An Ontology-Based Publish/Subscribe System*

Jinling Wang[1,2], Beihong Jin[1], and Jing Li[1]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
{jlwang,jbh,lij}@otcaix.iscas.ac.cn
[2] Graduate School of the Chinese Academy of Sciences, Beijing, China

Abstract. Expressiveness and matching efficiency are two key design goals of publish/subscribe systems. In this paper, we introduce the Semantic Web technologies into the publish/subscribe system and propose an ontology-based publish/subscribe (OPS) system. The system can make use of the semantic of events to match events with subscriptions, and can support events with complex data structure (such as graph structure). An efficient matching algorithm is proposed for the OPS system, which can match events with subscriptions in a speed much higher than conventional graph matching algorithms. Therefore, the main contribution of our work is that it greatly improves the expressiveness of the publish/subscribe system without the sacrifice of matching efficiency.

1 Introduction

Publish/subscribe (pub/sub) is a loosely coupled communication paradigm for distributed computing environments. In the pub/sub systems, *publishers* publish information to *event brokers* in the form of *events*, *subscribers* subscribe to a particular category of events within the system, and event brokers ensures the timely delivery of published events to all interested subscribers. The advantage of pub/sub paradigm is that publishers and subscribers are full decoupled in time, space and flow [1], so it is well suitable for the large-scale and highly dynamic distributed systems.

In different distributed systems, the information exchanged between participants differs greatly in formats and semantics. If the pub/sub system is to become a general infrastructure for distributed computing and support different applications, it should have strong expressiveness, i.e.:

- It should support events in different formats and semantics;
- It should provide a powerful subscription language, so that information consumers can easily express their interest in certain events.

For each published event, the pub/sub system should match it with subscriptions to find out the interested subscribers. A large-scale distributed system may have millions of subscribers, and events may be published frequently. Therefore, the efficiency of the matching algorithm significantly affects the performance and scalability of a pub/sub system.

* This work was supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2002CB312005; the National Hi-Tech Research and Development 863 Program of China under Grant No. 2001AA113010; and the National Natural Science Foundation of China under Grant No. 60173023.

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 232–253, 2004.

There is a close relation between the expressiveness and the efficiency of matching algorithm for a pub/sub system. Generally speaking, the more expressive a pub/sub system is, the more difficult it is to design an efficient matching algorithm, and vice versa. On the one hand, the pub/sub system should have strong expressiveness to support more applications; on the other hand, the system should keep a high matching efficiency to ensure the scalability of the system. Therefore, expressiveness and scalability are two key goals of a pub/sub system that needs trade-off [2].

Although much work has been done on the research of pub/sub systems, there are still some problems in the expressiveness of existing pub/sub systems, such as:

1. The existing systems mainly use the structural information of events to match them with subscriptions, and they generally have no sense of the semantic of events. If the pub/sub system could match events with subscriptions based on both the semantic and the structure of events, it would be more intelligent and could better serve the distributed applications.
2. The existing systems can only support events with relational data structure (such as “attribute=value” pairs) or tree data structure (such as XML), but some distributed systems may require events to have more complex format (such as graph structure). Furthermore, events from different publishers may be in different formats. Therefore, a unified mechanism is needed to process events with different formats at the same time.

To solve the above problems, we introduce the Semantic Web technologies into the pub/sub system and propose an Ontology-based Publish/Subscribe (OPS) system. In the OPS system, the domain concepts involved in all events are integrated together to form a concept model, and the system matches events with subscriptions both semantically and syntactically. Inside the OPS system, each event is represented as a Resource Description Framework (RDF) [3] graph, which is a kind of directed labeled graph. As Tim Berners-Lee has stated [4], data in almost any form can be broken down into a representation as a directed labeled graph, and then be represented as RDF graph. Therefore, the OPS can support events in almost any complex format. When an event is published, it is firstly converted into a RDF graph before further processing. For subscribers, the received events are always in RDF format.

In the OPS system, subscriptions are represented as graph patterns, so the matching algorithm is in fact a kind of graph matching algorithm. Based on the characteristic of the RDF graph and a few constraints on the graphs and graph patterns, we designed a highly efficient matching algorithm for the OPS system. Experimental results show that under the same environment and workload, the matching time of our algorithm is much lower than existing graph matching algorithms. While there are 10,000 graph patterns in the system, the matching time for an input graph is just 1-2 seconds.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we introduce the data model of the OPS system. In Section 4, we introduce the subscription language supported by the OPS system. In Section 5, we give the matching algorithm. In Section 6, we present and analyze the experimental results. Finally, in Section 7, we conclude the paper with a summary.

2 Related Work

Pub/sub systems are generally divided into two categories: *subject-based* and *content-based*. In subject-based systems (such as IBM MQSeries [5]), each event belongs to one of a fixed set of *subjects* (also called *topics*, *channels*, or *groups*). Publishers are required to label each event with a subject name; subscribers subscribe to all events under a particular subject. In content-based systems, each subscriber defines a subscription according to the internal structure of events; all events that meet the constraints of the subscription will be sent to the subscriber. The content-based systems are more expressive and flexible than the subject-based pub/sub systems; they enable subscribers to express their interests in a finer level of granularity.

Existing content-based pub/sub systems can be further divided into two sub-categories: *Map-based* and *XML-based*. In Map-based systems, each event is a set of "attribute=value" pairs, and subscriptions are usually conjunctions of simple predicates on data attributes, which are called *flat patterns*. Known prototype systems include SIENA [6], Gryphon [7], JEDI [8], etc. In XML-based pub/sub systems, each event is an XML document, and subscriptions are usually XPath expression or its variations, which contain not only constraints on the structure of the XML documents but also constraints on certain elements and attributes. Such subscriptions are called *tree patterns*. Known prototype systems include XFilter [9], XTrie [10], WebFilter [11], etc.

Our OPS system differs from the existing content-based pub/sub systems in the following ways:

1. Most existing systems are not aware of the semantic of events, whereas the OPS system can match events with subscriptions based on both the semantic and the structure of events.
2. Events in the OPS system are represented as graphs rather than "attribute=value" pairs or XML, so the system can support events with more complex formats.
3. Subscriptions in the OPS system are *graph patterns*, which are more expressive than flat patterns and tree patterns.

In recent years, there are also some works on the research of semantic matching for pub/sub systems, such as S-ToPSS [12] and CREAM [13]. Our work differs from their works in that events are represented as RDF graphs and subscriptions are represented as graph patterns. Furthermore, our work is focused on designing an efficient matching algorithm for the system, while the matching efficiency issues are seldom touched in their works.

On the other hand, there have been a lot of algorithms for graph matching by the graph theory community. In this community, graph matching is divided into two types: exact matching and approximate matching; the matching problem in the OPS system belongs to the first one. The exact graph-matching problem is in fact the subgraph isomorphism problem. The classical algorithms for subgraph isomorphism are based on backtracking in a search tree, and using refinement procedures to prevent the search tree from growing unnecessarily large. Common refinement procedures include Ullman algorithm [14], forward checking [15], graph partition [16], etc. These algorithms can only work on the matching of one input graph and one graph pattern at a time. In [17, 18, 19], algorithms were proposed for applications where an input graph should be matched with a database of graph patterns. In the algorithms proposed in [17, 18], all graph patterns are organized into a hierarchical index structure, and the system traverses the hierarchical structure to find the matched graph patterns for a given input graph. However, these algorithms cannot find all matched graph patterns for an input graph. In the algorithm proposed in [19], every graph pattern is continuously decomposed into sub-graphs, until each sub-graph contains only one vertex. On the arrival of an input graph, the system first matches it with the smallest sub-graphs, and then assembles the matched sub-graphs into larger sub-graphs; finally get all matched graph patterns. But the algorithm can only support a database of tens or hundreds of graph patterns, and cannot serve the large-scale pub/sub systems where there are thousands or millions of subscriptions.

