Complex Event Service Provision and Composition based on Event Pattern Matchmaking

Feng Gao
INSIGHT @ NUI Galway,
The DERI Building,
Galway, Ireland.
feng.gao@nuigalway.ie

Edward Curry
INSIGHT @ NUI Galway,
The DERI Building,
Galway, Ireland.
edward.curry@nuigalway.ie

Sami Bhiri
Computer Science
Department,
TELECOM SudParis, France.
sami.bhiri@telecom-sudparis.eu

ABSTRACT

Service computing has gained great success because it decouples service providers and consumers. Because of this decoupling, it is possible to reuse software applications without knowing their implementation details. However, business applications consuming complex events (and complex event processing systems in general) are not maximising the full benefits from service computing. Current service models are not suitable for describing complex events and the requirements for enabling pattern-based complex event service composition are not fully addressed. In this paper, we propose a complex event service description model that extends OWL-S and captures the exact semantics of complex events, including their patterns and attributes. We propose algorithms to create event service compositions based on event patterns. These algorithms are capable of selecting the composition plans with lowest estimated data traffic demands over the service network. Moreover, we show how to improve the efficiency of event service composition by indexing event patterns.

1. INTRODUCTION

In the context of Business Process Management (BPM), business processes are typically interested in complex events with business values rather than primitive events representing simple changes of states. For example, a supermarket manger may be interested in of the sales for a certain product during the past season rather than a barcode scan event at a counter; a bank is interested in detecting fraud use of credit cards by identifying a set of unusual purchase events rather than a single payment transaction. That said, a business/complex event is typically detected based on the occurrence of a primitive event sequence. The term *Event Pattern* is widely used in research and industry to describe how a complex event can be detected from a set of correlated events called its member events. Event patterns can be formally described with Event Pattern Languages (EPL), e.g., RAPIDE[11], and event engines will subscribe to the primitive event streams specified in the patterns and then evaluate the rules and constraints upon real-time event data streams.

Despite the extensive research efforts spent on Complex Event Processing (CEP), providing CEP applications as reusable services that allow the composition of complex event services based on event patterns efficiently is still a challenging task. Many existing event service description and discovery mechanisms are topic or content based, which is sufficient for reusing primitive/simple event services. However, it is impossible to reuse a complex event without knowing its exact semantics expressed in the pattern.

Providing events, especially complex events through services has many benefits. First, the loose-coupling nature of service computing helps to automate the process of business event implementation, which is a non-trivial task in BPM. Typically, to implement a business event, a developer needs to write middleware to prepare event streams for event engines using the data coming from event sources. Then a business analyst/expert must familiarize themselves with the specific data formats and encoding of the event streams provided by event middleware as well as the syntax of platform-dependent event pattern languages, so that they can specify the event patterns accordingly. With the ability of event service description and discovery, event publishers can advertise their event semantics and data formats publicly and event consumers can specify their request (including event patterns) in a platform-independent way.

A second important advantage of providing event services is that the CEP capability can be reused. By reusing business event services instead of subscribing directly to primitive event streams, the amount of events delivered through the network can be greatly reduced. This is important for CEP applications in general because it reduces the use of bandwidth and CPU, resulting in more efficient and in-time event detection. For example, if a user is interested in detecting heating system failures, he might specify an event pattern as below:

> *If the heater is on, and 3 temperature drop events happened in 5 mins, alert the technician to check the heating system.*

Suppose there are two sensors deployed in the same room: a temperature sensor reporting current temperature readings and a heater monitor reporting the status of the heater. If a complex event subscribes to these two sensor event streams directly, to detect a heater failure situation at least 5 events will be consumed: 4 temperature readings to detect 3 consecutive temperature drops, and 1 heater status report. If there exists already a temperature drop monitor that is reporting 3 temperature drops during the past 5 minutes, with the event service discovery and reuse mechanism the complex event can subscribe to this event stream instead of temperature reading events. As a result, the detection of the situation may consume just 2 events at least. The above scenario is a simplest demonstration of the benefits of reusing event services. In large-scale systems, e.g., smart city applications, the number of event notifications reduced by reusing existing event services can be significant.

To provide business/complex events as reusable services and facilitate more efficient event processing systems, the following sequence of questions needs to be answered.

1. How to describe event services properly so that event service matchmaking based on event patterns and event attributes can be realized?
2. How to determine if two event patterns are functionally equivalent (i.e., produce the same complex event notifications), provided that different event patterns may have identical meanings?
3. How to choose optimal event service composition plans that consumes the least amount of input event data?
4. How to derive event service compositions efficiently for very complicated event patterns (i.e., with a lot of event rules) and in a large scale event marketplace?

This paper provides answers to the above questions. The remainder of the paper is organized as follows. Section 2 discusses related work; Section 3 answers the first question by proposing an event service model. Section 4 answers the second question by presenting the abstract syntax of event patterns, and then it describes the operations and algorithms to reduce event patterns into canonical forms so that they can be compared. Section 5 answers the third and fourth questions, it first provides the definition of *Structurally Optimized* event service compositions based on Estimated Traffic Demand (ETD) of compositions, and provide way to calculate the ETD. Then it presents two composition algorithms creating structurally optimized event service compositions: a slow algorithm based on event substitution and a fast algorithm based on the reusability index of event patterns. Section 6 demonstrates the performance of the proposed algorithms with prototype experiments. Section 7 concludes the paper and discusses some possible future work.

2. RELATED WORK

Complex event service composition can be seen as a variant of service composition. However, current planning based service composition [13] only consider the matching of input/output message types and the evaluation of logical formulas in preconditions/effects. In complex event service composition, it not straightforward to define preconditions and effects for event detection tasks, nor is it enough to create composition plans based on matchmakings between event types. Rather, comparing event patterns/queries in event service descriptions/requests is essential to determine the reusability. In database systems, various techniques including query subsumption [4], multiple query optimization[15] and y-filter[5] etc. are developed to share and reuse partial results among similar queries in static databases. Our approach is inspired by query subsumption: we identify the subsumption (reusable) relationships between canonical event patterns.

Reusing event queries/subscriptons is also discussed in many other event based systems, including content-based event overlay networks [2, 3, 8, 7, 10, 6, 12] and CEP query optimization[1, 14]. In [6] and [3] reusability of event queries are evaluated based on the similarity between event attribute types and values, no event patterns are considered. In [2] and [7] only simple attribute filters and sequential event patterns can be reused by defining a subscription covering relation. In [8] and [10], brokers are equipped with event engines which makes them capable of processing more event operators, however, the decomposition of event subscriptions follows a top-down traversal on the query tree, until all primitive events identify their sources. Similarly in [1] and [14], two event queries are considered equivalent when their query trees are isomorphic. However, in our work, we prove that tree isomorphism is a sufficient but unnecessary condition for determining the equivalency of event patterns. Moreover, we also prove that it is not very efficient to perform a top-down traversing on query trees to decompose event patterns when we take combinations of sub-patterns into consideration.

E-Cube [9] is comparable to our approach, in the sense that it also analyzes the reusability between event patterns and organize them into a hierarchy. The main differences to our approach are 1) E-Cube only supports sequence and negation operators, whereas we support sequence, conjunction, disjunction and repetetion operators, 2) E-Cube has different rules in determining reusable relation between sequence patterns, e.g.: the event sequence (e_1, e_3) is reusable to (e_1, e_2, e_3) in E-Cube but not so in our approach, and 3) we provide an evaluation on the different performances of the hierarchy or non-hierarchy based composition algorithms.

