3

Content-Based Models and Matching

3.1 Content-Based Data and Filter Models

This section discusses some important content-based data models in conjunction with corresponding filter models. Informally, a *data model* defines how the content of notifications is structured, while a *filter model* defines how subscriptions can be specified, i.e., how notifications can be selected by applying filters that evaluate predicates over the content of notifications. The filter model always depends on the underlying data model, and there can be more than one filter model for a given data model. The data/filter model has to be chosen carefully because it has a large impact on the expressiveness and the scalability of a content-based notification service. In the following, we discuss tuples, structured records, semistructured records, and objects.

3.1.1 Tuples

In tuple-oriented models a notification is a *tuple*, i.e., an *ordered* set of *attributes*. All approaches using tuples deploy some sort of templates as subscription mechanisms. Similarly, to a query-by-example mask, a *template* specifies matching notifications by a partial tuple which can contain wildcards. The attributes in the notification are matched to the attributes in the template according to their position. For example, the notification (*StockQuote*, *"Foo Inc."*, 45) is matched by the subscription template (*StockQuote*, *"Foo Inc."*, $*$). "Matching by position" is inflexible because attributes cannot be optional. Tuples in conjunction with templates were first proposed by Gelernter in work onLinda Tuple spaces [174], which use typed attributes. The original version of Linda, however, did not support a subscription mechanism, but newer approaches based on Tuple spaces, e.g., JavaSpaces [366], do. Also, some notification services are built upon tuples: JEDI models a notification as a tuple of strings [91] in which the first string corresponds to the notification name, while the others are normal attributes. JEDI supports the equality and the prefix operator for matching. Bates et

al. [32] define notifications as instances of classes. An instance consists of a tuple of typed attributes derived from a class definition. Here, a template either specifies the exact value of an attribute or it does not care about the value. Concluding, tuples with templates provide a simple model that is not flexible enough because attributes of notifications and templates are matched to each other according to their position. This disadvantage is diminished by record-oriented models which use "matching by attribute names." However, "matching by position" is more efficient.

3.1.2 Structured Records

In this section structured records are discussed in detail. In a record-oriented model a notification consists of a *named* set of attributes. Record-oriented models can be divided into two categories, which are structured records and semistructured records, respectively. Roughly speaking, the models can be distinguished by the fact that in structured records attribute names are unique, while in the semistructured models several attributes with the same name can exist. In this section, structured records are discussed; semistructured records are discussed in Sect. 3.1.3.

Many systems model notifications similarly to structured records consisting of a set of name/value pairs called attributes. Examples are Siena [65], Gryphon [6, 26], Rebeca [136], JMS [364], and the Corba Notification Service [279]. In this model filters address attributes by their unique names and impose constraints on the values of the respective attributes. In most models a constraint is assumed to evaluate to *false* if the addressed attribute is not contained in the notification. Therefore, each constraint implicitly defines an existential quantifier over the notification. Besides *flat records* in which values are atomic types, *hierarchical records* in which attributes may be nested can also be supported easily by using a dotted naming scheme (e.g., *Position*.x).

Some systems (e.g., Siena) restrict constraints to depend on a single attribute (e.g., $\{x = 1\}$). This class of constraints is called *attribute filters*. Other systems, such as Elvin, allow constraints to evaluate multiple attributes which are combined by operators (e.g., $\{x + y = 5\}$). In general, multiple constraints can be combined to form filters by Boolean operators (e.g., $\{y < 3 \wedge x = 4\}$). Siena and Rebeca restrict filters to be conjunctions of attribute filters. On one hand, this restriction reduces the expressiveness of the filter model, but on the other hand it enables routing optimizations like covering (cf. Chap. 4) to be applied efficiently. The limitation is also not as serious as it seems first. For example, a filter that is defined by an arbitrary Boolean expression can always be converted to and treated as a collection of conjunctive filters.

Although records and tuples seem to be similar at a first glance, records are clearly more powerful because they allow for optional attributes in the notifications. They also avoid unnecessary "don't care" constraints in the templates, and enable the easy addition of new attributes without affecting existing filters.

Data Model

A notification is a message that contains information about an event that has occurred. Formally, a *notification* n is a nonempty set of *attributes* $\{a_1, \ldots, a_n\}$, where each a_i is a *name/value pair* (n_i, V_i) with name n_i and value v_i. It is assumed that names are unique, i.e., $i \neq j \Rightarrow n_i \neq n_j$, and that there exists a function that uniquely maps each n_i to a type T_j that is the type of the corresponding value v_i.

In the following we distinguish between *simple values* that are a single element of the domain of T_j, i.e., $v_i \in dom(T_j)$, and *multi values* that are a finite subset of the domain, i.e., $v_i \subseteq dom(T_j)$. An example of a simple notification is $\{(type, StockQuote), (name, \text{“}Infineon\text{”}), (price, 45.0)\}$.

Filter Model

A *filter* F is a stateless Boolean function that is applied to a notification, i.e., $F(n) \rightarrow \{true, false\}$. A notification *matches* F if $F(n)$ evaluates to *true*. Consequently, the set of matching notifications $N(F)$ is defined as $\{n \mid F(n) = true\}$. Two filters F_1 and F_2 are *identical*, written $F_1 \equiv F_2$, iff $N(F_1) = N(F_2)$. Moreover, they are *overlapping*, denoted by $F_1 \sqcap F_2$, iff $N(F_1) \cap N(F_2) \neq \emptyset$. Otherwise they are *disjoint*, denoted by $F_1 \not\sqcap F_2$.

A filter is usually given as a Boolean expression that consists of predicates that are combined by Boolean operators (e.g., *and*, *or*, *not*). A filter consisting of a single atomic predicate is a *simple filter* or *constraint*. Filters that are derived from simple filters by combining them with Boolean operators are *compound filters*. A compound filter that is a conjunction of simple filters is called a *conjunctive filter*. In the model proposed filters are restricted to be conjunctive filters. It is sufficient to consider conjunctive filters because a compound filter can always be broken up into a set of conjunctive filters that are interpreted disjunctively and can be handled independently.

An *attribute filter* is a simple filter that imposes a constraint on the value of a single attribute (e.g., $\{name = \text{“}Foo\ Inc.\text{”}\}$). It is defined as a triple $A_i = (n_i, Op_i, C_i)$, where n_i is an attribute name, Op_i is a test operator and C_i is a set of constants that may be empty. The name n_i determines to which attribute the constraint applies. If the notification does not contain an attribute with name n_i then A_i evaluates to *false*. Therefore, each constraint implicitly defines an existential quantifier over the notification. Otherwise, the operator Op_i is evaluated using the value of the addressed attribute and the specified set of constants C_i. It is assumed that the types of operands are compatible with the used operator. The outcome of A_i is defined as the result of Op_i that evaluates either to *true* or *false*. Furthermore, an attribute filter is provided that simply checks whether a given attribute is contained in n. For the sake of simplicity the more readable notation $\{price > 10\}$ is used instead of $\{(price, >, \{10\})\}$. In contrast to most other work (e.g.,)Siena, constraints that depend on more than one constant are considered in this chapter. This

enables more operators and enhances the expressiveness of the filtering model and can be done without affecting scalability.

By $L_A(A_i) \subseteq dom(T_k)$ the set of all values is denoted that cause an attribute filter to match an attribute, i.e., $\{v_i \mid Op_i(v_i, C_i) = true\}$. It is assumed that $L_A(A_i) \neq \emptyset$. An attribute filter A_1 *covers* an attribute filter A_2, written $A_1 \sqsupseteq A_2$, iff $n_1 = n_2 \ \wedge \ L_A(A_1) \supseteq L_A(A_2)$. For example, $\{price > 10\}$ covers $\{price \in [20, 30]\}$. A_1 and A_2 are *identical*, denoted by $A_1 \equiv A_2$, iff $n_1 = n_2 \ \wedge \ L_A(A_1) = L_A(A_2)$. A_1 and A_2 are *overlapping* iff $n_1 = n_2 \ \wedge \ L_A(A_1) \cap L_A(A_2) \neq \emptyset$, denoted by $A_1 \sqcap A_2$. Otherwise they are *disjoint*, denoted by $A_1 \not\sqcap A_2$. For example, $\{price > 10\}$ and $\{price < 20\}$ are overlapping, while $\{price < 10\}$ and $\{price > 20\}$ are disjoint.

