Composite Subscriptions in Content-Based Publish/Subscribe Systems

Guoli Li and Hans-Arno Jacobsen

Middleware Systems Research Group, University of Toronto,
Toronto, ON, Canada

Abstract. Distributed publish/subscribe systems are naturally suited for processing events in distributed systems. However, support for expressing patterns about distributed events and algorithms for detecting correlations among these events are still largely unexplored. Inspired from the requirements of decentralized, event-driven workflow processing, we design a subscription language for expressing correlations among distributed events. We illustrate the potential of our approach with a workflow management case study. The language is validated and implemented in PADRES. In this paper we present an overview of PADRES, highlighting some of its novel features, including the composite subscription language, the coordination patterns, the composite event detection algorithms, the rule-based router design, and a detailed case study illustrating the decentralized processing of workflows. Our experimental evaluation shows that rule-based brokers are a viable and powerful alternative to existing, special-purpose, content-based routing algorithms. The experiments also show that the use of composite subscriptions in PADRES significantly reduces the load on the network. Complex workflows can be processed in a decentralized fashion with a gain of 40% in message dissemination cost. All processing is realized entirely in the publish/subscribe paradigm.

1 Introduction

In distributed applications large numbers of events occur. In isolation these events are often not too interesting or useful. However, as correlations over time, for example, these events may represent interesting and useful information. This information is important for coordinating activities in a distributed system. Workflow processing and business process execution, where different stages of the flow or process execute on distributed nodes, are examples of distributed applications generating potentially huge numbers of events. The efficient correlation of these events reveals information about the status of the workflow. Events in a workflow could be the initiation, the termination, or the status of a task.

Distributed publish/subscribe systems are well-suited to handle large numbers of events. A publish/subscribe system is comprised of information producers who publish and information consumers who subscribe to information. The key benefit of publish/subscribe for distributed event-based processing is the natural decoupling of publishing and subscribing clients. This decoupling can enable the design of large, distributed, loosely coupled systems that interoperate through simple publish and subscribe-style operations.

However, current publish/subscribe approaches lack the ability to address event correlation and enable the coordination of activities associated with disparate clients in the content-based network. In order to allow publish/subscribe to support such distributed applications, first, an appropriate subscription language needs to be designed which offers a suitable view over available events to enable coordination. Second, event correlation requires the detection of distributed events. In publish/subscribe this is based on routing subscriptions and publications throughout the broker network and on efficient composite event detection algorithms realized on a single publish/subscribe broker.

Some work on detecting composite events in distributed publish/subscribe systems is starting to appear [21,22,5]. However, these approaches are mainly focusing on the design of the subscription language and do not address the event correlation problem central to our approach. We have developed an expressive content-based subscription language that is derived from the requirements of event-driven, decentralized workflow management and business process execution scenarios. To validate our approach we have implemented the language in PADRES (Publish/subscribe Applied to Distributed REsource Scheduling), a novel distributed, content-based publish/subscribe messaging system, and have built all the necessary infrastructure to support the deployment, monitoring, and execution of workflows and business processes. In essence, we have realized a decentralized workflow management and execution environment that builds directly on top of a standard publish/subscribe interface.

PADRES's subscription language is fully content-based, includes notions to express time, supports variable bindings, coordination patterns, and composite subscriptions. *Composite subscriptions* offer a higher level view for subscribers by enriching the expressiveness of the subscription language. A composite subscription consists of several *atomic subscriptions* linked by logical or temporal operators. An atomic subscription refers to the traditional notion of a subscription in publish/subscribe and is matched by a single publication event; a composite subscription is matched by a set of independent events potentially occurring at different locations and times. PADRES is based on a rule-based broker that implements composite event detection and introduces a novel distributed algorithm for composite subscription routing.

Support for composite subscriptions is essential for applications where it is impossible to detect a particular condition from isolated atomic events. For example, in workflow management systems, tasks can only be executed if certain conditions are met. A given task may require that two other tasks have successfully completed and a certain timing constraint is met. We will show experimentally that supporting composite subscriptions in content-based publish/subscribe systems has two key advantages. First, subscribers receive fewer messages and network traffic is reduced. Without composite subscriptions, the subscriber must subscribe to all the corresponding atomic events in order to receive the necessary information. The subscriber would be overwhelmed by an excessive amount of atomic events, most of which may be irrelevant and could be filtered out before reaching the subscriber. Second, the overall performance of the publish/subscribe system is improved by detecting composite events in the network, rather than at the edge of the network. Moreover, composite subscriptions reduce the complexity of subscriber components.

The rest of this paper is organized as follows. Section 2 presents background material and related work. An overview of PADRES is given in Section 3. Section 4 presents the PADRES subscription language, composite subscription routing and composite event detection in detail. A workflow management system case study built on PADRES is discussed Section 5. An experimental evaluation of PADRES and its potential for workflow management is presented in Section 6.

2 Background and Related Work

Content-based Routing. Content-based publish/subscribe systems typically utilize *content-based routing* in lieu of the standard address-based routing. Since publishers and subscribers are decoupled, a publication is routed towards the interested subscribers without knowing specifically where subscribers are and how many subscribers exist. The *content-based address* of a subscriber is the set of subscriptions issued by the subscriber. There are several interesting projects dealing with content-based routing, such as SIENA [3], REBECA [18], JEDI [6], Hermes [20] and Gryphon [19]. Covering and merging-based routing, which are optimizations for content-based routing, are discussed in SIENA [3], JEDI [6], REBECA [18], and PADRES [15]. In addition to publications and subscriptions, content-based routing can use *advertisements* [18,3], which are indications of the data that publishers will publish in the future. Advertisements are used to form routing paths along which subscriptions are propagated. Without advertisements, subscriptions must be flooded throughout the network. PADRES adopts the publication-subscription-advertisement model for content-based routing and suggests several novel features not realized in existing approaches. The novel features of PADRES discussed in this paper include a rule-based router design, algorithms to support composite subscription routing, composite event detection, coordination patterns for expressing workflows and business processes, and support for the decentralized deployment and execution of workflows and business processes.

Composite Events. An *event* is defined as a state transition. In the publish/subscribe literature, events describe state transitions of interest to subscribers. Events are often synonymously referred to as *publications*[1]. A *subscription* captures the interest of a subscriber to be informed about possible events. We generically refer to subscriptions, publications, and advertisement as *messages*, if no distinction is required.