3. SEMANTIC EVENT SERVICE MODEL

In this section we present a semantic event service model describing both event patterns and event attributes. This model allows (complex) event services to be discovered and composed based on event patterns and attributes. A high level overview of the event service model is shown in Figure 1. The event service model can be seen as an extension to OWL-S[1], which is a standard ontology for semantic web services, it can be integrated into the OWL-S model with concepts from an event ontology.

3.1 Service Model

We consider an event service should be described with a *Event Grounding* and a set of *Event Profiles*. The concept of *Event Grounding* is similar to *Service Grounding* in OWL-

[1]OWL-S: standardized semantic service description framework, `http://www.w3.org/Submission/OWL-S/`

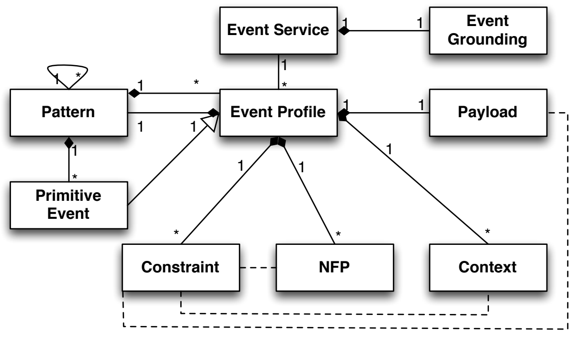

Figure 1: Overview of event service model

S, it tells a event consumer how to access the event service by providing information on service protocol, message formats etc. An *Event Profile* is comparable to the *Service Profile* in OWL-S, which describes the events transmitted by the service. Event profiles are key documents used for event service discovery. When an event service can produce different types of events, each type should be described by an event profile.

An event profile describes a type of event with four dimensions: *Pattern, Payload, Context* and *Non-Functional Properties* (NFP). *Pattern* refers to the member event correlations. An event pattern may have other patterns or (primitive) event services as components. An event profile without *Pattern* is considered a primitive event service, otherwise a complex event service. *Payload* refers to the type information for the event message and data. *Context* refers to the social and physical context of the event, e.g., by whom or at which location the event was produced. *Non-Functional Properties* refers to the Quality-of-Information (QoI) or Quality-of-Service (QoS) measures, e.g., precision, reliability, cost and etc. When the service model is used to formulate event service requests, *Constraints* can be specified by users to declare their requirements/preferences on the event payload, context or NFPs. We do not provide a dedicated *Service Model* as they do in OWL-S to describe the service composition details, because this information is specified already in event patterns.

3.2 Pattern Model

Since we are focusing on pattern-based event service discovery in this paper, we show more details on the *Pattern* models in Figure 2. The temporal relationships captured by an event pattern has three basic types: sequence, parallel conjunction and parallel alternation. If two events (or event patterns) are correlated by a sequence pattern, one should occur before the other, in parallel conjunction, both should occur and in parallel alternation, at least one should occur. Hence we define three types of patterns respectively: *Sequence, And* and *Or.* A special case of *Sequence* is that the sequence repeats itself for more than once, in this case the sequence can be modeled by a *Repetition* pattern, with a cardinality indicating the number of repetition.

Besides the temporal relationships, event pattern may also specify data constraints with *Filters* and *Aggregations*, also, a sliding window may be specified to indicate how many

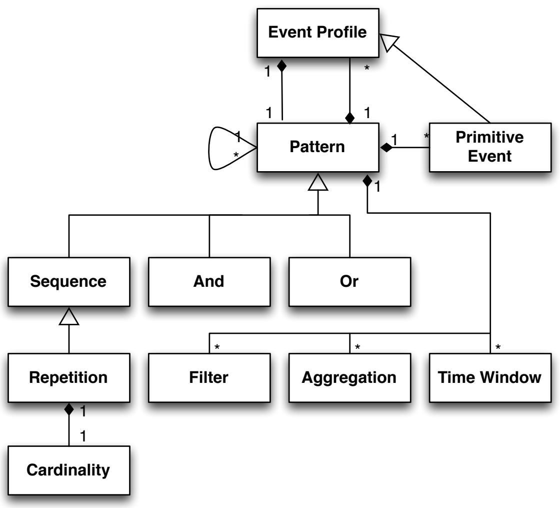

Figure 2: Event pattern model

events should the event engine keep in its memory at runtime. In the next section, we provide the abstract syntax of event patterns and elaborate in detail the semantics of the temporal and data constraints specified by event patterns.

4. ABSTACT SYNTAX OF COMPLEX EVENT PATTERNS

Using the semantic event service model, an event service provider can describe event services and store these service descriptions in a service repository; an event service consumer can formulate a event service query to specify his requirement on event services. These descriptions and queries are the key documents of the discovery and composition of event services. The discovery and composition of process break down to the matchmakings between event patterns and event attributes (i.e., NFPs and context) specified in event queries and descriptions. In this paper we focus on the matchmaking of event patterns, existing service matchmaking algorithms can be easily adapted for event attributes. To discuss the composition of event patterns, we need to describe them formally. In the following we give formal semantics for the event patterns described with our event service model.

An event can be described at two levels. The basic information required to reuse the event can be described at a surface level, it is sufficient to reuse primitive events using such basic information. The detailed information on the semantics of complex events can be described at a deeper level. Without such detailed semantics it is impossible to reuse complex events.

An *Event Declaration* describes the surface of a (complex) event without considering the event context and NFPs. It is a tuple $ed = (src, t, ep, D)$ where src is the service location where the events described by ed are hosted, t is the domain specific event type, ep is the event pattern for ed and D is its data payload.

An *Event Pattern* describes the detailed semantics of a complex event. It is a tuple $ep = (w, ED, OP, E, S, F)$ where

- w is a sliding window specified for p, we only consider w as a duration of time;
- ED is a set of member event declarations, we denote D' as the payload of $ed' \in ED$;
- OP is a set of operators, $op \in OP = (t_{op}, r)$ where $t_{op} \in \{Seq, Or, And, Rep\}$ is the type of operator, $r \in \mathcal{N}^+$ is the cardinality of repetition, $r > 1$ for repetition operators, and $r = 1$ otherwise;
- $E \subset (OP \times (OP \cup ED))$ is a set of asymmetric relations on operators and member events, it captures the provenance (i.e., causal) relation within ep, $\forall (op, n) \in E$, the execution of operator op relies on the execution result of n when $n \in OP$, or the occurrence of n when $n \in ED$;
- $S \subset (OP \cup ED) \times (OP \cup ED)$ is a set of asymmetric relations on operators and member events, it gives the temporal order within ep, $\forall (n_1, n_2) \in S, \exists n \in OP \wedge (n, n_1), (n, n_2) \in E \wedge n.t_{op} = (Seq|Rep)$, also, the occurrence of n_1 (if $n_1 \in ED$) or the last member event instance that completes the execution of n_1 (if $n_1 \in OP$) should happen before that of n_2;
- $F \subset (D \cup D)' \times (D \cup D' \cup V)$ is a set of filters representing binary relations (e.g., $>, <, =$) between two data payloads or payloads and values (V).

