Scalable Efficient Composite Event Detection[⋆]

K. R. Jayaram and Patrick Eugster

Department of Computer Science, Purdue University
{jayaram, peugster}@cs.purdue.edu

Abstract. Composite event detection (CED) is the task of identifying combinations of events which are meaningful with respect to program-defined patterns. Recent research in event-based programming has focused on language design (in different paradigms), leading to a wealth of prototype programming models and languages. However, implementing CED in an efficient and scalable manner remains an under-addressed problem. In fact, the lack of scalable algorithms is the main roadblock to incorporating support for more expressive event patterns into prominent event-based programming languages. This lack of scalable algorithms is a particularly acute problem in event stream processing, where event patterns can additionally be specified over time windows. In this paper we describe GenTrie, a deterministic trie-based algorithm for CED. We describe how complex event patterns are split, how each sub-pattern maps to a node in the trie, and demonstrate through empirical evaluation that GenTrie has higher throughput than current implementations of related languages.

1 Introduction

An event-based system consists of a set of software components that interact by notifying each other of events, where an event is any happening of interest – typically a change in the state of a component. Mouse clicks, keyboard events, timers, OS interrupts, sensor readings, stock quotes, and news articles are all examples of events. Events have data attributes attached to them. A stock quote, for instance has the name of the corporation and the price of the stock as attributes.

Event-based programming aims to eliminate coupling as much as possible in a software system, thereby reducing its overall complexity and making it easier to understand, debug, test, and maintain. Event-based programming underpins a wide variety of software – operating systems, graphical interfaces, news dissemination, algorithmic stock/commodity trading, network management and intrusion detection.

Simple event handling of *singleton* events is supported in most mainstream programming languages through libraries – examples are Java's JFC/Swing, RTSJ's AsyncEvents and C's POSIX condition variables. The event handler, also called a *reaction*, is executed when a corresponding event occurs, and is often

[⋆] This work is supported by NSF grants 0644013 and 0834619.

a method, executed asynchronously to the caller. Reacting to simple events, though common, is not sufficient to support a growing class of event-based applications which are centered around *patterns* of events, called *composite* (or *complex*) events. Composite events are defined by quantification over combinations of events exhibiting some desired relationships.

The efficacy of an event-based programming language is governed by (R1) its expressivity, i.e., the ability of the language to precisely capture the complex event patterns of interest and, (R2) the ability of the runtime to match events to patterns in a timely and efficient manner. As expected, R1 and R2 are related, and many language designers choose to restrict patterns to things that can be efficiently implemented with standard data structures and custom off the shelf components. The lack of efficient algorithms for *composite event detection* (CED) turns out to be a main reason why many programming languages like Polyphonic C# [1] or JoinJava [5] only support limited CED through simple event *joins* without predicates on event attributes. Detecting such unpredicated joins is possible in $O(1)$ [1].

The lack of efficient algorithms for event correlation is especially acute in distributed event *stream* processing, where events of different types arrive at different independent rates. If an event pattern p involves predicates on two types of events e_1 and e_2, with e_1 occurring more frequently than e_2, events of type e_1 may satisfy some of the predicates of p, thus *partially* matching p. In this example, an event-processing algorithm must store partially matched events and decide if and when events of any type expire, i.e., how long events of type e_1 should be stored waiting for event e_2 to occur. Research in event-based programming has focussed on programming language design, semantics of event joins and concurrency issues in dispatching reactions, but there is little research on scalable and efficient algorithms for actual event matching.

Fueled by these observations, this paper makes the following contributions:

1. An abstract model and a formal definition of CED.
2. An original *trie*-based algorithm for CED called GenTrie.
3. An empirical evaluation of GenTrie, compared to existing solutions.

While 1. allows us to compare the expressiveness of existing event-based programming languages and systems, our objective is not to determine whether any one is better than the others. Other distinctions may also exist (e.g. support for distribution, persistence, synchronous events) in the semantics. Concisely defining the problem of CED allows us to present our algorithm in an abstract manner, so that it can be used or adapted by any compiler designer for any existing or future language.

Roadmap. Section 2 introduces our model of CED and uses it to summarize related systems and languages for event-based programming. Section 3 presents GenTrie. Section 4 evaluates GenTrie on several benchmark programs introduced by others as well as through stress-testing to illustrate its scalability. Section 5 discusses options. Section 6 draws final conclusions.

2 Problem Description and State of the Art

2.1 Events, Patterns and Reactions

An event, sometimes explicitly referred to as a *simple* event to disambiguate it from more general *composite* (or *complex*) events, is a change in the state of a system component. Events have (type) names and data attributes. As an example, StockQuote(organization: ''IBM'', price: 56.78, opening: 57.90) represents a stock quote event. Events can be typed like methods, which can be exploited by representing event types and events by event method headers and event method invocations respectively. The signature of StockQuote events for instance can be represented as StockQuote(String organization, **float** price, **float** opening). A composite event is any *pattern* of simple events, i.e., it is a set of events satisfying a specific *predicate*. Examples are: (1) the average of 100 consecutive temperature readings from a sensor, (2) a drop in the price of a stock by more than 30% after a negative analyst rating, (3) the price of a stock exceeds its 52-week moving average, and (4) the debut of a new Volkswagen car with 5 positive reviews.

