Pattern Rewriting Framework for Event Processing Optimization

Ella Rabinovich
IBM Haifa Research Labs
Haifa 31905, Israel
ellak@il.ibm.com

Opher Etzion
IBM Haifa Research Labs
Haifa 31905, Israel
opher@il.ibm.com

Avigdor Gal
Technion – Israel Institute of Technology, Haifa 32000, Israel
avigal@ie.technion.ac.il

ABSTRACT
A growing segment of event-based applications require both strict performance goals and support in the processing of complex event patterns. Event processing patterns have multiple complexity dimensions: the semantics of the language constructs (e.g., sequence) and the variety of semantic interpretations for each pattern (controlled by policies). We introduce in this paper a novel approach for pattern rewriting that aims at efficiently processing patterns which comprise all levels of complexity. We present a formal model for pattern rewriting and demonstrate its usage in a comprehensive set of rewriting techniques for complex pattern types, taking various semantic interpretations into account. A cost model is presented, balancing processing latency and event throughput according to user's preference. Pattern cost is then estimated using simulation-based techniques.

This work advances the state-of-the-art by analyzing complex event processing logic and by using explicit means to optimize elements that were considered "black box." Our empirical study yields encouraging results, with improvement gain of up to tenfold relative to the non optimized solutions that are used in the current state-of-the-art systems.

Categories and Subject Descriptors
H3.4. Systems and software (Performance evaluation), D3.3 Language constructs and features (Patterns).

General Terms
Algorithms, performance, experimentation.

Keywords
Event processing, event patterns, event processing optimization, simulation-based optimization, bi-objective goal function.

1. INTRODUCTION AND MOTIVATION
Rapid evolution of the use of event-based applications in enterprises poses high agility and scalability requirements for event processing systems. Such requirements call for performance optimization to achieve low latency and high throughput. Among the event-based systems there is a growing segment of applications that have (typically conflicting) requirements for both strict performance and processing of event patterns with an increased complexity level. As an example, an auditing application may be used to verify a certification process that consists of multiple steps and needs to be finalized in the correct order and with an acceptable combination of completion status [24]; the complexity here stems from the quantity of steps in some workflows (>20) and the multiple assertions among events representing finalization of states. In the financial industry, an anti-money laundering system may monitor a continuous stream of customers' transactions for suspicious activities while looking for complex trends [23]. On the Internet, the variety of Web2.0 applications generate vast amount of real-time information that can be analyzed to infer social activities; again, using complex patterns.

Typical benchmark and optimization work in the event processing domain has been geared towards the simple functionality cases. Previous studies [1][14] indicate that there is a major performance degradation as application complexity increases (ratio of 1:90 for both latency and throughput between scenarios), thereby evidencing that optimization efforts comprise a great potential value for the high-end applications in the complexity scale.

The potential optimization value of pattern rewriting, along with rapid evolution of the complexity of event processing systems, stimulated our effort to develop a formal and methodical approach for event pattern rewriting, while considering the most complex cases of patterns and their various semantic interpretations. In this work, we provide a model and algorithms for complex event patterns rewriting as an optimization mechanism. The rewriting technique, underlying the core part of this work involves splitting a pattern or unifying several patterns by using patterns of the same type. We propose a formal model for rewriting of the two commonly used patterns, namely all and sequence, for (1) subsumption of the common logic, and (2) splitting a pattern into sub-patterns for parallel execution, where possible (e.g., multi-core platform). These techniques assist in reducing CPU consumption in the first case and improving a system throughput in the second.

We also propose a rewriting technique that provides a choice (realized by different pattern rewritings) between eager and lazy evaluation of the sequence pattern. We demonstrate that eager vs. lazy evaluation of the sequence pattern mirrors the latency vs. throughput tradeoff performance phenomenon. Driven by the possible pattern split alternatives, we generate a set of Pareto efficient solutions with respect to pattern performance indicators; this allows the system designer to select a favorable solution, by tuning a bi-objective performance goal function. Our empirical study produced encouraging results, demonstrating up to tenfold improvement in pattern latency, achieved by the rewritten version when compared to its original alternative.

Our approach advances the state-of-the-art in the domain of event processing optimization, by proposing enhanced solutions to

DEBS'11, July 11–15, 2011, New York, New York, USA.

complex patterns. Therefore, the contribution of this paper is twofold. First, we establish a methodical approach for event pattern rewriting. Second, we describe a framework for tuning the performance objectives of the sequence pattern, employing the pattern rewriting techniques, as a proof of usefulness of the rewriting approach.

The rest of the paper is structured as follows: Section 2 provides the background and sets the terminology used throughout the paper. In Section 3, we present the pattern rewriting model, formally proving pattern rewriting validity for one pattern and extending the discussion to additional cases. Section 4 describes in depth the idea of bi-objective optimization of the sequence pattern and Section 5 presents our empirical study. We discuss related work in Section 6 and conclusions in Section 7.

2. PRELIMINARIES

The terminology we use in this paper is based on the event processing model defined by Etzion and Niblett [5]. In this section we briefly overview the main constructs used in this work.

2.1 Event Processing Constructs

An *event* (e) is an occurrence within a particular system or domain; it is something that has happened, or is contemplated as having happened in that domain. An *event type* (E) is a specification for a set of events that share the same semantic intent and structure. Given an event *e*, we define its event type by *e.type*. An event type can represent an event arriving from a producer or an event produced by an event processing agent (see below). We denote such events as *raw* and *derived*, respectively.

An *event processing agent* (EPA) is a processing element that applies logic on a set of input events, to generate a set of output (derived) events. A *stateful event processing agent* maintains its internal state over successive invocations. For example, an EPA that performs aggregation of events over a time window is *stateful*, while an EPA that filters events by applying a predicate to a single event's attributes is considered *stateless*.

A *context* is a named specification of conditions that groups event instances so that they can be jointly processed. While there exist several context dimensions, in this work we refer to the temporal dimension, one of the most commonly used dimensions. A *temporal context* consists of one or more time intervals, possibly overlapping. Each time interval corresponds to a context partition, containing events that occur during that interval.

An *event processing network* (EPN) [11][19] is a conceptual model, describing the event processing flow execution. It consists of a collection of EPAs, producers, and consumers linked by event channels. An *event channel* is a processing element that receives events from one or more source elements (producer or EPA), makes routing decisions, and sends the input events unchanged to one or more target elements (EPA or consumer) in accordance with the routing decisions. A schematic presentation of an EPN, as presented by Moxey et al. [15], is illustrated in Figure 1. The EPN follows the event driven architecture where EPAs are communicating in asynchronous fashion by receiving and sending events. Although a channel can route several event types, in this work we assume that it is manifested as an edge in the graph, connecting a single source with a single sink carrying a single event type.

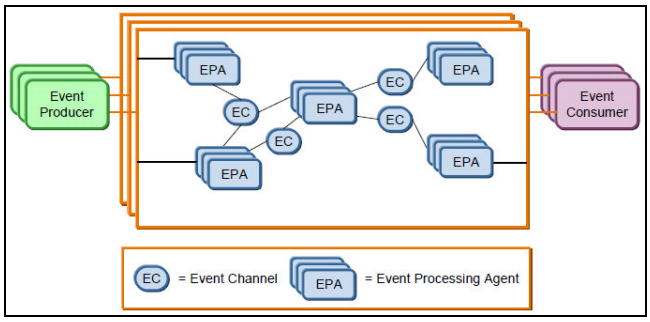

Figure 1: Schematic representation of EPN

2.2 Pattern Matching

Pattern matching (also known as pattern detection) is a type of EPA that enables the analysis of collections of events and the relationship between them. Informally, we say that a conditional combination of events matches a pattern if this combination satisfies the particular pattern definition. Pattern matching EPA examples are: all, sequence, absence, and any. Pattern matching EPAs are stateful, involving both detection of multiple event combinations and temporal semantics; thereby introducing the majority of performance optimization challenges.

A *relevant event types set* (RTS) of a pattern matching EPA is a list of event types to which a matching function is applied. Pattern *participant set* (PS) is a collection of event instances that occur within this pattern agent's context partition (time window). These events are instances of event types mentioned in the pattern's RTS. Pattern *matching set* (MS) is the output of pattern matching process; it is a subset of the *participant set*, satisfying a certain pattern definition.