An event pattern defined according to the above semantics can be organized into a tree structure to have a more intuitive representation, called an event syntax tree. In the following sections we elaborate how event patterns can be mapped to a syntax tree, and then derive the canonical form of syntax trees, so that we can determine if two event patterns are semantically equivalent (in order to discover event patterns) by comparing their canonical syntax trees.

4.1 Syntax Tree of Complex Event Pattern

An event syntax tree describes an event pattern with a tree. More formally, for an event pattern ep, a syntax tree $T(v) = (V, E)$, where $V = (OP \cup ED)$ is the set of vertices representing operators and member events, $v \in V$ is the root node, typically an operator, and E is the set of directed edges representing the provenance relation. If $(v_1, v_2) \in E$, v_2 is called a child node of v_1, and the event syntax tree represented by $T(v_2)$ is a direct sub-tree (DST) of $T(v_1)$. If $(v_1, v_2) \in S$, then v_1 is to the left of v_2. Each node in V is labeled with its type, repetition cardinality (omitted if $r = 1$) and data payload (if any). A filter on a single payload is attached to the node labeled by the payload. A filter on two payloads is attached to the lowest common ancestor (LCA).

In a syntax tree, the *depth* of a node is the number of edges connecting the node to the root, the *height* of a tree is the maximum depth of its leaves, the *degree* of a node is the number of its child nodes. An example of using a syntax tree to capture event semantics for a complex event is shown in Figure 3.

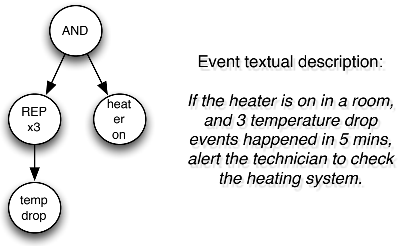
AND
REP x3
heat er on
temp drop
Event textual description:
If the heater is on in a room, and 3 temperature drop events happened in 5 mins, alert the technician to check the heating system.

Figure 3: Example of a syntax tree

4.2 Complete Syntax Tree

Event declarations (leaf nodes) in a syntax tree can expand into syntax trees to reveal the event patterns of their own. An event syntax tree is *complete* if all its leave nodes are primitive events. By checking recursively the event pattern definitions of the member event profiles, it is trivial to build the complete syntax tree for a complex event. A complete event syntax tree gives complete information on the logical rules specified for a complex event, as well as how the event is implemented over the event service network. We define a $f_{complete}$ function that creates the complete syntax tree as follows.

Definition 1 $f_{complete} : P \longrightarrow T$, *$P$ is a set of event patterns and T is a set of event syntax trees.* $p \in P, t \in T \wedge t = f_{complete}(p) \iff$ *t is the complete syntax tree derived from p.*

4.3 Irreducible Syntax Tree

Complete syntax trees are not sufficient for event pattern discovery and composition, because an event pattern can use different complete syntax trees to express its semantics. As such, we need to have a unique representation for each event pattern. An event syntax tree is *irreducible* if it contains the least number of nodes and edges while carrying the same semantics.

We consider two major types of syntax tree reducing operations: lift and merge. *Lifting* removes the redundant event operators, resulting in a lower height of the tree. *Merging* removes overlapping member event nodes, resulting in a less degree of the nodes in the tree[2]. Different rules apply when performing lift and merge operations on different types of nodes. Examples of lifting and merging operations are shown in Figure 4.

- **Sequential Lift:** when a node and its child are both sequence operators, the child node can be removed. Incoming edges on the child node are removed while all outgoing edges will attach their sources to the parent node. The payload and filters attached to the removed node are merged with the parent.

[2]The sequential and repetition merge exemplified in Figure 4 do not decrease the total number of nodes in the tree because they only merge two nodes, for conformance reasons we still consider such merging necessary to create irreducible trees.

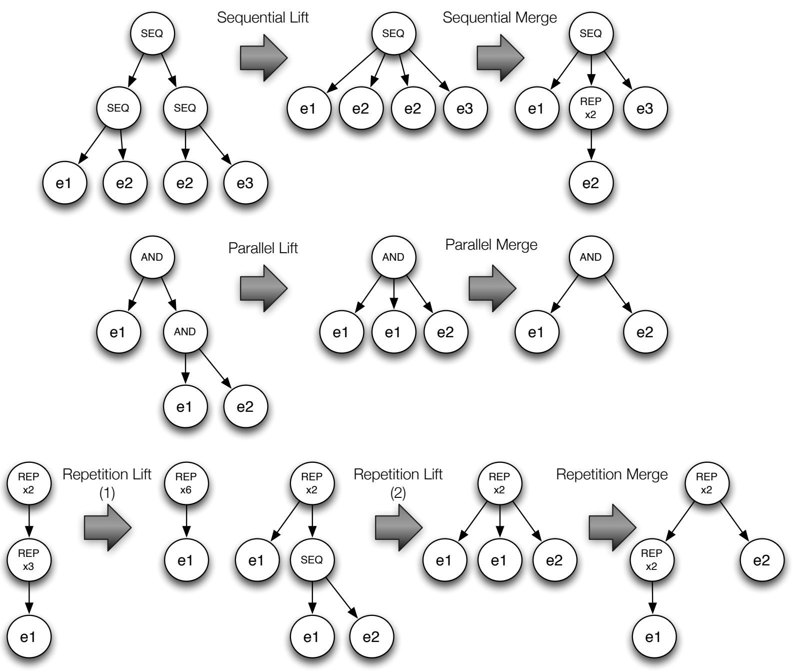

Figure 4: Examples of syntax tree reduction operations

- **Sequential Merge:** when a node is a sequence operator and there is a repeating sequence in its child nodes (recurring primitive event or DST sequences), a repetition node is inserted as a child node of the sequence operator. Repeated sequences are merged into one and relocated under the inserted repetition node. The cardinality of the repetition node is determined by the number of occurrences of the sequence.

- **Parallel Lift:** when a node and its child are the same type of parallel operator (conjunction or alternation), the child node can be removed (as the sequential lift).

- **Parallel Merge:** when child nodes of a parallel operator has duplicates (recurring primitive events or DSTs), duplications are removed. When the only differences of two child nodes n_1, n_2 (or DSTs $T(n_1), T(n_2)$) are the filters attached, and each filter in n_1 (or $T(n_1)$) is *covered*[3] by the corresponding filter in n_2 (or $T(n_2)$), then these two nodes (or DSTs) can be merged. For conjunction operators in this case, n_1 ($T(n_1)$) is kept, for alternation operators, n_2 ($T(n_2)$) is kept. Additionally, there is a special case for conjunctional merge: when a conjunction operator has two repetition DSTs with only different cardinalities, the DST with less cardinality is removed.

- **Repetitional Lift:** when a node is a repetition operator (with cardinality n), and it has only one child node which is also a repetition (with cardinality m), the child node is removed and the cardinality of the parent node is changed into $n \times m$. Otherwise, if the child node is a sequence operator, the child node is removed.

[3]f_1 covers $f_2 \iff P(f_1) \supseteq P(f_2)$, where $P(f_1), P(f_2)$ are the notifications produced by filters f_1, f_2, respectively.

- **Repetitional Merge:** merging operation for repetition nodes is the same as a sequential merge.