A *reaction* is a program fragment (usually a method body), executed by a software component upon receipt of a simple or composite event. In the case of a composite event, all simple events that are part of it are said to *share* the reaction (method body).

We use e and its indexed variants to range over event names, which are uniquely associated with types as well as events. The meaning of e will be clear from the context. For example $e(T_1\ a_1, T_2\ a_2)$ refers to an event type, and $e(T_1 : v_1, T_2 : v_2)$ refers to an event. a and T refer to data attributes and their types respectively. We use v to range over values, which for the sake of brevity, are assumed to be strings, integers, or floats. In event type $e_i(T_{i,1}\ a_{i,1}, ..., T_{i,r}\ a_{i,r})$, $a_{i,j}$ is a data attribute of type $T_{i,j}$. We also assume for simplicity in the following that any event has at least one attribute.

2.2 Formal Definition of Composite Events

As mentioned, an event pattern describes a composite event as a set of events with a predicate they must satisfy. The BNF syntax of an event pattern $\mathcal{P}$ is:

attributes	$t\ ::= T\ \ a \mid t,\ T\ \ a$
join	$j\ ::= e(t)[v] \mid j, e(t)[v]$
condition	$b\ ::= true \mid e[\mathsf{i}].a\ op\ v \mid e[\mathsf{i}].a\ op\ e[\mathsf{i}].a$
predicate	$p\ ::= \forall \mathsf{i} \in [v,v]\ p \mid p\ \&\&\ p \mid p\ \Vert\ p \mid (p) \mid !p \mid b$
boolean operator	$op ::= > \mid < \mid >= \mid <= \mid == \mid !=$
pattern	$\mathcal{P}\ ::= j\ \textbf{if}\ p$

The general form of an m-*way* event pattern $\mathcal{P}$, i.e., with m event types, where e_i contains r_i attributes is thus:

$$\mathcal{P} \;=\; e_1(T_{1,1}\ a_{1,1}, ..., T_{1,r_1}\ a_{1,r_1})[k_1], ..., e_m(T_{m,1}\ a_{m,1}, ..., T_{m,r_m}\ a_{m,r_m})[k_m]\ \ \textbf{if}\ \ p$$

We refer to k_i as *window size* of event type e_i in $\mathcal{P}$. It refers to the number of events of type e_i that are part of the composite event $\mathcal{P}$. An event in the window can be referred to in the predicate by using indices $1, ..., k_i$.

A *unary* condition compares an event attribute to a value; a *binary* condition compares two event attributes. *Intra-event* conditions are either unary conditions or binary conditions comparing two attributes of the same event type. *Inter-event* conditions are binary conditions comparing attributes of two distinct event types. More generally, a predicate is *n-ary* if the largest set of event types related transitively by inter-event conditions is of size n. For a predicate consisting only of intra-event predicates, n is trivially 1. We focus in the following on *well-formed* patterns, such that event attributes are compared to values of same types, and $n \leq m$.

As a concrete example, the composite event "*the release of a new Volkswagen Jetta with 5 positive reviews*" can be formally specified as:

$$\mathcal{P}_{vw} \quad = \quad \left\{ \begin{array}{c} \mathsf{VWRelease(String\ model, String\ date)[1],} \\ \mathsf{Review(String\ model, String\ textReview, float\ rating)[5]} \end{array} \right\} \quad \textbf{if}$$

$$\left(\begin{array}{lcr} \forall\ \mathsf{i}\ \in\ [1,5] & \mathsf{Review[i].rating}\ >\ 3.5 & \&\& \\ & \mathsf{VWRelease[1].model} == \mathsf{"Jetta"} & \&\& \\ \forall\ \mathsf{i}\ \in\ [1,5] & \mathsf{Review[i].model{==}VWRelease[1].model} & \end{array} \right)$$

Given a set $\mathcal{E}$ of events of t types $e_1, ..., e_t$, and an m-way event pattern $\mathcal{P}$, *composite event detection* (CED) is defined as the problem of finding a set $\mathcal{E}' \subseteq \mathcal{E}$, such that p is satisfied. We discuss limitations of our syntax of composite subscriptions together with semantic choices in Section 5.

Note that the events in a window do *not* have to be *consecutive* or *subsequent* – i.e. event $e[i]$ and $e[i+1]$ do not have to be consecutive, and $e[i]$ does not have to occur before $e[i+1]$. This however does not mean that windows of consecutive or subsequent events cannot be specified using the above syntax. A window of consecutive Review events can be specified by assuming that each event has a sequence number seq as one of its attributes, and by adding the following condition to the predicate: $\forall\ \mathsf{i}\ \in\ [1,4]\ \ \mathsf{Review[i].seq} <= \mathsf{Review[i+1].seq}$. A similar condition can be used to specify subsequent events, assuming that the time of occurrence of the event time is one of its arguments:
$\forall\ \mathsf{i}\ \in\ [1,4]\ \ \mathsf{Review[i].time} <= \mathsf{Review[i+1].time}$.