A basic form of pattern *derived event* (DE) generation is a composition of all event instances in a matching set, denoted by Compose(MS). Pattern derivation can alternatively be performed by computing output event attributes as a function of event values in a matching set.

Figure 2 demonstrates the concept of a pattern matching EPA. Event instances in the PS comply with event types defined by RTS, and MS contains a subset of PS satisfying the pattern.

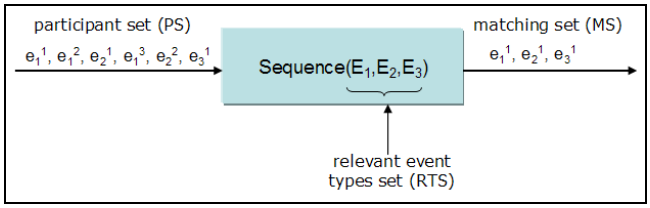

Figure 2: Pattern matching EPA

A *pattern assertion* (PA) is a condition that the matching set is required to meet for the pattern to be satisfied. Pattern assertion is a predicate, i.e., Boolean expression including variables (event attributes), mathematical (e.g., +, -, *, <, >, =) and logical (e.g., AND, OR, NOT) operators, and custom functions. We distinguish a pattern assertion that refers to a single event type (e.g., event *threshold condition* such as E_1.price>100), from pattern assertion that encapsulates several event types (e.g., E_1.price>E_2.price), further denoted by *cross-event pattern assertion*.

Pattern *policies* (Policies) [1][5] are used to fine-tune and disambiguate pattern matching semantics. In this work we

consider the following subset of pattern policies: *evaluation* policy determines when the matching sets are produced with possible values in {immediate, deferred}; *cardinality* policy determines how many matching sets are produced within a single context partition (e.g., time window) with possible values in {single, unrestricted}; *repeated type* policy handles multiple events of the same type, and has possible values in {first, last, override, every}; *consumption* policy determines the status of a participant event after it has been included in a matching set with possible values in {consume, reuse}. All policies are specified at the pattern level and *repeated type* and *consumption* policies can be overridden for specific event types.

2.3 Speculative Broker Scenario

We illustrate the basics of event processing, as demonstrated in sections 2.1 and 2.2 through a concrete use case.

Example 1:

A speculative broker is a broker who buys and immediately sells, with profit, the same stock at least three times within a ten minutes time window.

E_1: StockBuy event type with the following attributes: {transactionID, transactionTS, stockID, price, volume}

E_2: StockSell event type with the following attributes: {transactionID, transactionTS, stockID, price, volume}

Pattern: sequence(E_1^i, E_2^j, E_1^k, E_2^l, E_1^m, E_2^n), where the superscript numbers are used to distinguish among various instances of the same event type, such that i<k<m and j<l<n.

Pattern assertion:

(E_1^i.stockID==E_2^j.stockID AND E_1^i.price<E_2^j.price) AND
(E_1^k.stockID==E_2^l.stockID AND E_1^k.price<E_2^l.price) AND
(E_1^m.stockID==E_2^n.stockID AND E_1^m.price<E_2^n.price)

Pattern policies:

evaluation:	immediate
cardinality:	single
repeated type:	first
consumption:	---

Consider the ordered set of event instances along with the relevant attributes, as specified in Table 1. Events whose name starts with e1 and e2 are instances of E1 and E2 event types respectively.

The pattern matching set, and therefore the pattern derived event (Compose(MS)) will consist of a single matching set that contains the following event instances:

<e_1^1, e_2^1, e_1^2, e_2^3, e_1^3, e_2^4>

Note that a single matching set is reported due to the "single" cardinality policy; this policy allows disregarding the consumption mode, which is only relevant for multiple detections. The first instance is selected from within a sequence of events of the same type, e.g. e_2^1 is selected for the matching set from the existing <e_2^1, e_2^2> instances, due to the "first" repeated type. The "immediate" evaluation policy forces the detection report upon the e_2^4 instance arrival, and not at the end of a ten minutes time window, as would be the case with the "deferred" mode.

Changing the cardinality policy to "unrestricted", repeated type policy to "last" and setting the consumption policy to "reuse" would produce two matching sets:

<e_1^1, e_2^2, e_1^2, e_2^3, e_1^4, e_2^4> and <e_1^2, e_2^3, e_1^4, e_2^4, e_1^5, e_2^5>

Unlike the previous case, several matching sets are reported and the last event from all existing instances is selected for a matching set, e.g. e_2^2 from the possible <e_2^1, e_2^2>. Finally, the "reuse" consumption mode gives rise to a repetitive appearance of certain instances in multiple matching sets, e.g. e_1^4.

Table 1: Event sequence for speculative broker pattern

event name	transactionID	symbol	price	volume
e_1^1 (buy)	1110	ICO	160	50
e_2^1 (sell)	1145	ICO	175	25
e_2^2 (sell)	1177	ICO	175	25
e_1^2 (buy)	2011	CCHOF	200	75
e_2^3 (sell)	2014	CCHOF	207	75
e_1^3 (buy)	2175	WBD	170	40
e_1^4 (buy)	2200	WBD	170	25
e_2^4 (sell)	2201	WBD	177	65
e_1^5 (buy)	2210	ICO	160	50
e_2^5 (sell)	2245	ICO	175	50

3. PATTERN REWRITING MODEL

In this section we introduce the idea of event processing pattern rewriting (Section 3.1), along with the formal model that comprises the basis of our policy-based rewriting approach (sections 3.2 and 3.3). We prove rewriting validity for a representative pattern (Section 3.4) and extend the discussion to additional cases (Section 3.5).

3.1 Introduction to Pattern Rewriting

The idea of expression rewriting has been around for quite some time. Both unification and subsumption of logical assertions were introduced in the AI literature more than 30 years ago [16]. Some event processing related rewritings, such as extracting common sub-expression logic, were demonstrated in the SMILE project [21]. We now introduce an approach that aims at fitting the complexity of current event processing systems.

The rewriting technique, underlying the core part of this work involves either splitting a pattern or unifying several patterns by using patterns of the same type, as demonstrated in Figure 3. The sequence pattern at the top part of the figure (f1) is split into two successive sequence patterns (f2' and f2"), where DE is the derived event type produced by f2'; f2" monitors a series of DE, E3 and E4 instances, thus completing the sequence of E1, E2, E3 and E4, as required for detection of the original pattern.

Among the new and nontrivial challenges, posed by the event processing language expressiveness for pattern rewriting, are cross-event pattern assertions and pattern policies. Each aspect increases the complexity of the pattern rewriting procedure and requires a distinct analysis of rewriting alternatives, as well as an appropriate formal representation. The detailed exploration of these challenges is given in sections 3.3 and 3.4.

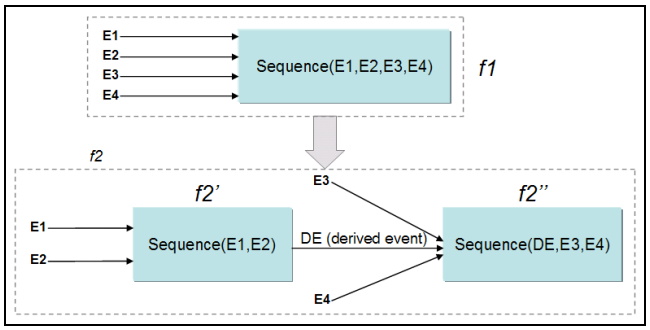

Figure 3: Pattern rewriting example

Another category of rewritings is inspired by the redundancy in event processing languages; i.e. there exist several alternatives for expressing the same logic. A simple example is the pattern sequence(E_1,E_2) that can be rewritten using all(E_1,E_2) and filter EPAs, while the filter EPA will filter the derived events that comply with E_1.timestamp<E_2.timestamp from all(E1,E2). A more complex example is the rewriting of all(E_1,E_2) into any(sequence(E_1,E_2),sequence(E_2,E_1)).

In this work we focus on the first category of rewriting by using patterns of the same type, proposing several rewriting alternatives, showing formally the rewriting correctness and demonstrating concrete usages.