In the described model a *filter* is defined as a conjunction of attribute filters, i.e., $F = A_1 \wedge \ldots \wedge A_n$. To enable efficient evaluation of routing optimizations like covering and merging, at most one attribute filter for each attribute is allowed. A notification n *matches a filter* F iff it satisfies all attribute filters of F. Moreover, a filter with an empty set of attribute filters matches any notification. An example for a conjunctive filter consisting of attribute filters is $\{(type = StockQuote), (name = \text{“Foo Inc.”}), (price \notin [30, 40])\}$.

The limitation to at most one attribute filter for each attribute is not as serious as it seems at first glance because the proposed model provides complex data types as attribute values and an extensible set of constraints that can be imposed. Moreover, it is often possible to merge several conjunctive constraints imposed on a single attribute into a single constraint on the same attribute. Especially suited for this kind of merging are constraints which are either contradicting (if they are conjuncted) or can be replaced by a single constraint of the same type. Such types of constraints and their corresponding attribute filters are called *conjunction-complete*. For example, interval constraints and constraints testing whether a point is in a given rectangle in a two-dimensional plane are conjunction-complete. As an example, $\{x \in [3, 7] \ \wedge \ x \in [5, 8]\}$ can be substituted by $\{x \in [5, 7]\}$. If a constraint type is not conjunction-complete it is often possible to substitute a set of such constraints by a single constraint of a more general type. For example, a set of ordering constraints defined on a totally ordered set (e.g., integer numbers) are either contradictory or can be replaced by a single interval constraint. As an example, $\{x \geq 3 \ \wedge \ x \leq 5\}$ can be merged to $\{x \in [3, 5]\}$.

Subscriptions and *advertisements* are simply filters that are issued by consumers and producers of notifications, respectively. There is no difference in their model, and hence, subscriptions and advertisements are the exact dual of each other. This is in contrast to Siena, where subscriptions and advertisements are not exactly complementary, raising a number of problems.

Generic Constraints and Types

Earlier work dealing with content-based notification selection mechanisms often tightly integrated the constraints that can be put on values and the types

of values supported by the matching and the routing algorithms [6, 26]. An exception is SIENA, where matching and routing algorithms are separated from constraints. However, SIENA only supports a fixed set of constraints on some predefined primitive types.

We propose to use a collection of abstract attribute filter classes. Each of these classes offers a generic implementation of the methods needed by the matching and the routing algorithms (e.g., a covering and a matching test) and imposes a certain type of constraint on an attribute that can be used with values of all types that implement the operators needed. The appropriate implementation of the operators is called by the constraint class at runtime using polymorphism. This enables new constraints and types to be defined and to be supported without requiring changes to the routing and or to the matching algorithms. Note that although an object-oriented approach is suggested, it is not mandatory to use it.

For example, a constraint class can realize comparison constraints on totally ordered sets. This class can be used to impose comparison constraints on all kinds of ordered values (e.g., integer numbers). Consider a type "person" that consists of first and second name, the date of birth, and the place of birth. This type is easily supported by providing implementations for the comparison operators which are called by the constraint class to provide the covering and matching methods using polymorphism.

In the following subsections, some generic attribute constraints are presented that cover a wide range of practically relevant constraints, but more important, they illustrate the feasibility of the approach. Of course, this collection is not exhaustive, but other constraints can be integrated easily. For example, intervals could be used as values. In this case the same operators as for set constraints can be used because intervals are essentially sets. The investigation of a subset of regular expressions seems to be promising, too. Most paragraphs also present a table that gives an overview of covering implication dealing with the discussed type of constraint. The meaning of a single row in the Tables 3.1 through 3.7 is: Given A_1 and A_2 as specified in column 1 and 2, $A_1 \sqsupseteq A_2$ iff the condition in column 3 is satisfied. In order to test whether a filter *covers* another, covering must hold for all attributes, as will be shown later.

General Constraints

Two general constraints are considered that can be imposed on all attributes regardless of the type of their value: *exists*(n) tests whether an attribute with name n is contained in a given notification, i.e., whether $\exists A_i.\, n_i = n$. The *exists* constraint covers all other constraints that can be imposed on an attribute.

Constraints on the Type of Notifications

Most work on notification services has a notion of types or classes of notifications. Usually, the type of a notification is specified by a textual string

that can be tested for equality and prefix. If a dot notation is used, a type hierarchy with single inheritance can be supported, allowing for the automatic propagation of interest in subclasses [32]. Unfortunately, multiple inheritance cannot be supported by a dotted naming scheme. In contrast to that, a direct support of notification types has a number of advantages. Such an approach can enable multiple inheritance and achieve a better programming language integration [120]. Moreover, type inclusion tests can be evaluated more efficiently than the corresponding string operation (i.e., whether the string starts with a given prefix) [388].

Consequently, a separate constraint that evaluates to *true* if n is an instance of type T and *false* otherwise, written n *instanceof* T, is defined. A constraint n *instanceof* T_1 covers a constraint n *instanceof* T_2 iff T_1 is either the same type or a supertype of T_2 (Table 3.1). It is assumed that the set of attributes that can be contained in a notification of type T is a superset of the union of all attribute names of all supertypes of T.

Table 3.1. Covering among notification types

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
n *instanceof* T_1	n *instanceof* T_2	$T_1 = T_2 \ \vee \ T_1$ *supertype of* T_2

Equality and Inequality Constraints on Simple Values

The simplest constraints that can be imposed on a value are tests for equality and inequality. Covering implications among these tests can always be reduced to a simple comparison of their respective constants (Table 3.2).

Table 3.2. Covering among (in)equality constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
$x = c_1$	$x = c_2$	$c_1 = c_2$
$x \neq c_1$	$x = c_2$	$c_1 \neq c_2$
	$x \neq c_2$	$c_1 = c_2$

Comparison Constraints on Simple Values

Another common class of constraints are comparisons on values for which the domain and the comparison operators define a totally ordered set (e.g., integers with the usual comparison operators). Again, covering among these tests can be reduced to a simple comparison of their respective constants. Table 3.3 depicts covering implications of inequality and greater than; for brevity the other comparison operators are omitted.

Table 3.3. Covering among comparison constraint on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
$x \neq c_1$	$x < c_2$	$c_1 \geq c_2$
	$x \leq c_2$	$c_1 > c_2$
	$x = c_2$	$c_1 \neq c_2$
	$x \geq c_2$	$c_1 < c_2$
	$x > c_2$	$c_1 \leq c_2$
$x > c_1$	$x = c_2$	$c_1 < c_2$
	$x > c_2$	$c_1 \leq c_2$
	$x \geq c_2$	$c_1 < c_2$

Interval Constraints on Simple Values

Interval constraints test whether a value x is within a given interval I or not, i.e., $x \in I$ and $x \notin I$, respectively, where I is a closed interval $[c_1, c_2]$ with $c_1 \leq c_2$. Here, computing coverage involves two comparisons (Table 3.4).

Table 3.4. Covering among interval constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
$x \in I_1$	$x \in I_2$	$I_1 \supseteq I_2$
$x \notin I_1$	$x \notin I_2$	$I_1 \subseteq I_2$

Constraints on Strings

Constraints on strings can be used to realize subjects. In addition to the comparison operators based on the lexical order, a prefix, a substring, and a postfix operator are defined. *s hasPrefix S* and *s hasPostfix S* mean that *s* has the prefix and the postfix *S*, respectively. *s containsSubstring* S_1 means that *s* contains the substring S_1. Computing coverage among them requires a single test (Table 3.5).