Compared with existing graph matching algorithms, the algorithm in the OPS system makes use of the characteristic of the RDF graph and adds a few constraints on the graphs and graph patterns, so it can achieve a matching efficiency much higher than existing ones. Furthermore, conventional graph matching algorithms are mainly focus on the matching efficiency when there are numerous vertexes and edges in graphs, while our algorithms are mainly focus on the matching efficiency when there are numerous graph patterns that need to be matched.

3 Data Model

In the OPS system, we use RDF and DAML+OIL [20] in the Semantic Web to describe the data model. The data model consists of following two parts:

1. Event model, which specifies the organization of data inside events. It is described with RDF.
2. Concept model, which specifies the concepts involved in the events, the relations between them, and the constraints on them. It is described with DAML+OIL.

Since RDF and DAML+OIL are mainly used to represent information on the Web, they use URI as the identifiers of different entities. However, an event-based system mainly cares about the events that are being exchanged, which usually don't have a URI. Therefore, when we represent events with RDF, they (and most entities inside them) are represented as *blank nodes* (nodes without URI). According to the RDF specification, a blank node can be assigned an identifier prefixed with "_:".

For the sake of clarification, we call nodes in a graph as *vertexes* and call nodes in a tree as *nodes* in the remainder of the paper.

3.1 Event Model

Inside the OPS system, each event is represented as a RDF graph. RDF is a way to represent fact using the (*subject*, *property*, *object*) triples. Each triple is called a *statement*, in which *subject* and *property* are URI, and *object* can be URI or literals. RDF data can be represented as a directed labeled graph, in which vertexes represent subjects and objects of statements, and arcs represent properties of statements. We call the RDF graph of an event as an *event graph*.

For example, in a pub/sub style Internet auction system, suppose Jinling Wang wants to sell an IBM Desktop PC at the price of $450, and the PC contains a 40G-size hard disk that was also produced by IBM, then the corresponding event graph is shown in Figure 1. For the sake of simplicity, we omit the "daml:Thing" vertex, the "rdf:Literal" vertex, and the arcs pointing to them in the Figure.

In the OPS system, we add the following restrictions on event graphs:

1. There is one and only one vertex in the graph that is called the *home vertex*, which describes the global information about the event (such as the type and the creation time of the event). We specify that the identifier of the home vertex is "_:H".

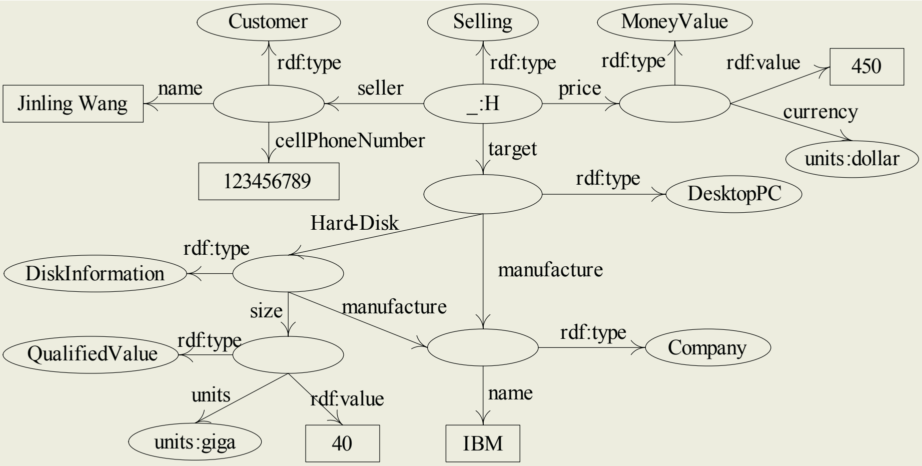


Fig. 1. An example of event graph.

2. There are paths from the home vertex to any other vertexes in the RDF graph.
3. Each vertex in the graph is a *typed vertex*, i.e., the graph specifies the class of each vertex. The publisher can specify multiple classes for a vertex, meaning that the entity simultaneously belongs to multiple classes.

3.2 Concept Model

The OPS system uses ontologies to represent the concept model of events, which describe not only the structural information but also the semantic information of events.

Ontology can be considered as a specification of a conceptualization [21]. It describes the concepts in a domain, the relations between them and the constraints on them. In the area of the Semantic Web technologies, one of the most influential ontology languages is DAML+OIL, which is used in the OPS system.

In the OPS system, the concept model of events is composed of the following three parts:

1. The description of classes and their hierarchical structure. An entity can belong to multiple classes. A class can have multiple parent classes, but the subclass-relations between classes must be acyclic (although there is no such restriction in DAML+OIL). For example, a part of the hierarchical structure of classes in an Internet auction system is shown in Figure 2(a).

   In this paper, we use two symbol $\sqsupseteq$ and $\sqsubseteq$ to represent the *containing* relations between two concepts. $X \sqsupseteq Y$ means X contains Y, and $X \sqsubseteq Y$ means X is contained by Y. The $\sqsupseteq$ and $\sqsubseteq$ relations are both reflexive and transitive.

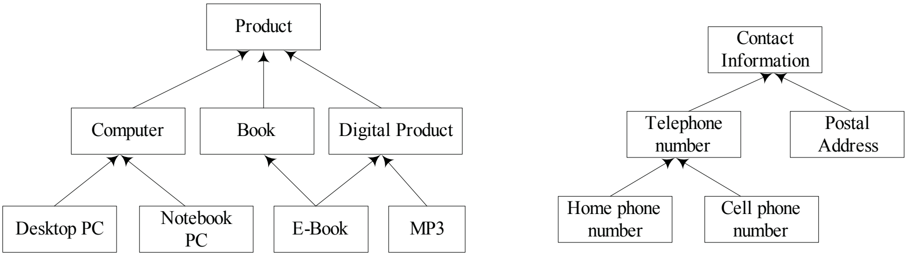
   For two class A and B, predicate "A rdfs:subClassOf B" can be represented as "$A \sqsubseteq B$".

   Suppose A is the class specified in a subscription and B is the class of an entity in an event, A can match with B if A is the ancestor of B. For example, in the Internet auction system, if someone is interested in all computer-selling events with price lower than \$400, he can use the *Computer* class to define his subscription, and all computer-selling events (no matter selling Desktop PCs or Notebook PCs) with price lower than \$400 will be sent to him. However, most existing content-based pub/sub systems can just support the hierarchy of event types, but cannot support the class hierarchy for entities inside the events.

2. The description of properties and their hierarchical structure. A property can have multiple parent properties, and the sub-property-relations between properties must be acyclic. For example, a part of the hierarchical structure of properties in an Internet auction system is shown in Figure 2(b).

   For two properties p_1 and p_2, predicate "p_1 rdfs:subPropertyOf p_2" can be represented as "$p_1 \sqsubseteq p_2$".

   Suppose p_1 is the property specified in a subscription and p_2 is the property appearing in an event, p_1 can match with p_2 if p_1 is the ancestor of p_2. For example, if there is a subscription "*telephoneNumber*=123456789" and an event that contains "*cellPhoneNumber*= 123456789", then the event can match with the subscription.



(a) The hierarchical structure of classes (b) The hierarchical structure of properties

Fig. 2. A part of classes and properties in an online auction system.

3. Meta-statement. We called the triple (*subject-class, property, object-class*) as a *meta-statement*. It specifies the allowed properties for a given class (*subject-class*), and the classes (*object-class*) that the values of these properties belong to.