3.2 Formal Definition of Pattern Detection

We employ the denotational semantics approach [20], which defines an event processing pattern as a function, mapping pattern's input into its output. The denotational approach provides natural and intuitive means for proving patterns' equivalence: we demonstrate that for the same input, two rewriting alternatives generate the same output. This formal evidence is a necessary and sufficient condition for a rewriting validity.

Due to space considerations, we refrain from the full-scale formal model of the observed pattern types and policies. We demonstrate the approach by modeling the sequence pattern with the policies: (1) evaluation="immediate", (2) cardinality="single" (3) repeated type="first" and (4) consumption="consume". RTS stands for relevant event types set, PA for pattern assertion, PS for participant set and MS for matching set.

We are given f(sequence, RTS, PA, Policies, PS) = MS, where

Policies = <immediate, unrestricted, first, consume>,

RTS = <E_1, E_2, E_3, … E_N>, PS = <e_1, …, e_M>, and

e.ts is the timestamp of event instance e.

We define an event collection EC to be a subsequence of PS. In what follows, events in an event collection of size N are enumerated as e_1, …, e_N, regardless of their enumeration in PS.

The conformance of an event collection EC with relevant event types set RTS is computed as follows:

$$\text{Conforms}(EC,RTS)=\begin{cases}\text{true} & \text{if } (e_i.\text{type} = E_i) \wedge e_1.ts < \ldots < e_N.ts, \\ & \forall(1\le i\le N) \\ \text{false} & \text{otherwise}\end{cases}$$

Definition 1: *A* pattern assertion PA over PS *is a function from the domain of subsequences of PS to {true, false}. PA(EC) is computed to be true if the assertion PA is satisfied by EC.*

We define a candidate matching sets CMS to be

CMS = {EC | Conforms(EC,RTS) ∧ PA(EC)}.

We say that a candidate matching set CMS is valid if it is not empty (CMS != ∅). Validity is denoted using the indicator ValidMatch(CMS). It is worth noting that the "first" repeated type policy induces lexicographical order on all event combinations that are candidates for a matching set, yielding chronological ordering between event instances, i.e. e1 ≺ e2 iff e1.ts<e2.ts. Therefore, event collections ordering is defined using lexicographical order notation.

Definition 2: *For EC' = <e'_1, …, e'_N>∈CMS and EC'' = <e''_1, …, e''_N>∈CMS we say that EC' precedes EC'', denoted by EC' ≺ EC'' iff ∃(1≤m≤N) ∀(i<m) (e'_i.ts==e''_i.ts ∧ e'_m.ts < e''_m.ts).*

Definition 3: *EC participants comply with the "first" repeated type if ∀(EC'∈CMS), EC' != EC, EC ≺ EC'). We denote such compliance with an indicator First(EC).*

Definition 4 *Given a candidate matching set CMS, a* pattern matching set *is defined to be:*

MS = (EC∈CMS | First(EC)).

The defined model serves validation of the rewriting procedure by proving the equivalence of two pattern alternatives, as presented in Section 3.4.

Employing cross-event pattern assertion introduces one of the main difficulties when approaching pattern rewriting. This challenge and its solution are detailed in the following section.

3.3 Assertion-based Rewriting

Splitting of a single pattern into two patterns, as demonstrated in Figure 3, infers assertion partitioning into two groups: an assertion related to the first pattern (f2') participants and an assertion related to second pattern (f2") participants. Such a partitioning gives rise to several assertion-related issues: (1) the direct connection of f2' (sequence(E_1,E_2)), with f2" (sequence(DE,E_3,E_4)) automatically implies "AND" operator between the two assertion parts, since both assertions need to be satisfied by the final matching, and (2) since f2' and f2" participant types do not overlap, the assertion is assumed to be *separable* (see below) into two independent parts, in terms of assertion variables.

We take the following steps to discover independent components in a pattern assertion, thus implying possible rewritings of the given pattern:

- Convert the pattern assertion expression into conjunctive normal form (CNF). Since CNF requires Boolean literals, each sub-expression (e.g., A.key>B.key) can be treated as a Boolean literal. The conversion to CNF is obtained by employing De Morgan laws [6].
- Identify independent participants' sub-groups in the newly created CNF assertion. This can be achieved by creating assertion variables dependency graph, where variables are represented by nodes and their *connections* by edges. Variable A is *connected* to variable B iff there exists a mathematical operator between them (e.g., A.key > B.key, A.key+B.key < 7) or variable A appears with variable B in the same OR clause.

Disconnected components in such a dependency graph induce partitioning of assertion variables into independent groups, thus creating independent patterns.

- Splitting an assertion into maximal number of independent partitions implies the finest granulation we can perform on the assertion expression; that is, the maximal split of the original pattern into interconnected sub-patterns.

As an example, consider the following pattern:

Example 2

sequence(A,B,C,D,E,F) with pattern assertion:
(A.key > B.key) AND (D.key > E.key) AND NOT ((E.key==F.key) AND (C.key==77))

Applying the above technique would result in the following steps:

Convert pattern assertion into CNF:

(A.key > B.key) AND (D.key > E.key) AND
(NOT(E.key==F.key) OR NOT(C.key==77))

Identify independent variables partitions, as depicted in Figure 4.

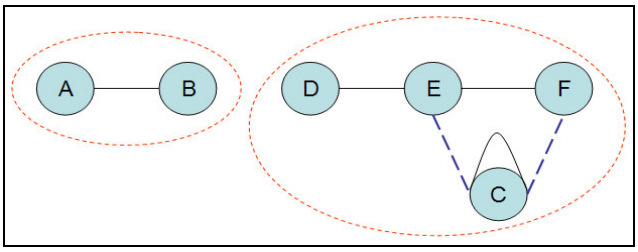

Figure 4: Disconnected components in assertion graph

Note that C is self-connected due to its comparison to constant and it is connected to E and F since they jointly appear in the same OR clause. The (interconnected) variables A and B are disconnected from the rest of the graph, thus creating two independent assertion variables parts. The pattern split induced by assertion partitioning generates two sub-patterns, as follows:

sequence(A,B) with pattern assertion:
(A.key > B.key)

followed by

sequence(C,D,E,F) with pattern assertion:
(D.key > E.key) AND (NOT (E.key==F.key) OR NOT (C.key==77))

In a more laconic writing: sequence(A,B,sequence(C,D,E,F))

When considering the sequence pattern, variable partitioning should preserve events order as specified by the relevant event types set. Consider again the sequence pattern in Example 2: in case the pattern assertion was slightly modified to produce {A,F} and {B,D,E,C} independent variables sets, such a partitioning would have been sufficient for pattern of type all, but not for sequence, where events order is of importance.

Expression evaluation optimization was extensively investigated in other areas, including compilers optimization, CSP (Constraint Satisfaction Problems) and Production Rules [12]. Several known techniques (e.g., converting assertions to CNF, identification of graph disconnected components) are leveraged at the demonstrated algorithm to allow assertion-based pattern rewriting. In this work we restrict ourselves to pattern assertions that are separable by "AND"; this can be extended to other connectors, such as "OR" and "XOR".

3.4 Patterns Equivalence Definition and Proof

Pattern assertion analysis is a fundamental step in the procedure of pattern rewriting. Pattern policies pose another nontrivial challenge of preserving the semantic interpretation of the pattern, while rewriting. We now present a formal proof of pattern rewriting equivalence for a basic set of policies and extend the discussion for more complex cases.

The formal definition introduced in Section 3.2 and assertion-based splitting techniques allow us to derive a formal justification for pattern rewriting, i.e. a proof that the original pattern is equivalent to the rewritten pattern in the sense that they produce an equivalent output to identical input. Complying with the query rewriting principle in database systems [3], we demonstrate that for the same input (pattern participant set), the original and the rewritten alternatives produce the same output (matching set).

When considering patterns we also need to refer to the temporal perspective, demonstrating that in the original and the rewritten alternative the equivalent output is produced at identical time points. Referring to absolute time points is not feasible for alternatives with different latencies, since execution of distinct rewritings may yield different absolute timestamps. Therefore, we consider two rewriting alternatives to be equivalent if they produce the same output in identical order, with respect to identical input events snapshot. The latter property can be viewed as temporal synchronization, thus satisfying the additional requirement introduced by patterns.

