Extending Rebeca to Support Concept-Based Addressing

J. Antollini, M. Antollini, P. Guerrero, M. Cilia*

Dept. of Computer Science,
Faculty of Sciences, UNICEN
Tandil, Argentina
{janto,manto,pguerrer,mcilia}@exa.unicen.edu.ar

Abstract. The event-based approach is well suited for integrating autonomous components or applications into complex systems by means of exchanging events. Event-based systems need an event dissemination mechanism to deliver relevant events to interested consumers. These events encapsulate data which can only be properly interpreted when sufficient context information about its intended meaning is known. In general, this information is left implicit and as a consequence it is lost when data/events are exchanged across system or institutional boundaries. For this reason, to exchange and process events in a semantically meaningful way, explicit information about their semantics in the form of additional metadata is required. This paper presents a solution to this problem and its application to an existing open source event notification service called Rebeca.

keywords: event-driven systems; distributed environments; data dissemination; notification services; publish/subscribe systems; data heterogeneity; semantic metadata

1 Introduction

Since the last few years the development of enterprise applications is moving away from tightly-coupled applications and towards systems of loosely-coupled, dynamically bound components. This trend fits the event-based application paradigm. The event-based approach is well suited for integrating autonomous components into complex systems by means of exchanging events. Because event-based systems do not require a-priori knowledge about the consumers of events they are easy to evolve and scale.

Notice that the exchanged events encapsulate data about a given happening of interest, which can only be properly interpreted and used when sufficient context information is known. In traditional centralized systems, this context information is typically known by the users and left implicit. When data and events are exchanged across component or institutional boundaries contextual information is usually lost. To process events in a semantically meaningful way, explicit information about the semantics of events and data is required.

Event-based systems need an event dissemination mechanisms (or notification service) with the purpose to timely deliver relevant events to interested consumers. Since a couple of years, the publish/subscribe interaction paradigm is gaining relevance. It basically consists of a set of clients that asynchronously exchange events, decoupled by a notification service that is interposed between them. Clients can be characterized as producers or consumers. Producers publish notifications, and consumers subscribe to notifications by issuing subscriptions, which are essentially stateless message filters. Consumers can have multiple active subscriptions, and after a client has issued a subscription the notification service delivers all future matching notifications that are published by any producer until the client cancels the respective subscription.

To the best of our knowledge (see Section 2 for details), in almost all publish/subscribe mechanisms only the data structure of events (data to be exchanged) is exposed to all participants. This reflects a low level support for event consumers that based on this scarce information must express their interest without having a concrete definition of meaning nor explicit assumptions made by event/data producers. Without this kind of information event producers and consumers are expected to fully comply with implicit assumptions made by participating software components or applications. Even in the cases of a very small set of applications within an enterprise this approach is questionable.

Since publish/subscribe mechanisms decouple producers and consumers, it must be noticed that they should share a common understanding in order to express their mutual interests. In other words, events

* also Dept. of Computer Science, TU Darmstadt, Germany

must be understandable beyond the closed confines of a single component or application. That includes the case when dealing with applications that interact across traditional borders regardless of economic, cultural or linguistic differences (e.g. system of units, currency or date time format). Because the source of an event cannot anticipate who is interested in a given event and where and when it must be delivered, a higher-level publish/subscribe infrastructure is needed. This basically adds to the current pub/sub mechanisms the following two requirements:

- the usage of a common vocabulary for defining interests, and
- the correct interpretation of information independently of its origin and place of consumption.

We have initially proposed the use of the *concept-based addressing* [1] as a part of a high-level dissemination mechanism for distributed and heterogeneous event-based applications. Its goal is to provide a higher level of abstraction to describe the interests of event producers and consumers [4]. This is achieved by supporting from the ground up ontologies which provide the base for correct data and event interpretation. Rather than requiring every producer or consumer to use the same homogeneous namespace (as is common in other publish/subscribe systems) we provide metadata and conversion functions to map from one context to another. This last feature allows event consumers to simply specify the context to which events need to be converted before they are delivered for client processing.

The initial implementation [3] was built on top of a commercial publish/subscribe product. Today we are moving our concept-based implementation to an open platform by extending an open source publish/subscribe notification service (called Rebeca [10]). In this paper we describe precisely how these extensions were carried out.

Notification services for large-scale environments cannot simply use flooding algorithms to route the notifications from producers to consumers [9]. However, clever strategies exist like *covering* and *merging* that allow the network's brokers to neatly select what to forward and what not. Our approach recognizes the existence of these strategies and shows how they can be exploited with the concept-based addressing model.

The rest of this paper is organized as follows. In section 2, related work on publish/subscribe systems is presented. In section 3, a background on the ideas that were combined in this work is given. Section 4 describes the proposed extensions to Rebeca providing details about how the data model is extended and how filters are represented, rewritten and evaluated. Finally, conclusions and future work are presented in section 5.

2 Related Work

Notifications about events and their timely delivery to the user represent valuable information. The dissemination of events is usually under the responsibility of notification/event services.