A *composite event* refers to a pattern of event occurrences of interest to a subscriber. These patterns may express temporal or causal relationships between

[1] One could further distinguish between the state transition (i.e., event) and the published information that reports on the transition (i.e., the *publication*).

different events. A pattern is matched, if the specified events have occurred, subject to optional timing constraints. Since several events are involved in the matching of a single subscription pattern the matching engine has to store partial matching states. In the literature, the term *composite event* has been used to refer to a subscription that expresses the pattern defining a composite event. To make the difference between the state transitions (i.e., the events) and the actual interest specification clearer, when discussing our work, we use the term *composite subscription* to refer to the pattern and use *composite event* to mean the distributed state transitions of relevance for the subscriber of the composite subscription. Also to distinguish composite subscriptions from traditional, non-composite subscriptions, we refer to the latter as *atomic subscriptions*.

The earliest approaches for enabling the processing of composite events were rule-based production systems established in artificial intelligence. One of the most widely used matching algorithms, the Rete algorithm is used in many expert systems today [9]. Rete compiles rules into a network. The design of Rete trades off space for processing efficiency. The Java Expert System Shell (*Jess*) [10] is a rule-based matching engine based on the Rete algorithm. Our PADRES broker is based on *Jess*. The Publication Routing Table (PRT) and Subscription Routing Table (SRT) are two *Jess* engines. We show how content-based publish/subscribe messages (i.e., subscriptions, composite subscriptions, publications, and advertisements) can be mapped to rules and facts processed by Rete-type rule engines.

Many early approaches for composite event processing relate to active databases and are based on centralized evaluation schemes [12,11,16,13,17,4]. These projects differ primarily in the mechanism used for event detection. Ode [12] uses a finite automaton and SAMOS [11] uses a Petri Net. Other approaches use trees as the data structure for representing and detecting composite events. The main reason for adopting trees is that they are simple and intuitive for representing composition. The traversal and manipulation of trees have been thoroughly studied in the past, and a large number of efficient algorithms have been developed [16,13,1,17]. GEM [16] and READY [13] are projects using tree-based approaches to process incoming events. Atomic events are leaf nodes and operators are inner nodes in the tree structure. The composite event is represented by the root of the tree. The main limitation of GEM is each composite event has its own tree, and identical subtrees cannot be shared among composite event trees. Similar to GEM and READY, EPS (Event Processing Service) [17] provides a tree-based event specification language. EPS alleviates the limitation of GEM by using a shared subscription tree to process incoming events. Snoop [4], also a tree-based approach, provides an expressive composite event specification language with temporal support. Snoop introduces the notion of consumption policies called *contexts*. They are used to capture application semantics by resolving which events are consumed from the event history for composite event detection in case of ambiguity. Composite subscriptions in PADRES are also represented by trees. Unique to PADRES is the mapping of atomic and composite subscriptions to rules and the support of full content-based, composite subscriptions. The rule-based processing has been thoroughly studied, leading to a large number of efficient algorithms for rule/fact matching. The rule-based approach employed in PADRES takes advantage of the existing research for the PADRES broker design. PADRES also supports a tree decomposition algorithm for composite subscription routing.

The specification and detection of composite events in the context of publish/subscribe systems has recently become an important research area [21,22,5]. Hermes [20] and Gryphon [19] provide parameterized atomic events to enrich the expressiveness of subscriptions. Courtenage [5] specifies composite events based on the λ-calculus. The approach lacks support for temporal constraints. CEA [21] proposes a Core Composite Event Language to express event patterns that occur concurrently. CEA constitutes a composite event detection framework built as an extension of an existing publish/subscribe middleware platform. The CEA language is compiled into automata for distributed event detection supporting regular expression-type patterns. CEA employs policies to ensure that mobile event detectors perform distributed event detection at favorable locations, such as close to event sources. REBECA [22] describes composite events using composite event filter expressions, which can be mapped to expressions of the Core Composite Event Language [21]. The subscription language design of PADRES has been inspired from requirements set forth by workflow and business process description languages and the requirements of distributed execution of these processes. Unique to PADRES is the use of variables in subscriptions to join atomic events. PADRES also supports language elements to express dependencies and condition-based repetition relationships of activities (i.e., while loops). Architecturally different from existing approaches, PADRES builds the composite subscription processing and composite event detection capability into the publish/subscribe layer.

3 PADRES System Description

The PADRES system consists of a set of brokers connected by a peer-to-peer overlay network. Clients connect to brokers using various binding interfaces such as Java Remote Method Invocation (RMI) and Java Messaging Service (JMS). Each PADRES broker employs a rule-based engine to route and match publish/subscribe messages, and is used for composite event detection. An overview of PADRES is provided in [8]. This paper focuses on the specification, detection, and use of composite events. PADRES provides four other novel features as well: monitoring support, historic query capability, fault detection and repair, and load balancing. A monitor module, which is an administrative client in PADRES, could display the broker network topology, trace messages, and measure the performance of the broker network. The historic data access module allows clients to subscribe to both future and historic publications. The fault tolerance module detects failures in the publish/subscribe layer and initiates failure recovery. The load balancing module handles the scenarios in which a broker is overloaded by a large number of publishers or subscribers. The detail

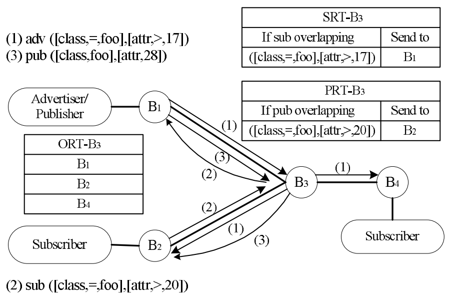

(1) adv ([class,=,foo],[attr,>,17])
(3) pub ([class,foo],[attr,28])
SRT-B3
If sub overlapping
Send to
([class,=,foo],[attr,>,17])
B1
PRT-B3
If pub overlapping
Send to
([class,=,foo],[attr,>,20])
B2
Advertiser/ Publisher
B1
ORT-B3
B1
B2
B4
B3
B4
Subscriber
B2
Subscriber
(1)
(2)
(3)
(2) sub ([class,=,foo],[attr,>,20])

Fig. 1. Broker Network

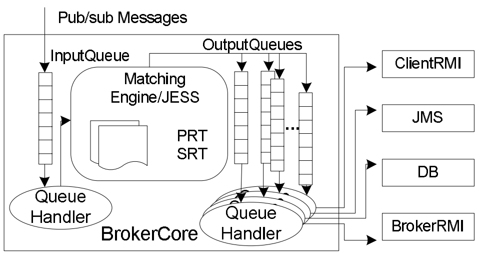


Fig. 2. Broker Architecture

of these features goes beyond the scope of this paper. Fig. 10 shows the protocol stack of PADRES. This section discusses the architecture of PADRES for processing of atomic subscriptions. The extension of PADRES to process composite subscription and the case study applying composite subscription processing to workflow management are discussed later.

