O. Benjelloun et al. An introduction to ULDBs and the Trio system. *IEEE Data Eng. Bulletin*, 2006.

[4] A. Carzaniga et al. Forwarding in a content-based network. In *SIGCOMM*, 2003.

[5] A. Compailla et al. Efficient filtering in publish-subscribe system using binary decision diagrams. In *ICSE*, 2001.

[6] G. Cugola et al. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE TSE*, 2001.

[7] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 2007.

[8] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 2007.

[9] Y. Diao et al. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

[10] F. Fabret et al. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.

[11] T. Fawcett et al. Activity monitoring: Noticing interesting changes in behavior. In *SIGKDD*, 1999.

[12] L. Fiege et al. Engineering event-based systems with scopes. In *ECOOP*, 2002.

[13] N. Fuhr and T. Rolleke. A probabilistic relational algebra for integration of information retrieval and database systems. *ACM TOIS*, 1997.

[14] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.

[15] N. Khoussainova et al. PEEX: Extracting probabilistic events from RFID data. In *ICDE*, 2008.

[16] I. Koenig. Event processing as a core capability of your content distribution fabric. In *Gartner EP Summit, Orlando, Florida*, 2007.

[17] C. L.Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

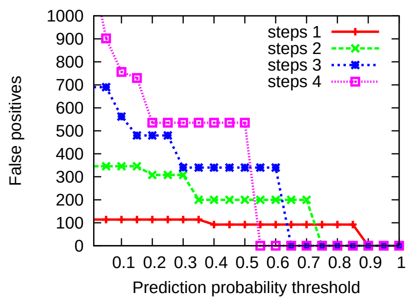
[18] G. Li et al. A unified approach to routing, covering and merging in publish/subscribe systems based on MBDDs. In *ICDCS*, 2005.

[19] G. Li et al. Adaptive content-based routing in general overlay topologies. In *ACM Middleware*, 2008.

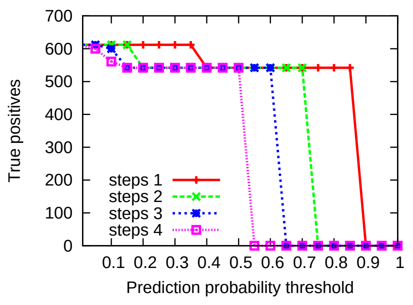
[20] I. Rose et al. Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds. In *NSDI*, 2007.

[21] S. Singh et al. Orion 2.0: Native support for uncertain data. In *SIGMOD*, 2008.

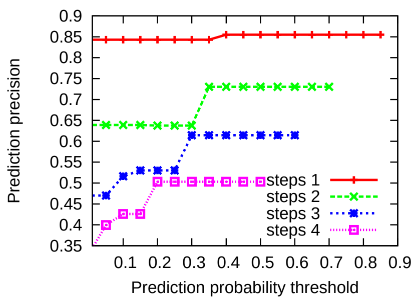
[22] S. Wasserkrug et al. Complex event processing over uncertain data. In *DEBS*, 2008.



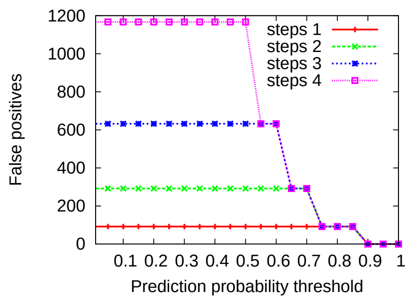
(a) False positives (full model)



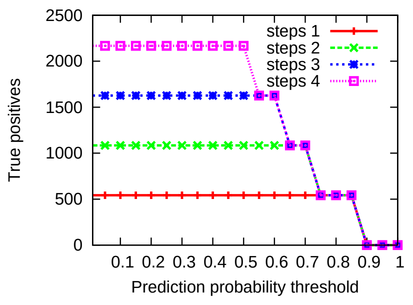
(b) True positives (full model)



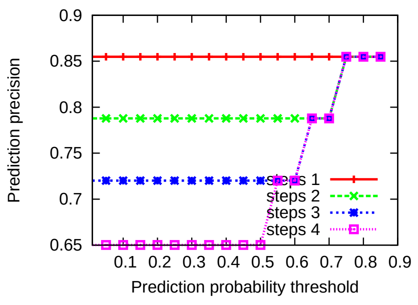
(c) Prediction precision (full model)



(d) False positives (generated)



(e) True positives (generated)



(f) Prediction precision (generated)

Figure 11: Evaluation on real data, compared with a model with full knowledge