Table 3.5. Covering among constraints on strings

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
s hasPrefix S_1	*s hasPrefix* S_2	S_2 *hasPrefix* S_1
s hasPostfix S_1	*s hasPostfix* S_2	S_2 *hasPostfix* S_1
s hasSubstring S_1	*s hasSubstring* S_2	S_2 *hasSubstring* S_1

Set Constraints on Simple Values

Set constraints on simple values test whether or not a value is a member of a given set. For computing coverage among two of these constraints, a single set inclusion test is sufficient (Table 3.6). Its complexity depends on the characteristics of the underlying set. Set constraints can be combined with comparison constraints if the domain of the value is a totally ordered set.

Table 3.6. Covering among set constraints on simple values

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
$x \in M_1$	$x \in M_2$	$M_1 \supseteq M_2$
$x \notin M_1$	$x \notin M_2$	$M_1 \subseteq M_2$

Set Constraints on Multi Values

The idea of multi values is to allow a value to be a set of elements. This enables set-oriented operators which are defined on a multi value $X = \{v_1, \ldots, v_n\}$. For example, the following common operators can be defined:

$$\begin{aligned} X \ \textit{subset} \ M &\Leftrightarrow X \subseteq M \\ X \ \textit{superset} \ M &\Leftrightarrow X \supseteq M \\ X \ \textit{contains} \ a_1 &\Leftrightarrow a_1 \in X \\ X \ \textit{notcontains} \ a_1 &\Leftrightarrow a_1 \notin X \\ X \ \textit{disjunct} \ M &\Leftrightarrow X \cap M = \emptyset \\ X \ \textit{overlaps} \ M &\Leftrightarrow X \cap M \neq \emptyset \end{aligned}$$

To determine covering with respect to these constraints either the evaluation of a set inclusion test or of a set membership test is needed (Table 3.7).

Table 3.7. Covering among set constraints on multi values

A_1	A_2	$A_1 \sqsupseteq A_2$ iff
X *subset* M_1	X *subset* M_2	M_1 *superset* M_2
X *contains* a_1	X *superset* M_2	$a_1 \in M_2$
X *superset* M_1	X *superset* M_2	M_1 *subset* M_2
X *notContains* a_1	X *disjunct* M_2	$a_1 \in M_2$
X *disjunct* M_1	X *disjunct* M_2	M_1 *subset* M_2
X *overlaps* M_1	X *overlaps* M_2	M_1 *superset* M_2

Support for Routing Optimizations

For routing algorithm such as identity-based, covering-based, or merging-based routing (cf. Chap. 4) as well as for enabling the use of advertisement, some routing optimization must be efficiently computable.

Identity of Conjunctive Filters

In the following it is shown how identity of conjunctive filters can be reduced to the respective attribute filters. An identity test among filters is necessary to implement identity-based routing.

Lemma 3.1. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters, the following holds: the fact that* F_1 *and* F_2 *contain the same number of attribute filters and that* $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$ *implies that* F_1 *and* F_2 *are identical.*

Proof. The proof is rather trivial. A notification that matches F_1 satisfies all attribute filters A_i^1. For each of these A_i^1 there is an identical A_j^2. Hence, A_j^2 is matched, too. As F_1 and F_2 contain the same number of attribute filters, this implies that all attribute filters of F_2 are matched, too. Therefore, F_2 is also matched. As the same argumentation can be applied to notifications that match F_2, this implies that F_1 and F_2 match identical sets of notifications, i.e., they are identical. □

It is necessary to restrict filters to contain at most one attribute filter for each attribute in order to strengthen Lemma 3.1 to an equivalence. As a simple example, $\{x > 5 \ \wedge \ x < 5\}$ is identical to $\{x \neq 5\}$, although neither $\{x > 5\} \equiv \{x \neq 5\}$ nor $\{x < 5\} \equiv \{x \neq 5\}$.

Lemma 3.2. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds:* $F_1 \equiv F_2$ *implies* $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$.

Proof. The proof is by contradiction. We assume that

1. $F_1 \equiv F_2$
2. $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$ does not hold

and prove that this cannot hold.

The second assumption implies that there is an A_i^1 for which no identical A_j^2 exists. This means that either no attribute filter with the same name is contained in F_2 or that $L(A_i^1) \neq L(A_j^2)$. In the first case, a notification can be constructed that does not contain the respective attribute and which matches F_2 but does not match F_1. Hence, F_1 and F_2 cannot be identical and the first assumption is violated. In the second case, a notification can be constructed, where the value of the respective attribute is in $L(A_i^1)$ but not in $L(A_j^2)$ if $L(A_i^1) \supset L(A_j^2)$. This notification matches F_1 but not F_2. The other way

around, a notification can be constructed, where the value of the respective attribute is in $L(A_j^2)$ but not in $L(A_2^j)$ if $L(A_i^1) \subset L(A_j^2)$. This notification matches F_2 but not F_1. At least one of these two cases needs to occur because $L(A_i^1) \neq L(A_j^2)$. Hence, F_1 and F_2 cannot be identical and the first assumption is violated. The above cases cover all possible cases. □

Lemma 3.3. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds:* $F_1 \equiv F_2$ *implies that* F_1 *and* F_2 *contain the same number of attribute filters.*

Proof. By Lemma 3.2 and the fact the identity relation among filters is symmetrical. □

Corollary 3.1. *Two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute are identical iff they contain the same number of attribute filters and* $\forall A_i^1 \exists A_j^2.\ A_i^1 \equiv A_j^2$.

Proof. By Lemmas 3.1, 3.2, and 3.3. □

The above corollary essentially states that two filters are identical iff they constrain the same attributes and iff the attribute filters of each constrained attribute are pairwise identical (Fig. 3.1).

$$
\begin{array}{ccccc}
F_1 = & \{x \geq 2\} & \wedge & \{y > 5\} \\
| & | & & | \\
\equiv & \equiv & & \equiv \\
| & | & & | \\
F_2 = & \{x \geq 2\} & \wedge & \{y > 5\}
\end{array}
$$

Fig. 3.1. Identity of filters consisting of attribute filters

Covering of Conjunctive Filters

In the following it is shown how covering of conjunctive filters can be reduced to the respective attribute filters. A covering test among filters is necessary to implement covering-based routing.

Lemma 3.4. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters, the following holds:* $\forall i \exists j.\ A_i^1 \sqsupseteq A_j^2$ *implies* $F_1 \sqsupseteq F_2$.

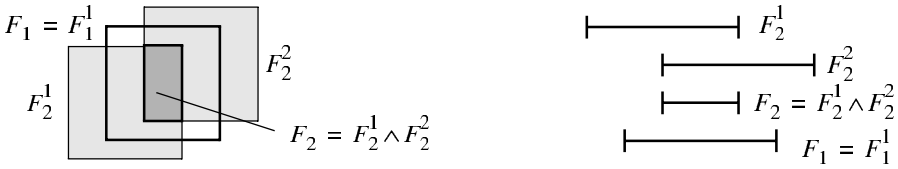

Fig. 3.2. $F_1 \sqsupseteq F_2$ although neither $F_1^1 \sqsupseteq F_2^1$ nor $F_1^1 \sqsupseteq F_2^2$ (two examples)

Proof. Assume $\forall i \exists j.\, A_i^1 \sqsupseteq A_j^2$. Prove $F_1 \sqsupseteq F_2$. If an arbitrary notification n is matched by F_2 then n satisfies all A_2^j. This fact together with the assumption implies that n also satisfies all A_1^i. Therefore, n is matched by F_1, too. Hence, $F_1 \sqsupseteq F_2$. □