A split of a pattern into two consecutive patterns, as depicted in Figure 3, is defined as follows:

Definition 5 *(split):*

Given f1 of type sequence, a split *of f1, denoted by f2, is a set (f2',f2''), such that:*

f1(sequence, RTS, PA, Policies, PS) = MS1,

f2'(sequence, RTS', PA', Policies', PS') = MS2',

f2''(sequence, Compose(MS2').type ∪ RTS'', PA'', Policies'', Compose(MS2') ∪ PS'') = MS2'' = MS2

RTS = RTS' ∪ RTS'', PS = PS' ∪ PS'',

Compose(MS) is a union of events composing all instances in MS,

CMS1 is a candidate matching set of f1,
CMS2' is a candidate matching set of f2',

PA can be separated at event type Ei, further denoted as "separable at i", PA' is pattern assertion of f2', PA'' is pattern assertion of f2'',

Policies is set of policies of f1, Policies' is set of policies of f2' and Policies'' is set of policies of f2''

Using Definition 5 we now demonstrate the formal method for proving equivalence of patterns. We show that the sequence pattern with the following set of policies: (1) evaluation="immediate", (2) cardinality="single" and (3) repeated type="first", can be split into two sequence patterns. For the sake of conciseness we denote by ValidMatch(f) the application of ValidMatch to the outcome of f(pattern type, RTS, PA, Policies, PS), which is an MS.

Proposition1: *Given f1 and split f2=(f2',f2'') of f1, it holds that:*

1. ValidMatch(f2) → ValidMatch(f1)

2. ValidMatch(f1) → ValidMatch(f2)

3. MS1 = MS2

Proof:

1. ValidMatch(f2) → ValidMatch(f1):
the same as proving !ValidMatch(f1) → !ValidMatch(f2)

!ValidMatch(f1) → CMS = ∅ →

!∃(EC=<e_1, …, e_N>⊆PS | Conforms(EC,RTS) ∧ PA(EC)) →
(since the assertion is separable at i)

!∃(EC'=<e_1, …, e_i>⊆PS') | Conforms(EC',RTS') ∧ PA'(EC')) OR

!∃(EC''=<e_{i+1}, …, e_N>⊆PS'' | Conforms(EC'',RTS'') ∧ PA''(EC''))→

CMS' = ∅ OR CMS'' = ∅

!ValidMatch(f2') OR !ValidMatch(f2'') →

!ValidMatch(f2)
□

2. ValidMatch(f1) → ValidMatch(f2):

We assume that Compose(MS1).ts = e_i.ts and pattern split temporal correctness is forced by implementation

ValidMatch(f1) → CMS != ∅

∃(EC=<e_1, …, e_N>⊆PS | Conforms(EC,RTS) ∧ PA(EC)) →
(since the assertion is separable at i)

∃(EC'=<e_1, …, e_i>⊆PS' | Conforms(EC',RTS') ∧ PA'(EC')) ∧

∃(EC''=<e_{i+1} , …, e_N>⊆PS'' | Conforms(EC'',RTS'') ∧ PA''(EC'')) →

ValidMatch(f2') ∧ ValidMatch(f2'') ∧
Compose(MS1).ts<e_{i+1}.ts →

ValidMatch(f2)
□

3. MS1 = MS2

This is trivial for #1; for #2:

ValidMatch(f1) → CMS1 != ∅ →

MS1 = (EC=<e_1, …, e_N>∈CMS1 | First(EC)) →

MS1 = (EC'=<e_1, …, e_i>∈CMS2' ∪ EC''=<e_{i+1}, …, e_N> |
First(EC' ∪EC'')) →(since the assertion is separable at i)

MS1 = (EC'=< e_1, …, e_i>∈CMS2' | First(EC')) ∪
(EC''=< e_{i+1}, …, e_N> | First(EC'')) → (according to def. of MS)

MS1 = MS2' ∪ (EC''=<e_{i+1}, …, e_N> | First(EC'')) →

*MS1 = MS2' ∪ EC''**

$PS_{f2''}$ = <Compose($MS^1_{f2'}$), …, Compose($MS^K_{f2'}$)> ∪ PS''

MS2 = (Compose($MS_{f2'}$∈CMS2') ∪ $EC_{f2''}$=<e_{i+1}, …, e_N>) |
First(Compose($MS_{f2'}$) ∪ $EC_{f2''}$) →
(since the assertion is separable at i)

MS2 = (Compose($MS_{f2'}$∈CMS2') | First(Compose($MS_{f2'}$)) ∪
($EC_{f2''}$=<e_{i+1}, …, e_N> | First ($EC_{f2''}$)) →

MS2 = (Compose(MS2') ∪ EC'') → Compose(H) = {h| h∈H}*

MS2 = (MS2' ∪ EC'') →*

MS2 = MS1
□

3.5 More Pattern Equivalences

While the equivalence demonstrated in the previous section is quite intuitive, other kinds of rewriting are more complex, requiring modification of pattern assertion and individual patterns policies to make the equivalence valid. In this section we show additional pattern rewritings. Due to space consideration, we refrain from presenting formal correctness statements.

We begin with the pattern sequence(E_1,E_2,E_3) comprising pattern assertion that is separable at E_3 and the following set of policies: (1) evaluation="immediate", (2) cardinality="single" and (3) repeated type="last". When rewritten into two patterns: sequence(sequence(E_1,E_2),E_3), the new pattern policies should change slightly to satisfy the desired equivalence. For event scenario <e_1^1, e_2^1, e_2^2, e_3^1>, f1 will report the <e_1^1, e_2^2, e_3^1> as the matching set. If we keep working with the same policies at the rewritten alternative, f2' will report <e_1^1, e_2^1> as a matching set and f2" will attach the last e_3^1 event to this pair, eventually reporting <e_1^1, e_2^1, e_3^1>, which is incorrect. Figure 5 demonstrates the temporal fallacy of this example, leading to the undesired result. Due to the temporal decoupling between f2' and f2", f2' will report its matching set "too early," ignoring instances that arrived after the first detection (due to the "single" cardinality policy, and "immediate" evaluation policy), thus losing synchronization with respect to the original pattern f1.

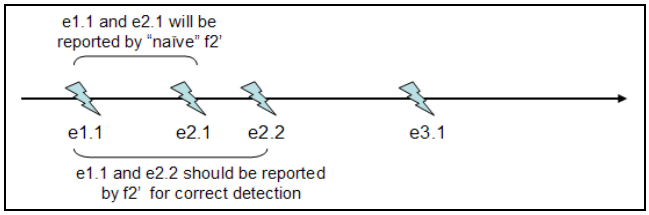

Figure 5: Temporal inconsistency keeping original policies

The solution for this undesired situation lies in modifying the pattern policies. Changing f2' cardinality policy to "unrestricted" (unbounded number of detections) and setting its consumption policy to "reuse" (keeping event instance for further detections after it was included in a matching set) will achieve the desired effect, resulting in equivalence of the original and the rewritten alternatives: f2' will produce two matching sets: <e_1^1, e_2^1> and <e_1^1, e_2^2>, and f2" will consider the last derived event; therefore, upon e_3^1 arrival it will eventually emit <e_1^1, e_2^2, e_3^1>, as required.

The example above can be concluded by a summary of the original and the rewritten pattern alternatives, driven by pattern policies as demonstrated in Table 2. f1 refers to the original pattern, while f2' and f2" refer to the rewritten first and second patterns, respectively.

Note that while the example refers to the sequence pattern, a theoretical analysis infers the same solution for a simpler case, the all pattern. This observation also holds for all further detailed rewritings, and its proof is similar to the demonstrated one up to the event ordering requirement removal.