   For example, an Internet auction system may have the following meta-statements:

   (Selling, seller, Customer)
   (Selling, target, Product)
   (Product, manufacture, Company)
   (Customer, name, xsd:string)
   …

   There is also a hierarchical structure for the meta-statements. For two meta-statement ms_1=(sc_1, p_1, oc_1) and ms_2=(sc_2, p_2, oc_2), we say ms_2 is the *ancestor* of ms_1 (denoted as $ms_1 \sqsubseteq ms_2$) if the following predicate is held:

   $$(sc_1 \sqsubseteq sc_2) \wedge (p_1 \sqsubseteq p_2) \wedge (oc_1 \sqsubseteq oc_2)$$

   It means that if a statement satisfies the type constraints of ms_1, it also satisfies the type constraints of ms_2.

4 Subscription Language

Since events are represented as RDF graphs in the OPS system, the subscription is in fact a graph pattern built on the RDF graph syntax, which specifies the shape of the graph as well as the constraints on some vertexes and arcs in the graph. Based on a number of RDF query languages such as SquishQL [22], RDQL [23] and RQL [24], we design a subscription language for the OPS system.

In the OPS system, a subscription is the conjunction of a number of *statement patterns*; each statement pattern specifies a statement in event graphs. The format of a statement pattern is as follows:

(*subject*, *object*, *meta-statement*, [*filter_func(object)*])

The subject and object in the statement pattern specify the subject and object of a statement in the event graph. They can be variables or specific values, and variables can match with any values. The variable names always begin with "?", such as ?1 and ?2.

The meta-statement in the statement pattern specifies the type constraints of statements. Suppose the meta-statement in a statement pattern is (*sc*, *sp*, *oc*) and there is a statement *S*=(*s*, *p*, *o*), the following predicates must be held if *S* matches the statement pattern:

s rdf:type *sc*
p rdfs:subPropertyOf *sp*
o rdf:type *oc*

When the object in a statement pattern is a variable and the class of the object is literal, the statement pattern can include a filter function *filter_func(object)*, which is a boolean expression used to further confine the value of object. Supported operations in the filter function include the relational operations such as >, <, = and the regular expression operations.

For example, in the Internet auction system, if someone is interested in all computer-selling events with price lower than $400, he can define the following subscription:

(_:H, ?1, (Selling, target, Computer))
(_:H, ?2, (Selling, price, MoneyValue))
(?2, units:dollar, (MoneyValue, currency, daml:Thing))
(?2, ?3, (MoneyValue, rdf:value, xsd:decimal),?3<400.00)

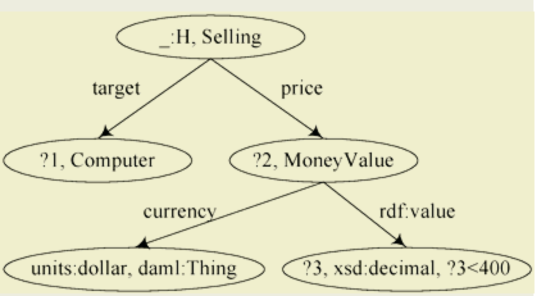


Fig. 3. Anexample of subscrition graph.

Inside the OPS system, each subscription is represented as a graph (called *subscription graph*), in which each vertex corresponds to a vertex in the event graph and each arc corresponds to an arc in the event graph. For example, the preceding subscription can be represented as a graph shown in Figure 3.

Each vertex in the subscription graph has a label (*id*, *class*, [*filter_func(id)*]), in which *id* is the *subject* or *object* in statement patterns, *class* is the class of *id*, and *filter_func(id)* is the filter function on the value of *id* if *id* is a variable and the class of *id* is literal. Each vertex has a unique *id* in the graph.

The label on the arc in the subscription graph is the property name, which forms a meta-statement together with the class of the starting vertex and the class of the end vertex. Each arc plus its starting vertex and ending vertex forms a statement pattern.

In the OPS system, we add the following restrictions on a subscription:

1. There is at least one statement pattern in which the subject is "_:H". We call the vertex with *id*="_:H" in the subscription graph as the *home vertex* of the subscription graph.
2. There are paths from the home vertex to any other vertexes in the subscription graph.

5 Matching Algorithm

The key points of our algorithm are as follows:

1) Each subscription is decomposed into a set of statement patterns, which are the basic units of matching;
2) The index structure of statement patterns is built on the basis of the concept model;
3) Statement patterns with same contents are matched only once;
4) The decomposition of subscription is gradually performed to avoid the creation of unnecessary statement patterns;
5) The event graph and subscription graphs are all traversed in broad-first order to form BFS trees, the matching of two BFS trees resulting in an AND-OR tree.

5.1 Formal Definition of the Matching Problem

Suppose there is an arc a^G in a graph G. We use function $SV(a^G)$ to denote the starting vertex of the arc, function $EV(a^G)$ to denote the ending vertex of the arc, and function $label(a^G)$ to denote the label of the arc.

Suppose the i^{th} vertex of an event graph EG is v_i^{EG}. We denote the *id* of the vertex as id_i^{EG} and the set of classes of the vertex as $classes_i^{EG}$.

Suppose the i^{th} vertex of a subscription graph SG is v_i^{SG}. We denote the *id* of the vertex as id_i^{SG}, the class of the vertex as $class_i^{SG}$, and the filter function of the vertex as $filter\_func_i^{SG}$. If the vertex does not have a filter function, then $filter\_func_i^{SG}$ always returns *true*.

We use function *isVariable(id)* to denote the predicate "*id* is a variable".

Since the *id* of each vertex is unique in subscription graphs and event graphs, we do not strictly distinguish a vertex and its *id* in the following discussion.

Definition 1. An event graph *EG* matches a subscription graph *SG* if and only if the following conditions are held:

1) For each vertex v_i^{SG} in *SG*, there is a corresponding vertex v_j^{EG} in *EG*, and

$$\exists c\text{: } c \in classes_j^{EG} \wedge (c \sqsubseteq class_i^{SG}) \wedge (\, id_i^{SG}{=}id_j^{EG} \vee isVariable(id_i^{SG})\,) \wedge filter\_func_i^{SG}(id_j^{EG})$$

We denote the mapping between v_i^{SG} and v_j^{EG} as $v_i^{SG}{\leftrightarrow}v_j^{EG}$.

2) For two vertexes v_i^{SG}, v_j^{SG} in *SG* and two vertexes v_x^{EG}, v_y^{EG} in *EG*:

$$v_i^{SG}{\leftrightarrow}v_x^{EG} \wedge v_j^{SG}{\leftrightarrow}v_y^{EG} \wedge v_i^{SG}{\neq}v_j^{SG} \Rightarrow v_x^{EG}{\neq}v_y^{EG}$$

3) For each arc a_i^{SG} in *SG*, there is a corresponding arc a_j^{EG} in *EG*, and

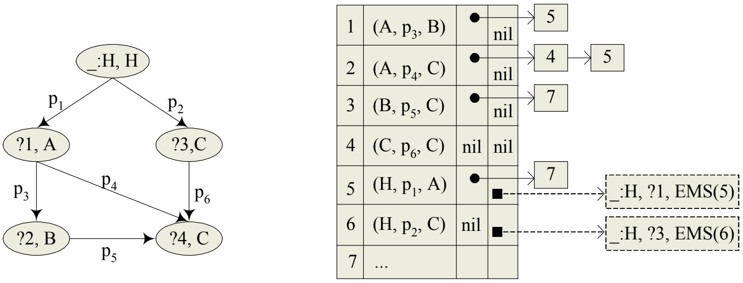
$$SV(a_i^{SG}){\leftrightarrow}SV(a_j^{EG}) \wedge EV(a_i^{SG}){\leftrightarrow}EV(a_j^{EG}) \wedge (label(a_j^{EG}) \sqsubseteq label(a_i^{SG}))$$

We denote the mapping between a_i^{SG} and a_j^{EG} as $a_i^{SG}{\leftrightarrow}a_j^{EG}$.

5.2 Index Structure

Based on the hierarchical structure of classes, the hierarchical structure of properties and the user-defined meta-statements in the concept model, we can figure out all valid meta-statements in the system. We store all these valid meta-statements in an array (called *Extended Meta-Statement array*, abbreviated as *EMS array*), which is the basis of the index structure of the OPS system. The items in the EMS array are sorted in alphabetical order, so that the binary-search algorithm can be used when we look up an item.