If several attribute filters can be imposed on the same attribute then $\forall i \exists j. A_i^1 \sqsupseteq A_j^2$ is not a necessary condition for $F_1 \sqsupseteq F_2$ (Fig. 3.2). For example, $\{x \in [5, 8]\}$ covers $\{x \in [4, 7] \wedge x \in [6, 9]\}$, although $\{x \in [5, 8]\}$ covers neither $\{x \in [4, 7]$ nor $\{x \in [6, 9]\}$. If conjunctive filters are restricted to have at most one attribute filter for each attribute, then Lemma 3.4 can be strengthened to an equivalence:

Lemma 3.5. *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds: $F_1 \sqsupseteq F_2$ implies $\forall i \exists j.\, A_i^1 \sqsupseteq A_j^2$.*

Proof. Assume $\neg(\forall i \exists j.\, A_i^1 \sqsupseteq A_j^2)$. Prove $\neg(F_1 \sqsupseteq F_2)$. A notification n is constructed that matches F_2 but not F_1 to prove that F_1 does not cover F_2. The assumption implies that there is at least one A_k^1 that does not cover any A_j^2. If there exists an A_l^2 that constrains the same attribute as such an A_k^1 then choose for this attribute a value that matches A_l^2 but not A_k^1. Such a value exists because $L_A(A_k^1) \neq \emptyset$ and $A_k^1 \not\sqsupseteq A_l^2$. Add name/value pairs for all other attributes that are constrained in F_2 such that they are matched by the appropriate attribute filters of F_2. The constructed notification matches F_2 but not F_1. Therefore, F_1 does not cover F_2. □

Corollary 3.2. *Given two filters $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ and $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ that are conjunctions of attribute filters with at most one attribute filter per attribute, the following holds: $F_1 \sqsupseteq F_2$ is equivalent to $\forall i \exists j.\, A_i^1 \sqsupseteq A_j^2$.*

Proof. By Lemmas 3.4 and 3.5. □

The above corollary essentially states that a filter F_1 covers a filter F_2 iff for each attribute filter in F_1 there is an attribute filter in F_2 that is covered by the former (Fig. 3.3).

$$
\begin{array}{ccccc}
F_1 = & \{x \geq 2\} & \wedge & \{y > 5\} & \\
| & | & & | & \\
\sqsupseteq & \sqsupseteq & & \sqsupseteq & \\
| & | & & | & \\
F_2 = & \{x = 4\} & \wedge & \{y = 7\} & \wedge \{z \in [3,5]\}
\end{array}
$$

Fig. 3.3. Covering of filters consisting of attribute filters

Overlapping of Conjunctive Filters

In the following it is shown how overlapping of conjunctive filters can be reduced to the respective attribute filters. An overlapping test among filters is necessary to use advertisements for routing optimizations.

Lemma 3.6. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters,* $\exists A_i^1, A_j^2. \left(n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\right)$ *implies that* F_1 *and* F_2 *are disjoint.*

Proof. Proof: Suppose that F_1 and F_2 contain attribute filters A_i^1 and A_j^2 such that $\left(n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\right)$. This means that both filters require the existence of an attribute with name n_i^1 and that the value of this attribute must match $L_A(A_i^1)$ in order to make a notification match F_1 and $L_A(A_j^2)$ in order to match F_2. As $L_A(A_i^1)$ and are $L_A(A_j^2)$ disjoint, this implies that a given notification can be matched either by F_1 or by F_2. Hence, F_1 and F_2 are disjoint. □

It is necessary to restrict filters to contain at most one attribute filter for each attribute in order to strengthen Lemma 3.6 to an equivalence. As a simple example, $\{x \in \{3,5\} \ \wedge \ x \in \{4,5\}\}$ is disjoint with $\{x \in \{3,5\} \ \wedge \ x \in \{3,4\}\}$ although there are no disjoint attribute filters.

Lemma 3.7. *Given two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute, the fact that* F_1 *and* F_2 *are disjoint implies that* $\exists A_i^1, A_j^2. \left(n_i^1 = n_j^2 \ \wedge \ L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\right)$.

Proof. Proof: The proof is by contradiction. Suppose that F_1 and F_2 are disjoint and that there are no A_i^1, A_j^2 such that $n_i^1 = n_j^2 \wedge L_A(A_i^1) \cap L_A(A_j^2) = \emptyset$. We construct a notification that matches F_1 and F_2 to imply a contradiction in following way: For each attribute that is constrained in F_1 or F_2 add an attribute whose value satisfies the attribute filters contained in F_1 and F_2 regarding this attribute. This value must exist because there are no A_i^1, A_j^2 such that $n_i^1 = n_j^2 \wedge L_A(A_i^1) \cap L_A(A_j^2) = \emptyset$. Hence, the constructed notification matches F_1 and F_2, and therefore F_1 and F_2 are not disjoint. □

Corollary 3.3. *Two filters* $F_1 = A_1^1 \wedge \ldots \wedge A_n^1$ *and* $F_2 = A_1^2 \wedge \ldots \wedge A_m^2$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute are disjoint, i.e., not overlapping, iff* $\exists A_i^1, A_j^2. \left(n_i^1 = n_j^2 \wedge L_A(A_i^1) \cap L_A(A_j^2) = \emptyset\right)$.

Proof. By Lemmas 3.6 and 3.7. □

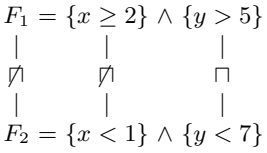

Fig. 3.4. Disjoint filters consisting of attribute filters

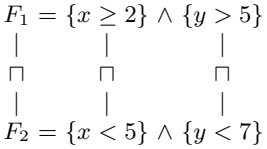

Fig. 3.5. Overlapping filters consisting of attribute filters

The above corollary essentially states that two filters are disjoint iff for an attribute that is constrained in both filters the corresponding attribute filters are disjoint (Fig. 3.4). Hence, two filters are overlapping iff no such attribute filters exist (Fig. 3.5).

Merging of Conjunctive Filters

Merging-based routing algorithms use abstract merging operations. In this section merging of conjunctive filters is discussed. The aim of filter merging is to determine a filter that is a merger of a set of filters. Merging of filters can be used to drastically reduce the number of subscriptions and advertisements that have to be stored by the brokers.

Perfect Merging

A set of conjunctive filters with at most one attribute filter for each attribute can be perfectly merged into a single conjunctive filter if, for all except a

single attribute, their corresponding attribute filters are identical and if the attribute filters of the distinguishing attribute can be merged into a single attribute filter. For example, the two filters $F_1 = \{x = 5 \wedge y \in \{2, 3\}\}$ and $F_2 = \{x = 5 \wedge y \in \{4, 5\}\}$ can be merged to $F = \{x = 5 \wedge y \in \{2, 3, 4, 5\}\}$. Moreover, a set of attribute filters imposed on the same attribute with name n can be merged to an *exists*(n) test if at least one of them is satisfied by any value. Note that an existence test is equivalent to no constraint if the attribute is mandatory for the corresponding type of notification.

An algorithm that determines the possibly empty set of filters which are candidates to be merged with a given filter is depicted later. From the set of merging candidates the set of attribute filters to be merged can easily be extracted. This set is used as input of a merging algorithm which has a specialized implementation for each type of constraint. In the general case purely algebraic merging techniques have exponential time complexity. Alternatively, a predicate proximity graph can be used to implement a greedy algorithm [218]. For many practical cases (e.g., set operators) efficient algorithms exist. Only in rare cases is it necessary to use an exhaustive combinatorial or a suboptimal greedy algorithm.

The characteristics of the constraints that are used to define attribute filters are important for merging. Constraints which only exist in a normal and a negated form can be directly merged by using some basic laws of Boolean algebra. For example, the filters $F_1 = (y = 3 \wedge x = 5)$ and $F_1 = (y = 3 \wedge x \neq 5)$ can be merged to $F = (y = 3 \wedge \exists x)$. In general, constraints are not restricted to be the negated form of each other, and hence better merging can be achieved by taking the specific characteristics of the imposed constraints into account.