Table 2: Pattern rewriting by policies modification (1)

Pattern type: sequence/all			
Policy	f1	f2'	f2"
Evaluation	immediate	immediate	immediate
Cardinality	single	unrestricted	single
Repeated Type	last	last	last
Consumption	-	reuse	-

Another nontrivial example involves the pattern sequence(E_1,E_2,E_3) such that the pattern assertion is separable at E_3, and the following set of policies (1) evaluation="immediate", (2) cardinality="unrestricted", (3) repeated type="override" and (4) consumption="consume". Again, we consider a split into two patterns sequence(sequence(E_1,E_2),E_3). Similarly to the previous example, we should assign the "reuse" consumption policy to f2'. However, while in the former example the entire processing was terminated after the first detection ("single" cardinality policy), in this case we are not limited to a single detection ("unrestricted" cardinality policy). This poses an additional difficulty: after any detection at f2", the entire matching set should be consumed, at odds with f2' consumption mode, which is "reuse". In order to prevent f2" from reporting "consumed" events over and over again, we would like to prevent the consumed events from arriving at f2", filtering them out on their way to f2". This can be achieved by assigning a filter EPA between f2' and f2" in the rewritten alternative, which filters in only relevant events, and filters out all the consumed events by querying an in-memory consumed events pool. This rewriting is schematically demonstrated in Figure 6.

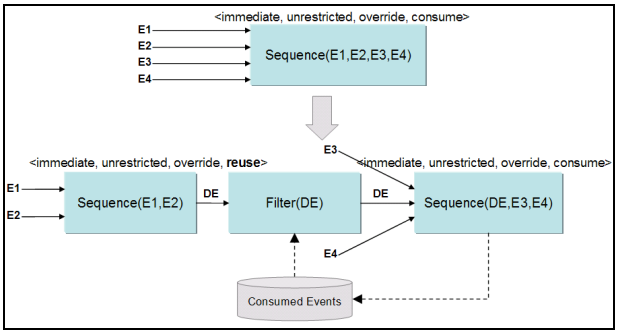

Figure 6: Pattern rewriting using filter EPA

A summary of the original and rewritten pattern alternatives in the example above is presented in Table 3.

Table 3: Pattern rewriting by policies modification (2)

Pattern type: sequence/all			
Policy	f1	f2'	f2"
Evaluation	immediate	immediate	immediate
Cardinality	unrestricted	unrestricted	unrestricted
Repeated Type	override	override	override
Consumption	consume	reuse	consume
filter between f2' and f2"			

The most complex case we have encountered is the one combining both "last" repeated type policy from the first example and "unrestricted" cardinality policy from the second one. Consider the following event instances collection: <e_1^1, e_1^2, e_2^1, e_2^2, e_3^1, e_3^2>. For consumption policy="consume", the original pattern will produce two matching sets: <e_1^2, e_2^2, e_3^1> and <e_1^1, e_2^1, e_3^2>. Applying previously mentioned rewriting techniques ("reuse" consumption policy and filter EPA) will be insufficient, producing the incorrect second matching set: <e_1^2, e_2^2, e_3^1>, <e_1^2, e_2^1, e_3^2>. Note that the pair <e_1^1, e_2^1> required for the correct second detection was not derived by f2' at all. We need to force f2' to report all possible combinations, and then select the correct ones carefully upon f2" detection attempt. Such an effect can be achieved by using "every" repeated type policy at f2', instead of the original "last" policy.

A side effect of using "every" policy will cause redundant matching sets generation at f2' and their arrival to f2"; that is, for a given example, at the first detection point, the internal state of f2" will contain the <e_1^2, e_2^1> derived event, that should not be used after the first detection, which consumes e_1^2. In order to get rid of derived events, already stored at f2" internal state, but (partially) consumed, we constrain f2" to include only instances not consumed yet at its matching sets, by extending f2" pattern assertion with validation check against a consumed events pool.

Figure 7 and Table 4 summarize the described observations:

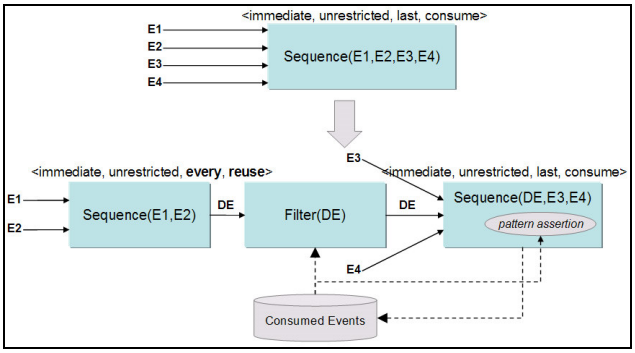

Figure 7: Pattern rewriting using extended pattern assertion

Table 4: Pattern rewriting by policies modification (3)

Pattern type: sequence/all			
Policy	f1	f2'	f2"
Evaluation	immediate	immediate	immediate
Cardinality	unrestricted	unrestricted	unrestricted
Repeated Type	last	each	last
Consumption	consume	reuse	consume
filter between f2' and f2"			
extended pattern assertion in f2"			

In order to complete the picture of pattern rewriting with respect to the most useful set of policies, we bring an additional mapping (Table 5), employing the already mentioned rewriting techniques. (the "reuse" consumption policy, filter and extended assertion).

Table 5: Pattern rewriting by policies modification (4)

Pattern type: sequence/all			
Policy	f1	f2'	f2"
Evaluation	immediate	immediate	immediate
Cardinality	unrestricted	unrestricted	unrestricted
Repeated Type	first	each	first
Consumption	consume	reuse	consume
filter between f2' and f2"			
extended pattern assertion in f2"			

Policies not covered here (e.g. "deferred" evaluation policy and "reuse" consumption policy), pose similar rewriting challenges, thus we omit them in this paper.

The mapping of policies, as presented in this section, provides a initial set of techniques for pattern rewriting. An individual fine tuning (e.g., forcing ascending lexicographical order for the "first" repeated type policy in f2") for some of the presented rewritings is required; this is achieved by further extending of the pattern assertion on top of the predefined policies-based rewriting schemes detailed in tables 2 to 5.

3.6 Summary

Although the pattern assertion analysis presented in Section 3.3 was mainly motivated by the desired pattern rewriting, it can be used as white-box pattern optimization technique. One can examine pattern assertion, in terms of its independent parts, and apply an optimized assertion evaluation upon pattern detection attempt. As an example, the pattern detection procedure can be discarded when identified that a certain (independent) assertion part can not be satisfied with respect to a pattern internal state; that instead of performing a naïve nested loop on all candidates, in order to find an event combination satisfying the entire assertion.

A trivial observation from the analysis presented in this section is that rewriting techniques are also valid for a simpler case, when there is no pattern assertion, i.e. the pattern assertion is "true". This kind of assertion can be observed as separable at any i for RTS=<E_1, …, E_N>, thus inferring N-1 splitting alternatives.

In this section we have demonstrated rewriting techniques for the all and sequence patterns, splitting a single pattern into two consecutive patterns or unifying them. A formal definition and a proof were demonstrated for a representative case, evidencing rewriting logical validity and supporting our intuition of employing policies modification for rewriting.

4. BI-OBJECTIVE OPTIMIZATION OF SEQUENCE PATTERNS

In this section we discuss how the rewriting means gained by now are utilized to improve an application's performance according to predefined objectives. Section 4.1 defines various performance indicators of event processing systems, Section 4.2 identifies the tradeoff between these indicators and in sections 4.3 and 4.4 we discuss how pattern rewriting techniques can be leveraged to achieve the desired performance objectives.

4.1 Pattern Performance Objectives

Due to the rapid growth of event processing technology, large variety of application domains, and lack of standards [13], performance metrics are subject to various interpretations, often leading to incomparable product benchmarks. Among the most commonly used performance indicators are throughput and latency (response time), as well as scalability, security, correctness and other non-functional requirements.

When considering throughput and latency, various performance optimization goals exist, including max input/processing/output throughput, minmax latency, and minavg latency. The specific optimization goal function is typically derived from the application's requirements. For example, minmax latency should be employed by real-time systems that aim to provide an upper bound over the worst case performance, while minavg latency is a reasonable goal function for non real-time applications. Our work focuses on minavg latency and max input throughput performance functions, targeting what we believe are the most commonly used performance indicators in event processing applications.