In CORBA, a primitive event service [7] was introduced to provide a mechanism for asynchronous interaction between CORBA objects. Here, an event channel acts as a mediator between suppliers and consumers of events. To overcome deficiencies of this service specification, the notification service [14] was proposed as a major extension with support for quality of service specifications and basic event filtering.

The Java Message Service (JMS) [16] provides the Java technology platform with the ability to process asynchronous messages. JMS was originally developed to provide a common Java interface (API) to legacy Message Oriented Middleware (MOM) products like IBM MQ-Series (called today WebSphere MQ) or TIB/Rendezvous. In this way, the JMS API provides portability of Java code allowing the underlying messaging service to be replaced without affecting existing code. JMS provides two models for messaging among clients: point-to-point (using a queue) and publish/subscribe (by means of topics). Under the topic model, consumers not only subscribe to a channel but can also specify additional boolean predicates by means of a restricted set of SQL `WHERE` expressions [8]. As a recognition of the relevance of publish/subscribe for enterprise applications, JMS was incorporated as integral part of the Enterprise Java Beans (EJB) component model in the EJB 2.0 specification [5].

In recent years, academia and industry have concentrated on publish/subscribe mechanisms because they allow loosely coupled exchange of asynchronous notifications, facilitating extensibility and flexibility. The channel model has evolved to a more flexible subscription mechanism, known as subject-based addressing, where a subject is attached to each notification [13]. Subject-based addressing features a set of rules that defines a uniform name space for messages and their destinations. This approach is inflexible if changes to the subject organization are required, implying fixes in all participating applications.

To improve expressiveness of the subscription model the content-based approach was proposed where predicates on the content of a notification can be used for subscriptions. This approach is more flexible but requires a more complex infrastructure [2, 9]. Many projects in this category concentrate on scalability issues in wide-area networks and on efficient algorithms and techniques for matching and routing notifications to reduce network traffic [15, 11, 6]. Most of these approaches use simple boolean expressions as subscription patterns since more powerful expressions cannot be treated.

As it was mentioned before, only the data structure of events is exposed to all participants. This puts in evidence the low level support regarding data exchange which directly impacts on the broad usability of such an important piece of the communication infrastructure.

3 Background

This section provides the background to understand the ideas that were combined in this work.

3.1 The Concept-based Addressing Model

As it was mentioned before, the publish/subscribe paradigm provides two important benefits: it naturally decouples producers and consumers and makes them anonymous to each other. However, it must be noticed that producers and consumers should share a common understanding in order to express their mutual interests. But, this may be not the case when dealing with loosely-coupled applications that interact across traditional borders. It implies that the information to interpret message content is left implicit. Thus, it would be desirable to add meta information to the data or, in other words, information that describes the information.

The goal of *concept-based* publish/subscribe is to provide a higher level of abstraction to describe the interests of publishers and subscribers and to make assumptions explicit. It provides a flexible interaction mechanism for open, distributed and heterogeneous event-based applications. This is achieved by supporting from the ground up ontologies (see the MIX Model in section 3.3) which provide the base for correct data and event interpretation. Rather than implicitly assume an homogeneous namespace among publishers and subscribers (as is common in other publish/subscribe systems) it provides metadata and conversion functions to solve data heterogeneity issues. The concepts specified in the ontology provide a common vocabulary (or dictionary) for which no further negotiation is necessary.

3.2 REBECA Architecture

Next we describe the current implementation of the REBECA notification service [9], so as to show the starting point where changes were applied.

The REBECA communication interface to the service is rather simple. The set of producer clients interacting with the system publish notifications by invoking the *pub(n)* operation, giving the notification n as parameter. The published notification can potentially be delivered to all connected consumers via an output operation called *notify(n)* (a callback function called on the registered subscribers to deliver a notification). Consumers register their interest in specific kinds of notifications by issuing subscriptions via the *sub(F)* operation which takes a filter F as parameter. Each client can have multiple active subscriptions which must be revoked separately by using the *unsub(F)* operation.

The service implementation is distributed to meet scalability considerations. The communication topology of the publish/subscribe system is given by a graph, which is assumed to be acyclic and connected. Figure 1 shows an example of this graph with the basic components. Each *client* consists of the application logic itself and a *local broker proxy*. The *edges* are point-to-point communication links that obey FIFO message ordering. Inside the *broker's network* there are *inner* or *border brokers*, which are processes that route the notifications along multiple hops to the appropriate clients, using the content-based addressing model with several routing strategies.