A class of constraints that is *complete under disjunction* allows a set of constraints of this class to be merged into a single constraint of the same class. Examples for disjunction-complete constraints are *set inclusions* (e.g., $x \in \{2, 3, 7\}$) and *set exclusions* (e.g., $x \notin \{2, 3, 7\}$) while *comparison constraints* (e.g., $x < 4$) are not disjunction-complete. If a constraint class is not disjunction-complete it may still be possible to carry out merging if a specific *merging condition* is met. For example, a set of *interval tests* (e.g., $x \in [2, 4]$ and $x \in [3, 5]$) can be merged into a single interval test (here, $x \in [2, 5]$) if the intervals form a connected set. Otherwise, merging may be possible if a more general constraint is considered as merging result. For example, two comparison constraints (e.g., $x < 4$ and $x > 7$) can be merged to an interval test (here, $x \notin [4, 7]$).

Merging on the level of attribute filters is implemented by each generic attribute filter class. Table 3.8 presents some perfect merging rules. The meaning of a single row is that A_1 and A_2 can be perfectly merged to the indicated merger (column 4) if the given merging condition (column 3) holds. The first two rules can also be applied to equality and inequality tests because $x = a_1 \Leftrightarrow x \in \{a_1\}$ and $x \neq a_1 \Leftrightarrow x \notin \{a_1\}$.

Table 3.8. Perfect merging rules for attribute filters

A_1	A_2	Condition	$A_1 \cup A_2$
$x \in M_1$	$x \in M_2$	-	$x \in M_1 \cup M_2$
$x \notin M_1$	$x \notin M_2$	$M_1 \cap M_2 = \emptyset$	$\exists x$
		$M_1 \cap M_2 \neq \emptyset$	$x \notin M_1 \cap M_2$
X *overlaps* M_1	X *overlaps* M_2	-	X *overlaps* $M_1 \cup M_2$
X *disjunct* M_1	X *disjunct* M_2	$M_1 \cap M_2 = \emptyset$	$\exists X$
		$M_1 \cap M_2 \neq \emptyset$	X *disjunct* $M_1 \cap M_2$
$x = a_1$	$x \neq a_1$	$a_1 = a_2$	$\exists x$
$x < a_1$	$x > a_2$	$a_1 > a_2$	$\exists x$
	$x \geq a_2$	$a_1 \geq a_2$	
$x \leq a_1$	$x > a_2$	$a_1 \geq a_2$	$\exists x$
	$x \geq a_2$		

Imperfect Merging

At a first glance, imperfect merging seems to be less promising, but in situations in which perfect merging is either too complex or not computable it is a good compromise. Clearly, there exists a trade-off between filtering overhead and network resource consumption. Imperfect merging may result in notifications being forwarded that do not match any of the original subscriptions, but on the other hand, it reduces the number of subscriptions and advertisements that must be dealt with.

In order to use imperfect merging, heuristics are necessary that define in what situations and to what degree imperfect merging should be carried out. For example, filters that differ in few attribute filters could be merged imperfectly by imposing on each attribute a constraint that covers all original constraints. In order to decide whether two given filters should be merged a heuristic that allows the amount of introduced imperfection to be estimated is needed. This could also be accomplished by explicitly replacing an attribute filter with another that only tests for the existence of the given attribute or by simply dropping the attribute filter. Statistical online evaluation of filter selectivity would be also a good basis for merging decisions that enables adaptive filtering strategies. Imperfect merging requires further investigation.

Algorithms

In this section algorithms are presented that are superior to the naïve algorithms (cf. Sect. 3.2.1). The presented algorithms use the generic approach presented in the previous section: Each generic constraint class (e.g., constraints on ordered values) offers specialized indexing data structures to efficiently manage constraints on attributes. For example, hashing is used for equality tests. In the following, algorithms for matching, covering, and for

detecting merging candidates are described that are all based on the predicate counting algorithm (cf. Sect. 3.2.2). Algorithms for detecting identity and overlapping among filters can be derived similarly.

Matching Algorithm

The naïve algorithm separately matches a given notification against all filters to determine the set of matched filters. This implies that the same attribute filter may be evaluated many times. More advanced algorithms avoid this. Some of these require a costly compilation step (e.g., [181]) that makes them less suitable for publish/subscribe systems in which subscriptions change dynamically. In contrast to that, the algorithm presented here allows filters to be added or removed at any time. The algorithm is based on the idea of *predicate counting* [305, 404] and makes use of our generic approach. The algorithm is depicted in Fig. 3.6. It determines all filters that match a given notification.

```
1  Matching Algorithm
   Input: notification n, set of filters F
   Output: the set M of all filters in F that match n.
   {
      <For each filter in F a counter is initialized to zero.>
6     for <each A_i contained in n> {
         for <each filter S in F that has a constraint on A_i that
               is satisfied by the value of the corresponding
               attribute of n> {
            <Increment the counter of S>
11       }
      }
      M:=<all filters in F whose counter is equal to their
            number of attribute filters>
   }
```

Fig. 3.6. Matching algorithm based on counting satisfied attribute filters

Covering Algorithm

Covering-based routing is built upon two tests: a first test that determines all filters that cover a given filter, and a second one determines all filters that are covered by a given filter. The naïve implementation simply tests each filter against all others sequentially. The algorithms presented here are more efficient. They are derived from the matching algorithm presented above (Figs. 3.7 and 3.8).

```
Covering Algorithm I
Input: filter F1, set of filters F
Output: the set C of all filters in F that cover F1.
{
   <For each filter in F a counter is initialized to zero.>
   for <each Ai contained in F1> {
      for <each filter S in F that has a constraint Aj that
            covers Ai> {
        <Increment the counter of S>
      }
   }
   C:=<all filters in F whose counter is equal to their
          number of attribute filters>
}
```

Fig. 3.7. Covering algorithm that determines all *covering* filters

```
Covering Algorithm II
Input: filter F1, set of filters F
Output: the set C of all filters in F that are covered by F1.
{
   <For each filter in F a counter is initialized to zero.>
   for <each Ai contained in F1> {
      for <each filter S in F that has a constraint Aj that
      is covered by Ai> {
        <Increment the counter of S>
      }
   }
   C:=<all filters in F whose counter is equal to the
          number of attribute filters of F1>
}
```

Fig. 3.8. Covering algorithm that determines all *covered* filters

Merging Algorithm

We present an algorithm that determines all possible merging candidates. These are those filters that are identical to a given filter in all but a single attribute. The algorithm avoids testing all filters against all others. It counts the number of identical attribute filters to find merging candidates (Fig. 3.9).

The further handling of the set of merging candidates depends on the constraints involved. For all constraints discussed (e.g., set constraints on simple values) there exists an efficient algorithm which outputs a single merged filter and a set of filters not included in the merger. For other constraints, an optimal algorithm requires exponential time complexity [87]. In this case the

use of greedy algorithms or heuristics (e.g., using a predicate proximity graph) seems to be promising.

```
Merging Algorithm
Input: filter F_1, set of filters F
Output: set M of all merging candidates
{
   <For each filter in F a counter is initialized to zero.>
   for <each A_i contained in F_1> {
      for <each filter S in F that has a constraint A_j that
      is identical to A_i> {
         <Increment the counter of S>
      }
   }
   M:=<all filters in F whose counter is one smaller than or
         equal to their number of attribute filters>
}
```

Fig. 3.9. Merging algorithm based on counting identical attribute filters

3.1.3 Semistructured Records

In the previous section structured records have been discussed in detail. In this section a model for semistructured records is presented. The structured and the semistructured model are mainly distinguished by the following fact: In the structured model attribute names are unique, and hence an attribute name uniquely addresses a single attribute. On the contrary, in the semistructured model sibling attributes can have the same name, and therefore names address sets of attributes.