Stimulated by the conceptual similarity of communication [10] and event processing networks, we define event processing application *processing throughput*, hereafter denoted by *throughput*, as an average rate of events the system can process. We use events per second (event/s) to measure throughput. Similarly to [2] and [13], we define a system *latency* as a delay between the last input event causing a certain scenario detection and the detection itself, resulting in derivation of an output event. An application's latency is usually measured in milliseconds (ms).

The throughput and latency of a single pattern detection EPA (which is a special case of an event processing network), are defined as the average rate of events that this EPA can process and the average delay from the detecting event arrival to pattern's detection, respectively. For instance, if E_4 is derived from the pattern matching of sequence(E_1,E_2,E_3), latency is the time interval between the detection time of E_3 (time it arrived to the system) and the detection time of E_4 (time it is derived by the system).

Performance tuning of pattern matching EPA is important when done as part of the entire application optimization; it becomes critical in systems where a pattern embodies entire path in an EPN, as well as in highly distributed systems, where a single pattern can be assigned to a processing node.

4.2 Performance Objectives Tradeoff

Throughput vs. latency tradeoff is a general phenomenon in performance oriented systems. We sometimes face a choice between doing eager and lazy evaluation, where eager evaluation performs computing when possible in anticipation that it will be used later, and lazy evaluation is done on demand. Recall that the latency of sequence(E_1, …, E_N) is defined as a response time between E_N instance arrival and emission of the derived event, if detected. According to this definition, the eager evaluation choice biases towards low latency and reduced throughput, since the (occasionally redundant) pre-processing is done in advance. The lazy evaluation, however, biases towards high throughput and increased latency, since the necessary processing is only triggered by E_N arrival. Intuitively speaking, we observe that improved latency comes at the cost of decreased throughput.

Pattern rewriting offers natural means for tuning the throughput vs. latency tradeoff for the sequence pattern. Consider again the speculative broker scenario: maintaining the entire logic in a single pattern (sequence(E_1^i, E_2^j, E_1^k, E_2^l, E_1^m, E_2^n)) produces a solution with high throughput. However, splitting the pattern into two patterns (sequence(sequence(E_1^i, E_2^j, E_1^k, E_2^l), E_1^m, E_2^n)), leaving only a light processing delta for the last, potentially detecting event (E_2^n), produces a low latency oriented alternative.

While the two rewritings above serve two extremes, highest throughput and lowest latency respectively, there is a range of Pareto optimal solutions on the throughput vs. latency tradeoff spectrum, realized by additional rewriting alternatives; these solutions are discussed in Section 5.2. Constrained by pattern assertion decomposability, the only additional rewriting in our example is sequence(sequence(E_1^i, E_2^j), E_1^k, E_2^l, E_1^m, E_2^n).

4.3 Bi-objective Goal Function

Our model enables a system designer to define a bi-objective performance function that stems from the application's characteristics. This is done by assigning a scalar weight for each objective to be optimized, i.e. weight of α to pattern throughput (th) and a complementary weight of 1-α to its latency (lt). Striving to minimize latency and to maximize throughput, the general form of the bi-objective performance function is min α*lt+(1-α)*(1/th). In addition to application requirements, the goal function is also affected by the application's properties. For example, for a pattern triggered by low-rate events we will rather focus on improving latency, adjusting the α weight accordingly.

Given a bi-objective performance function, a rewrite that optimizes this function is selected using an empirical simulation-based approach (see Section 4.4), given that the analytic approach poses challenges, including difficulty to estimate pattern internal state size and assertion satisfactory probability with respect to pattern internal state at a certain time point.

Application rewriting decisions can be done off line, therefore we can tolerate simulation-based optimization duration. In cases where the rewriting decision has time constraints, such as on-line ("hot") updates of an application definitions, a heuristic-based approach can be taken. While the heuristic approach discussion is beyond the scope of this paper, we discuss simulation-based techniques in details in the next section.

4.4 Simulation-based Optimization

Given a sequence pattern and a bi-objective goal function (g) of the form g=α*lt+(1-α)*(1/th), simulation-based optimization empirically analyzes all rewriting alternatives and selects the one optimizing the goal function, i.e. gaining the minimal value for g.

Applying the described rewriting techniques on a given pattern, we generate a number of logically equivalent alternatives $A=\{A_1, \ldots A_K\}$ that are subject for analysis. Our goal is to select the rewriting alternative that minimizes the *expectation* (E) estimator of g, i.e. we aim to find $\text{argmin}_{Ai}(\hat{g})$, where $\hat{g}$ is the estimated Expectation of α*lt + (1-α)*(1/th).

A close look at the presented goal function reveals anomaly that stems from its bi-objective character. In case both latency and throughput indicators have roughly the same scale and same relative importance, the function above will lead to a deceptive decision: it will constantly favor the latency part (α*lt) over the throughput part ((1-α)*(1/th)), due to the element of division in the latter one. A simple, yet satisfactory solution is to normalize both of them to the same order of magnitude.

Due to the lack of standard approaches to such normalization (to the best of our knowledge), we compute the throughput *coefficient* C by performing a series of latency and throughput estimations and multiply the right hand side of the goal function by it; intuitively speaking, we are adjusting both sides of the function g to the same scale. The final form of g therefore is

$$g = \alpha*lt + C*(1-\alpha)*(1/th)$$

and we aim to find $\text{argmin}_{Ai}(\hat{g})$.

For a set A of rewriting alternatives, expectation estimator and its 95% confidence interval are calculated for each $A_i \in A$ $1 \le i \le K$ by computing average and variance of N individual measurements:

$$\hat{g}_i = \frac{1}{N}\sum_{j=1}^{N} g_i^j \;,\; S_i^2 = \frac{1}{N-1}\sum_{j=1}^{N}(g_i^j - \hat{g}_i)^2$$

With sufficiently large number of measurements, e.g. N>120, the 95% confidence interval of $\hat{g}_i$ is calculated using the Central Limit Theorem, to be:

$$g_i \in \hat{g}_i \pm \frac{1.96 S_i}{\sqrt{N}}$$

After producing a set of estimators $\{\hat{g}_1, \ldots, \hat{g}_K\}$, the minimal one is selected, inducing the optimal rewriting alternative with respect to a given bi-objective goal function.

5. EVALUATION

Our evaluation has two main goals. Firstly, assessing the potential efficiency gains of pattern optimization; and secondly, providing empirical support for rewriting equivalence in addition to the theoretical background demonstrated in Section 3.

We conducted an empirical study for the sequence pattern, experimenting on a subset of pattern assertions, policies and relevant events sets. In our experiments we varied control parameters that impact performance the most. Among these parameters are event arrival rate (such as λ for Poisson distribution) and pattern assertion satisfaction probability for a randomly chosen attributes values. The latter can be tuned by varying the event attributes' domain size.

5.1 Experiment Setup

In our simulation framework event distribution types (e.g. Poisson or uniform), and arrival rates are configurable parameters. In addition, each pattern is examined within the temporal window (context) it is associated with, e.g. ten minutes window in the speculative broker scenario (Example 1).

Our framework assigns uniform distribution to event attribute values, i.e. each value in the domain has equal probability for random sampling. We use a uniform distribution of integer event attributes in [0,100] domain. Finally, the normalization coefficient C in the goal function was set to 3000 in all experiments.

The simulation was performed on a Lenovo T60 ThinkPad, with Intel Core 2 Duo 1.83-GHz processor, 2.00 GB memory, using Microsoft Windows XP operating system and Java 1.6.

5.2 Experiments

Extending the speculative broker scenario in Example 1, the following pattern was chosen for the basic experiment:

Pattern: sequence(E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8),
s.t. each event type Ei has a single integer attribute ID

Pattern assertion:

E_1.ID==E_2.ID AND E_3.ID==E_4.ID AND
E_5.ID==E_6.ID AND E_7.ID==E_8.ID

Pattern policies:

evaluation:	immediate
cardinality:	unrestricted
repeated type:	---
consumption:	consume
Pattern context:	10 minutes (fixed window)

The system parameters were configured as follows:

All event types follow the Poisson distribution with average rate of 50 events per second ($\lambda(E1)=\ldots=\lambda(E8)=50$).