- **Special Lift:** when a sequence or parallel operator has only one child, this operator is removed. Such situations only happen during the reduction process.

4.4 Syntax Tree Reduction Algorithm

The algorithm to create irreducible syntax trees is shown in Algorithm 1. The algorithm traverses a syntax tree from the bottom to the top. The algorithm starts with lifting the whole tree to remove redundant operators. Then, it tries to merge sub-trees on the maximum depth, i.e., sub-trees whose root depths are equal to the height of the whole tree minus one. If these sub-trees are merged, we check if they can be lifted again because merging could create further redundant operators. After merging and lifting all sub-trees on same depth, we decrease the depth and repeat the merging and lifting process until the whole tree is merged (and possibly lifted again).

In the algorithm, line 2 uses the method *getHeight* to compute the height (maximum maximal depth) of a syntax tree. Line 9 uses the method *getSubTreesByDepth* to retrieve all sub-trees within a syntax tree whose root is of a certain depth. The *merge* method used in Line 11 merges the direct sub-trees of a certain node. The *liftTree* method in Line 7 and 13 carries out the lifting operations on a sub-tree.

Using the above algorithm, we define f_{reduce} and $f_{canonical}$ functions as follows:

Definition 2 $f_{reduce} : T \longrightarrow T$, *$T$ is a set of event*

Algorithm 1 Creates an irreducible syntax tree from a complete syntax tree ST.

```
Require: Syntax Tree $ST$.
 1: procedure REDUCE($ST$)
 2:     $height$ = getHeight($ST$)
 3:     if height<1 then
 4:         exit
 5:     end if
 6:     $root \leftarrow$ getRoot($ST$)
 7:     LIFTTREE($root, ST$)
 8:     for $height - 1 \rightarrow rootDepth \rightarrow 0$ do
 9:         $nodesToMerge \leftarrow$ getNodesByDepth($ST, rootDepth$)
10:         for $node \in nodesToMerge$ do
11:             MERGE($node, ST$)
12:             if $merged$ then
13:                 LIFTTREE($node, ST$)
14:             end if
15:         end for
16:     end for
17: end procedure
```

syntax trees. $T_1, T_2 \in T \wedge T_1 = f_{reduce}(T_2) \iff T_1$ *is the result of executing Algorithm 1 with* T_2 *as input.*

Definition 3 $f_{canonical} = f_{reduce} \circ f_{complete}$

5. EVENT PATTERN DISCOVERY AND COMPOSITION

With the capability of deriving canonical event patterns, we now elaborate the mechanisms of complex event service discovery and composition. We start with the definition of event composition. An event composition is a process that takes a *query tree* and a set of event services as inputs and produces a set of *composition plans* as outputs.

A query tree is a complete syntax tree created by complex event designer/modeler. We assume all the primitive events in a query have user-defined event types in their event declarations, but the event source locations are missing. The mission of event composition is to find out where should these primitive events come from. Of course, the mission can be accomplished by simply discovering primitive event services using the event types and then filling the source locations for the primitive event declarations in the query, but that will demand a lot of data traffic from the primitive event services (as we have discussed ealier). Therefore, we need to reuse complex event services as well.

When a complex event service is reused, we replace an appropriate portion (sub-tree or part of sub-tree) of the query tree with the event declaration of the complex event service, which transforms the portion into a event declaration node (a leaf node) with a complex event type and a service location. When all the leave nodes of a query have such type and location information, we call the query to be *bound.* When the query is bound, we call it a composition plan. An example of a composition plan created with a query and a set of event services in shown in Figure 5. When the composition plan is generated, it can be implemented by transforming the plan into an event/stream query (e.g., EPL,EP-Sparql), along with a set of service subscription commands.

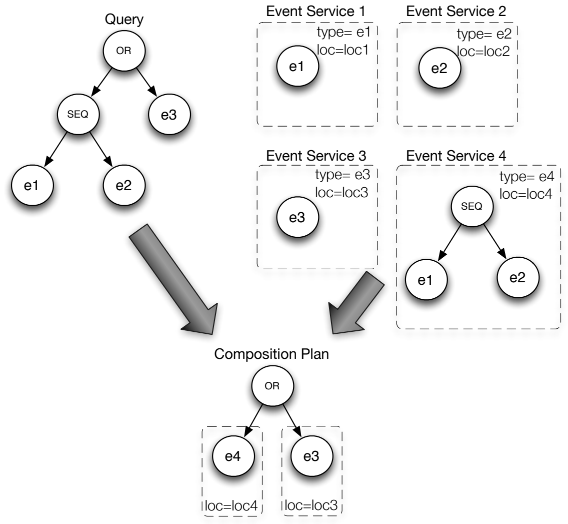

Figure 5: Example of a composition plan

The algorithms for event pattern composition have the following assumptions:

1. all events are instantaneous, which means each event has only one timestamp. In a complex event, the timestamp of the last detected member event is used as its timestamp;
2. all events delivered by event services are error-free, synchronized and complete;
3. all events have similar payload size;
4. in general, complex events are less frequently detected than their member events.[4]

The first two assumptions draw the scope for our discussion: we only deal with instantaneous events (while an event with a duration can be seen as a sequence of two instantaneous start and end events), and we do not deal with data quality or quality-of-service issues. The third and fourth assumptions allow us to propose a heuristic for achieving the goal of reusing event patterns: to minimize traffic, a complex event service composition should contain as few as possible the member event services, meanwhile, it should choose more coarse-grained member events. An event composition plan is said to be *structurally optimized* when it demands the least amount of traffic over the network per unit time.

[4]This assumption does *not* hold for alternation event patterns and their member events.

In the following, we first give formal definitions on structurally optimized event compositions, we also give means to select the optimized event compositions from a set of composition plans. Then, we present a slow algorithm which derives optimized event compositions by traversing top-down in the query tree to find *substitutes* for its sub-trees. Finally, we present a fast event composition algorithm based on the event pattern *reusability index*.

5.1 Structural Optimization based on Traffic Estimation

In Section 1 we informally define the term *structurally optimized* event composition without details on how different event compositions are evaluated and compared with regard to their degrees of optimization. Intuitively, they should be measured by the number of member event notifications delivered over the network per unit time. Ideally, if all the member event services are up and running and they provide statistics on the frequencies of event notifications, we can simply sum up these frequencies and derive the traffic demands for each composition. However, in realistic scenarios we cannot assume all event services provide such frequency monitoring operations. Even if they do, there are cases when a user needs to deploy a batch of complex event services, in which some services may be used in others' compositions and they do not have any statistics on their frequencies. Therefore, the ability to estimate the traffic demands and notification frequencies of complex events is necessary.