In the following, a model for semistructured records is presented in which notifications are essentially XML [399] documents. The filtering mechanisms are similar to but less powerful than XPath [398]. After the model has been introduced, how routing optimizations can be achieved is discussed.

According to Bunemann [55] semistructured data can be characterized as some kind of graphlike or treelike structure that is often called *self-describing* because the schema of the data is contained in the data itself. At the moment, the most prominent semistructured data model is XML [399]. Similarly, to structured records, a semistructured record is a set of nested attributes, but in contrast to structured records, in semistructured records sibling attributes can have the same name. In consequence, a single attribute can no longer be uniquely addressed by its name alone. Instead, names (e.g., *car.price*), which are usually called *paths* in this context, select sets of attributes. Therefore, filtering strategies assuming that a single attribute is addressed by a given

name cannot directly be used in this scenario. One way to approach this problem is to use path expressions (e.g., XPath [398]), which select a set of attributes and impose constraints on the selected attributes.

Clearly, the semistructured model is more powerful than structured records, but work in this area related to content-based routing is still in its early stages. Lately, using XML and path expressions has gained increased attention. Nguyen et al. [271] and Chen et al. [77] described approaches for XML continuous queries. Altinel and Franklin [12] presented an efficient method for filtering XML documents using XPath expressions. All this work concentrates on efficient local matching and does not deal with distributed content-based routing. First ideas on how to support routing optimizations like covering and merging for semistructured records was presented by Mühl and Fiege [264]. These ideas are discussed later in this section.

Data Model

In the semistructured data model a *notification* is a well-formed XML document [399] and consists of a set of elements that are arranged in a hierarchy with a single root element uniquely named "notification". Each *element* consists of a set of attributes whose names must be distinct and a set of subordinate *child elements*, which are named but whose names must not necessarily be distinct. An *attribute* A is a pair (n_i, v_i) with name n_i and value v_i. Names of attributes must be unique with respect to elements. A simple notification that describes an auction is shown in Fig 3.10. In this example, the element *auction* has two subelements that are named *item*. Furthermore, the element *cpu* contains an attribute *clock* whose value is 800. Note that XML documents can contain free text between the opening and the closing tag of an element. Here, this text is simply ignored.

Filter Model

In the semistructured filter model a filter is a conjunction of path filters. Each of the path filters selects a subset of the elements in a notification by an element selector and places constraints on the attributes of the selected elements by an element filter, which consists of a set of attribute filters. In the following, this model is described in full detail.

An *element selector* selects a subset of the elements of a notification and is specified by an attribute path. It is distinguished between absolute and abbreviated paths. An *absolute path* is a slash-separated string that starts with a single slash (e.g., */notification/auction*). An *abbreviated path* is a slash-separated string that starts with two slashes (e.g., *//cpu*). An absolute (abbreviated) path selects all elements whose path is equal to (ends with) the given path. For example, *//item* selects both *item* elements of the notification in Fig. 3.10.

```
<notification>
  <auction
    endtime="05/18/02 22:17:42"
    minprice="50">
    <seller
      name="Smith"
      id="1234"/>
    <item>
      <board
        manufacturer="Elitegroup"
        type="K7S5"
        socket="Socket A"/>
    </item>
    <item>
      <cpu
        manufacturer="AMD"
        type="Athlon"
        socket="Socket A"
        clock="800"/>
    </item>
  </auction>
</notification>
```

Fig. 3.10. A simple notification

An *attribute filter* is a pair $A = (n, Q)$ consisting of a name n (e.g., *manufacturer*) and a constraint Q (e.g., $=$ *"AMD"*). An element *matches an attribute filter* if the element contains an attribute with name n whose value v satisfies Q, e.g. (*manufacturer*, *"AMD"*). This means that an attribute filter evaluates to false if the element does not contain an attribute with name n. Therefore, an attribute filter implicitly defines an existential quantifier over an element.

An *element filter* C is a conjunction of a nonempty set A of attribute filters $\{A_1, \ldots, A_i\}$, i.e., $C = \wedge_i A_i$. Hence, an element *matches an element filter* iff all attribute filters are satisfied. An example of an element filter based on the syntax of XPath is [*@manufacturer* $=$ *"AMD"* $\wedge$ *@clock* ≥ 700]. Note that in this notation attribute names are prefixed by an "@".

A *path filter* $P = (S, C)$ consists of an element selector S and an element filter C. A notification n *matches a path filter* P if at least one element of n is selected by S that matches C. It is possible to extend this model in such a way that an interval constraint can be imposed on both the number of elements that match an element filter and the number of elements that must not match. These extensions are not discussed for brevity. An example of a complete path filter based on an absolute path is: */notification/auction/item/cpu*[*@manufacturer* $=$ *"AMD"* $\wedge$ *@clock* ≥ 700].

A *filter* F is a conjunction of path filters $\{P_1, \ldots, P_n\}$. Hence, a notification *matches a filter* if all path filters are satisfied. The set of all notifications that match a given filter F is $N(F)$.

Covering

This section discusses how covering among filters can be detected in the semistructured model. Similar results can easily be obtained for identity and overlapping, too. These are not discussed for brevity.

Let $L_A(A)$ be the set of all values that cause an attribute filter A to match an attribute. An attribute filter $A_1 = (n_1, Q_1)$ *covers* an attribute filter $A_2 = (n_2, Q_2)$, denoted by $A_1 \sqsupseteq A_2$, iff $n_1 = n_2 \wedge L_A(A_1) \supseteq L_A(A_2)$. For example, [@*clock* $\geq$ 600] covers [@*clock* $\geq$ 700].

Let $L_E(C)$ be the set of all elements that match an element filter C. An element filter C_1 *covers* an element filter C_2, denoted by $C_1 \sqsupseteq C_2$, iff $L_E(C_1)$ is a superset of $L_E(C_2)$. For example, [*@clock* $\geq$ 600] covers [*@manufacturer* = *"AMD"* $\wedge$ *@clock* $\geq$ 700]. Furthermore, C_1 is *disjoint* with C_2 with respect to the constrained attributes if there exists no attribute that is constrained in both element filters. For example, [@*minprice* < 100] is disjoint with [*@name* = *"Pu"*] with respect to their constrained attributes.

Corollary 3.4. *Given two element filters C_1 and C_2, neither of which contains two attribute filters with the same name, the following holds: $C_1 \sqsupseteq C_2$ is equivalent to $\forall j \exists i.\ A_i^1 \sqsupseteq A_j^2$.*

Let $L_S(S)$ be the set of all elements that are selected by an element selector S. An element selector S_1 *covers* an element selector S_2, denoted by $S_1 \sqsupseteq S_2$, iff $L_S(S_1) \supseteq L_S(S_2)$. S_1 is *disjoint* with S_2, iff $L_S(S_1) \cap L_S(S_2) = \emptyset$.

In the model presented here, an absolute path covers another absolute path iff both are identical. An absolute path only covers an abbreviated path iff the former is */notification* and the latter is *//notification*, as the root element has a unique name. An abbreviated path covers another (abbreviated or absolute) path iff the former is a suffix of the latter (without the leading // or /). For example, *//cpu* covers *//item/cpu* because the former path selects all elements named *cpu*, while the latter only selects those elements named *cpu* which are a subelement of an element with name *item*.

Let $L_P(P)$ be the set of all elements that match a path filter P. A path filter $P_1 = (S_1, C_1)$ *covers* another path filter $P_2 = (S_2, C_2)$, written $P_1 \sqsupseteq P_2$, iff $L_P(P_1) \supseteq L_P(P_2)$. For example, the path filter *//cpu*[*@manufacturer* = *"AMD"*] covers *//cpu*[*@manufacturer* = *"AMD"* $\wedge$ *@clock* $\geq$ 700]. P_1 is *disjoint* with P_2, iff either S_1 is disjoint with S_2 or if C_1 is disjoint with C_2 with respect to their constrained attributes.