Data Model REBECA is an event notification service framework. Its data model is an essential part of it and it's composed by two closely interrelated components: *Notifications* and *Subscriptions*. The former

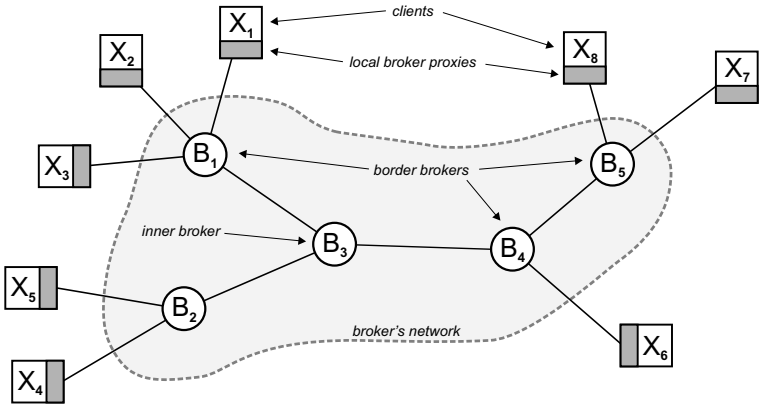

Fig. 1. REBECA architecture

are messages that reify and describe `Event` occurrences emitted by publishers, while the latter are message `Filter`s that express subscribers interests.

The `Event` class is the base class of all notifications. Basically it encapsulates message data. The `Filter` class is the base class of all filters. Its most important functionality is to determine if a given event *matches* a filter. Also, it defines other important methods to detect similarities between filters (more details later in this section).

Routing Strategies The REBECA event notification service has a routing algorithm framework which allows several routing strategies to be implemented.

Probably, the simplest way to implement a distributed publish/subscribe system is by *flooding* notifications across the brokers network. This implies that any published notification is processed by every broker. Although it could be well suited to systems in which subscriptions are changing at a very high rate, a high volume of notifications may be forwarded unnecessarily because a notification is sent to every broker (regardless of whether or not it has a local client with a matching subscription).

This drawback led to the idea of filter-based routing. Here, each broker has a routing table that is used to route notifications based on their content to local clients and neighbors. Compared with flooding, filter-based routing reduces the quantity of notifications that are forwarded, but introduces the necessity to maintain routing tables.

With *Simple Routing*, new and cancelled subscriptions are simply flooded into the broker network such that they reach every broker. This strategy is simple but it implies that every broker has global knowledge about all active subscriptions (i.e., any routing table contains a routing entry for every active subscription).

By taking into account similarities among the subscriptions, global knowledge of active subscriptions is avoided. In this way, the notification service performance and scalability is enhanced (for concrete results the reader is referred to[12]). The following strategies do so but differ in the resulting routing table size.

When using *Identity Routing*, a subscription is not forwarded to a neighbor if an *identical* subscription is already forwarded to that neighbor. Two filters are said to be identical if they match exactly the same set of notifications.

Another strategy is *Covering Routing*. A filter *covers* another filter if the former matches at least all notifications the latter matches. A subscription is not forwarded to a neighbor if a subscription that covers the former is forwarded to that neighbor.

A different approach is *Merging Routing*. A broker can *merge* the filters of existing routing entries and forward this merger to a subset of its neighbors. The generated mergers are forwarded in a way such that only interesting notifications are delivered to a broker.

Routing with Advertisements implies that advertisements are issued by producers to indicate their intention to publish certain kinds of notifications. Similar to subscriptions, each publisher can have multiple advertisements.

3.3 Model for Data Exchange

To exchange data among loosely coupled systems, it must be taken into account that the structure and semantics of individual data items may vary, even if they describe objects of the same class of real world phenomena. Therefore, *context information* concerning the organization and meaning of data has to be given on an extensional level, i.e., on the level of data values. For this reason, we need description models that allow a flexible association of metadata with the available data items. We next describe MIX, the model used to extend REBECA to support semistructured data.

Metadata based Integration model for data X-change, or MIX for short, can be understood as a self-describing data model. This is because information about the structure and semantics of the data is not provided as a separately specified data schema, but is given as part of the available data itself. Thus, MIX allows a flexible association of context information in the form of metadata.

This model is based on the concept of a `SemanticObject`. It represents a data item together with its underlying `SemanticContext` which consists of a flexible set of meta-attributes that explicitly describe the implicit assumptions about the meaning of the data item. However, because we cannot explicitly describe all modelling assumptions the semantic context always has to be understood as a partial representation. In addition, each semantic object has a concept *label* associated with it that specifies the relationship between the object and the real world aspects it describes. These labels have to be taken from a commonly known vocabulary, or `Ontology`. In the MIX model, an ontology is a finite set of concepts and their relationships.

The association of context information with a given data value serves as an explicit specification of the implicit meaning of the data. This allows the determination of semantically equivalent semantic objects even if they are represented differently, i.e., relative to different contexts. For example

<Distance, 3850, { <Unit, "mile" >, <Scale, 1 >} >, and

<Distance, 3.85, { <Unit, "mile" >, <Scale, 1000 >} >

are semantically equivalent, because they represent the same information and we can specify a conversion function by which one representation can be transformed into the other. Such conversion functions are a prerequisite for the integration of semantic objects coming from different sources, by converting these objects, as far as possible, to a common context. In detail, two kind of conversion functions can be distinguished. The first are those embedded (or are defined) in the concept. The second are those defined in a function conversion manager (because they change along time). These conversion functions are automatically called to appropriately evaluate expressions.