Given an event declaration $ed = (src, t, ep, D)$ and the complex syntax tree $T_c(v) = (V, E) = f_{complete}(ep)$ where $v \in V$ is the root node, we denote $\nu(n)$ as the frequency estimation of the member event represented by the sub tree $T_c(n) \subseteq T_c(v)$. Obviously, $\nu(v)$ is the frequency estimation of event described by *ed*. The traffic demand of *ep* is denoted $\mathit{Traffic}(ep) = \sum \nu(n)$ where $T_c(n)$ is the complete syntax tree of a member event service directly used in the composition of *ep*. Given node $n \in V$, $m \in V'$ where V' is the set of child nodes of n, the relation of $\nu(n)$ and $\nu(m)$ is given by Equation 1.

$$\nu(n) \begin{cases} = freq(n) & \text{if n is a primitive event then its frequency is given directly by } freq(n) \\ = \sum \nu(m) & \mathit{type}(n) = \mathit{Or} \\ = \min\{\nu(m)\} & \mathit{type}(n) = \mathit{And} \\ \leq \min\{\nu(m)\} & \mathit{type}(n) = \mathit{Seq} \\ \leq \dfrac{\min\{\nu(m)\}}{r} & \mathit{type}(n) = \mathit{Rep}, r = \mathit{card}(n) \end{cases} \tag{1}$$

In the above equation, we expect the *freq* function to give the frequency of a primitive event directly. The *type* function identifies the operator type for a node and the *card* function gives the cardinality of a repetition.

The equation allows us to calculate the maximum estimated frequencies for a set of member event services $(\max\{\nu(n)\})$, with which we can derive the maximum traffic demand estimation for an event composition plan that directly consumes these services. Then by choosing the plans with the minimal estimation, we can determine which plans are structurally optimized. However there is a limitation of Equation 1: filters are not considered. Indeed, filters may have a strong impact on the frequency. Unfortunately, it is impossible to estimate the impact without knowing beforehand the value range of data payloads and their distributions over the range.

5.2 Event Pattern Composition based on Substitution

Based on the above unique representation of event syntax trees and the exact semantics of event patterns, we now give the definition for the *substitute* relation between event patterns in Definition 4.

Definition 4 *substitute* $\subset P \times P$ *where* P *is a set of event patterns.* $substitute(p_1, p_2)$ *holds for* $p_1, p_2 \in P \iff f_{canonical}(p_1) = f_{canonical}(p_2)$.

From the above definition, if an event pattern is a substitute of another, they are semantically equivalent and can be seen as exact matches for each other during event service discovery.

Intuitively, to create an event composition, a top-down approach that finds substitutes for the event pattern (or its sub-patterns) is necessary. In the following we describe the mechanisms to create event compositions based on substitution, and then choose the structurally optimized compositions which demand the least amount of traffic over the network.

5.2.1 Substitution based Event Composition Algorithm

The top-down event composition algorithm based on substitution (Algorithm 2) traverses a query tree from the root node to the leaves to find substitutes for subtrees or different partitions of subtrees.

The *getSubstitutes* method in line 2 is a key operation in Algorithm 2, it retrieves the complex event declarations whose patterns are substitutes to the query. The algorithm first tries to find a identical tree from a list of candidate canonical trees for the whole query tree. If there is a match, it will replace the query tree with the matching event declaration node.

When there's no direct match for a query, the algorithm tries to find substitutes for sub-trees (or sub-tree partitions) of the query. If the root node of the query is a repetition operator, it will first change the cardinality of the operator to its factors (starting from the biggest factor) and try to find substitutes for all factors (including 1, which makes the repetition a sequence), if it failed, the algorithm is recursively invoked for each direct sub-trees (DSTs) of the root.

Algorithm 2 Creates optimal composition plans.

```
Require: Query Tree: $ST$, Candidate Trees: $cand$ Query Root: $root$.
 1: procedure COMPOSE($ST$, $cand$, $root$)
 2:     $matchingED \leftarrow$ getSubstitutes($ST, cand$)
 3:     if $matchingED \neq \emptyset$ then
 4:         replacePattern($ST, matchingED, root$)
 5:     else if $root.type$ = REPETITION then
 6:         $hasReplacement \leftarrow false$
 7:         for $f \in$ getFactors($root.r$) $\cup 1, r > f >= 1$ do
 8:             $newRoot \leftarrow$ createRepetition($root.r/f$)
 9:             $root$.setCardinality($f$)
10:             $matching \leftarrow$ getSubstitute($ST, cand$)
11:             if $matching \neq \emptyset$ then
12:                 $ST \leftarrow ST$.addRoot($newRoot$)
13:                 replacePattern($ST, matching, root$)
14:                 $hasReplacement \leftarrow true$
15:                 $break$
16:             end if
17:         end for
18:         if $hasReplacement = false$ then
19:             for $dst \in$ getDSTs($root, ST$) do
20:                 COMPOSE($ST, cand, dst.getRoot()$)
21:             end for
22:         end if
23:     else
24:         $hasMatchedPartition \leftarrow$ ANALYZEPARTITION($ST, cand, root$)
25:         if $hasMatchedPartition = false$ then
26:             for $dst \in$ getDSTs($root, ST$) do
27:                 COMPOSE($ST, cand, dst.getRoot()$)
28:             end for
29:         end if
30:     end if
31: end procedure
32:
Require: Query Tree: $ST$, Candidate Trees: $cand$ Query Root: $root$.
Ensure: Boolean $result$.
33: procedure ANALYZEPARTITION($ST, cand, root$)
34:     $partitions \leftarrow$ getDSTPs($ST, root$)
35:     $replacements \leftarrow \emptyset$
36:     for $dstp \in partitions$ do
37:         $replacement \leftarrow$ getSubstitutes($dstp, cand$)
38:         $replacements \leftarrow replacements \cup replacement$
39:     end for
40:     if $replacements \neq \emptyset$ then
41:         $replacements \leftarrow$ getBestReplacements($replacements$)
42:         replaceAll($ST, replacements, root$)
43: return $true$
44:     else
45: return $false$
46:     end if
47: end procedure
```

If the root is a sequence, conjunction or alternation operator, the algorithm will create different non-overlapping partitions (using the *getDSTPs* method in line 34, example of the operation is illustrated in Figure 6) with its DSTs. Then, the algorithm will try to find substitutes for each part in each DST partitions. Once all substitutes for a partition are found, the algorithm adds the composition plan for the partition to a list. After all possible partitions are investigated, the composition plan with the lowest traffic demand in the list will be picked (line 41) and corresponding replacements are made. If no partitions have complete substitutions, the composition algorithm is invoked on each DSTs of the root node.

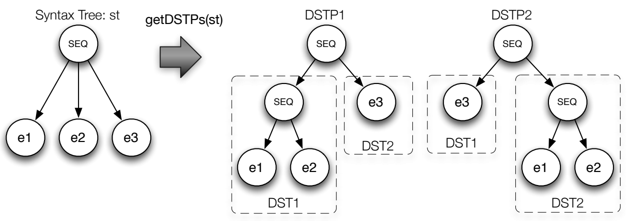

Figure 6: Example of creating DST combinations

5.2.2 *Complexity Analysis*

Algorithm 2 guarantees the creation of structurally optimized composition plans because all sub-trees and possible partitions of sub-trees are examined. However, it comes with the price of very high time complexity. The basic operation of the algorithm is the *getSubstitutes* method, which checks the *graph isomorphism* between a query and a candidate. The *getSubstitutes* operation needs to be executed for every sub-tree and sub-tree partition for the query, comparing with every existing candidate. Given n candidates, for a query with average height h and node degree d, the time complexity of the composition algorithm w.r.t. *getSubstitutes* is $\mathcal{O}((2^d)^h n)$. Clearly, the algorithm cannot scale and we need to have a much faster way to compose complex event services.