3.1 Message Format

The PADRES subscription language is based on the traditional `[attribute, operator, value]` predicates used in several existing content-based publish/subscribe systems [3,18,19,7]. An atomic subscription is a conjunction of predicates. For example, an atomic subscription in workflow management may be (`[class, =, job-status]`, `[appl, =, payroll]`, `[job-name, isPresent, *]`). The comma between predicates indicates the conjunction relation. This subscription is matched by publications of all jobs involved in application *payroll*. We support operators, such as $=$, $>$, $<$, $\geq$, $\leq$, and *isPresent*. The special operator *isPresent* means an attribute could be any value in a given range. Each subscription message has a *mandatory* tuple describing the *class* of the message. The *class* attribute provides a guaranteed selective predicate for matching, similar to the *topic* in topic-based publish/subscribe systems[2]. Other predicates are constraints on particular attributes. Advertisements have the same format as atomic subscriptions. Publications are sets of `[attribute, value]` pairs. There is a match between a subscription and a publication if each predicate in the subscription is satisfied by a corresponding `[attribute, value]` pair in the publication. A match between a subscription and a advertisement means the sets of publications matching the advertisement and the subscription are overlap.

3.2 Network Architecture

The overlay network connecting the brokers is a set of connections that form the basis for message routing. The overlay routing data is stored in Overlay

[2] The PADRES language is fully content-based based on a rich predicate language.

Routing Tables (ORT) at each broker. Specifically, each broker knows its neighbors from the ORT. Message routing in PADRES is based on the publication-subscription-advertisement model established by the SIENA project [3]. We assume that publications are the most common messages, and advertisements are the least common ones. A publisher issues an advertisement before it publishes. An advertisement allows the publisher to publish a set of publications matching this advertisement. Advertisements are effectively flooded to all brokers along the overlay network. A subscriber may subscribe at any time. The subscriptions are routed according to the Subscription Routing Table (SRT), which is built based on the knowledge of advertisements. The SRT is essentially a list of `[advertisement,last hop]` tuples. If a subscription overlaps an advertisement in the SRT, it will be forwarded to the last hop broker the advertisement came from. Subscriptions are routed hop by hop to the publisher, who advertises information of interest to the subscriber. Meanwhile, the subscription will be used to construct the Publication Routing Table (PRT). Like the SRT, the PRT is logically a list of `[subscription,last hop]` tuples, which is used to route publications. If a publication matches a subscription in the PRT, it will be forwarded to the last hop broker of that subscription until it reaches the subscriber. A diagram showing the overlay network, SRT and PRT is provided in Fig. 1. In this figure, step *1)* an advertisement is propagated from B_1. Step *2)* a matching subscription enters from B_2. Since the subscription overlaps the advertisement at broker B_3, it is sent to B_1. Step *3)* a publication is routed along the path established by the subscription to B_2. A subscription/advertisement covering and merging scheme [15] is used to optimize content-based routing by reducing network traffic and routing table size, especially for applications with highly clustered data.

3.3 Broker Architecture

The PADRES brokers are modular software components built on a set of queues: one input queue and multiple output queues. Each output queue represents a unique message destination. A diagram of the broker architecture is provided in Fig. 2. The matching engine between the input queue and output queues is built using *Jess*. It maintains the SRT and PRT, which are Rete trees [9]. For example, in the PRT, subscriptions are mapped to rules, and publications are mapped to facts, as shown in Fig. 3. An atomic subscription message is mapped to the antecedent of a rule; the actions to be taken if the subscription is matched are mapped to the consequent of the rule. The antecedent encodes the message filter condition and the consequent encodes the notification semantic.

The matching between subscriptions and publications is transformed to the matching between rules and facts, which is performed by the rule-based broker. When a new message is received by the broker, it is placed in the input queue. The matching engine takes the message from the input queue. If the message is a publication, it is inserted into the PRT as a fact. When a publication matches a subscription in the PRT, its next hop destination is set to the last hop of the subscription, and it is placed into the corresponding output queue(s). If the message

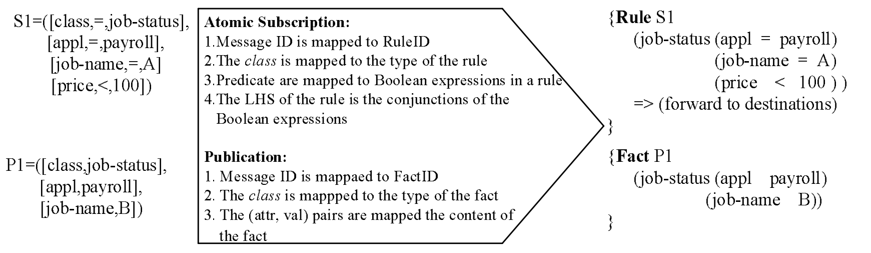

S1=([class,=,job-status],
[appl,=,payroll],
[job-name,=,A]
[price,<,100])
P1=([class,job-status],
[appl,payroll],
[job-name,B])
Atomic Subscription:
1.Message ID is mapped to RuleID
2.The class is mapped to the type of the rule
3.Predicate are mapped to Boolean expressions in a rule
4.The LHS of the rule is the conjunctions of the Boolean expressions
Publication:
1. Message ID is mappaed to FactID
2. The class is mappped to the type of the fact
3. The (attr, val) pairs are mapped the content of the fact
{Rule S1
(job-status (appl = payroll)
(job-name = A)
(price < 100))
=> (forward to destinations)
}
{Fact P1
(job-status (appl payroll)
(job-name B))
}

Fig. 3. Mapping Subscriptions/Publications to Rules/Facts

is a subscription, the matching engine first routes it according to the SRT, and, if there is an advertisement overlapping the subscription, the subscription will be inserted into the PRT as a rule. Essentially, the rule-based broker performs matching and decides the next hop destinations of the messages as a router. This novel rule-based approach allows for powerful subscription language and notification semantics and naturally enables composites subscriptions.

4 Composite Subscription Processing

4.1 Composite Subscription Language

The composite subscription language is inspired by the requirements of workflow management and business process execution. The language should be powerful enough to eventually describe workflows defined using the Business Process Execution Langauge (BPEL4WS) [14], which is a standard language for business processes. PADRES supports *parallelization*, *alternation*, *sequence* and *repetition* compositions. PADRES also supports *variable bindings* that serve to correlate and aggregate publications by specifying constraints on attribute values between different atomic subscriptions. A composite subscription is represented by a subscription tree, where the internal nodes are operators and leaf nodes are atomic subscriptions, as shown in Figure 4 (b).