2.3 Event Joins

Event joins (or just joins) are one of the most common forms of composite events supported in programming languages; they are characterized by predicates of the form *true* and window sizes of $k = 1$. Joins were popularized by languages based on the *join calculus* [18] which predominantly reify events through asynchronous methods. A join has a method body associated with it, which is executed in a separate thread when all the m events in the join occur. The detection of a join is $O(1)$ and reaction dispatch can be performed in $O(m)$, e.g., as described below:

1. A queue q_i is used to store all events of type e_i.
2. Associate a bit array B of size m with an m-way join, where bit B_i corresponds to e_i in the join.
3. Upon arrival of an event of type e_i, enqueue it in q_i, and if now $|q_i| = 1$, set B_i to 1.
4. Detecting a join involves checking whether each bit in the bit array is 1. This can be accomplished in $O(1)$ time through a logical OR operation, i.e., checking whether B OR 0x0 $= 2^m$.
5. When an event join is detected, dequeue each event e_i from queue q_i. If now $|q_i| = 0$, set B_i to 0. This can be done in $O(m)$. The dequeued events are then consumed by the reaction executed in a separate thread.

Note that sometimes one synchronous event is permitted per join, in which case the reaction can be piggy-backed on the thread corresponding to that event. The presence of synchronous events does not affect the complexity of detecting event joins. Similarly, certain languages support unicasting of events while others offer multicast or both. This does not affect the detection complexity either, though some redundancy might be avoidable if a set of receivers have the exact same patterns and all corresponding events are multicast.

2.4 State of the Art of CED

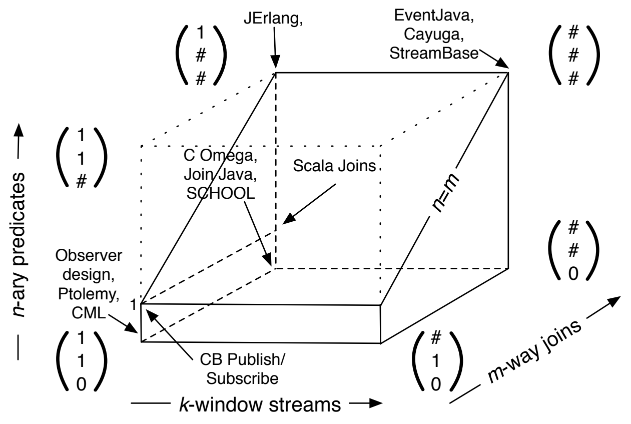


Fig. 1: Overview of features of event-based programming languages and systems. # represents the absence of a bound.

Patterns in CED can be considered along three dimensions according to which programming languages and systems for CED can be classified (see Figure 1):

1. The (maximum) window sizes k for streams of individual event types (k-size windows).
2. The (maximum) number m of correlated event types ($m-$way joins).
3. The (maximum) number of event types n involved in predicates ($n-$ary predicates).

The first two dimensions can be viewed as representing time and space dimensions respectively and are clearly orthogonal. The third dimension, as mentioned, is not independent of the space dimension which leads to the division of the 3-dimensional space in Figure 1.[1] # stands for "unbounded".

Based on the admitted values for $\langle k, m, n\rangle$ we can coarsely classify languages and systems as follows (due to space limits the list is not exhaustive):

Simple event handlers $\langle 1, 1, 0\rangle$. The observer design pattern and most library-based event handlers or simple languages like Ptolemy [8] support reactions to single event instances, without predicates. Languages which support only *staged* correlation where the consumption of a first event conditions the consumption of second one etc. also fall into this category (e.g. CML [7]).

Simple predicated event handlers $\langle 1, 1, 1\rangle$. This includes content-based publish/subscribe multicast systems such as Siena [9] or languages inspired by the model (e.g., ECO [10], Java$_{PS}$ [11]), as well as Actor-based languages or Actor libraries supporting predicates on individual messages (e.g., Erlang [2], Scala Actors [4], AmbientTalk [12]).

Join languages $\langle 1, \#, 0\rangle$. This category corresponds to the join calculus family. Examples are given by JoinJava [5], SCHOOL [6], Russo's library for VisualBasic [17], or Polyphonic C# [1] – now Cω. The work of Sulzmann et al [13] is another citizen of this class.

Predicated join languages $\langle 1, \#, \#\rangle$. Second generation join languages (e.g., JErlang [16]) support n-ary predicates but no streams. Scala Joins [3] are special, isolated, case of predicated join language supporting only intra-event conditions ($\langle 1, \#, 1\rangle$).

Generic correlation $\langle \#, \#, \#\rangle$. Database-derived systems such as Cayuga [14], Borealis [15] or the commercial StreamBase (www.streambase.org) support all features of CED. EventJava [2] mirrors these at the language level. Several content-based publish/subscribe systems have been extended for generic correlation, such as PADRES [19] which uses Jess [21].

3 GenTrie

In this section, we describe GenTrie, an algorithm that constructs an event-flow graph as a *generalized trie* to detect composite events.

[1] n could be defined as the total number of involved *events*, but we have not encountered any systems supporting streams *without* predicates (or without joins), thus n depends here only on m and not on k.