5.3 Event Pattern Composition based on Reusability Index

To accelerate the composition, a natural thought is to index the event syntax trees, so that for a certain sub-query (sub-tree or sub-tree combination), the number of examined candidates can be reduced. Additionally, if the index can tell which parts of a query can reuse existing syntax trees, the number of examined sub-queries can also be reduced. Therefore, we propose to build a reusability index for event syntax trees. In the following we first define the reusable relation between syntax trees. Then we use this relation to organize syntax trees into a hierarchy. Finally we show how this hierarchy is used to accelerate the event composition.

5.3.1 *Reusability of event patterns*

An event pattern is *reusable* to another, if the detection of the former can be used in the detection of the latter. We distinguish between *directly reusable* and *in-directly reusable* relations. An event pattern ep_1 is directly reusable to ep_2, denoted $R_d(ep_1, ep_2)$, *iff* ep_1 is a sub-pattern of ep_2, i.e., the canonical syntax tree of ep_1 is a sub-tree of the canonical syntax tree of ep_2. We formally define directly reusable relation R_d in Definition 5.

Definition 5 *$R_d \subset P \times P$ where P is a set of event patterns. $R_d(p_1, p_2)$ holds for $p_1, p_2 \in P \iff \exists T(v) \subseteq f_{canonical}(p_2)(f_{canonical}(p_1) = T(v))$. We denote p_1 is directly reusable to p_2 on node v.*

An event pattern ep_1 is in-directly reusable to ep_2, denoted $R_i(ep_1, ep_2)$, *iff* ep_1 is not directly reusable to ep_2, but ep_1 can be transformed into ep_1' using a sequence of operations on the canonical syntax tree of ep_1, as a result, it makes $R_d(ep_1', ep_2)$ hold. These operations have four types: $F_{filter} : T \times F \longrightarrow T$ attaches filters to the roots of syntax trees; $F_{multiply} : T \times \mathcal{N}^+ \longrightarrow T$ multiplies the cardinality of repetition of the roots; $F_{append} : T \times T \longrightarrow T$ adds a sequence of DSTs to the sequential roots as prefixes or suffices; $F_{add} : T \times T \longrightarrow T$ adds a set of DSTs to the parallel roots. In the above function definitions, T is a set of syntax trees, F is a set of filters. We formally define in-directly reusable relation R_i in Definition 6

Definition 6 *$R_i \subset P \times P$ where P is a set of event patterns. Given $T = \{$the set of all syntax trees$\}$, $\mathcal{N}^+ = \{$positive integers$\}$, $F = \{$the set of all filters$\}$, $F_{filter}, F_{multiply}, F_{append}, F_{add} = \{$sets of transformation functions$\}$; $R_i(p_1, p_2)$ holds for $p_1, p_2 \in P \iff \neg R_d(p_1, p_2) \wedge \exists p_1' \in P, T' \subset T, F' \subset F, n \in \mathcal{N}^+, T_1 = f_{canonical}(p_1), T_1' = f_{canonical}(p_1'), r = \{$the root node of $T_1\}, f_f \in F_{filter}, f_m \in F_{multiply}, f_{add} \in F_{add}, f_{app} \in F_{append}(R_d(p_1', p_2) \wedge$*

$$T_1' = \begin{cases} f_f(f_m(T_1, n), F') & type(r) = Rep \\ f_f(f_m(f_{app}(T_1, T'), n), F') & type(r) = Seq \\ f_f(f_m(f_{add}(T_1, T'), n).F') & type(r) = And|Or \end{cases}$$

Similarly, we denote p_1 is in-directly reusable to p_2 on node r.

We now formally define the reusable relation on event patterns R in Definition 7. An example of reusable relations is depicted in Figure 7

Definition 7 $R = (R_d \cup R_i)$

5.3.2 *Event Pattern Reusability Hierarchy*

With the reusable relation, we can build a hierarchy of canonical event syntax trees, called an Event Reusability Hierarchy (ERH). An ERH is a Directed-Acyclic-Graph (DAG), denoted $ERH = (T, R)$ where T is a set of nodes (canonical trees derived from patterns) and $R \subset T \times T$ is a set of edges (reusable relations) connecting nodes. Given an ERH $erh = (T, R)$, P is the set of event patterns of trees in T, $\forall (t_1, t_2) \in E$, $R(p_1, p_2)$ holds and $\nexists p_3 \in P$ such that

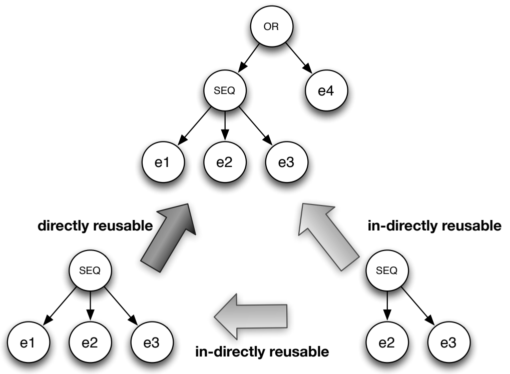

Figure 7: Example of event pattern reusability

$R(p_1, p_3) \wedge R(p_3, p_2)$, where $p_1, p_2 \in P$ are event patterns of t_1, t_2. According to this definition, if we build an ERH for the three event patterns in Figure 7, the edge at the top-right is ignored. The nodes do not reuse any other nodes are called roots in the ERH, the nodes cannot be reused by other nodes are leaves.

Constructing an ERH requires iteratively inserting canonical trees of event patterns into the hierarchy. If not all nodes can be inserted into a single ERH, we obtain a set of separated ERHs, called a Event Reusability Forest (ERF). The algorithm that inserts a node into a given ERF is shown in Algorithm 3.

Algorithm 3 Insert an canonical tree t to reusability forest *erf*.

```
Require: Canonical Tree $t$, ERF $erf$.
 1: procedure INSERT($t$, $erf$)
 2:     $roots \leftarrow$ getRoots($erf$), $leaves \leftarrow$ getLeaves($erf$)
 3:     $erf$.addNode($t$)
 4:     $parents \leftarrow$ getReusable($roots, t$)
 5:     drawEdges($parents, t$)
 6:     $childNodes \leftarrow$ getChildNodes($parents, erf$)
 7:     $parents \cup$ getReused($childNodes, t$)
 8:     drawEdges($parents, t$)
 9:     remove redundant edges
10:     if $parent$ is modified then
11:         go to 6
12:     end if
13:     perform reversed operations on $leaves$
14: end procedure
```

The above algorithm takes the canonical tree t of event pattern ep and an event reusability forest as inputs. As the first step, it finds all $p \in P$ where P is the set of nodes in the forest such that $R(p, t)$ holds, starting from the roots (line 4). Then the algorithm draws all edges for (p, t) and removes the redundant edges. As the second step, it draws all necessary edges for (t, p'), where $p' \in P \wedge R(t, p')$ holds. During the navigation of nodes, if a tree $t' = t$ is found, the algorithm terminates. This step is omitted in Algorithm 3 for brevity.

As mentioned above, finding reusable components or substitutes for a certain pattern can be achieved by the first step of the node insertion algorithm. Compared to Algorithm 2 in which all nodes may need to be compared, we can now use Algorithm 3 to prune the irrelevant parts of the hierarchy and reduce the number of comparisons required.