The operator to represent the *parallelization* pattern is `AND`, denoted by the symbol `(&)`. The composite subscription (s_1 `&` s_2) is matched when both s_1 and s_2 are matched, irrespective of their matching order. The operator `&` is to connect two or more subscriptions, and it is different from the conjunction operator between predicates in an atomic subscription that requires to be matched by one publication. The *alternation* pattern represents the matching of any of two specified subscriptions using operator `OR`, denoted as `(`$\|$`)`. The composite subscription ($s_1 \| s_2$) is satisfied when either s_1 or s_2 is matched by a publication. Furthermore, composite subscriptions in PADRES can have variables bound to values in the publications. Variables are represented by \$ in subscription predicates. Parenthesis are used to specify the priority of operators. In the example below, the composite subscription consists of three atomic subscriptions, linked using `&` and ||, and requires the values of the attribute *appl* in the matching publications to be equal. This is expressed using the variable symbol $X.

```
{Rule (((job-status (appl = $X) (job-name = A)(state = succ)) &
       (job-status  (appl = $X) (job-name = B)(state = succ)))||
       (job-status  (appl = $X) (job-name = C)(state = succ)))
  =>  (forward a notification to proper destinations)}
```

Events in applications may have sequential relations, that is, one event happens before the occurrence of another event. The *sequence* pattern describes this kind of event relation. The composite subscription $(s_1;_{[timespan:ts]}\, s_2)_{[within:wi]}$ is matched when a publication p_2 matching s_2 occurs provided publication p_1 matching s_1 has already occurred. The *timespan* parameter specifies the minimum time step of the two publications; the *within* parameter limits the maximum time span between them. In the *sequence* pattern, a *time* predicate is added to standard subscriptions. Suppose s_1 and s_2 subscribe to job `A` and job `B` respectively, as in the previous example. The composite subscription is mapped to a rule as described below. This pattern requires that the time p_2 is published is greater than that of p_1.

```
{Rule ((job-status ...(job-name = A)(time = $Y)...)  &
       (job-status ...(job-name = B)(time > $Y+ts)(time < $Y+wi)))
  =>  (forward a notification to proper destinations)}
```

The *repetition* pattern describes an aperiodic or periodic event. PADRES can describe the repetition events as `Repetition(S, n, attr, v)`. It means publications matching S happen n times and attribute *attr* increases by step v, or decreases if v is negative. The iteration is controlled the value of *attr* with step v. A *repetition* pattern can be mapped to a rule as below.

```
{Rule ((job-status ...(job-name = A)(attr = $Z)...)  &
       (job-status ...(job-name = A)(attr = $Z+v)...)&
                        ...                          &
       (job-status ...(job-name = A)(attr = $Z+(n-1)v)...))
  =>  (forward a notification to proper destinations)}
```

Composite subscriptions can be composed in a nested fashion using the above operators to create more complex composite subscriptions. Mapping composite subscriptions to rules consists of three steps: first, each atomic subscription is mapped to part of the antecedent. Second, connect each part of the antecedent using logical operators and variables. Third, activites to be taken after matching are mapped to the consequent of the rule. In the PADRES broker, both atomic and composite subscriptions are mapped to rules. That is, extending this subscription language does not require significant changes in the matching engine.

4.2 Composite Subscription Routing

In a large-scale publish/subscribe system, publications are issued at geographically dispersed sites. A centralized composite event detection scheme constitutes a potential bottleneck and consists of a single point of failure. All atomic publications have to be centrally collected in order to detect an occurrence of a

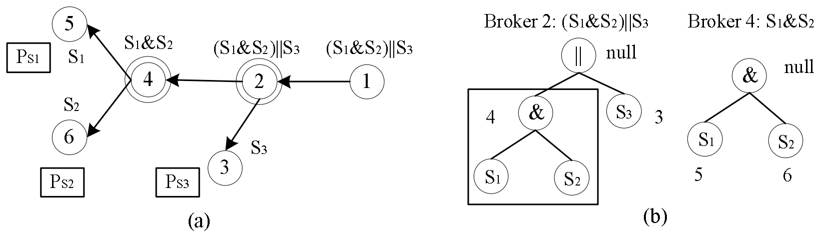


Fig. 4. Composite Subscription Routing

composite event. Our distributed solution consists in detecting parts of an event pattern and aggregating the parts. A notification message signifying the occurrence of the composite event is sent to the subscriber only after all the parts are detected. The main difficulties of distributed event detection are routing composite subscriptions, including where and how to decompose a composite subscription, and routing the individual parts of the subscription. The location of detection should be as close to publishers as possible to ensure that the publications contributing to a given composite subscription are not unnecessarily disseminated throughout the broker network. In other words, the composite subscription should be forwarded to the publishers within the broker network as far as possible before it is decomposed. As a result, bandwidth usage is reduced. Following the example in Fig. 4 (a), suppose a composite subscription $((s_1$ & $s_2)$ $\|$ $s_3)$ arrives from broker 1, and its matching publications arrive from broker 3, 5, and 6. The composite subscription is split into parts along the routing path, since the matching publications may arrive from different brokers. Atomic subscriptions s_1 and s_2 are detected at broker 5 and 6 respectively and the detection results are combined at broker 4 for $(s_1$ & $s_2)$. Moreover, the detection results could be shared among subscribers that have common subexpressions of composite subscriptions in order to save bandwidth and computational effort.

Each atomic subscription in a composite subscription could find its destination(s) from SRT. If all atomic subscriptions have the same next hop destination, a broker should forward the composite subscription as a whole to the destination; otherwise the composite subscription should be split into parts according to different destinations, and each part should be forwarded to its own destination. In Fig. 4 (b), since all matching publications are coming from broker 2, broker 1 routes the composite subscription as a whole. At broker 2 publications matching s_1 and s_2 arrive from broker 4 according to the SRT, while s_3's publications will arrive from broker 3. As a result, the composite subscription is split into two parts: $(s_1$ & $s_2)$ and s_3. The first part is sent to broker 4, where it is split into s_1 and s_2, and sent to broker 5 and 6 respectively. The second part s_3 is routed to broker 3. The routing scheme is to detect the event pattern matching a composite subscription at a location which is as close as possible to the data sources. A composite subscription is mapped to a rule, and a publication is mapped to a fact at a single broker. The rule-based broker matches facts against rules and decides where to route the notification if there is a match. Therefore, the broker acts as both a message router and a composite event detector. The advantage of using a rule-based matching engine is that it enables composite subscriptions naturally without significant changes to the broker.