[2] http://www.erlang.org

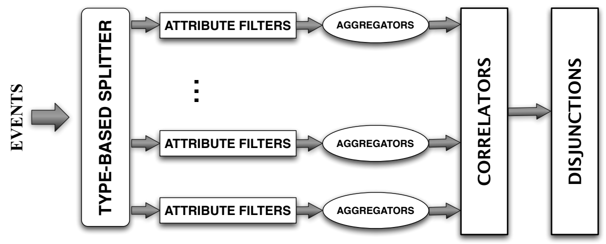


Fig. 2: An overview of GenTrie

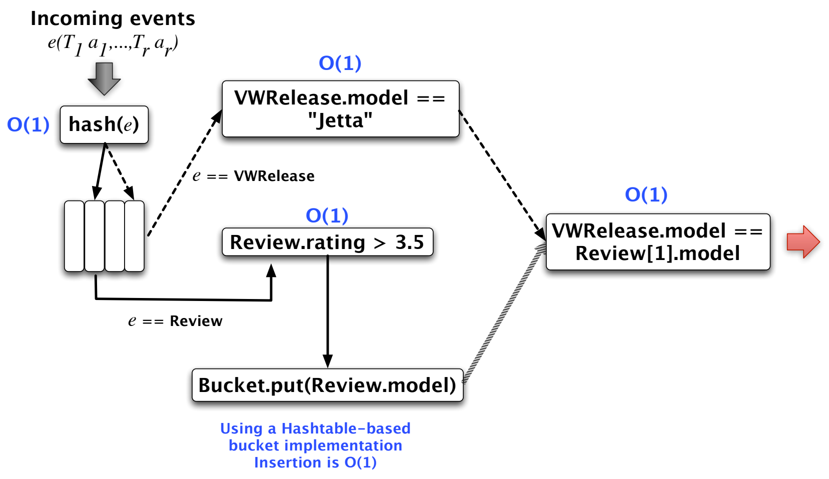


Fig. 3: GenTrie by example – "Release of a new Volkswagen Jetta with five positive reviews".

3.1 Overview

Each node of the trie corresponds to a (intra- or inter-event) condition of a pattern. Figure 2 gives a high-level overview of GenTrie, whose stages are explained in this section through the "Volkswagen" Jetta example outlined in Section 2. Figure 3 illustrates the stages schematically. The five stages of GenTrie are:

1. *Type-based event filtering and splitting*: The input to this stage is a heterogenous input stream of simple events of different types, from which composite events have to be detected. In this stage, the input stream is split into several sub-streams for *individual event types*.
2. *Attribute filtering*: The input to this stage is a given stream of events of the same type. In this stage, each event is matched against all corresponding *intra-event conditions*, keeping track of the satisfied ones.

3. *Aggregation*: Some *inter-event conditions* aggregate *events of the same type* in corresponding windows. In this stage, events of the same type involved in such aggregations are grouped together.
4. *Correlation*: This stage combines event streams of different types and evaluates the remaining *inter-event conditions* on events, and groups of events of *different types*.
5. *Disjunctions*: For predicates involving disjunctions, this stage combines events that satisfy each of their components.

3.2 Predicate Simplification

Predicate simplification consists of removing quantification ($\forall$) and negation (!) and is done as a pre-stage preferably at compilation. Quantification can be simplified by the following equivalence rules:

$$\begin{array}{ll}
\forall\ i\ \in\ [v_1, v_2]\ (p_1\ \&\&\ p_2) & \doteq \forall i \in [v_1, v_2]\ p_1\ \&\&\ \forall\ i\ \in\ [v_1, v_2]\ p_2 \\
\forall\ i\ \in\ [v_1, v_2]\ (p_1 \,||\, p_2) & \doteq \begin{pmatrix} (\{{}^{v_1}/_{\mathrm{i}}\}p_1 \,||\, \{{}^{v_1}/_{\mathrm{i}}\}p_2)\ \&\& \\ \dots\qquad\qquad \&\& \\ (\{{}^{v_2}/_{\mathrm{i}}\}p_1 \,||\, \{{}^{v_2}/_{\mathrm{i}}\}p_2) \end{pmatrix} \\
\forall\ i\ \in\ [v_1, v_2]\ (e_1[i].a_1\ op\ e_2[i].a_2) & \doteq \begin{pmatrix} e_1[v_1].a_1\ op\ e_2[v_1].a_2\ \&\& \\ \dots\qquad\qquad \&\& \\ e_1[v_2].a_1\ op\ e_2[v_2].a_2 \end{pmatrix} \\
\forall\ i\ \in\ [v_1, v_2]\ (e[i].a\ op\ v) & \doteq \begin{pmatrix} e[v_1].a\ op\ v\ \&\& \\ \dots\qquad \&\& \\ e[v_2].a\ op\ v \end{pmatrix}
\end{array}$$