4 Extending the REBECA Data Model

The REBECA data model was extended to support concept-based addressing by adopting the MIX model. The following subsections explain this extensions in detail.

4.1 Events

The basic idea was to specialize the `Event` class with the purpose to support the MIX model, delegating to a semantic object the required functionality. Thus, a new `SemanticEvent` class was defined, which has a `SemanticObject` instance variable actually containing the information.

To enable REBECA events to be transported among TCP-connected event brokers, they must implement the JAVA `Serializable` interface. Thus, each event's attribute must in turn be serializable. Fortunately, `Ontology` objects were implemented with this in mind, so nothing was needed with this respect. Figure 2 (a) shows a UML diagram of the new subclass.

4.2 Filters

Similarly, the idea was to specialize the `Filter` class with a new `SemanticFilter` subclass (figure 2 (b)). This new subclass makes use of one of the Ontology's packages that represents expressions as concepts. Hence, this subclass has an `Expression` instance variable. Just like every other Ontology concept, these expressions have their associated semantic context. This allows expressions to make explicit assumptions about the contained values. For instance, a temperature sensor would use a `Temperature` object, specifying the unit of measure

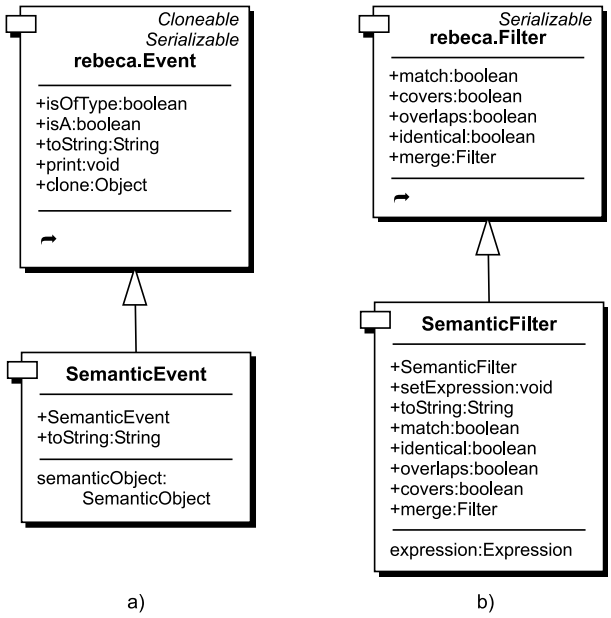

Fig. 2. Partial class diagram of the REBECA data model

used, such as Celsius. The evaluation of expressions take into consideration this meta-information (semantic context) avoiding a wrong interpretation of events.

As every `Filter` subclass, this new subclass implements several methods such as the *matching* method, and other operations used to check relationships between filters: *identity, overlapping, covering* and *merging*. These methods are explained later in this section.

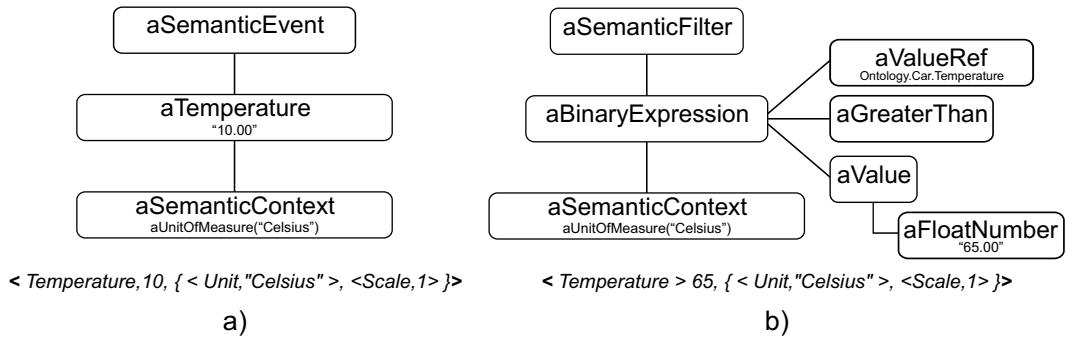

Fig. 3. Rendering of a `SemanticEvent` (a) and a `SemanticFilter` (b) objects, their relationships and MIX representations

In Figure 3 we exemplify the described classes with a static snapshot of their object representation, their relationships at a point in time and their MIX representation: a `SemanticEvent` with a `Temperature SemanticObject` (a) and a `SemanticFilter` with a `BinaryExpression` (b).

Event matching: A filter can be thought of as a stateless boolean function that is applied to the content of a notification. Formally, a notification n matches a filter F if $F(n)$ evaluates to $true$. Consequently, the set of matching notifications $N(F)$ is $\{n \mid F(n) = true\}$.