5.3.3 Event Composition Algorithm with ERF

Although we may improve the efficiency of the event composition by reducing the number of comparisons required, it comes with the price of more complicated comparisons. Reusability checking is based on *subgraph isomorphism*, which is a generalization of substitute checking and is NP-complete. Moreover, the full potentials of the reusability index are not exploited.

In fact, once a query tree representing an event pattern is inserted into the ERF, the components needed in the composition plans of the event pattern are prepared, even if no identical syntax trees are found for the whole query. All we need to do is to gather the parent nodes of the inserted query and replace appropriate parts of the query with the event declarations of these parent nodes. In cases when in-directly reusable nodes/sub-trees are replaced, additional transformation functions are invoked. If the replacement results in a bound query, the composition plan is derived, otherwise, the composition fails due to the lack of required primitive event services. The algorithm that accomplish this task is given in Algorithm 4.

Algorithm 4 Event composition for query Q with ERF *erf*.

```
Require: Query Tree: $Q$, ERF $erf$.
Ensure: Composition Plan $Q$.
 1: procedure COMPOSEWITHINDEX($Q, erf$)
 2:     INSERT($Q, erf$)
 3:     if an identical node is found then
 4:         $EDs \leftarrow$ event declarations of the identical node
    return getOptimal($EDs$)
 5:     end if
 6:     $parents \leftarrow$ getParents($Q, erf$)
 7:     for $p \in parents$ do
 8:         if $R_d(p, Q)$ then
 9:             directlyReplace($Q, p$)
10:         else
11:             inDirectlyReplace($Q, p$,getDSTs($p$))
12:         end if
13:     end for
14:     if $Q$ is bound then return $Q$
15:     end if
16:     Fail
17: end procedure
```

The *directlyReplace* operation in line 9 replace the sub-tree in Q that is identical to p with the optimal event declarations of p. When $R_i(p, Q)$ holds and $p' = F(p)$ is the transformed pattern, according to Definition 6, $f_{canonical}(p') \subseteq f_{canonical}(Q) \wedge$ the set of DSTs of $f_{canonical}(p)$ is a subset of the DSTs of $f_{canonical}(p')$. The *inDirectlyReplace* operation will replace all DSTs of p' in Q which are identical to the DSTs of p with the event declarations of p, with necessary filter attachments and cardinality changes.

Compared to Algorithm 2 which requires $\mathcal{O}(n(2^d)^h)$ graph isomorphism checks to find proper substitutes, Algorithm 4 only needs $\mathcal{O}(m)$ subgraph isomorphism checkings to find reusable components, while $m \leq n$ if all input parameters are the same. The efficiency of event composition is greatly improved by Algorithm 4. However, Algorithm 4 does not guarentee the results are structurally optimized, because with in-directly reusable components it only tries to cover the DSTs of the in-directly reused node with the candidate components once, i.e., different partitions of candidates are not exhausted and evaluated. Nevertheless, the algorithm still tends to reuse more coarse grained components, which gives relatively good composition plans.

6. EVALUATION

In this section, we evaluate the performance of the proposed algorithms with prototypes and simulation datasets. Three sets of experiments are conducted to evaluate the performance of the event query reduction (Algorithm 1), event reusability forest construction (Algorithm 3) and event compositions (Algorithm 2 and 4). In this section, we first present our general experiment settings, then, we elaborate the detailed settings for each experiment and explain the results.

6.1 General Experiment Settings

All experiments are carried out on a Macbook Pro with a 2.53 GHz duo core cpu and 4 GB 1067 MHz memory. Prototypes are developed in Java. The Java Virtual Machine is configured with a minimal heap size of 64 MB and a maximal heap size of 256 MB.

To have more accurate results, all results are averaged from 5 to 10 test iterations for each test setting. To ensure the test results are unbiased, we develop an Event Pattern Generator (EPG) to create random event patterns/queries. The EPG will choose the leaf nodes used in event patterns randomly from 10 different primitive events. The event operators are also randomly created as roots or intermediate nodes in query trees. To ensure the random event pattern creation stops at some point, also to have some control on the size of patterns (number of nodes in the query tree) created, EPG receives two parameters: height and degree, specifying the maximal tree height and degree of the output. EPG is used to create simulated datasets in our experiments.

6.2 Performance of Event Query Reduction

Query reduction is a basic and important operation in our prototype. To test its performance, we use EPG to generate 5000 event patterns. The query trees of these patterns have a maximum degree and height of 5. We invoke the query reduction algorithm on each pattern and group the execution time by the size of patterns. In this way we can derive the minimal, maximum and average reduction time for event patterns of different sizes. To obtain more accurate results, groups with less than 10 patterns are excluded from the results. Figure 8 shows the results of the experiment.

In the results we can see that most event patterns can be reduced to their canonical forms efficiently, in fact, 92% of the event patterns are reduced in less than 100 ms. However several "spikes" occur in the maximal reduction times, 2.4% event patterns took more than one second to be reduced, and in extreme cases it goes up to 8 seconds. After

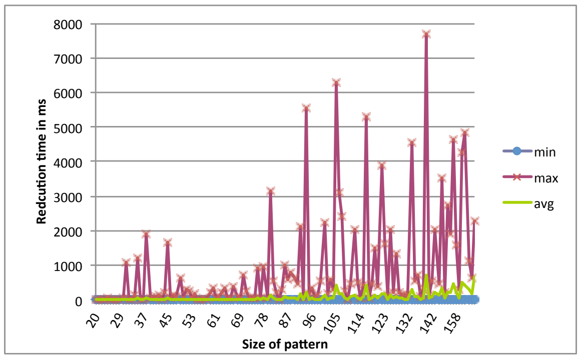

Figure 8: Execution time of query reduction

investigating the data set we find that these extremely long reduction time are due to the nested repetition nodes in the query tree. Since the repetition nodes are transformed into sequence operators during merge operations, they may significantly increase the total size of the pattern. As a result the merge operation may take much more time. A partial solution to the problem is to use faster graph isomorphism algorithms and accelerate the merge operations. In conclusion, the query reduction algorithm is efficient for most event queries.

6.3 Performance of Event Reusability Forest Construction

We evaluate the feasibility of ERF construction by measuring the time required. The EPG is used to create 100 to 1500 random event patterns with different maximal degree and height parameters. Then, we invoke the ERF construction algorithm on these sets of patterns and observe the time needed. Figure 9 shows the results of the experiment.

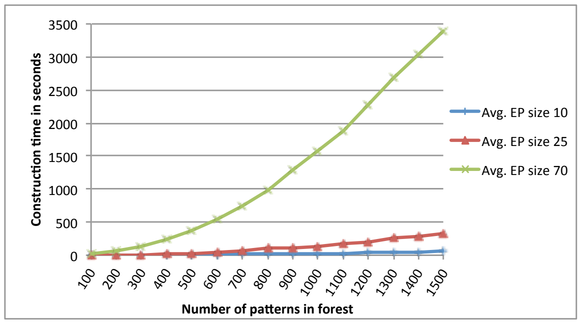

Figure 9: Execution time of hierarchy construction

In the above results, the lowest blue line indicates the time needed for constructing an ERF with sets of random event patterns with an average pattern size of 10 nodes, for 1500 event patterns, it took 58 seconds to complete the construction. Similarly, the red line in middle represents the set of event patterns with 25 nodes in average and completes the construction in 323 seconds. Finally, the green line represents the set of event patterns with 70 nodes in average and took nearly a hour to construct the hierarchy. The results indicate that for event patterns with around 25 nodes, inserting it into a 1500-node forest could take hundreds of milliseconds. However, inserting a large and complex event pattern with about 70 nodes into a large forest with 1500 very complicated event patterns could take more than 2 seconds.