Pattern assertion analysis draws four possible rewritings:

sequence(E_1, E_2 ,E_3, E_4, E_5, E_6, E_7, E_8) denoted by 0:4
sequence(E_1,E_2, sequence(E_3, E_4, E_5, E_6, E_7, E_8)) denoted by 1:3
sequence(E_1, E_2 ,E_3, E_4, sequence(E_5, E_6, E_7, E_8)) denoted by 2:2
sequence(E_1,E_2 ,E_3, E_4, E_5, E_6, sequence(E_7, E_8)) denoted by 3:1

The simulation results are shown in Table 6.

Table 6: Simulation results

#	rwrt.	throughput (event/s)	latency (ms)
1.	0:4	113	134
2.	1:3	80	50
3.	2:2	104	30
4.	3:1	44	14

In this case the highest throughput is achieved at the unified alternative (0:4); that, along with the worst latency, demonstrate the approach of performing pattern computation in lazy evaluation, i.e. on arrival of the (potentially detecting) E_8 event instance. Another extreme is the 3:1 alternative where almost the entire computation is done in eager evaluation, thus favoring latency over throughput. The first rewriting (0:4) will be selected by assigning $\alpha=0$ at the goal function, attempting to minimize $g=C*(1/th)$, and the last one (3:1) by assigning $\alpha=1$, thus gaining minimum for g=lt.

An interesting observation from Table 6 is that rewriting #2 was outperformed by rewriting #3 both in terms of throughput and latency, i.e. alternative #3 strictly dominates alternative #2, thus excluding #2 from the efficient set of rewritings. The complete mapping of optimal rewriting alternatives by α and goal function values inferred by it is presented in Table 7. Note that complying with the observation above, the 1:3 rewriting alternative (#2) is never chosen as optimal solution.

Table 7: Mapping of optimal rewritings by goal functions

#	rwrt.	$\alpha=0$	$\alpha=1$	$\alpha=0.5$	$\alpha=.25$	$\alpha=.80$
1.	0:4	26.55	134	147.27	53.41	112.51
2.	1:3	37.50	50	43.75	40.63	45.77
3.	2:2	28.85	30	29.42	29.13	29.77
4.	3:1	68.18	14	41.09	54.64	24.84

While the latency metric behaves in an intuitive way, constantly decreasing as more emphasis is given to eager evaluation, the throughput is more difficult to predict. As opposed to latency, a rewriting throughput is affected by both sides of a split, as well as by seemingly contradictory factors: easily satisfiable pattern assertion increases pattern throughput due to the decrease in processing time of a potentially detecting event, however each such detection involves a generation of a derived event and an invocation of the consecutive pattern part. As a result pattern rewriting throughput behaves in a non-monotonic fashion; thus there is a set of Pareto optimal solutions, drawing rewrites favorable over others for the same value of α. A Pareto efficient set of solutions for this experiment is illustrated in Figure 8.

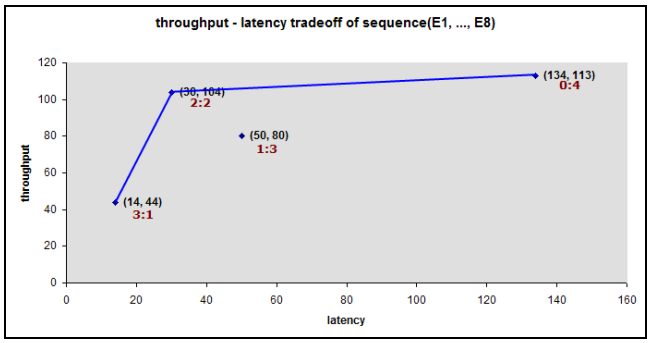

Figure 8: Pareto frontier for sequence(E1, …, E8)

Additional experiment was held on sequence with sixteen participating events (E_1, E_2, …, E_{16}) and an accordingly extended pattern assertion. A Pareto frontier generated by the simulation in this case is demonstrated in Figure 9.

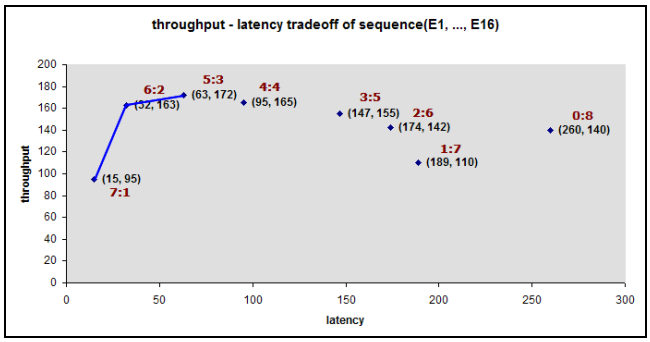

Figure 9: Pareto frontier for sequence(E1, …, E16)

An interesting observation from Figure 9 is that the initial pattern alternative (non-split pattern version – 0:8) is dominated by other alternatives with respect to both throughput and latency, making this most common implementation of sequence inferior to other alternatives. This observation remains sound when approaching rewriting of patterns with big relevant types set, making rewriting remarkably beneficial in these cases.

5.3 Sensitivity Analysis

Next we perform a sensitivity analysis of the goal function g in order to assess the relation between different parameters and the value of g. Such an analysis, yielding insights about the way various parameters impact the goal function, can serve as a basis for a heuristic approach for selection of a rewriting alternative.

We experimented with the 2:2 rewriting (alternative #3 in tables 6 to 7), varying event arrival rates and pattern assertion satisfaction probability. The schematic representation of this rewriting is depicted in Figure 10.

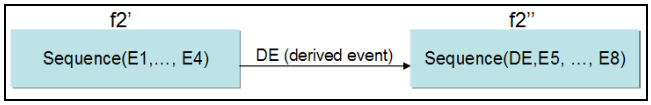

Figure 10: Experimental pattern rewriting

Experiment 1: Reducing f2' detecting event arrival rate, $\lambda(E_4)$: 50→25. The simulation results are presented in Table 8.

Table 8: Experiment 1 results

#	alt.	$\lambda(E_4)$=50		$\lambda(E_4)$=25	
		through.	latency	through.	latency
3.	2:2	104	30	134	26

While the latency remains nearly unchanged, the throughput improves. That can be explained by the reduced detection rate at f2', leading to reduced derived event generation rate and f2" invocation, which in turn results in a lower average processing time for each event instance; and thus, higher average throughput.

Experiment 2: Reducing f2" detecting event arrival rate, $\lambda(E_8)$: 50→25. Table 9 presents the simulation results.

Table 9: Experiment 2 results

#	alt.	$\lambda(E_8)$=50		$\lambda(E_8)$=25	
		through.	latency	through.	latency
3.	2:2	104	30	98	45

The reduced arrival rate of E8 gives rise to longer detection attempts at f2", since more E_5, … E_7 candidates are accumulated upon E_8 arrival, resulting in a higher latency.

Experiment 3: Relaxing pattern assertion at f2' by reducing f2' event attributes domain: [0,100]→[0,10]; thus increasing assertion satisfactory probability. Simulation results are presented in Table 10.

Table 10: Experiment 3 results

#	alt.	A at (E_1…E_4): [0,100]		A at (E_1…E_4): [0,10]	
		through.	latency	through.	latency
3.	2:2	104	30	67	27

Comparing this experiment with Experiment 1, the increased detection rate at f2' causes more detections, generations of derived events and invocations of f2". This, in turn, results in higher time for handling a single event, yielding lower throughput.

Experiment 4: Relaxing pattern assertion at f2" by reducing f2" event attributes domain: [0,100]→[0,10]. Table 11 presents simulation results.

Table 11: Experiment 4 results

#	alt.	A at (E_5…E_8): [0,100]		A at (E_5…E_8): [0,10]	
		through.	latency	through.	latency
3.	2:2	104	30	127	11

Pattern assertion relaxation at f2" makes detection easier, and therefore the entire rewriting benefits. A quick detection naturally leads to an improved overall latency.

5.4 Discussion

Our experiments fit the intuition for extreme cases, demonstrating the lowest and highest latency biased results for the eager and the lazy evaluation respectively. Reasonable results, with respect to latency, are provided for other rewriting alternatives, lying in between the two extremes.