Composite subscriptions in PADRES are represented by a tree structure. When a broker receives a composite subscription, it performs the following steps. First, a destination tree is built bottom-up for the composite subscription according to the SRT, which knows where all the atomic subscriptions came from. Leaf nodes of the tree are destinations of atomic subscriptions; an internal node is the destination of its child nodes if the two child nodes have the same destination, or *null* otherwise. If a node is *null*, all its parent nodes are *null*. Each node in the composite subscription tree has a corresponding node in the destination tree. The recursive algorithm for building such a tree is presented in Fig. 5. The average time complexity of this algorithm is O(N) and the average space complexity is O(N+$logN$), where N is the number of atomic subscriptions in a composite subscription. Second, the composite subscription tree is split according to its destination tree. The decomposition process of a composite subscription tree is top-down. If the destination of a node in the composite subscription tree is *null*, the subscription represented by the node is split into two parts, one for each child node. Otherwise the node and its subtree are kept as a whole unit. The algorithm is given in Fig. 6. The time and space complexity of this algorithm is the same as algorithm *buildDestinationTree(cs)*. Last, each part resulted from the decomposition is routed to its destination, and the composite subscription is mapped to a rule and inserted into the PRT for later event detection. The process happens at each broker on the routing path. As a result, all the atomic subscriptions are routed to their destinations as specified by the destination tree and the broker network is ready to detect composite events in a distributed mode. Moreover, after composite subscriptions are split into atomic subscriptions, the covering-based and merging-based routing techniques can be applied to create compacted PRTs/SRTs at brokers and further reduce the network traffic [18,15].

```
buildDestinationTrees(cs):
Input: composite subscription cs
Output: a destination tree T

Initialize T according to cs
If (cs.root is leaf node) {
        T.destination = cs.root.destination
  }Else{
        T.left = buildDestinationTree(cs.root.left)
        T.right = buildDestinationTree(cs.root.right)
        If (T.left.destination == T.right.destination) {
                T.destination = T.left.destination
        }Else{
                T.destination = null
        }
}
Return T
```

Fig. 5. Algorithm for Building a Destination Tree

```
decomposition(cs, T):
Input: composite subscription cs; destination tree T
Output: a set of subscriptions S

Initialize S = empty
If (T.destination == null){
        S = S ∪ decomposition(cs.root.left, T.left) ∪
             decomposition(cs.root.right, T.right)
}Else{
        S = S ∪ cs
}
Return S
```

Fig. 6. Algorithm of Decomposing a Composite Subscription

There are several advantages of using distributed composite event detection. Redundant detection is eliminated by sharing the detection results among subscribers. For the overlapping expressions of composite subscriptions issued by clients, the detection is executed once, and subscribers close to each other can reuse the detection results. Distributed detection also reduces network traffic. A composite subscription is forwarded into the network as far as possible before it is split. As a result, the number of subscriptions injected into the network does not increase significantly for composite subscriptions. Furthermore, composite events are detected close to their data sources in the network and are not widely disseminated. A single notification is sent after a match, instead of a set of individual notifications for each matching publication, reducing the number of publications routed in the federation.

4.3 Distributed Composite Event Detection

Each broker is an atomic/composite event detector. It processes a large number of publications/subscriptions and maintains them as rules/facts in its matching engine. The broker matches the rules against the facts. The occurrence of a composite event is marked by the occurrence of the last event that completes the composite event. When a publication is received, it is inserted as a fact. The fact may match part of a rule, or several rules. Then the rule(s) are maintained in the engine in a partial match state. If the fact does not fire a rule, the matching engine updates the partial match state with the new fact. If the fact fires a rule, that is, the fact makes a partially matched rule a full match then associated composite subscription is satisfied. A notification message with a set of matching publications, called a detection set, as its payload is issued as result. The main problem in composite event detection is consuming the publications received by the brokers, e.g. among all the matching publications what should go to the detection set. To be more flexible, our matching engine provides all the possible combinations of matching publications. Consider the composite subscription `((`s_1 `&` s_2`)` `&` s_3`)`, where s_i matches publication type e_{ij}, i=1 $\sim$ 3 and j is the instance number of e_i. Subscription is issued after e_{22}. Our composite event detection semantic is based on the constraint that at least one of the events in the detection set must be issued after the composite subscription. This is to remain compatible with standard publish/subscribe approaches, where subscriptions refer to information published in the future. The subscription is inserted into the PRT as a rule. The matching engine filters out the solution set $< e_{11}, e_{21}, e_{31}$, which is older than the subscription. The rule is partially matched in the matching engine. Four possible composite event patterns matching the subscription are given in Fig. 7 when e_{32} arrives.

4.4 Unsubscription of Composite Subscriptions

In PADRES, if a client wants to revoke a subscription, it issues an unsubscription message. To maintain the consistency of routing tables in the broker network, *ack* messages are used to ensure the unsubscription process is successful. An

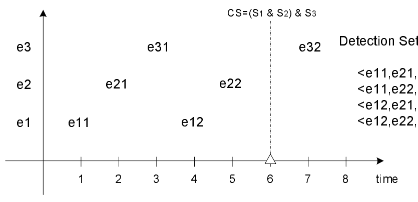


Fig. 7. Event Consuming

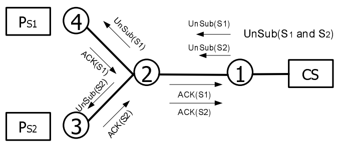


Fig. 8. Unsubscription

ack message is sent if a broker removes a subscription from its matching engine. The unsubscription message is sent periodically every t_1 *ms* until its *ack* is received.[3] When a broker receives an unsubscription, the following three steps are performed: first, it checks the SRT to find the list of neighbor brokers to which it previously routed the subscription (or part of the subscription). Second, if the list is empty, it removes the subscription from its routing table, and sends back an *ack* message. Otherwise, it splits the unsubscription if necessary, forwards the unsubscription(s) to the brokers in the list, and waits for *ack* messages from them. Last, the broker cannot safely delete the subscription until it collects all the *ack* messages back from its neighbors. An *ack* message is sent back to the broker/client who forwards the unsubscription. Fig. 8 shows an example of the unsubscription process.

5 Case Study: Event-Based Workflow Management

A workflow management system performs coordinated execution of workflows. A *workflow*, also called an *application*, is a set of business-related activities that are invoked in a specific sequence to achieve a business goal. An *activity* is a computer *job*, such as a Unix job, a Windows NT job or a database job, which is executed by a *job execution agent*. The agents are distributed in the network, working in coordination with each other. The workflow manager starts an execution *instance* of a workflow by issuing a *workflow trigger*, a message starting the execution of a workflow.