6.4 Performance of Event Composition

To evaluate the composition algorithms, we compose the same sets of queries based on the same sets of candidate replacements/reusable components for both indexed and unindexed algorithms and compare their results. More specifically, we use EPG to create 500 and 1000 event patterns with an average pattern size of 25 nodes as candidates. Then we use EPG again to create 3 sets of event patterns as queries. There are 100 event patterns in each query set and their average pattern size are 10, 14 and 25 nodes. Figure 10 shows the time needed for each query set against each candidate set using indexed or unindexed composition algorithms.

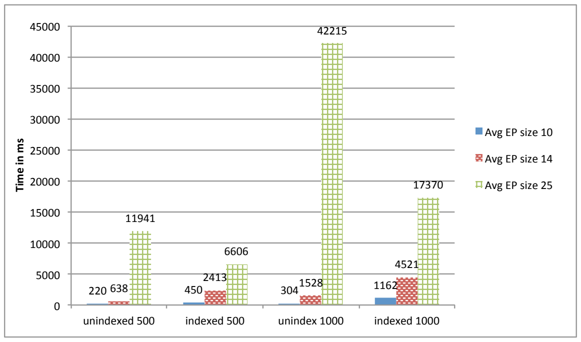

Figure 10: Execution time of composition: indexed vs. unindexed

The results indicate that for small event patterns, the unindexed approach out-performs the indexed one, but for large event patterns the indexed algorithm is much faster. This is aligned with our discussions in Section 5: reusability checking is more complicated than graph isomorphism, but the number of subgraphs compared is much less for reusability checking in large graphs. In fact, we also tested with 70-node queries, the indexed approach took 75 and 157 seconds to complete but the unindexed algorithm terminates before finish due to insufficient memory.

Another factor causing the slow indexed composition on small patterns is that the shape of the forest is too "flat" for random patterns, i.e., very few candidate event patterns reuse others. In fact, we observed that about 80% of the nodes in the random forests are roots which do not reuse other patterns. In such forests, the advantage of navigating the forest/hierarchy to avoid unnecessary comparisons is not significant. In real world scenarios, we have reasons to believe users may use existing event patterns as templates to create new ones, so that the probability of reusability can be higher than randomly created datasets. To evaluate the impact, we assign a reuse probability from 10% to 90% to the EPG and make it reuse existing patterns with the assigned rate. Figure 11 demonstrates the impact of reuse probability. It shows the time required for composing 100 14-node queries on 1000 14-node candidates created with different reuse probabilities. The results indicate that even for simple event patterns, the indexed approach is faster than the unindexed one when the probability of reuse is above 70%.

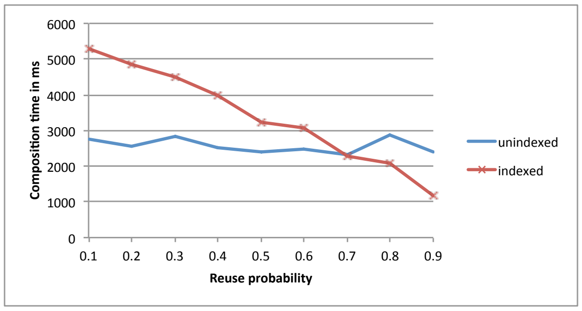

Figure 11: Impact of reuse probability on indexed composition

7. CONCLUSIONS AND FUTURE WORK

In this paper, we provide business events through services and enable the reuse of business event rules and patterns by providing pattern based event service discovery and composition. We propose an event service model to describe event service requests and advertisements with both event patterns and attributes. Then we provide an event pattern reduction algorithm to reduce event patterns into canonical forms so that they can be compared for semantic equivalence. Based on the semantic equivalence checking and the method to estimate traffic demands for pattern composition plans, we develop a event pattern composition algorithm that traverses the query tree in a top-down manner and finds replacements to derive *structurally optimized* compositions with the least estimated traffic demand. To improve the efficiency of composition algorithm, we propose to index event patterns with a *reusable* relation and develop another composition algorithm based the reusability index. In our prototype experiments we demonstrate the feasibility of our proposed approaches and prove that the indexed composition algorithm is more efficient when dealing with large event queries and for a large number of candidate event services.

Although we focus only on pattern based matchmaking for event services, in future we plan to take constraints on event attributes, especially non-functional properties (NFP) into consideration while creating event compositions. We need to develop algorithms to delegate constraints from complex event service to its members and propagate NFPs from member events to complex ones. Moreover, we will consider the means of adding more event operator (e.g.: aggregation) and transforming event patterns into stream queries so that they can be executed. Experiments on realistic datasets are also on agenda.

Acknowledgments

This work is supported by the EU FP7 CityPulse Project under grant No.603095. http://www.ict-citypulse.eu

8. REFERENCES

[1] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, Aug. 2008.

[2] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, Aug. 2001.

[3] E. Curry. Increasing mom flexibility with portable rule bases. *Internet Computing, IEEE*, 10(6):26–32, Nov 2006.

[4] S. M. Deen and M. Al-Qasem. A query subsumption technique. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications*, DEXA '99, pages 362–371, London, UK, UK, 1999. Springer-Verlag.

[5] Y. Diao and M. J. Franklin. High-performance xml filtering: An overview of yfilter. *IEEE Data Eng. Bull.*, pages 41–48, 2003.

[6] S. Hasan, S. O'Riain, and E. Curry. Approximate semantic matching of heterogeneous events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 252–263, New York, NY, USA, 2012. ACM.

[7] J. Keeney, D. Roblek, D. Jones, D. Lewis, and D. O'Sullivan. Extending siena to support more expressive and flexible subscriptions. In R. Baldoni, editor, *DEBS*, volume 332 of *ACM International Conference Proceeding Series*, pages 35–46. ACM, 2008.

[8] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware '05, pages 249–269, New York, NY, USA, 2005. Springer-Verlag New York, Inc.

[9] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta. E-cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 889–900, New York, NY, USA, 2011. ACM.

[10] Z. Long, B. Jin, F. Qi, and D. Cao. Reuse strategies in distributed complex event detection. In *Quality Software, 2009. QSIC '09. 9th International Conference on*, pages 325–330, 2009.

[11] D. Luckham. The power of events: An introduction to complex event processing in distributed enterprise systems. In N. Bassiliades, G. Governatori, and A. Paschke, editors, *RuleML*, volume 5321 of *Lecture Notes in Computer Science*, page 3. Springer, 2008.

[12] G. Mühl. *Large-scale content-based publish-subscribe systems*. PhD thesis, TU Darmstadt, 2002.

[13] J. Rao and X. Su. A survey of automated web service composition methods. In J. Cardoso and A. Sheth, editors, *Semantic Web Services and Web Process Composition*, volume 3387 of *Lecture Notes in Computer Science*, pages 43–54, 2005.

[14] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, 2009.

[15] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, Mar. 1988.