Performance indicators show that the rewriting that favors latency over throughput carries more than fourfold increase in latency, along with almost half the throughput, when compared to its throughput-oriented alternative; therefore both versions yield a performance benefit. Moreover, in a basic experiment (Table 6), almost tenfold latency improvement was obtained (134 ms in 0:4 vs. 14 ms in 3:1). The Pareto frontier of goal function values points out alternatives unconditionally favorable over others, assisting in the choice of the desirable rewriting alternative.

Although the discussion on indicator tradeoffs and experiments were conducted on the sequence pattern, the entire approach can be extended to additional pattern types, e.g. all. The theoretical background presented in Section 3 remains sound for pattern of type all, and the empirical evaluation is easily extendable.

Commercial products present benchmarks with seemingly much higher throughput; however these benchmarks are typically conducted for simpler cases, e.g. stateless EPAs and simple patterns [1][14]. As noted in the introduction, experimental studies indicate that the performance indicators between simple and complex scenarios have ratio of 1:90 both in terms of latency and throughput. Our implementation in this work introduces results that are fairly comparable to commercial tools for high end cases in the complexity scale.

6. RELATED WORK

Various types of optimization techniques have been discussed in the event processing literature [8][9][22][25]. According to Etzion and Niblett [5], these optimizations are typically based on a single objective, optimizing either latency or throughput, and can be classified into three categories: (1) optimization related to application components assignment: partitioning, distribution, parallelism and load balancing; (2) optimization related to the coding of a specific component: code optimization and state management; (3) optimization related to the execution process: scheduling and routing optimizations. Our work falls into the second category. The idea of rewriting in event processing networks was suggested by Perrochon et al. [17] more than a decade ago. Perrochon mentions some performance oriented optimization techniques, such as compression (e.g., grouping two agents into a single one), common sub-expression elimination (e.g., extracting commonly used part into a separate agent) and reordering (e.g., pushing the filter to an early point of the event processing network). [17] offers an informal description of the techniques above, neither considering pattern assertion, nor pattern policies.

Simple cases of pattern rewriting for application optimization are discussed by Poul et al. [18]. The authors present an event processing language that facilitates rewriting of "next" (sequence) and "union" (any) patterns for optimized deployment of event processing system across multiple machines. Considering only a binary sequence pattern, their rewriting rationale is to minimize the rate of intermediate derived events generation given event arrival distribution and rate. The proposed model assumes a single set of policies, and does not employ pattern assertion.

Another work that mentions pattern rewriting is [2] where the authors present an approach for communication-efficient complex event detection over distributed sources. A multi-step detection plan for a complex event is generated on the basis of event frequency statistics, postponing the monitoring of high frequency events to later steps in the plan. Events relevant to each step are pulled from a distributed source, carrying a potential reduction in transmission cost. As an example, the "seq"(sequence) operator is rewritten into several consequent sequence sub-operators, and execution of each sub-operator is conditional upon the detection of earlier ones,

eliminating the need for communication in some cases. While the authors proposed a cost-based model for n-ary event operators, pattern assertion and policies are not considered.

Several research efforts were devoted to the formal definition of event processing languages. Early works [17] only cover a subset of the language constructs with partial functionality (limited set of operators, no policies). Recent papers [4][7] present a more comprehensive approach, but still lack some language aspects (n-ary event operators, some policies, cross-event pattern assertions). One of the most mature works in this field is [1] by Adi et al., which we refined to better fit the purpose of this work.

7. CONCLUSION AND FUTURE WORK

A methodic approach for pattern rewriting, presented in this paper, serves as a step forward in rewriting-based optimization of event processing applications targeting systems with both pattern complexity and performance requirements. The ability to automatically rewrite certain application parts and thereby gain an optimized, yet logically equivalent alternative, exists in most declarative languages (e.g. database queries). This capacity is still missing from the event processing domain and is required to help making event processing systems more pervasive and mature.

The pragmatic impact of this work will be in deploying the proposed techniques within existing products, by automatically rewriting application patterns for subsuming common logic and splitting for parallel execution (where possible) in the deployment phase. Rewriting the sequence pattern can be employed as an advanced technique for bi-objective tuning of the performance indicators, thereby yielding an optimization value.

The potential value of pattern rewriting leaves much for further exploratory and practical activities. Our future plans include the investigation of additional rewritings and suggesting an algorithm for rewriting of an event processing network. In the context of the sequence pattern, we plan to continue exploring the heuristic-based approach for selection of the rewriting alternative, thus providing a solution for online rewriting decisions.

8. REFERENCES

[1] Adi A., Etzion O. Amit - the situation manager. The VLDB Journal — The International Journal on Very Large Data Bases. Volume 13 Issue 2, 2004.

[2] Akdere M., Cetintemel U., Tatbul N. Plan-based Complex Event Detection across Distributed Sources. In proceedings of the VLDB Endowment, Vol. 1. 2008.

[3] Calvanese D., Giacomo G., Lenzerini M., and Vardi M. What is Query Rewriting? In proceedings of the 7th International Workshop on Knowledge Representation meets Databases (KRDB), 2000.

[4] Cugola G., Margara A. TESLA: A Formally Defined Event Specification Language. In proceedings of DEBS 2010.

[5] Etzion O., Niblett P. Event Processing in Action, Manning Publications, 2010.

[6] Goodstein R.L. Boolean Algebra. Dover Pubns, 2007.

[7] Hinze A. and Voisard A. EVA: An EVent Algebra Supporting Adaptivity and Collaboration in Event Systems. ICSI Technical Report TR-09-006. International Computer Science Institute, 2009. http://www.icsi.berkeley.edu/cgi-bin/pubs/publication.pl?ID=002680.

[8] Khandekar R.et al. COLA: Optimizing Stream Processing Applications via Graph Partitioning, Middleware 2009.

[9] Lakshmanan G., Rabinovich Y., Etzion O. A stratified approach for supporting high throughput event processing applications. In proceedings of DEBS 2009.

[10] Leon-Garcia A., Widjaja I. Communication Networks: Fundamental Concepts and Key Architectures. Second edition. McGraw-Hill. 2004.

[11] Luckham, D. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley, Boston, 2002.

[12] Maheshwari S. Optimize the performance of condition evaluation part of rule sets. http://drupal.org/node/761624.

[13] Mendes M., Bizarro P., Marques P. A Framework for Performance Evaluation of Complex Event Processing Systems. In proceedings of DEBS 2008.

[14] Mendes M., Bizarro P., Marques P. Benchmarking event processing systems: current state and future directions. WOSP/SIPEW 2010: 259-260.

[15] Moxey C. et al: A Conceptual model for Event Processing Systems, an IBM Redguide publication. http://www.redbooks.ibm.com/redpapers/pdfs/redp4642.pdf

[16] Nilsson N. Principles of artificial intelligence. Tioga, 1980.

[17] Perrochon L., Kasriel S. and Luckham D. Managing Event Processing Networks. Technical report CSL-TR-99-788, Stanford University Computer Systems Lab, 1999. ftp://reports.stanford.edu/pub/cstr/reports/csl/tr/99/788/CSL-TR-99-788.pdf.

[18] Poul N., Migliavacca M., Pietzuch P. Distributed Complex Event Processing with Query Rewriting. In proceedings of DEBS 2009.

[19] Sharon, G. and Etzion, O. Event Processing Networks: model and implementation. IBM System Journal, 2008, 47(2), pages 321-334.

[20] Stoy J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, Massachusetts, 1977.

[21] Strom R., Dorai C., Buttner G., Li Y. SMILE: distributed middleware for event stream processing. In proceedings of the 6th international conference on Information processing in sensor networks, 2007.

[22] Tatbul N., Cetintemel U., Zdonik S. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In proceedings of VLDB 2007.

[23] FIU's (Financial Intelligence Units) in action. http://www.egmontgroup.org/library/download/21, page 80.

[24] W.M.P. van der Aalst, K.M. van Hee, J.M. van der Werf, and Verdonk M. Auditing 2.0: Using Process Mining to Support Tomorrow's Auditor. IEEE Computer, 43(3):90-93, 2010.

[25] Wolf J., Bansal N., Hildrum K., Parekh S., Rajan D., Wagle R., Wu K., Fleischer L. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems, Middleware 2008: 306-32.