Negation can be removed easily using the following rules:

$$\begin{array}{ll}
!(p_1\ \&\&\ p_2) \doteq\ !p_1 \,||\, !p_2 & !(p_1 \,||\, p_2) \doteq\ !p_1\ \&\&\ !p_2 \\
!!p \doteq p & !(\forall i \in [v_1, v_2]\ p) \doteq \{{}^{v_1}/_{\mathrm{i}}\}!p \,||\, \dots \,||\, \{{}^{v_2}/_{\mathrm{i}}\}!p \\
!(e[i].a{<}e'[i].a') \doteq e[i].a{>=}e'[i].a' & !(e[i].a{<=}e'[i].a') \doteq e[i].a{>}e'[i].a' \\
!(e[i].a{>}e'[i].a') \doteq e[i].a{<=}e'[i].a' & !(e[i].a{>=}e'[i].a') \doteq e[i].a{<}e'[i].a' \\
!(e[i].a{==}e'[i].a') \doteq e[i].a!{=}e'[i].a' & !(e[i].a!{=}e'[i].a') \doteq e[i].a{==}e'[i].a'
\end{array}$$

Now we only have conjunctions and disjunctions left in the predicate. Since we can do away with negation by changing individual conditions it is easy to see that any predicate can be be transformed into a disjunctive normal form (DNF), i.e., of the form $p_1 \,||\, \dots \,||\, p_n$ where each p_i is a conjunction of several conditions.

3.3 Type-based Event Filtering and Splitting

The objective of this stage is to split a single input event stream into several streams, one corresponding to each type. This allows for the early application of intra-event conditions which involve attributes of a single event type. Filtering out events of no interest reduces the load on the correlation module. To efficiently implement this stage, a hash table is used. The key of the hash table is the event type, and the value consists of a filter node explained below. The hash of an event type (which is a String) can be performed in constant time [22]. This stage also adds a sequence number to each non-filtered event, to represent the order in which the algorithm received input events. In the example, this means separating out all events of types VWRelease and Review into separate sub-streams.

3.4 Attribute Filtering

A pattern may contain several intra-event conditions, each of which compares an attribute of an event type to a value or another attribute of the same event. If there are l_i intra-event conditions for event type e_i, a sequence of l_i filter nodes (in any order) is constructed, the output of each node being the input of the next. A pointer to the first attribute filter is stored in the hash table, as explained above. Each attribute filter processes an input event in constant time. Thus, if an event pattern $\mathcal{P}$ has m event types, and l_i intra-event conditions for event type e_i, the total number of attribute filters created are $\sum_{i=0}^{m} l_i$. Figure 3 illustrates how all intra-event conditions are linked to each other in the case of the running example.

3.5 Aggregation

In this stage, an aggregation node is created to process certain windows of events. Creating aggregation nodes is an optimization of GenTrie. GenTrie can detect composite events without aggregation nodes, but aggregation nodes simplify the detection of certain event windows. If an inter-event condition is of the form $\forall i \in [i_1, i_2] e[i].a \; op \; v$, or $\forall i \in [i_1, i_2] e[i].a \; op \; e'[1].a'$, where *op* is either $==$ or $!=$, then aggregation nodes are created. This is done by *bucketing* events, similar to the strategy used by a Bucket Sort or Bin Sort algorithm [22].

In the example, this step consists of collecting five Review events for each model. All Review events with the same model are put into the same bin. This is done by hashing Review.model, again in constant time. Once a bin gets five events, they are removed from the bin and sent to the correlation node.

3.6 Correlation

The predicate, being in DNF, is of the form $p_1 \,||\, ... \,||\, p_n$ where each p_i is a conjunction of several conditions. One correlation node is created for each condition in the p_i, which is of the form $e_1.a_1 \; op \; e_2.a_2$, with the output of one node being piped to the next. Then,

a. if *op* is $==$, a_2 of all incoming e_2 events are stored in a hash table. Then $e_2.a_2$ that equals $e_1.a_1$ can be found in constant time ($O(1)$) by hashing $e_1.a_1$.
b. if *op* is $<, <=, >, >=$, a_2 of all incoming e_2 events are stored in a B+ tree [22]. Insertion of a_2 into B+ tree containing n elements is $O(log_b n)$, where b is the *degree* or *fanout* of the B+ tree, i.e. each node in the B+ tree contains at least b children but no more than $2b$ children [22]. B+ trees store only

keys in their internal nodes, and all data is stored in the leaves, which are linked to each other. It is well known that B+ trees are optimized for range queries [22]. If a B+ tree stores numerical values (either integers or floats), finding all values less than (greater than) c, for example, is $O(log_b n + k)$, where k is the number of values less than (greater than) c in the B+ tree. Hence, given an event of type e_1, finding all events of type e_2 that satisfy $e_1.a_1$ *op* $e_2.a_2$ is $O(log_b n + k)$.

c. if *op* is !=, a_2 of all incoming e_2 events are stored in a B+ tree [22]. The algorithm searches the B+ tree for $e_1.a_1$, and returns all the leaves of the B+ tree not equal to $e_1.a_1$ in $O(log_b n + k)$, where k is the number of values not equal to $e_1.a_1$ in the B+ tree.

In the example, since all groups of five Review events arriving at this node have the same Review.model, it is sufficient to check if the model attribute of one event matches that of VWRelease – again in constant time. Hence, using an aggregation node reduces the complexity of this correlation.