Corollary 3.5. *Given two path filters $P_1 = (S_1, C_1)$ and $P_2 = (S_2, C_2)$, the following holds: $P_1 \sqsupseteq P_2$ is equivalent to $S_1 \sqsupseteq S_2 \wedge C_1 \sqsupseteq C_2$.*

A filter F_1 *covers* a filter F_2, denoted by $F_1 \sqsupseteq F_2$, iff $N(F_1) \supseteq N(F_2)$.

Corollary 3.6. *Given two filters* $F_1 = P_1^1 \wedge \ldots \wedge P_n^1$ *and* $F_2 = P_1^2 \wedge \ldots \wedge P_m^2$ *which are conjunctions of disjoint path filters the following holds:* $F_1 \sqsupseteq F_2$ *is equivalent to* $\forall i \exists j.\ P_i^1 \sqsupseteq P_j^2$.

For example, the filter $\{//cpu[@type = \text{“}Athlon\text{”}]\}$ covers $\{//seller[@name = \text{“}Pu\text{”}] \wedge //cpu[@type = \text{“}Athlon\text{”} \wedge @clock \geq 600]\}$.

3.1.4 Objects

Using objects as notifications is widely used in GUIs (e.g., Java AWT [358]) and visual components (e.g, JavaBeans [359]). The Java Distributed Event Specification [361], which is built upon Java RMI, also uses objects. The difference between this approach and a notification service is that consumers must directly register with the source of an event. Eugster and Guerraoui [124] present how to use structural reflection for content-based filtering of notifications. The object-oriented model is most flexible and powerful, but routing optimizations like covering and merging are difficult to achieve if filters can contain arbitrary code. Mühl and Fiege [264] have presented first ideas on how to support routing optimizations like covering and merging for objects. These ideas are discussed later in this section.

A purely object-oriented approach models notifications and filters as objects. A clear advantage of such a model is that it can easily be integrated with object-oriented programming languages. In contrast to that, models that are based on, e.g., name/value pairs, can only operate on serialized instances of objects violating object encapsulation. Unfortunately, routing optimizations, and in particular, covering and merging, are difficult to achieve if filters can contain arbitrary code. In this section three scenarios for which covering and merging can be supported are described.

Calling Methods on Attribute Objects

Regardless of whether the data models depend on structured or on semistructured records, it is possible to embed objects in notifications. In this case public members can be accessed and public inspector methods can be invoked on the embedded object after it has been instantiated. The returned member or the return value of the inspector method can either be a Boolean value that is directly interpreted as result of the attribute filter or a value that is used in order to evaluate the actual constraint.

For example, suppose that an instance of a class `StockQuote` has been embedded in a notification as an attribute with name *quote*. Then an attribute filter that evaluates this attribute could be specified like this: $\{quote.id() = \text{“}IBM\text{”}\}$. For example, this filter covers $\{quote.isRealTime() \wedge quote.id() = \text{“}IBM\text{”} \wedge quote.Price() > 45.0\}$. Moreover, it could be merged with a filter $\{quote.id() = \text{“}MSFT\text{”}\}$ to a filter $\{quote.id() \in \{\text{“}IBM\text{”}, \text{“}MSFT\text{”}\}\}$.

As stated in [121, 124], structural reflection (e.g., supported by Java) can be used to invoke the specified methods. Unfortunately, the model does not allow us to detect all covering relations among filters. For example, a filter $\{quote.Volume() > 10,000\}$ covers a filter $\{quote.Price() > 100 \ \wedge \ quote.Quantity() > 100\}$ because the volume is defined as the product price multiplied by the quantity.

Filtering on Notification Classes

Here, notifications are objects and consequently they are an instance of some class. Hence, class filters can be used that evaluate the class of a notification: A notification matches a filter if it is assignable to the specified class. It is also possible to support covering and merging. A class filter covers another class filter if an instance of the latter class can be assigned to an instance of the former one. A set of class filters can be merged perfectly if they either contain a class which covers all other classes or if they represent all direct subclasses of their common superclass. Figure 3.11 shows the implementation of a `ClassFilter` in Java. The integration with content-based filtering can be achieved by supporting filters that are conjunctions of a class filter and a specialized filter object whose match method is invoked if the class filter returned true.

Specialized Filter Objects

Another possibility is to use specialized filter objects, an approach that can also be combined with class filters. Such a filter implements a `match` method that evaluates whether a notification matches this filter instance or not. Moreover, it can also implement methods for covering and merging. Figure 3.12 shows the implementation of a `QuoteFilter` in Java. Note that the filters can also be built upon a more generic filter library, which offers, for example, set-oriented filters.

3.2 Matching Algorithms

Matching is probably the most fundamental functionality in a publish/subscribe system. A matching algorithm determines the filters, and thus the recipients, that are matched by a given notification. In this chapter several common approaches are discussed, including brute force, predicate counting, decision trees, binary decision diagrams, and efficient XML matching.

One must carefully distinguish between *notification matching* and *notification forwarding*. While matching aims at determining all filters that match a given notification, notification forwarding aims at determining all destinations for which a filter exists that matches a given notification. This means that for

```
class ClassFilter {
   protected Class class;

   public boolean covers(ClassFilter filter) {
      return class.isAssigneableFrom(filter.class);
   }

   public static ClassFilter merge(ClassFilterSet filters) {
      Class superClass=filters.getCommonSuperClass();
      if (superclass!=null) {
         if (filters.contain(superClass))
            return new ClassFilter(superClass);
         if (filters.containAllSubclasses(superClass))
            return new AllSubclassesFilter(superClass);
      }
      return null;
   }

   public boolean match(Notification n){
      return class.isInstance(n);
   }
}
```

Fig. 3.11. Implementation of a `ClassFilter` in Java

```
public class QuoteFilter {

   public boolean covers(QuoteFilter qf){
      return getSymbolSet().isSuperSet(qf.getSymbolSet());
   }

   public static QuoteFilter merge(QuoteFilter[] qf){
      return new QuoteFilter(QuoteFilter.
            unionOfSymbolSets(qf));
   }

   public boolean match(Event e) {
      if (!(e instanceof QuoteEvent))
         return false;
      return (qf.getSymbolSet().contains(
               ((QuoteEvent)e).getSymbol()));
   }
}
```

Fig. 3.12. Implementation of a `QuoteFilter` in Java

the latter it may not be necessary to determine all matching filters. However, most algorithms do not exploit this difference. They determine all matching filters and derive the set of destinations by "or-ing" the individual destination of each filter. In the following, we concentrate on notification matching.

3.2.1 Brute Force

This is the simplest algorithm. It tests the given notification sequentially against all filters. The main advantage of this algorithm is that it can be used for all kind of filters; for example, it does not presume that filters are conjunctive filters. Moreover, it does not require some kind of preprocessing as other algorithms do. The main disadvantage of this naïve algorithm is its degraded performance. This is because the same predicate is evaluated many times if it is part of many filters. Moreover, the dependencies among predicates are not exploited. For example, the algorithm does not exploit that if the predicate $\{x = 5\}$ is matched, the predicate $\{x = j\}$ for any $j \neq 5$ cannot be matched.

3.2.2 Counting Algorithm

Yan and Garcia-Molina have proposed to use the counting algorithm for document matching [404]. This algorithm separates filter matching from predicate matching. This way, the algorithm avoids evaluating predicates more than once. In the following, we depict the algorithm for conjunctive filters consisting of attribute filters.

For each filter there is a counter that is initialized to 0. Then, all matching attribute filters are determined. For each matching attribute filter, the counters of those filters are incremented which contain the attribute filter as conjunctive term. After all matching attribute filters have been processed, those filters whose counter equals the number of predicates this filter consists of match the given notification.