The functionality to determine whether an event matches a filter (or not) is located in the `SemanticFilter`'s `match()` method. This functionality is usually required by a routing engine in order to deliver the messages. The `SemanticFilter` delegates its work to two methods already defined in the `Expression` class:

- `resolveValueRefs(...)`, which searches the expression for `ValueRef` objects and binds them to the appropriate message values.
- `eval()`, which calculates the result in a bottom-up fashion. This evaluation implies the correct interpretation of the context of the event in question.

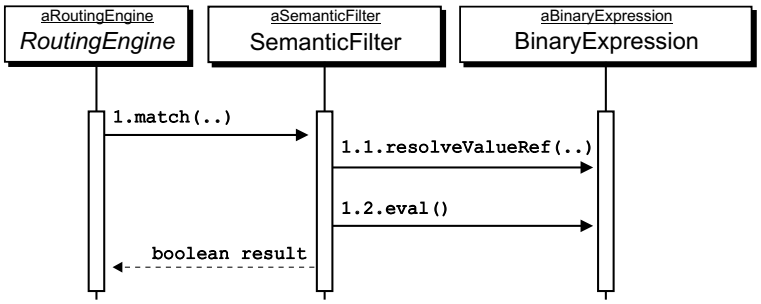

Fig. 4. Decomposition of Event Matching steps

Thus, each time a new notification arrives, the following steps are carried out (illustrated as a UML sequence diagram in Figure 4):

```
1) The broker's routing engine calls the match() method on the Filter, passing the Event
   as parameter. This method, in turn,
   1.1) resolves every reference of the incoming Event received as parameter, and
   1.2) evaluates the expression and returns its boolean value
```

Expression Satisfiability: Before developing the approaches taken to test for filter relationships, we describe the notion of expression satisfiability. The idea is to write (i.e., convert) the expression in its Disjunctive Normal Form, i.e., as a series of disjunctive terms concatenated by `OR` ($\vee$) operators. These disjunctive terms, in turn, are series of propositional variables concatenated by `AND` ($\wedge$) operators. Expressions written in the Disjunctive Normal Form are satisfiable iff *at least one* of those disjunctive terms is satisfiable. To check for a disjunctive term's satisfiability, every term's propositional variable[1] must be compared against every other term's propositional variable *that references the same concept*. If there are no contradictions within the term, then it can be stated that at least a set of values (i.e., a notification) that makes the entire expression satisfiable exists. With this building block, the next approaches can be easily constructed.

More specifically, the satisfiability of a filter F can be checked following the next steps:

```
1. Push all negations down in F using Table 1 (which is based on the De Morgan's and Complement laws),
   until they cancel with other negations, or they apply only to propositional variables.
2. Eliminate every occurrence of the 'NOT' operator changing the comparison operator, as Table 2 indicates.
```

These laws allow us to successively apply negation inwards on expressions in parenthesis.

```
3. Transform the resulting expression F into its Disjunctive Normal Form (DNF), using the distributive
   law (AND over OR) to move the ANDs down and the ORs up in the expression tree.
4. Find a disjunctive term 'T' in F, such that:
   (a) 'T' has no pairs (X,Y) of propositional variables that reference the same ValueRef concept, or
       that
```

[1] A *propositional variable* is a `BinaryExpression` composed of a `ValueRef`, a `ComparisonOperator` and a `Value`, e.g. `InTemperature` $<$ `40`.

(b) for every pair (X,Y) of propositional variables in 'T' that reference the same ValueRef concept, X and Y are mutually satisfiable (which is checked using Table3[2]).

To illustrate these steps, we will follow the next example. Lets assume that F is
$\neg$ { [($Odometer < 800$) $\wedge$ ($\neg$ $PartFailure =$ "$Engine$")] $\vee$
($\neg$ $Odometer > 2000$) }

- Step 1: Pushing negations down
  [($\neg$ $Odometer > 800$) $\vee$ ($PartFailure =$ "$Engine$")] $\wedge$ ($Odometer > 2000$)
- Step 2: Elimination of NOT operators
  [($Odometer \leq 800$) $\vee$ ($PartFailure =$ "$Engine$")] $\wedge$ ($Odometer > 2000$)
- Step 3: Conversion into the Disjunctive Normal Form
  [($Odometer \leq 800$) $\wedge$ ($Odometer > 2000$)] $\vee$ [($PartFailure =$ "$Engine$") $\wedge$ ($Odometer > 2000$)]
- Step 4: Disjunctive term analysis
  - The first disjunctive term has a pair of propositional variables that reference the same concept: ($Odometer \leq 800$) and ($Odometer > 2000$). Nevertheless, these terms are *not* mutually satisfiable, so this first term is not satisfiable.
  - The second disjunctive term doesn't have pairs of propositional variables that reference the same ValueRef concept. Hence *this term* is satisfiable, which in turn means that *the entire expression* is satisfiable.