Each item in the EMS array contains two lists: the *ancestor list* and the *waiting-pattern list*. The ancestor list records the sequence numbers of all ancestors of the meta-statement. The waiting-pattern list includes the corresponding statement patterns that are waiting for matching. In the initial state, the waiting-pattern lists just include the statement patterns with *subject*="_:H". For example, suppose a system just contains one subscription as shown in Figure 4(a), the initial state of the EMS array can be shown in Figure 4(b). The first list of each item in Figure 4(b) is the ancestor list (drawn in real lines), and the second list of each item is the waiting-pattern list (drawn in broken lines). *Nil* means the null pointer.



(a) A subscription (b) The initial state of the EMS array

Fig. 4. A subscription and the initial state of the EMS array.

For the sake of simplicity, in Figure 4 and the following examples, we use capital letters (such as A, B) to denote classes, p_i (such as p_1, p_2) to denote properties except "rdf:type", and EMS(i) to denote the meta-statement in the i^{th} item of the EMS array.

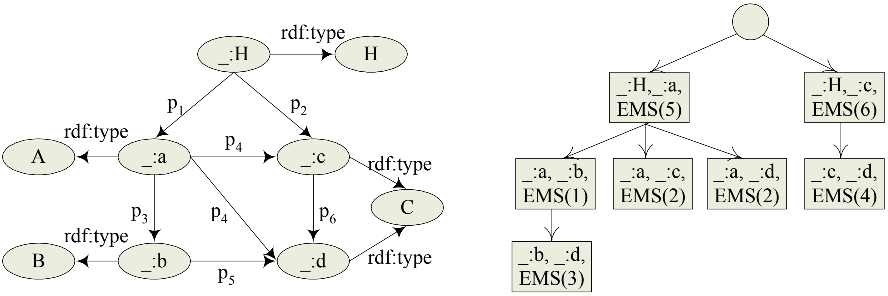
5.3 Traversal of RDF Graphs

When an event arrives, the OPS system will traverse the event graph from the home vertex in a breadth-first order, so that every arc with label≠"rdf:type" is traversed once and only once. For each traversed arc, the system will generate one or several triples with the following format:

(*subject, object, meta-statement*)

We call the triples as *typed-statements*, in which *subject* is the identifier of the starting vertex of the arc, *object* is the identifier of the ending vertex of the arc, and *meta-statement* is the corresponding meta-statement of the statement. The rule for creating meta-statements for a given statement is as follows: suppose the statement is (s, p, o) and the created meta-statement is (ts, tp, to), then ts is the class of s specified in the event graph, tp equals p, and to is the class of o specified in the event graph. One statement can generate multiple typed-statements.

The traversal of an event graph results in a tree structure, in which all nodes except the root node are typed-statements. We call the tree as the *BFS tree* of the event. For example, Figure 5(a) shows an event graph, and Figure 5(b) shows the corresponding BFS tree.



(a) An event graph

(b) The BFS tree of the event

Fig. 5. The RDF graph and the BFS tree of an event.

5.4 Matching Process and the Matching Tree

For each generated typed-statement during the traversal of the event graph, the OPS system will find the corresponding item in the EMS array according to its meta-statement, and matches it with the waiting-pattern list of the item. After that, the system will find all ancestors of the meta-statement (according to the ancestor list of the item), and then matches the typed-statement with the waiting-pattern lists of those ancestors.

For a statement pattern sp=(s_1, o_1, ms_1, $filter\_func_1$) and a typed-statement ts=(s_2, o_2, ms_2), the necessary and sufficient condition for sp matching with ts is as follows:

$$(s_1=s_2 \vee isVariable(s_1)) \wedge (o_1=o_2 \vee isVariable(o_1)) \wedge (ms_1 \sqsupseteq ms_2) \wedge filter\_func_1(o_2)$$

The matching results in the mapping of two pairs of vertexes: $s_1 \leftrightarrow s_2$ and $o_1 \leftrightarrow o_2$.

For example, the typed-statement (_:H, _:a, EMS(5)) in Figure 5(b) can match with the statement pattern (_:H, ?1, EMS(5)) in Figure 4(a) with vertex pairs {"_:H"↔"_:H", ?1↔"_:a"}. To accept these vertex pairs, other statement patterns should also be matched, such as (_:a, ?2, EMS(1)) and (_:a, ?4, EMS(2)). Therefore, the matching process of a subscription and an event is actually the process of trying and evaluating different mapping solutions between the vertexes of the two graphs.

Now we study the matching process of a single subscription and an event. At the beginning, all statement patterns with *subject*="_:H" have already been put into the waiting-pattern lists in the EMS array. For each typed-statement in the event, the system will match it with the statement patterns in the corresponding waiting-pattern lists in the EMS array. For each matched statement pattern, a *partial mapping solution* is created, which records all vertex pairs resulting from the current matching and previous matchings. Suppose a statement pattern sp=(s_1, o_1, ms_1, $filter\_func_1$) has matched with a typed-statement ts=(s_2, o_2, ms_2), the system will act as follows:

1) If o_1 is not in the current path from the home vertex to s_1 in the subscription graph, the system will find out all statement patterns in the subscription graph with *subject*=o_1, and then create new statement patterns based on them, in which variables are replaced with the specific values according to the known vertex pairs. We call these new statement patterns as *derived statement patterns*. All derived statement patterns will be put into the waiting-pattern lists to wait for matching.
2) If o_1 is already in the current path from the home vertex to s_1 in the subscription graph, the system will not create any new derived statement patterns.

The matching state of a subscription can be represented as a tree structure (called *matching tree*), as shown in Figure 6. The matching process of a subscription can then be regarded as the process of creation and verification of the matching tree.

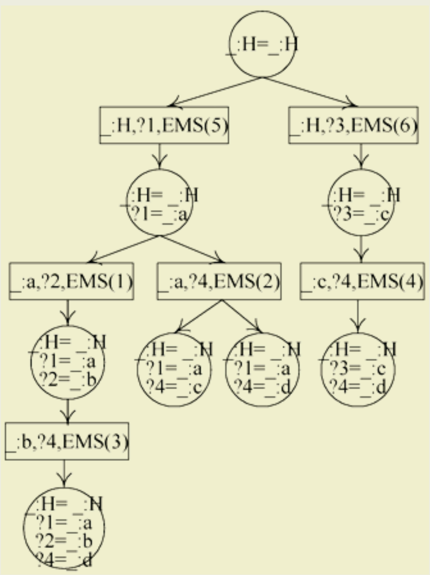


Fig. 6. An example of matching tree.

Figure 6 shows the matching state of the event in Figure 5(a) and the subscription in Figure 4(a). In the figure, the circle node represents a partial mapping solution, and the rectangle node represents a statement pattern. The root node is a circle node with vertex pair "_:H"↔"_:H" (we use symbol "=" to represent "↔" in the figure), and the children of the root node are statement patterns with *subject*="_:H" in the subscription graph. A circle node can have multiple rectangle nodes as its children, meaning the derived statement patterns of the partial mapping solution. Only after all its children being successfully matched, can a circle node be accepted as successful. A rectangle node can also have multiple circle nodes as its children, meaning the multiple matching solutions for the same statement pattern. As long as any one of its children succeeds, the rectangle node also succeeds. Therefore, the circle node implies the “and” relation of its children and the rectangle node implies the “or” relation of its children, so the whole matching tree is actually an AND-OR tree.