The publish/subscribe messaging paradigm efficiently supports the decentralized execution of event-driven, loosely coupled applications, such as workflows and business processes. Since routing is content-based, the workflow manager does not need to maintain the address information of each job execution agent and route the messages to and from agents, as those messages are automatically delivered using content-based routing. Moreover, no centrical workflow manager is required, as workflow processing is fully decentralized. Job execution agents are lightweight components without special logic for workflow management. They only need the capability to send and receive messages and execute jobs. The

[3] If the *ack* does not arrive in t_2 *ms*, we assume the neighbor broker has failed. A fault tolerant module is called to recover SRTs/PRTs. The details are beyond the scope of this paper.

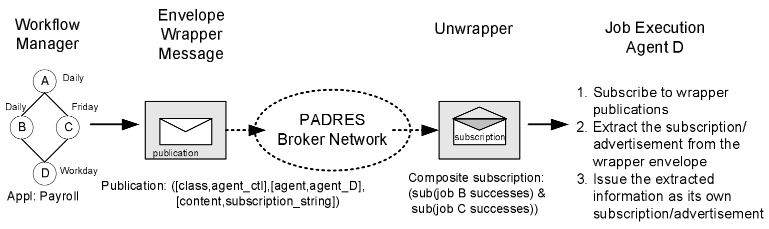

Workflow Manager
Envelope Wrapper Message
Unwrapper
Job Execution Agent D
A
Daily
B
C
Friday
D
Workday
Appl: Payroll
publication
PADRES Broker Network
subscription
Publication: ([class,agent_ctl],[agent,agent_D], [content,subscription_string])
Composite subscription: (sub(job B successes) & sub(job C successes))
1. Subscribe to wrapper publications
2. Extract the subscription/ advertisement from the wrapper envelope
3. Issue the extracted information as its own subscription/advertisement

Fig. 9. Envelope Wrapper Message

agents are publish/subscribe clients, who subscribe and publish to exchange information using the publish/subscribe network. PADRES, which introduces composite subscriptions in addition to the standard publish/subscribe features, illustrates the successful application of the publish/subscribe paradigm to workflow management. The overall architecture for supporting workflow processing is shown in Fig. 10. The publish/subscribe-based workflow management system includes four components: workflow transformation, workflow deployment, workflow execution and workflow monitoring.

Workflow Transformation. Workflows are specified as XML documents detailing the job execution information and the various dependencies between jobs. The XML documents are converted into a set of subscriptions and advertisements. Fig. 9 shows an example of a workflow consisting of four jobs. Job `D` depends on job `B` and job `C`, respectively, subject to certain constraints, such as time and resources. Composite subscriptions are used to express all job dependencies and constraints. A job can be run only when its job dependency subscription is matched. Advertisements enable job execution agents to publish job status information after completing a job. In a workflow, the jobs that have no predecessors are called *start jobs*, for instance, job `A` is a start job in *payroll*. Start jobs subscribe to a *workflow trigger*.

Workflow Deployment. The goal of workflow deployment is to send the subscriptions and advertisements generated from the workflow definition file to the corresponding *job execution agents*. For example, the agent for job `D` should subscribe to execution status information of job `B` and job `C`. To send job dependency subscriptions to job execution agents, the workflow manager uses an *envelope wrapper*[4] message pattern to wrap the subscription inside an envelope message that is complies with the publish/subscribe messaging paradigm. Each envelope wrapper is a publication which indicates its destination agent. Agents receive the wrapper messages by subscribing to the wrapper. For instance, agent `D` subscribes to ([class,=,agent_ctl],[agent,=,agent_D]) in Fig. 9. As a result, agent `D` receives the wrapper with a composite subscription embedded in the message. Agents unwrap envelope messages by extracting the subscriptions from the envelopes, and issue them as their own subscriptions. The same process applies for advertisements. As a result, the agents are ready to receive and publish work-

[4] The class of the envelope wrapper message is *agent_ctl*.

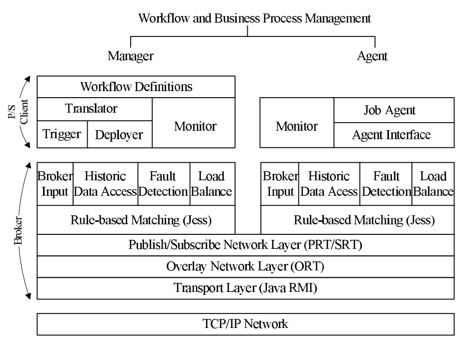


Fig. 10. PADRES Protocol Stack

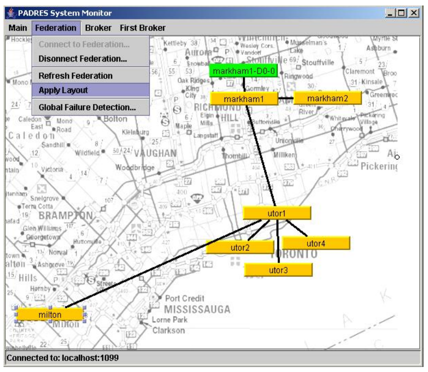


Fig. 11. PADRES System Monitor

flow execution information. This deployment process is performed entirely using publish/subscribe interactions.

Workflow Execution. The job execution agents are both subscribers and publishers. The dual roles enable them to exchange messages within the publish/subscribe messaging system, enabling a coordinated execution of the workflow. A particular instance of a workflow is started by a *trigger*. It fires all start jobs. When these start jobs are finished, they trigger their subsequent jobs. Execution continues until all the jobs defined in the workflow are finished. The key to workflow execution is job dependency subscriptions, which determine the order of execution of jobs. All the message routing is automatic and transparent to the workflow management layer.

Workflow Monitoring. A workflow management system maintains a trace of job executions and provides a control and monitoring interface. The monitor may be a separate publish/subscribe client in Fig. 10. An important function of a workflow management system is monitoring. Real-time monitoring fits directly in the content-based publish/subscribe paradigm. The monitor simply subscribes to job execution status information publications of a particular set of jobs. As a result, when the job is completed, the monitor knows the execution status information. PADRES also provides a graphical interface which allows the monitor to visualize the network topology and message routing in order to gain an intuitive picture of the workflow execution as shown in Fig. 11. All the monitoring functions are entirely based on the publish/subscribe layer's primitives.