3.7 Disjunction

An event predicate being in DNF as described above, a sequence of correlation nodes are created for each inter-event condition of each predicate p_i. The output of the sequence, which is a set of events, is connected to a *union* node to handle disjunctions. The union node performs a set union on all its inputs. In other words, if a set of events $\mathcal{E}$ matches both p_i and p_j, then $\mathcal{E}$ is delivered by the algorithm only once. Set union is also implemented in linear time using the disjoint set data structure [22].

4 Evaluation

In this section we (1) show that GenTrie performs as well as existing solutions for languages with less expressiveness and that (2) such solutions can not compensate for our additional features without significant penalty; finally, we also (3) stress-test GenTrie to assess its scalability.

4.1 Santa Claus Problem

The *Santa Claus* problem was first proposed by Trono [23], and used by Benton [24] among others to test the expressiveness of concurrent programming languages. Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions: [a] if awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on holidays, and [b.] if awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys. A waiting group of reindeer must be served by

Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshaling the reindeer or elves into a group must not be done by Santa.

We implemented the Santa Claus problem analogously to the proposition by Benton [24]. The arrival of a reindeer or an elf is an event. Event patterns do not have any predicates. We generate "reindeer-arrival" and "elf-arrival" events randomly at different frequencies, and measure the number of synchronizations (either reindeer or elves) per second. Figure 4a compares the performance of GenTrie with that of Scala Joins and Cω. In Figure 4a, the abscissa plots the ratio of the number of times per second all nine reindeer arrive to the number of times three out of ten elves arrive. Hence, if the ratio is 10:900, the ideal number of synchronizations per second is 910. As can be seen from Figure 4a, the performance of GenTrie is comparable to Scala Joins and Cω for simple event joins, though it is more complicated due to its support for (more expressive) predicates. As the number of synchronizations per second increases, GenTrie increasingly outperforms the other approaches.

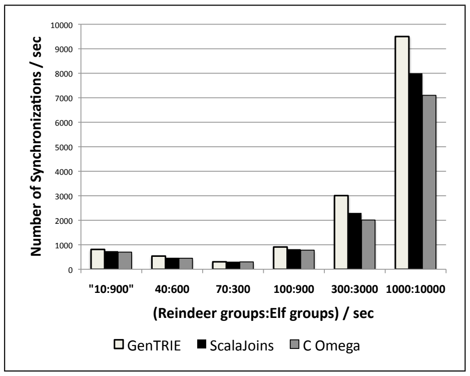
4.2 Stock Monitoring

We take the stock monitoring component of an algorithmic trading application to evaluate the efficiency of GenTrie and to illustrate that manual implementations of streams and intra-event conditions are inefficient. The application used 200 stocks of 10 categories (finance, technology, minerals, power etc.), and three trading strategies, namely Target Volume Participation Strategy (TVPS) (cf. [25]), Static Order Book Imbalance (SOBI) [3], and Volume Weighted Average Price (VWAP) [4]. The application has 250 event types, 120 correlation patterns, and window sizes of 10. Figure 4b shows the event-processing throughput using each strategy. Figure 4b shows that the throughput of GenTrie is 2.5$\times$ that of the Jess [21] implementation of the well-known Rete [20] algorithm used by various systems for composite event detection including an earlier incarnation of our own EventJava [2] framework. GenTrie's throughput is also 10$\times$ that of Scala Joins and Cω. The general nature of Rete allowed us to implement the examples quite easily, while in Scala Joins and Cω we had to manually compensate in the application for the missing features, essentially leading to a staged event matching as for instance, CML. Note that the implemented semantics are equivalent, i.e., the same composite events are identified.

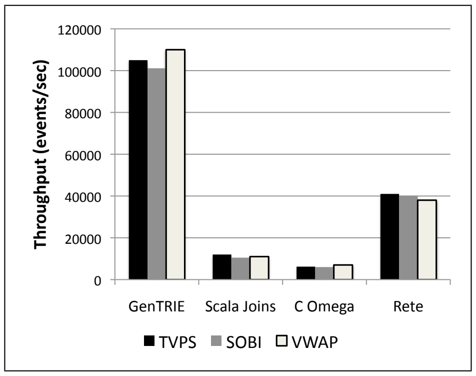
4.3 Scalability

While the need to use real applications for evaluation is obvious, individual applications can not fully *stress-test* GenTrie. Given the wide variance in system loads produced by different event-processing applications and by a same application over time, stress-testing plays an important role though in evaluating the scalability of event-processing algorithms. We thus algorithmically generate

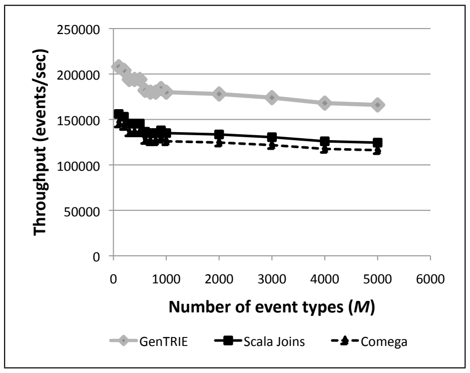
[3] http://www.cis.upenn.edu/~mkearns/projects/sobi.html
[4] http://www.investopedia.com/terms/v/vwap.asp