During the matching process, the system may generate multiple statement patterns with same contents. If the system put all these statement patterns into the waiting-pattern lists, a typed-statement would match with identical statement patterns for multiple times, which is undesirable. To avoid this phenomenon, a straightforward idea is to let a statement pattern be shared by multiple matching trees. However, since different statement patterns imply different path information in matching trees, it is very difficult for a statement pattern to be shared by multiple matching trees. To solve this problem, we use an approach similar to the *Observer* design pattern [25]. When the system put a statement pattern (suppose it to be *A*) into a waiting-pattern list, it first examine whether there is an existing statement pattern with the same contents in the list. If there exists such a statement pattern (suppose it to be *B*), then *A* will not be put into the list, but be registered to *B*. In the future, whenever *B* successfully matches with a typed-statement, it will notify all the statement patterns that have been registered to it. In this way, statement patterns with same contents will be matched only once.

5.5 Verification of Matching Trees

After the traversal of an event graph, the OPS system has created the matching trees for all subscriptions. Then the system will judge from the matching trees whether a subscription is successfully matched. We call the process as the *verification* of the matching trees. The OPS system uses two methods to verify the matching trees: Boolean Expression Based Verification (BEBV) and State Based Partial Verification (SBPV).

In the BEBV method, each leaf node in a matching tree is given a boolean expression, and the system calculates the expression for the root node to judge the result of matching. The calculation rules are as follows:

1) If a leaf node is circle node with vertex pairs $\{v_1^{SG} \leftrightarrow v_{x1}^{EG}, v_2^{SG} \leftrightarrow v_{x2}^{EG}, \ldots, v_k^{SG} \leftrightarrow v_{xk}^{EG}\}$, its boolean expression is $(v_1^{SG} \leftrightarrow v_{x1}^{EG}) \wedge (v_2^{SG} \leftrightarrow v_{x2}^{EG}) \wedge \ldots \wedge (v_k^{SG} \leftrightarrow v_{xk}^{EG})$.
2) If a leaf node is rectangle node, its expression is *false*.
3) The expression of a non-leaf circle node is the conjunction of the expressions for its children, and the expression of a non-leaf rectangle node is the disjunction of the expressions for its children.
4) For any vertex v_i^{SG} in the subscription graph and two vertex v_x^{EG} and v_y^{EG} in the event graph:

$$(v_i^{SG} \leftrightarrow v_x^{EG}) \wedge (v_i^{SG} \leftrightarrow v_y^{EG}) \wedge (v_x^{EG} \neq v_y^{EG}) = false.$$

It means that one vertex in the subscription graph cannot simultaneously maps to two different vertexes in the event graph.

5) For any vertex v_x^{EG} in the event graph and two vertex v_i^{SG} and v_j^{SG} in the subscription graph:

$$(v_i^{SG} \leftrightarrow v_x^{EG}) \wedge (v_j^{SG} \leftrightarrow v_x^{EG}) \wedge (v_i^{SG} \neq v_j^{SG}) = false.$$

It means that one vertex in the event graph cannot simultaneously maps to two different vertexes in the subscription graph.

According to the above rules we can calculate the boolean expression of the root node for all matching trees. If the expression of the root node in a matching tree is *false*, then the matching fails. Otherwise the matching is successful.

However, it would be very inefficient if the system calculates boolean expressions for every matching tree. To improve the matching efficiency, we design another verification method – SBPV method, which can check out most unmatched subscriptions with very low cost, but cannot tell whether a subscription is successfully matched. Only after a subscription has passed the SBPV check should it perform the time-consuming BEBV check.

In the SBPV method, each node in the matching trees has two possible states: *unchecked* and *checked*. The *checked* state means that the node has passed the SBPV check, and the *unchecked* state means that the node has not passed the check. The initial state of every node is *unchecked*. For a rectangle node, its state turns to *checked* as long as the state of one of its children turns to *checked*. For a circle node, its state turns to *checked* only after the states of all its children turn to *checked*.

In the SBPV method, each circle node has an *unCheckedChildren* field, recording the number of its children whose states are *unchecked*.

During the creation of a matching tree, when a partial mapping solution cannot create new statement patterns as its children, the system begins to backtrack on the matching tree. The backtracking procedure is as follows:

1) Set the state of the current node as *checked*.
2) If the current node is a circle node:
    a) If the current node is the root node, the procedure finishes.
    b) If the current node is not the root node, examine whether the state of its parent is *checked*. If it is already *checked*, the procedure finishes, otherwise set the parent node of the current node as the current node and execute the procedure recursively.
3) If the current node is a rectangle node, subtract 1 from the value of *unCheckedChildren* field of its parent node. If the value turns to 0, set the parent node as the current node and execute the procedure recursively, otherwise the procedure finishes.

The backtracking procedure of the SBPV check is shown in Figure 7. There is a label *id(state, unCheckedChildren)* besides each circle node and a label *id(state)* besides each rectangle node. In the labels, *id* means the identifier of the

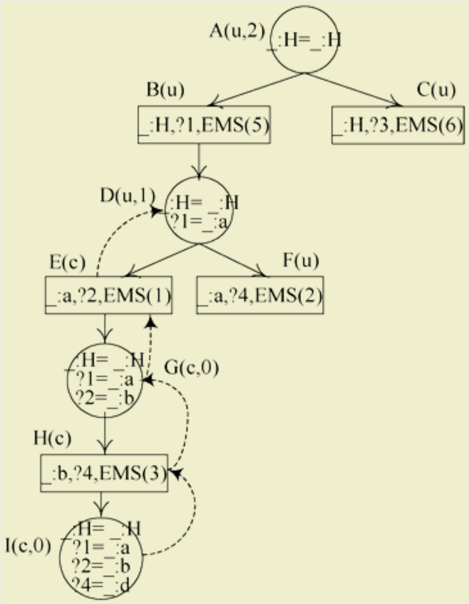


Fig. 7. The backtrack process of the SBPV method.

node, and the *state* field has two possible values: *c* and *u*, meaning *checked* and *unchecked* respectively. The broken lines in Figure 7 show the backtracking process. When node *I* cannot create new statement patterns, The states of node *I*, *H*, *G*, *E* are set to *checked*, and the value of the *unCheckedChildren* field in node *D* turns to 1 from 2.

When the SBPV check finishes, if the state of the root node in a matching tree is *unchecked*, then the subscription cannot match with the event. Since the SBPV check can be performed simultaneously with the creation of the matching tree, the system just needs to do the BEBV check for the matching trees in which the state of the root node is *checked* after the traversal of an event graph. Therefore, the performance of the system can be greatly improved.

5.6 Correctness Proof of the Algorithm

In the following discussion, we call the expression of a node in the matching tree calculated with the BEBV method as the *BEBV expression* of the node, and the state of a node in the matching tree calculated with the SBPV method as the *SBPV state* of the node.

When the BEBV expression of the root node in a matching tree is not *false*, we can cut some unnecessary branches out from the tree to create a *reduced matching tree*. The rules for creating a reduced matching tree are as follows:

1) Transform the BEBV expressions of all nodes into the simplest disjunctive normal form, i.e., the disjunction of non-false conjunction expressions.
2) Cut some branches off from the tree layer by layer from top to bottom:
   a) For the root node, randomly select one of the disjuncts from the disjunction as its new expression, and delete other disjuncts.
   b) For any non-leaf circle node (including the root node): Let the new expression is *P*, and *P* must be a conjunction expression. Suppose the node has *n* children and the expression of the k^{th} child is $p_{x1}^{\ k} \vee p_{x2}^{\ k} \vee \ldots \vee p_{xm}^{\ k}$, in which $p_{xi}^{\ k}$ (i=1..m) is a conjunction expression. Each child must has a disjunct (suppose it to be $p_{xk}^{\ k}$ for the k^{th} child), so that $p_{x1}^{\ 1} \wedge p_{x2}^{\ 2} \wedge \ldots \wedge p_{xn}^{\ n} = P$. Therefore, we can set these disjuncts as the new expressions for the children; i.e., the new expression of the k^{th} child is $p_{xk}^{\ k}$.
   c) For any rectangle node: Since the expression of its parent is not *false*, the expression of itself is not *false* too, so it must be a non-leaf node. Let the new expression is *P*, and *P* must be a conjunction expression. Suppose the node has *n* children and the expression of the k^{th} child is $p_{x1}^{\ k} \vee p_{x2}^{\ k} \vee \ldots \vee p_{xm}^{\ k}$, in which $p_{xi}^{\ k}$ (i=1..m) is a conjunction expression. *P* must be equal to a conjunction expression in one of its children, supposing $P = p_{xk}^{\ k}$. Then we can keep this child and delete all other children and the corresponding sub-trees. The new expression of the remaining child is set to $p_{xk}^{\ k}$.