There are several advantages to use a publish/subscribe system for workflow management. First, workflows are by nature event-driven. A workflow is started by a trigger and is driven by publication messages of finished jobs. Control messages are automatically and transparently routed to the appropriate agents in the publish/subscribe layer. Second, workflows are easily scalable to multiple platforms, as the publish/subscribe architecture supports cross-platform applications in a distributed environment. Moreover, large-scale applications can be supported easily. Third, the management of workflow definitions is flexible. It is easy to add, modify or delete jobs from a workflow. The modification can be performed dynamically. Furthermore, job monitoring is a natural fit for the publish/subscribe paradigm, since managers can subscribe to job execution information. Fourth, multiple workflows can be deployed into the broker federation at the same time. Concurrent execution of several workflow instances is possible. Finally, the distributed application deployment provides a robust workflow management mechanism. Deploying a workflow application into a distributed network, instead of using a central manager to control the execution of the workflow, avoids a single point of failure.

6 Evaluation

We implement PADRES in Java with JDK1.4.2 using *Jess* as a matching engine and RMI as the native transport protocol. All our experiments are performed on a computer with an Intel Xeon 3GHz processor and 2GB RAM, of which 1GB is allocated to the JVM. Due to lack of benchmarks or real application data, we generate the subscriptions and publications using a workload generator which produces the data by selecting between 3 and 6 attributes from a list of twenty attributes $\{a_i,\ i = 1...20\}$ and selecting values from given value ranges, [1..100] by default. We generate two kinds of data sets. Attributes and values in the first data set are selected randomly following a uniform distribution. The second data set follows a Zipf distribution, in which attributes are chosen from the attribute set $\{a_i,\ i = 1...20\}$, where the probability of selecting a_i is $\frac{1}{i}$, and value v_i is chosen with the probability of $\frac{1}{v_i}$. For evaluating the distributed workflow management system, we deployed a distributed network of 5 overlay brokers, one with 10 job agents and one with 30 job agents, each representing a separate workflow. In our experimental evaluation we focus on proving the viability of composite subscriptions to encode workflows and business processes and the use of publish/subscribe for the decentralized execution of these workflows. Furthermore, we aim to evaluate the performance and overhead associated with composite event detection and the effect of composite subscriptions on network traffic for the execution of workflows. A small network is fully sufficient for this purpose. The evaluation of large-scale broker networks comprising hundreds of nodes is deferred to future work.

Publication Matching Time. We generate 200,000 subscriptions and 5,000 publications for both uniform distribution and Zipf distribution to evaluate the publication matching time of PADRES brokers. Fig. 12 shows the average matching time of publications against atomic subscriptions. The matching time is given using a logarithmic scale. Each data point is obtained by averaging the time taken to process 5,000 publications. We compare our broker based on the *Jess* rule-based matching engine with two other methods. One is a naive matching algorithm which linearly scans the routing table to find the matched subscriptions. The other is a matching algorithm that is similar to the predicate counting algorithm [2]. This algorithm calculates distinct predicates only once. Our exper-

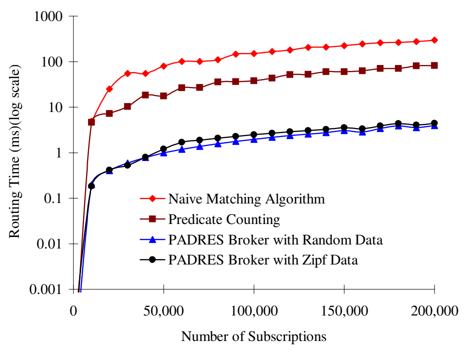


Fig. 12. Publication Matching Time

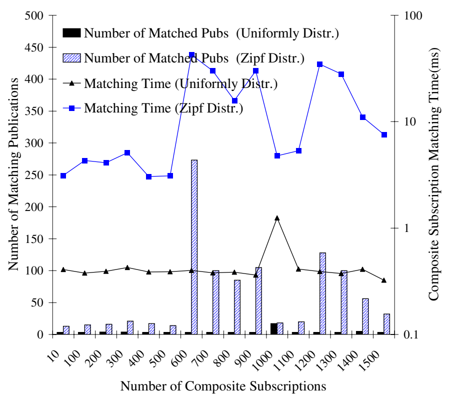


Fig. 13. Composite Event Detection

iments show that the rule-based matching engine using a Rete network is very efficient. It takes only 4.52ms to route a publication against 200,000 subscriptions for both sets. The well-known Rete algorithm trades space for time. (Matching a publication against 200,000 subscriptions, the PADRES broker uses 644MB of memory while the predicate counting algorithm uses about 38MB memory space.)[5] The matching time does not increase significantly with an increase in the number of subscriptions for both data sets. This indicates that the Rete-based approach is suitable for large scale publish/subscribe systems and can process a large number of publication and subscription messages efficiently.

Composite Subscription Matching Time. The performance of composite subscription matching is shown in Fig. 13. We first inject 1,000 publications into the broker, and then insert 2,000 composite subscriptions, each of which consists of 3 atomic subscriptions. Fig. 13 shows the average detection time per composite subscription against the publications and the number of matched publications. Each data point in Fig. 13 represents the average detection time for 50 composite subscriptions. In the uniformly distributed data set, the number of matched publications per composite subscription[6] does not change significantly, as a result, the composite subscription matching time is stable. In the Zipf data set, more publications are matched and the composite subscription matching time varies according to the number of matched publications. The results show that, given the publication set, the detection time does not increase with the number of composite subscriptions in the matching engine for both data sets. The matching time is effected by the number of matched publications. That is, the more publications match a subscription, the longer it takes the matching engine to process the subscription. From the experiment, we notice that if there is

[5] We maintain two *Jess* Retes in the matching engine as SRT and PRT. To support composite subscription, publications are maintained in PRT as facts which consume the space.

[6] The matched publications maybe count multiple times in different detection sets.

Table 1. Composite Subscription Routing Delay

Number of Atomic Subscriptions	2	3	4	5	6
Routing Delay (ms)	3.210	5.367	9.287	11.437	12.074

no publication matching a composite subscription, the matching engine stops the matching in $0.01ms$ no matter how many composite subscriptions are resident in the broker. The number of publications resident in the matching engine affects the detection time as well. The larger the number of publications, the more publications are matched, and the longer matching time it takes per composite subscription.

Routing Delay. We route a composite subscription according to its destination tree. The routing delay for a composite subscription at a broker includes the time to build the destination tree and to split the composite subscription. Table. 1 shows that the routing delay increases with the number of atomic subscriptions included in a composite subscription. This substantiates the time complexity of the two algorithms we discussed in Section 4.2 are *O(N)*, where N is the number of atomic subscriptions in a composite subscription. The more complex a composite subscription is, the longer it takes to route the subscription.