Operation	Changes to
$NOT\ (X\ AND\ Y)$	$(NOT\ X)\ OR\ (NOT\ Y)$
$NOT\ (X\ OR\ Y)$	$(NOT\ X)\ AND\ (NOT\ Y)$
$NOT\ (NOT\ X)$	X

Table 1. De Morgan's and Complement laws table

Operation	Changes to
$\neg$ ($Op_1 < Op_2$)	$Op_1 \geq Op_2$
$\neg$ ($Op_1 \leq Op_2$)	$Op_1 > Op_2$
$\neg$ ($Op_1 > Op_2$)	$Op_1 \leq Op_2$
$\neg$ ($Op_1 \geq Op_2$)	$Op_1 < Op_2$
$\neg$ ($Op_1 = Op_2$)	$Op_1 \neq Op_2$
$\neg$ ($Op_1 \neq Op_2$)	$Op_1 = Op_2$

Table 2. Elimination of the NOT operator

		Y					
	$\backslash$	$v < c_2$	$v \leq c_2$	$v = c_2$	$v > c_2$	$v \geq c_2$	$v \neq c_2$
X	$v < c_1$	T	T	$c_2 < c_1$	$c_1 > c_2$	$c_1 > c_2$	T
	$v \leq c_1$	T	T	$c_2 \leq c_1$	$c_2 < c_1$	$c_2 \leq c_1$	T
	$v = c_1$	$c_2 > c_1$	$c_2 \geq c_1$	$c_1 = c_2$	$c_2 < c_1$	$c_2 \leq c_1$	$c_1 \neq c_2$
	$v > c_1$	$c_2 > c_1$	$c_2 > c_1$	$c_2 > c_1$	T	T	T
	$v \geq c_1$	$c_2 > c_1$	$c_2 \geq c_1$	$c_2 \geq c_1$	T	T	T
	$v \neq c_1$	T	T	$c_1 \neq c_2$	T	T	T

Table 3. Propositional variables satisfiability test

[2] In this table, T means *always* satisfiable (**T**autology), v is the referenced concept, and c_i is a valued constant

Filter Identity: As we stated previously, two filters F_1 and F_2 are *identical* (denoted $F_1 \equiv F_2$) iff they match the same set of notifications. Alternatively, we can define filter identity with the function $N(F)$, defined as *the set of all the notifications that match the filter* F. With this definition in mind, two filters F_1 and F_2 are identical iff $N(F_1) = N(F_2)$. Intuitively, one could think of generating these sets of notifications and making a *set equality* test. Nevertheless this approach is computationally not feasible.

To determine the filter *identity*, three approaches can be adopted. The first of them is to check for structural identity. With this approach, two filters are identical iff they have the same object representation. For this purpose, a new *traverser*[3] subclass called `IdentityTraverser` having an `identical(F1,F2)` method was created. It takes two expressions as arguments and determines if they have the same structure.

However, two filters may have totally different structures and still be equivalent, having two structurally different filters, although they *match* exactly the same set of notifications.

This fact led us to the second approach. To generalize the identity comparison, we elaborated the following strategy:

$$F_1 \equiv F_2 \iff N(F_1) = N(F_2)$$

from the point of view of a notification n, it must happen that

$$\forall n/(n \in N(F_1) \iff n \in N(F_2))$$

This expression can be converted to

$$\forall n/(n \in N(F_1) \Rightarrow n \in N(F_2)) \wedge (n \in N(F_2) \Rightarrow n \in N(F_1))$$

equivalently

$$\forall n/(n \notin N(F_1) \vee n \in N(F_2)) \wedge (n \notin N(F_2) \vee n \in N(F_1))$$

then, applying distributive law,

$$\forall n/(n \in N(F_1) \wedge n \in N(F_2)) \vee (n \notin N(F_1) \wedge n \notin N(F_2))$$

this is equivalent to state that

$$\forall n/(n \in N(F_1) \wedge n \in N(F_2)) \vee (n \in N(\neg F_1) \wedge n \in N(\neg F_2))$$

alternatively,

$$\forall n/(n \in N(F_1 \wedge F_2)) \vee (n \in N(\neg F_1 \wedge \neg F_2))$$

or, what is the same,

$$\forall n/(n \in N((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$$

if we change the quantifiers,

$$\neg\exists n/(n \notin N((F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2))$$

or that

$$\neg\exists n/(n \in N(\neg[(F_1 \wedge F_2) \vee (\neg F_1 \wedge \neg F_2)])$$

with the De Morgan's laws, we can convert this into

$$\neg\exists n/(n \in N((\neg F_1 \vee \neg F_2) \wedge (F_1 \vee F_2))$$

transforming it to its DNF, we get

$$\neg\exists n/(n \in N([(\neg F_1 \vee \neg F_2) \wedge F_1] \vee [(\neg F_1 \vee \neg F_2) \wedge F_2])$$

and (finally) cancelling complementary terms

$$\neg\exists n/(n \in N((\neg F_2 \wedge F_1) \vee (\neg F_1 \wedge F_2)).$$

With this in mind, the problem can be reduced to the satisfiability check of the expression

$$(\neg F_2 \wedge F_1) \vee (\neg F_1 \wedge F_2)$$

If it is not satisfiable, then $F_1 \equiv F_2$; otherwise, $F_1 \not\equiv F_2$. The steps would be:

```
1) Create a new filter F as:
```

[3] An `ExpressionTraverser` is a mechanism defined in the `Mix.utils` package to traverse a tree-form expression. Subclasses of it can be implemented to perform new functions while the expression is being traversed

$$(\neg F_2 \wedge F_1) \vee (\neg F_1 \wedge F_2)$$

```
2) Check if F is satisfiable: if it is not, then F1 and F2 are
identical; otherwise, they are not.
```

A third approach is to view the problem in terms of *filter covering* (described in 4.2): $F_1 \equiv F_2$ iff $F_1 \supseteq F_2 \wedge F_2 \supseteq F_1$, as we will see next.