Obviously, the reduced matching tree has the following features:

1) It is still an AND-OR tree;
2) Each rectangle node has only one child;
3) All leaf nodes are circle nodes;
4) The expression of every node is a non-false conjunction expression.

Lemma 1. For each statement pattern sp^{SG} in the subscription graph SG, there is a rectangle node sp^T in the reduced matching tree T, so that sp^T equals sp^{SG} or sp^T is the derived pattern of sp^{SG}.

Proof. Suppose there is a statement pattern $sp_i^{SG}=(s_i, o_i, ms_i, filter\_func_i)$ in SG. There must be at least one acyclic path $sp_1^{SG}\circ sp_2^{SG}\circ\ldots\circ sp_i^{SG}$ from the home vertex to s_i in SG, in which each item is a statement pattern. The subject of sp_1^{SG} is the home vertex. We can use the mathematical induction to prove sp_i^{SG} has a corresponding rectangle node in T.

1) The statement pattern sp_1^{SG} is put into the matching tree as the child of the root node at the beginning of the matching process. When we cut branches from the matching tree to form a reduced matching tree, we will not delete the children of the root node, so sp_1^{SG} is also in the reduced matching tree T.
2) For $2 \le k \le i$, suppose sp_{k-1}^{SG} has a corresponding rectangle node in T, and let it be sp_{k-1}^{T}. Suppose the object of sp_{k-1}^{SG} is v_k^{SG}. Since v_k^{SG} is not in the path of $sp_1^{SG}\circ sp_2^{SG}\circ\ldots\circ sp_i^{SG}$, the object of sp_{k-1}^{T} is also v_k^{SG}. The node sp_{k-1}^{T} must have a child (let it be pm_k) in T, i.e., it can match with a typed-statement in the event graph and form a partial mapping solution. Since v_k^{SG} is not in the path of $sp_1^{SG}\circ sp_2^{SG}\circ\ldots\circ sp_i^{SG}$, the system should create derived patterns for all statement patterns with $subject=v_k^{SG}$ in SG and put them into the waiting-pattern lists. Since the subject of sp_k^{SG} is v_k^{SG}, it must have a derived pattern (let it be sp_k^{T}) in the matching tree as the child of pm_k. Since pm_k is in T, so are all its children (including sp_k^{T}). Therefore, sp_k^{SG} also has a corresponding rectangle node in T.

Lemma 2. For each vertex v_i^{SG} in the subscription graph SG, there is at least one leaf node in the reduced matching tree T that includes the mapping from v_i^{SG} to a vertex in the event graph.

Proof. For the home vertex of SG, its mapping pair exists in every leaf nodes of T. For any other vertex v_i^{SG} in SG, since there is at least one path from the home vertex to v_i^{SG}, there is at least one statement pattern sp_i^{SG} with $object= v_i^{SG}$. From Lemma 1 we can know there is a corresponding rectangle node (let it be sp_i^{T}) for sp_i^{SG} in T. The node sp_i^{T} must have a child (let it be pm_i), i.e., it can match with a typed-statement in the event graph and form a partial mapping solution, so pm_i include a mapping pair from v_i^{SG} to a vertex in the event graph. The mapping pair must exist in all leaf nodes that are descendants of pm_i.

Lemma 3. If the expression of a node in the reduced matching tree includes a vertex pair, then the expressions of all its ancestor nodes also include the vertex pair.

Proof. Suppose there is a node n_i in the reduced matching tree, the expression of which include a vertex pair $v_i^{SG}\leftrightarrow v_j^{EG}$. There are two possible cases:

1) Node n_i is a circle node. Since n_i is the only child of its parent, the expression of its parent is equal to the expression of n_i, which certainly includes $v_i^{SG}\leftrightarrow v_j^{EG}$.

2) Node n_i is a rectangle node. The expression of its parent is the conjunction of the expressions of n_i and its siblings, and all these expressions are non-false conjunction expressions, so the expression of the parent node must include $v_i^{SG} \leftrightarrow v_j^{EG}$.

Therefore, the expression of the parent node of n_i must include $v_i^{SG} \leftrightarrow v_j^{EG}$. By repeating the process recursively, we can know that all ancestors of n_i include $v_i^{SG} \leftrightarrow v_j^{EG}$.

Lemma 4. The expression of the root node in the reduced matching tree includes the mapping pairs for all vertexes in the subscription graph.

Proof. From Lemma 2 we know that for each vertex in the subscription graph, there is at least one leaf node in the reduced matching tree that includes the mapping pair for it. According to Lemma 3, we can know that the expression of the root node includes the mapping pairs of all vertexes in the subscription graph.

Lemma 5. Suppose the BEBV expression of the root node in a matching tree is not *false*. It cannot become *false* if we add sub-trees under any rectangle node of the tree.

Proof. Suppose we add a sub-tree under a rectangle node n_a in the matching tree, and the root of the sub-tree is n_b. Suppose the expression of n_b is P_b, and the original expression of n_a is P_a. After the addition of n_b, the expression of n_a becomes $P_a \vee P_b$.

Now we will prove that for any node n_k in a matching tree, when its expression changes from P_k to $P_k \vee P_x$, the expression its parent n_p changes from P_p to $P_p \vee P_y$, in which P_x and P_y are boolean expressions. Suppose the node n_p has m children $n_1, \ldots, n_k, \ldots, n_m$ (k ≤ m).

1) Suppose n_k is a rectangle node. Then n_p is a circle node, and the original expression of n_p is:

$$P_p = P_1 \wedge \ldots \wedge P_k \wedge \ldots \wedge P_m$$

The new expression of n_p is:

$$\begin{aligned} P_p' &= P_1 \wedge \ldots \wedge (P_k \vee P_x) \wedge \ldots \wedge P_m \\ &= P_1 \wedge \ldots \wedge P_k \wedge \ldots \wedge P_m \vee P_1 \wedge \ldots \wedge P_x \wedge \ldots \wedge P_m \\ &= P_p \vee P_1 \wedge \ldots \wedge P_x \wedge \ldots \wedge P_m \\ &= P_p \vee P_y \quad (\text{Let } P_y = P_1 \wedge \ldots \wedge P_x \wedge \ldots \wedge P_m) \end{aligned}$$

2) Suppose n_k is a circle node. Then n_p is a rectangle node, and the original expression of n_p is:

$$P_p = P_1 \vee \ldots \vee P_k \vee \ldots \vee P_m$$

The new expression of n_p is:

$$\begin{aligned} P_p' &= P_1 \vee \ldots \vee (P_k \vee P_x) \vee \ldots \vee P_m \\ &= P_1 \vee \ldots \vee P_k \vee \ldots \vee P_m \vee P_x \\ &= P_p \vee P_x \end{aligned}$$

We can calculate the expressions of the ancestors of n_a from bottom to top until the root node. Suppose the original expression of the root node is P_r, then the new expression of the root node is $P_r'=P_r \vee P_y$. Since P_r is not *false*, P_r' is not *false* too.

Theorem 1. For any subscription S and event e in the OPS system, suppose the resulting matching tree is MT, and the BEBV expression of the root node in MT is P_{root}, then:

$$(P_{root} \neq false) \Leftrightarrow (e \text{ matches } S)$$

Proof. We first prove $(P_{root} \neq false) \Rightarrow (e \text{ matches } S)$.