Network Traffic Overhead. Detecting composite events in the broker network reduces the message traffic received by clients. We compare two scenarios. In the first scenario, a client issues 200 composite subscriptions, each consisting of 5 atomic subscriptions. In the second scenario, instead of composite subscriptions, the client issues the 1000 atomic subscriptions that make up the original 200 composite subscriptions. After 40,000 publications are injected into the broker network, we measure the number of notifications received by the client in the different scenarios, as shown in Fig. 14. The result shows that the number of notifications sent to the client is greatly reduced by the composite subscriptions, yielding an overall reduced message traffic. For this scenario, the reduction is up to 65%.

Distributed Workflow Deployment and Execution. We measure the network traffic overhead of a workflow deployment and execution to show the effect

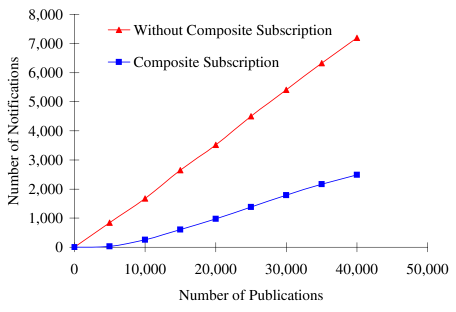
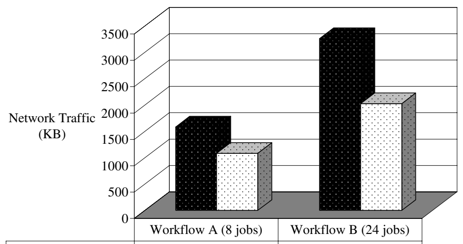


Fig. 14. Number of Notifications



	Workflow A (8 jobs)	Workflow B (24 jobs)
■ W/t Composite Sub.	1582	3264
□ Composite Sub.	1084	2026

Fig. 15. Workflow Traffic

of composite subscriptions for workflow processing. We design two workflows: workflow A is a workflow with 8 jobs which includes the *payroll* example, a diamond workflow shown in Fig. 9, twice in sequence. Workflow B is a workflow with 24 jobs, which is workflow A followed by 4 concurrent diamond workflows. The manager dispatches the workflow to agents[7]., and the agents submit advertisements and subscriptions, which represent the job dependencies. Without composite subscriptions, agents have to subscribe to several atomic subscriptions instead of a single composite one. When a composite subscription issued by an agent is matched, only one notification message is sent back to the agent, as opposed to several individual atomic notifications. So more messages are disseminated in the broker network. To simplify the measurements, we assume each publication and subscription message is 1KB. We measure the traffic overhead of the workflow deployment and 10 execution instances in Fig. 15. The results show that composite subscriptions reduce the network bandwidth by about 40% for both workflows.

7 Conclusions

In this paper, we introduce the PADRES project. PADRES is a distributed publish/subscribe system building on and extending existing content-based routing approaches. PADRES offers an expressive subscription language, including unique features such as composite subscriptions, various coordination patterns, a notion of time and time-based subscriptions, and variable bindings. PADRES fully integrates these features in a standard content-based subscription language. The choice of language features has been derived from the requirements of workflow management and business process execution use cases. For example, structured coordination activities, such as `sequence` and `while loops`, today available in BPEL4WS, are expressible.

The PADRES brokers build on a rule-based approach to perform content-based event matching and composite event detection. We present two algorithms for composite subscription routing and distributed composite event detection. The experimental evaluation of PADRES shows that the rule-based broker design is an efficient alternative to existing content-based message routing, matching, and distributed event detection algorithms. For example, the routing overhead is on the order of a few milliseconds for hundreds of thousands of subscriptions.

A distributed, decentralized workflow management system based on PADRES is presented to validate the approach. The case study proves the viability of the approach and introduces the concepts of decentralized deployment, execution, and monitoring of workflows entirely in the publish/subscribe layer. Our experiments show that through the use of composite subscriptions, subscribers receive less notification messages. As a result, the overall network traffic overhead is reduced. The experiments for workflow management further substantiate this conclusion by showing that more benefits are gained from composite

[7] This is done through the publish/subscribe based injection mechanism described in Section 5.

subscriptions, for both workflow deployment and execution, leading to about 40% fewer messages overall.

Acknowledgements

We would like to thank the PADRES team for their help and feedback in carrying out this research. The team is currently comprised of Eli Fidler, Vinod Muthusamy, Pengcheng Wan, Alex Cheung, and Serge Mankovski (Cybermation, Inc.) Between May 2003 and April 2005, the PADRES project was supported by Cybermation, Inc., CITO, and NSERC.

References

1. A. Aho, J. Hopcroft, and J. Ullman. Data structures and algorithms. *Reading, MA: Addison-Wesley; 1983*, 1983.
2. G. Ashayer, H. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *DEBS'02 Workshop at ICDCS'02*, Vienna, Austria, 2002.
3. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
4. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1):1–26, 1994.
5. S. Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems(DEBS'02)*, 2002.
6. G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9), 2001.
7. F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, 2001.
8. E. Fidler, H.-A. Jacobsen, G. Li, , and S. Mankovski. Distributed publish/subscribe for workflow management. *International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI'05), Leisester, UK*, 2005.
9. C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, 1982.
10. E. J. Friedman-Hill. Jess, The Rule Engine for the Java Platform. `http://herzberg.ca.sandia.gov/jess/`.
11. S. Gatziu and K. R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proceedings of the 4th Intl. Workshop on Research Issues in Data Engineering (RIDE): Active Database Systems, Houston, Texas*, 1994.
12. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model & implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, 1992.
13. R. E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. *In 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

14. IBM and Microsoft. Business process execution language for web services version 1.0. `http://dev2dev.bea.com/techtrack/BPEL4WS.jsp`.
15. G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. *International Conference on Distributed Computing Systems (ICDCS'05), Columbus, Ohio, USA*, 2005.
16. M. Mansouri-Samani and M. Sloman. GEM: A generalized event monitoring language for distributed systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), June 1997.
17. D. Moreto and M. Endler. Evaluating composite events using shared trees. *IEE Proceedings - Software*, 148(1):1–10, 2001.
18. G. Mühl. *Large-scale content-based publish/subscribe systems*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, 2002.
19. L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting IP multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms*, pages 185–207, 2000.
20. P. R. Pietzuch and J. Bacon. Hermes: A distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 611–618. IEEE Computer Society, 2002.
21. P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, January/February 2004.
22. A. Ulbrich, G. Mühl, T. Weis, and K. Geihs. Programming abstractions for content-based publish/subscribe in object-oriented languages. In *CoopIS/DOA/ODBASE (2)*, pages 1538–1557, 2004.