Performing these steps is computationally expensive and they will be executed with every issued subscription. In an open environment, where there many kinds of subscriptions and events are common, it seems to be the case that many of these expression identity comparisons will fail. Given that making out if two expressions are not identical is much less expensive than verifying if they are identical, heuristics could be used. In order to reduce this effort, the approach used to analyze both expressions is to look for a referenced concept (i.e., a `ValueRef`) in one expression missing in the other one. If this is the case, then, we can state that both expressions are not identical. In this environment, performing this pre-analysis will considerably reduce the time required to compare two different expressions.

Filter Overlapping: Formally, a filter F_1 overlaps another filter F_2 (denoted $F_1 \sqcap F_2$) iff $N(F_1) \cap N(F_2) \neq \emptyset$. Thus, we could say that F_1 overlaps F_2 if

$$\exists n / n \in N(F_1) \wedge n \in N(F_2)$$

or equivalently,

$$\exists n / n \in N(F_1 \wedge F_2).$$

With this in mind, the following approach arises:

```
1) Create a new filter F as:
```

$$F_1 \wedge F_2$$

```
2) Check if F is satisfiable: if it is not, then F1 and F2
overlap; otherwise, they do not.
```

We illustrate these steps with the following two examples.

Example 1: (F_1 and F_2 overlap)

$$F_1 = (InTemperature < 10) \wedge (OutTemperature = 30)$$

$$F_2 = (InTemperature > 20) \vee (OutTemperature = 30)$$

– Step 1: Intersecting both filters, F becomes

$$F = [(InTemperature < 10) \wedge (OutTemperature = 30)]$$

$$\wedge[(InTemperature > 20) \vee (OutTemperature = 30)]$$

– Step 2: The disjunctive form of F is:

$$F = \{[(InTemperature < 10) \wedge (OutTemperature = 30)] \wedge (InTemperature > 20)\}$$

$$\vee\{[(InTemperature < 10) \wedge (OutTemperature = 30)] \wedge (OutTemperature = 30)\}$$

– Step 3: Disjunctive term analysis

- The first disjunctive term has a pair (X,Y) of propositional variables that reference the same concept ($InTemperature$). By checking Table 3, we conclude that this pair is not satisfiable (intuitively $InTemperature$ can not be less than 10 and greater than 20 *at the same time*), hence this term is not satisfiable (i.e., it's a contradiction).

- The second disjunctive term has a pair (W,Z) of propositional variables that reference the same concept ($OutTemperature$). By checking Table 3 we realize that this pair is satisfiable. Since there are no more pairs, this term *is* satisfiable, hence the entire expression is satisfiable (recall that it is in its Disjunctive Normal Form). Thus we conclude that F_1 and F_2 overlap.

Example 2: (F_3 and F_4 *don't* overlap)

$$F_3 = [(InTemperature < 10) \wedge (OutTemperature = 30)] \vee (Speedometer \neq 20)$$

$$F_4 = (Speedometer = 20) \wedge (InTemperature > 20)$$

– Step 1: Intersecting both filters, F becomes

$$F = \{[(InTemperature < 10) \wedge (OutTemperature = 30)] \vee$$

$$(Speedometer \neq 20)\} \wedge [(Speedometer = 20) \wedge (InTemperature > 20)]$$

– Step 2: The disjunctive form of F is:

$$F = \{[(InTemperature < 10) \wedge (OutTemperature = 30)] \wedge [(Speedometer = 20) \wedge$$

$$(InTemperature > 20)]\} \vee \{(Speedometer \neq 20) \wedge [(Speedometer = 20) \wedge (InTemperature > 20)]\}$$

– Step 3: Disjunctive term analysis
- The first disjunctive term has a pair (X,Y) of propositional variables that reference the same concept ($InTemperature$). By checking Table 3, we conclude that this pair is not satisfiable, hence the term is not satisfiable.
- In the second disjunctive term there is a pair (W,Z) of propositional variables that reference the same concept ($Speedometer$). By checking Table 3, we conclude that this pair is not satisfiable (intuitively $Speedometer$ can not be equal to 20 and distinct from 20 *at the same time*). Hence, the entire expression is not satisfiable. Thus we conclude that F_3 and F_4 do not overlap.