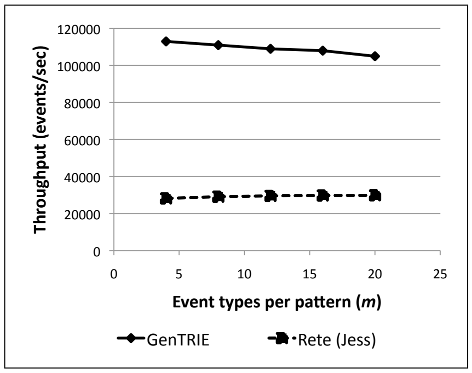
(a) Santa Claus problem



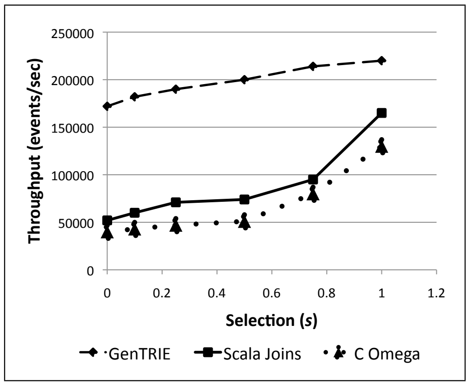
(b) Three Algorithmic stock trading strategies



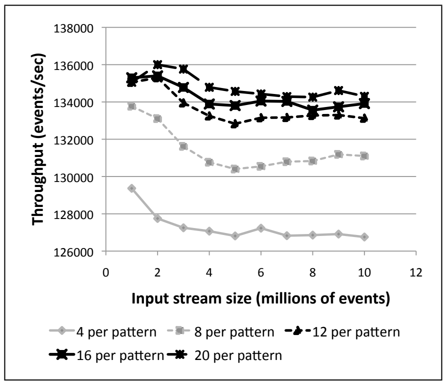
(c) Performance while increasing M



(d) GenTRIE vs. Rete (Jess)



(e) Varying selection (s)



(f) Stability of GenTrie with varying m.

Fig. 4: Performance of GenTrie vs. Scala Joins, Jess and Cω

event types and event patterns, and randomly generate events to match these patterns. We strive to keep the generated event patterns as close as possible to real world patterns, by following benchmarks used by systems mirroring the features supported by EventJava (e.g., Cayuga [14]).

Parameters. Some of the parameters of GenTrie which govern its event-processing throughput are:

1. The number of event types involved in the application – M.
2. The number of event types per correlation pattern – m. Thus the number of correlation patterns is M/m.
3. The *selection* s of the predicates in the event pattern, i.e., the probability that an event matches the event pattern. A selection of 1 implies that no event is ever filtered out, whereas a selection of zero implies that all events are filtered away before any correlation. This can be achieved easily by generating events and predicates with unary conditions such that the events contain values that never match the conditions.

Results. Figure 4c compares the join processing throughput of GenTrie against Scala Joins and Cω. Each join in this experiment contained $m = 4$ events and a *true* predicate. Figure 4c shows that GenTrie's throughput is 25% higher than that of Scala Joins and 30% higher than Cω.

Figure 4d compares scalability of GenTrie with respect to the number of event types per pattern m, in the presence of streams and predicates, to Rete. This experiment used $M = 5000$ event types, and streams windows of size $k = 5$. Predicates transitively linked all involved events of all types, exhibiting a combined selectivity of roughly 10%. The throughput of GenTrie here is around 4× that of Rete; neither algorithm's performance varies significantly with an increasing m. This difference is representative of a large number of scenarios that we tested, but which can't be covered due to space restrictions.

Figure 4e shows the throughput of GenTrie for a varying selection s, comparing also to Scala Joins and Cω. To make the comparison as fair as possible selection was achieved with unary conditions only (unary conditions are achieved via `if` statements inside reactions for Cω). For non-zero selection, the throughput of GenTrie is approximately 3× that of Scala Joins and 4× that of Cω.

In addition to M, m and s, another important criterion is the *stability* of an algorithm's throughput over time. To evaluate this, we use a stream of 10 Mio events, and sample the throughput at intervals of 1 Mio events. Figure 4f shows that the throughput of GenTrie remains fairly stable over time independently of m – variations in throughput are less than 2%. For an evaluation of the stability of Rete, refer [2].

5 Discussion

Multiple patterns and fairness. Like many other sources of semantic differences between languages and systems, we have not considered interaction across patterns. Languages based on the join calculus [18] typically allow several patterns/reactions to involve a same event type, and non-deterministically choose which reaction may consume a corresponding event. Such *exclusive* disjunctions (*X-OR* semantics) are rather easily implemented (pragmatically) in such languages devoid of predicates as events can be assigned to one pattern or another without further inspection, but the implementation intricacies may contribute to *fairness* issues [6] across patterns if a program relies on this non-determinism. In GenTrie, we have chosen to support disjunction || in predicates as opposed to *forcing* programs to declare a separate pattern/reaction pair for each disjoined predicate on the same set of events (as is common in certain publish/subscribe systems [9]), as this may in some systems be interpreted as X-OR semantics and in others lead to non-exclusive disjunctions. X-OR semantics can be achieved in GenTrie by duplicating events across queues corresponding to competing patterns, but keeping the copies linked to each other to ensure that matching and consumption of one instance leads to discarding all of them.