1) Since the expression of the root node of the matching tree is not *false*, we can create a reduced matching tree based on it. From Lemma 4 we know that the expression of the root node of the reduced matching tree includes the mapping pairs for all vertexes in the subscription graph, so we can get the vertex mappings from the subscription graph to the event graph.
2) Suppose there is a statement pattern $sp_i^{SG}=(s_i, o_i, ms_i, filter\_func_i)$ for arc a_i^{SG} in the subscription graph. According to Lemma 1, it must have a corresponding rectangle node (let it be sp_i^T) in the reduced matching tree, and sp_i^T has a child (let it be pm_i). Suppose pm_i is the result of the matching between sp_i^T and a typed-statement $ts_j=(s_j, o_j, ms_j)$ in the event graph, then pm_i must includes the vertex pairs $\{s_i \leftrightarrow s_j, o_i \leftrightarrow o_j\}$. According to Lemma 3, the vertex pairs are consistent with the mapping solution in the root node of the reduced matching tree. Suppose the arc for ts_j in the event graph is a_j^{EG}, then we can get an arc mapping from a_i^{SG} to a_j^{EG}. In this way, we can get all arc mappings from the subscription graph to the event graph.

According to Definition 1, we can conclude that the event matches with the subscription.

Now we prove $(e \text{ matches } S) \Rightarrow (P_{root} \neq false)$.

Suppose there is a known mapping from the vertexes and arcs of the subscription graph to those of the event graph. Since the event graph is traversed in a breadth-first order, after the traversing of the first-layer type-statement in the BFS tree of the event, there is at least one circle node under every first-layer rectangle node in the matching tree, in which the vertex pairs are consistent with the known vertex mapping. We keep these circle nodes and delete all other circle nodes (and their corresponding sub-trees) in this layer.

The remaining circle nodes may have created new rectangle nodes (the second-layer rectangle nodes) in the matching tree. Since there is a know mapping from the arcs of the subscription graph to those of the event graph, after the traversing of the second-layer typed-statement in the BFS tree of the event, every second-layer rectangle node in the matching tree can match with at least one typed-statement and form a circle node, in which the vertex pairs are consistent with the known vertex mapping. We keep these circle nodes and delete all other circle nodes (and their corresponding sub-trees) in this layer.

According to the above rules we can cut some branches out from the matching tree from top to bottom, and the resulting tree is a reduced matching tree. The expression of the root node of the reduced matching tree is the known vertex mapping solution, which is certainly not *false*. Then we can add the deleted sub-trees into the reduced matching tree and return it to the original matching tree. According to Lemma 5, the expression of the root node of the resulting matching tree is also not *false*.

Lemma 6. The SBPV state of every node in the reduced matching tree is *checked*.

Proof. We can use the mathematical induction to prove it. Suppose the depth of the reduced matching tree is n.

1) The nodes in layer n of the tree are all leaf nodes. As all leaf nodes in the reduced matching tree are circle nodes, the state of the nodes are *checked*.
2) For $1 \leq i < n$, suppose the states of nodes in layer i+1 are all *checked*. For the leaf nodes in layer i, since they are circle nodes, their state are *checked*. For the non-leaf nodes in layer i, since the states of all of their children are *checked*, their states are also *checked*. Therefore, the state of every node in layer i is *checked*.

Lemma 7. If we add sub-trees under any rectangle node of the reduced matching tree, the SBPV state of the root node will not become *unchecked*.

Proof. Suppose we add a sub-tree under a rectangle node n_i in the reduced matching tree, and the root of the sub-tree is n_j. Since n_i already has a child whose state is *checked*, the state of n_i remains *checked* no matter what the state of n_j is, so the state of all ancestor nodes (including the root node) of n_i also remain *checked*.

Theorem 2. For any subscription S and event e in the OPS system, suppose the resulting matching tree is MT, and the SBPV state of the root node in MT is $State_{root}$, then:

$$(State_{root} = unchecked) \Rightarrow (\ e \text{ doesn't match } S)$$

Proof. We can prove its inverse and negative proposition, i.e., (e matches S) $\Rightarrow$ ($State_{root} = checked$).

If an event can match with a subscription, we can know from Theorem 1 that the BEBV expression of the root node in the matching tree is not *false*, so we can build a reduced matching tree based on the matching tree. According to Lemma 6, the state of the root node in the reduced matching tree is *checked*. Then we can add the deleted sub-trees into the reduced matching tree and return it to the original matching tree. According to Lemma 7, the state of the root node in the resulting matching tree is also *checked*.

6 Experimental Evaluation

In this section we evaluate the performance of the OPS system with a variety of simulated workloads. The prototype system was implemented in Java, and the performance tests discussed below were performed on a common Notebook PC with an Intel Pentium IV CPU at 1.6GHz and 512MB RAM running Windows 2000 Sever and JDK 1.4.1.

To demonstrate the efficiency of our algorithm, we also implemented a recently proposed graph-matching algorithm in [19], and compared the performance of the two algorithms under the same environment and workloads. For the sake of simplicity, we call our algorithm as the *OPS* algorithm and their algorithm as the *Decomposition* algorithm in the follows.

Suppose the concept model contains *C* numbers of classes and *P* numbers of properties. There are no sub-class relations among classes and no sub-property relations among properties. In the following experiments, the value of *P* is fixed to be 10, and each class has exactly 2 properties.

Suppose there are *S* numbers of subscriptions in the system. Every subscription has 10 vertexes and 11 arcs. The *id* of the home vertex was "_:H", and the *id* of all other vertexes were variables. The *class* of every vertex was randomly selected from the total classes, and there were no filter functions in any vertexes.

Every generated event had 50 blank nodes, and there were 55 arcs among the blank nodes.

We define a parameter *matching rate*, meaning the ratio of matched subscriptions to total subscriptions for a given event. The value of matching rate considerably affects the performance of graph matching algorithms.

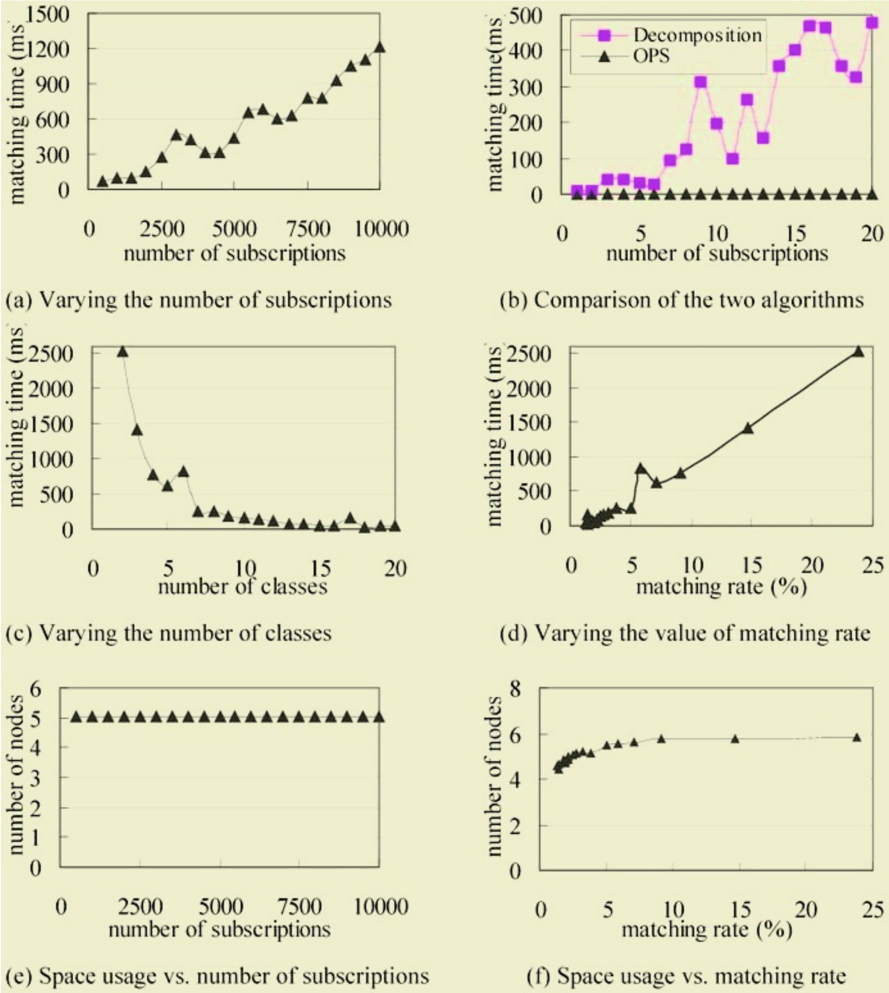
Figure 8(a) shows the matching times of the OPS algorithm under different numbers of subscriptions. In the experiments, the value of *C* is 10, and the value of *S* varies from 500 to 10,000. The resulting matching rates are always around 3%. From the figure we can see that the matching time is just 1.2 seconds when the number of subscriptions is 10,000.