Filter Covering: As we stated previously, a filter F_1 covers another filter F_2 (denoted $F_1 \sqsupseteq F_2$) iff $N(F_1) \supseteq N(F_2)$. From the point of view of a notification n, it must happen that

$$\forall n/n \in N(F_2) \Rightarrow n \in N(F_1)$$

this is equivalent to state that

$$\neg\exists n/n \in N(F_2) \wedge n \notin N(F_1)$$

or, what is the same,

$$\neg\exists n/n \in N(F_2) \wedge n \in N(\neg F_1)$$

finally,

$$\neg\exists n/n \in N(F_2 \wedge \neg F_1).$$

This means that if a notification n can be found such that $n \in N(F_2 \wedge \neg F_1)$, then F_1 does not cover F_2. With this in mind, the needed steps to test for filter covering could be the following:

```
1) Create a new filter F as:
```

$$F_2 \wedge \neg F_1$$

```
2) Check if F is satisfiable: if it is not, then F1 covers F2;
otherwise, it does not.
```

Filter Merging: Formally, a filter F is a *merger* of (or covers) a set of filters $\{F_1, ..., F_n\}$, denoted $F \sqsupseteq \{F_1, ..., F_n\}$, iff $N(F) \supseteq \{\bigcup_i N(F_i)\}$. F is said to be a *perfect* merger if the equality holds, and an *imperfect* merger otherwise. In practice, the implemented method returns a new filter F which is the disjunction (or union) of two filters F_1 and F_2. This new filter is a perfect merger of F_1 and F_2.

5 Conclusions and Future Work

The expressiveness of the notification selection mechanism used by the consumers to describe their interests is crucial for the flexibility of a notification service. Moreover, the concept-based addressing model enlarges the scope of a notification service by enabling it to pass through institutional, cultural and linguistical boundaries.

The event-driven paradigm is gaining momentum and is being used in distributed loosely-coupled systems. The concept-based approach enhances the scope of use of event notification services by enabling it to cross component, institutional or cultural boundaries. It supports a meaningful exchange of data within a publish/subscribe interaction. The open source notification service REBECA was extended to support the concept-based approach. Details about these extensions were described in this paper.

The object representation used for the described concept-based data model is more complex than the previously used (name-value pairs) data model. Also, the operations needed in order to find a relationship between filters (identity, overlapping, etc.) are computationally more expensive than before. Here we have a clear trade-off between the broker's processing effort and the network traffic. Therefore, this can be seen as a starting point for a detailed analysis of the performance of each routing strategy and its applicability to different environments.

As it was mentioned in this paper notification services concentrate on finding matching notifications according to consumers' interests (or subscriptions). These subscriptions are limited to predicates on notification content but no correlation among notifications can be expressed. This topic is part of our ongoing research.

References

1. C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An infrastructure for meta-auctions. In *Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, San José, California, USA, Jun 2000.
2. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. January 2000.
3. M. Cilia, C. Bornhövd, and A. Buchmann. Moving Active functionality from Centralized to Open Distributed Heterogeneous Environments. In *Proceedings of the 9th IFCIS Conference on Cooperative Information Systems (CoopIS'01)*, volume 2172 of *LNCS*, pages 195–210, Trento, Italy, September 2001. Springer.
4. Mariano Cilia. *An Active Functionality Service for Open Distributed Heterogeneous Environments*. PhD thesis, Darmstadt Univ. of Technology, 2002.
5. L. DeMichiel, L.U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans. Technical Report Version 2.0, Sun Microsystems, JavaSoftware, August 2001.
6. F. Fabret, F. Llirbat, J. Pereira, A. Jacobsen, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. pages 115–126, 2001.
7. Object Management Group. Event service specification. Technical report formal, Famingham, MA., May 1997.
8. Kim Haase. *Java Message Service API Tutorial*. Sun Microsystems, 2002.
9. Gero Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Darmstadt Univ. of Technology, http://elib.tu-darmstadt.de/diss/000274/, 2002.
10. Gero Mühl and Ludger Fiege. The REBECA Notification Service, 2001. http://www.gkec.informatik.tu-darmstadt.de/rebeca/.
11. Gero Mühl, Ludger Fiege, and Alejandro Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *LNCS*, pages 224–238. Springer, 2002.
12. Gero Mühl, Ludger Fiege, Felix C. Gärtner, and Alejandro P. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proc. MASCOTS 2002*, 2002.
13. Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus — an architecture for extensible distributed systems. In Barbara Liskov, editor, *Proceedings of the 14th Symposium of Operating Systems Principles (SIGOPS)*, pages 58–68, Asheville, NC, USA, December 1993. ACM Press.
14. Object Management Group (OMG). Corba notification service specification. Technical report telecom/98-06-15, Object Management Group (OMG), Fammingham, MA, may 1998.
15. Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. Exploiting IP Multicast in Content-based Publish-Subscribe Systems. volume 1795 of *LNCS*, pages 185–207. Springer, 2000.
16. Sun Microsystems, Inc. Java Message Service Specification 1.1, 2002.