Garbage collection. In the presence of predicates, some events may a priori never be consumed. Program analysis together with annotations could be used in more strongly coupled systems to statically ensure that this does not occur. A pragmatic approach which is viable for many loosely coupled systems consists in defining garbage collection policies based on the application at hand, e.g., bounding queues (keeping *first received* or *most recent* events), assigning timestamps and timeouts. In the benchmarks which used predicates, once an event matches a pattern, older unmatched events of the same type are discarded. This is particularly relevant in the case of algorithmic trading, and can be used to make event matching *order-preserving* [2]. Discarding older unmatched events might prevent some matches after garbage collection, and should be used instead of other approaches like bounding queues only when warranted by the application that uses GenTrie for event matching.

6 Conclusions and Future Work

We have presented a generic model of complex event detection (CED), and an efficient and scalable algorithm for CED implemented in EventJava [2]. We are currently in the process of extending the pattern grammar to become yet more expressive. Two thrusts focus on (1) *parametric* patterns which support variables besides values in unary boolean expressions ($e.a\ op\ x$ with x a program variable) and (2) supporting *combinators* on events. While (1) is already supported at local scope, we are interested in a distributed solution for EventJava and its decentralized runtime environment; this requires specific support to propagate variable changes across nodes in an efficient manner ensuring properties such as ordering of the appearance of these changes. For (2) we are in a first step interested in supporting operations on scalar event attributes (e.g. $e_1.a_1 + e_2.a_2 == e_3.a_3$) before investigating full support for methods as these may have side-effects.

References

1. N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. ACM TOPLAS, 26(5):769–804 (2004)
2. P. Eugster and K. R. Jayaram. EventJava: An Extension of Java for Event Correlation. *ECOOP 2009*, pp. 570–594.
3. P. Haller and T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. *COORDINATION '08*, pp. 135–152.
4. P. Haller and M. Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. Theoretical Computer Science, 410(2-3): 202–220.
5. S.G. Von Itzstein and D.A. Kearney. The Expression of Common Concurrency Patterns in Join Java. *PDPTA '04*, pp. 1021–1025.
6. A. Petrounias and S. Eisenbach. Fairness for Chorded Languages. *COORDINATION 2009*, pp. 86-105.
7. J. H. Reppy and Y. Xiao. Specialization of CML Message-passing Primitives. *POPL 2007*, pp. 315–326.
8. H. Rajan and G.T. Leavens. Ptolemy: A Language with Quantified, Typed Events. *ECOOP '08*, pp. 155–179.
9. A. Carzaniga, M.J. Rutherford, A.L. Wolf. Design and Evaluation of a Wide-area Event Notification Service. ACM TOCS, 19(3): 332-383 (2001)
10. M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. *PDSE 2000*, pp. 83–92.
11. P. Eugster. Type-based Publish/Subscribe: Concepts and Experiences. ACM TOPLAS, 29(1) (2007)
12. T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad hoc Networks. *SCCC 2007*.
13. M. Sulzmann, E.S.L. Lam, and P. Van Weert. Actors with Multi-headed Message Receive Patterns. *COORDINATION 2008*, pp. 315–330.
14. A.J. Demers, J. Gehrke, M. Hong, M. Riedewald, W.M. White. Towards Expressive Publish/Subscribe Systems. *EDBT 2006*, pp. 627–644.
15. M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis Distributed Stream Processing System. *SIGMOD 2005*, pp. 13–24.
16. H. Plociniczak. JErlang: Erlang with Joins. http://www.doc.ic.ac.uk/teaching/distinguished-projects/2009/h.plociniczak.pdf
17. C. V. Russo. Join Patterns for Visual Basic. *OOPSLA 2008*, pp. 53-72.
18. C. Fournet and C. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. *POPL '96*, pp. 372–385.
19. G. Li, H. A. Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. *Middleware 2005*, pp. 249–269.
20. C. Forgy: Rete: a Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artificial Intelligence 19(1): 17–37 (1982)
21. E. Friedman-Hill: Jess, http://www.jessrules.com/jess/. (2008)
22. T. H. Cormen, R. L.Rivest, C. Leiserson, C. H. Stein.: Introduction to Algorithms. MIT Press (2009)
23. J. A. Trono: A New Exercise in Concurrency. SIGCSE Bulletin, 26(3): 8-10 (1994)
24. N. Benton: Jingle Bells: Solving the Santa Claus Problem in Polyphonic C#, http://research.microsoft.com/en-us/um/people/nick/polyphony/santa.pdf (2003).
25. Vhayu: Vhayu Velocity – Algorithmic Trading Case Study. http://www.vhayu.com/Content/CollateralItems/AlgoTradingCaseStudy.pdf (2008).