Figure 8(b) shows the comparison of the matching times of the two algorithms. In the experiments, the value of *C* is 10, and the value of *S* varies from 1 to 20. From the figure we can see the OPS algorithm is much faster in event matching than the Decomposition algorithm. When the there are just 20 subscriptions, the matching time of the Decomposition algorithm reaches 500ms, while the matching time of the OPS algorithm is merely 1ms. Therefore, the conventional graph matching algorithms are not suitable for the pub/sub systems where there are large numbers of subscriptions.

Figure 8(c) shows the matching times of the OPS algorithm under different numbers of classes. In the experiments, the value of *S* is 1,000, and the value of *C* varies from 2 to 20. From the figure we can see that the matching time decreases dramatically from 2500ms to about 50ms. We believe the real reason is the changing of the matching rate, i.e., the matching rate decreases greatly when the number of classes increases, which leads to the decrease of matching time. Figure 8(d) shows the same experimental results, in which the x-axis represents the matching rate rather than the number of classes. From the figure we can see that the matching time is almost linear in the value of matching rate.

Now we evaluate the space usage of the OPS algorithm. In the OPS algorithm, the space is mainly used in the creation of matching trees for subscriptions. Figure 8(e) shows the average number of nodes in each matching tree under different numbers of subscriptions. In the experiments, the value of *C* is 10, and the number of subscriptions varies from 500 to 10,000. From the figure we can see that the average number of nodes in each matching tree is always around 5. Therefore, the space usage in the OPS algorithm is linear in the number of subscriptions.

Figure 8(f) shows the average number of nodes in each matching tree under different values of matching rates. In the experiments, the value of *S* is 1,000, and the value of *C* decreases from 20 to 2, so the matching rate increases from 1.3% to 23.8%. From the figure we can see that the average number of nodes in each matching tree increases just a little while the matching rate increases a great deal. Therefore, the space usage in the OPS algorithm is sub-linear in the value of matching rate.



(a) Varying the number of subscriptions

(b) Comparison of the two algorithms

(c) Varying the number of classes

(d) Varying the value of matching rate

(e) Space usage vs. number of subscriptions

(f) Space usage vs. matching rate

Fig. 8. The experimental results.

7 Conclusions

In this paper, we described the data model, subscription language and matching algorithm of an ontology-based publish/subscribe system. Through the combination of the publish/subscriber technologies and the Semantic Web technologies, the system can make use of the semantic of events to match events with subscriptions, and can support events with complex data structure (such as graph structure). Furthermore, we design a highly efficient matching algorithm for the OPS system, which can match RDF graphs with graph patterns in a speed much higher than the conventional graph matching algorithms. Therefore, the main contribution of our work is that it greatly improves the expressiveness of the pub/sub system and at the same time it keeps a high matching efficiency.

References

1. P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec: The many faces of publish/subscribe. ACM Computing Surveys 35(2) (2003) 114-131
2. Antonio Carzaniga, David S. Rosenblum, and Alexander L: Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. In 19th ACM Symposium on Principles of Distributed Computing. (2000)
3. O. Lassila and R. R. Swick: Resource Description Framework (RDF) Model and Syntax Specification. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/ (1999)
4. T. Berners-Lee: Using XML for Data. http://www.w3.org/DesignIssues/XML-Semantics.html. (2001)
5. IBM: Internet Application Development with MQSeries and Java. Vervante Corporate Publishing (1997)
6. Carzaniga, D. S. Rosenblum, and A. L. Wolf: Design and evaluation of a wide-area event notication service. ACM Trans. on Computer Systems 19(3) (2001) 332-383
7. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra: Matching events in a content-based subscription system. In: Proceedings of the Eighteenth ACM Symposium on Principles of Distributed Computing (1999) 53-61
8. G. Cugola, E. D. Nitto, and A. Fuggetta: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Trans. on Software Engineering 27(9) (2001) 827-850
9. M. Altinel and M. J. Franklin: Efficient Filtering of XML Documents for Selective Dissemination of Information. In: Proceedings of 26th International Conference on Very Large Data Bases. (2000) 53-64
10. C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi: Efficient Filtering of XML Documents with XPath Expressions. The VLDB Journal 11(4) (2002) 354-379
11. J. Pereira, F. Fabret, F. Llirbat, H.-A. Jacobsen, and D. Shasha: WebFilter: A High Throughput XML-based Publish and Subscribe System. In: Proceedings of 27th International Conference on Very Large Data Bases. (2001) 723-724
12. M. Petrovic, I. Burcea, and H.-A. Jacobsen: S-ToPSS: Semantic Toronto Publish/Subscribe System. In: Proceedings of 29th International Conference on Very Large Data Bases. (2003) 1101-1104
13. M. Cilia, C. Bornhoevd, and A. P. Buchmann: CREAM: An Infrastructure for Distributed, Heterogeneous Event-based Applications. In: Proceedings of the International Conference on Cooperative Information Systems. (2003) 482-502
14. J. R. Ullmann: An Algorithm for Subgraph Isomorphism. Journal of the ACM 23(1) (1976) 31-42
15. R.M. Haralick and G.L. Elliot: Increasing Tree Search Efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14 (1980) 263-313
16. R.E. Blake: Partitioning Graph Matching with Constraints. Pattern Recognition 27(3) (1994) 439-446
17. H. Sossa and R. Horaud: Model Indexing: The Graph-Hashing Approach. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition. (1992) 811-814
18. K. Sengupta and K.L. Boyer: Organizing Large Structural Modelbases. IEEE Trans. on Pattern Analysis and Machine Intelligence 17(4) (1995)
19. B. T. Messmer and H. Bunke: Efficient Subgraph Isomorphism Detection: A Decomposition Approach. IEEE Trans. on Knowledge and Data Engineering 12(2) (2000) 307-323
20. F. V. Harmelen, P. F. Patel-Schneider and I. Horrocks: Reference description of the DAML+OIL (March 2001) ontology markup language. http://www.daml.org/2001/03/reference. (2001)
21. T. R. Gruber: A translation approach to portable ontologies. Knowledge Acquisition 5(2) (1993) 199-220

22. L. Miller, A. Seaborne, and A. Reggiori: Three Implementations of SquishQL, a simple RDF Query Language. In: Proceedings of the First International Semantic Web Conference. (2002) 423-435
23. HP Labs: RDQL: RDF Data Query Language. http://www.hpl.hp.com/semweb/rdql.htm
24. G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl: RQL: A Declarative Query Language for RDF, In: Proceedings of the Eleventh International World Wide Web Conference. (2002) 592-603
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design pattern, elements of reusable object-oriented software. Addison-Wesley (1994)