The simplest strategy to find all matching predicates is to sequentially test each attribute filter as to whether or not it is matched by the given notification. A more advanced strategy is to use multilevel index structures that depend on the type of constraint (e.g., a hash table can be used for equality tests). The first level of the index (the attribute name index) is used to look up all attribute filters constraining an attribute by its name. The second level (the operator index) is used to look up all of those constraints that use a given operator (e.g., equivalence or greater than). The third level (the value index), finally, allow to find all of those attributes for the respective attribute and operator that are satisfied. In this way all matching attribute filters can be found without testing all attribute filters for satisfaction.

Figure 3.13 shows a simple example, where a notification is matched against three filters F_1, F_2, and F_3. From these filters only F_1 is matched by the notification.

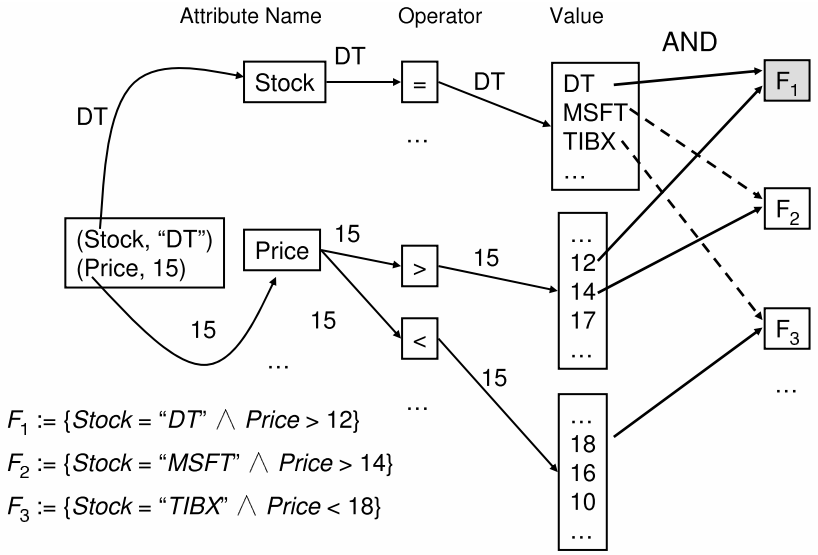

Fig. 3.13. Using a multilevel index structure for the counting algorithm

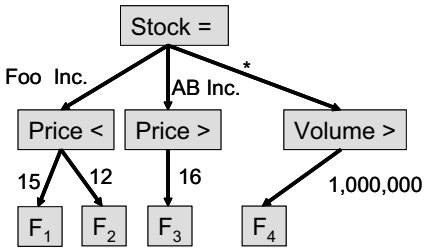

Fig. 3.14. An exemplary decision tree

3.2.3 Decision Trees

Aguilera et al. [6] have proposed using decision trees for matching in publish/-subscribe systems. A *decision tree* arranges tests, test results, and filters in a tree; usually conjunctive filters consisting of attribute filters are assumed. In the tree, nonleaf nodes are tests (e.g., *price* <), while leaf nodes represent filters. Finally, edges are test constants (e.g., 10). The decision tree is usually traversed in depth-first order. The traversal follows an edge if the notification matches the attribute filter that is formed by the test and the test constants (e.g., $price < 10$). The filters that are reached, match the given notification. Figure 3.14 shows an exemplary decision tree. The tree contains the filters $F_1 = \{Stock = \text{“}Foo\ Inc\text{”} \wedge Price < 15\}$, $F_2 = \{Stock =$

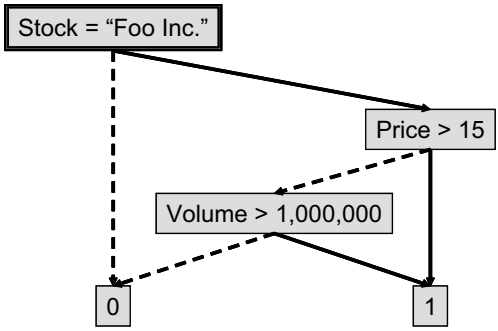

Fig. 3.15. An exemplary binary decision diagram

”Foo Inc.” $\wedge$ *Price* $< 12\}$, $F_3 = \{$*Stock* $=$ *”AB Inc.”* $\wedge$ *Price* $> 16\}$, and $F_4 = \{$*Volume* $> 1,000,000\}$.

3.2.4 Binary Decision Diagrams

Campailla et al. [58] suggested using *binary decision diagrams (BDDs)* for matching in publish/subscribe systems. BDDs are not restricted to conjunctive filters. They can be used to express arbitrary Boolean functions. In the following, we describe the basics of BDDs and how they can be used in publish/subscribe systems.

BDDs are directed acyclic graphs. In a BDD, there are two terminal nodes (i.e., nodes without outgoing edges) with the labels 1 and 0. These stand for the predicates *true* and *false*, respectively. Each nonterminal node corresponds to a predicate (e.g., *price* < 10) and has two outgoing edges, the *low edge* and the *high edge*. A subset of the nodes is marked as *output nodes*; each output node represents a filter. Figure 3.15 shows a simple BDD with a single output node. The solid lines are the high edges while the dashed lines are the low edges. The filter that corresponds to the output node is $\{$*Stock* $=$ *“Foo Inc.”* $\wedge$ (*price* $> 15 \vee$ *Volume* $> 1,000,000)\}$.

A filter is evaluated by traversing the BDD starting from the given output node (Fig. 3.16). While traversing the BDD, the high edge is followed if the predicate corresponding to the visited node is fulfilled by the given notification; the low edge is followed otherwise. A notification matches a filter if finally the node 1 is reached; if 0 is reached, the notification does not match. For example, the notifications $\{\{$*Stock*, *”Foo Inc.”*$\}, \{$*Price*$, 16\}, \{$*Volume*$, 10,000\}\}$ and $\{\{$*Stock*, *”Foo Inc.”*$\}, \{$*Price*$, 14\}, \{$*Volume*$, 1,000,000\}\}$ match the BDD shown in Fig. 3.15.

Evaluating all filters separately can be avoided by using *ordered binary decision diagrams (OBDDs)*. In a OBDD, the nodes are numbered such that for every path, the numbers of the visited nodes are strictly monotonically increasing. This means that the nodes 0 and 1 are numbered by n and $n-1$,

```
  v := <output node of filter>;
2 while <v is not a terminal node> do
     if eval[v] then
        v := high[v];
     else
        v := low[v];
7    endif
  endwhile
  matched := label[v];
```

Fig. 3.16. Evaluating a filter using a binary decision diagram

```
1 for v := n downto 1 do
     if <v is terminal node> then
        value[v] := label[v];
     else
        a := eval[v];
6       value[v] := a and value[high[v]] or
                    not a and value[low[v]];
     endif
  endfor
```

Fig. 3.17. Evaluating an ordered binary decision diagram

respectively. OBDDs are evaluated bottom-up by visiting the nodes in decreasing order starting by node n. If the visited node is a terminal node, a value of 1 is assigned if node 1 is visited and 0, otherwise. If a nonterminal node v is visited it is assigned the value $p(v) \wedge low(v) \vee \neg p(v) \wedge high(v)$, where $p(v)$ is the result of the predicate corresponding to node v, and $low(v)$ and $high(v)$ are the values assigned to the node to which the low and the high edge originating at v are leading, respectively. A filter is matched, if to its output node 1 is assigned; otherwise it is not matched. The algorithm is shown in Fig. 3.17.

A *reduced ordered binary decision diagram (ROBDD)* is an OBDD from which redundant nodes and isomorphic subgraphs are removed. It is known from the research on Boolean function minimization that ROBDDs exhibit exponential grow for some Boolean functions (e.g.,the chessboard function). The predicate numbering has a large effect on the size of the ROBDD, too. While some functions require exponential size only for a subset of the potential predicate orderings, other functions require exponential size for all possible variable orderings. Finding the optimal ordering is known to be NP-hard. BDDs can easily be logically combined. For example, the BDD of a negated function is the BDD of the function, where the nodes 0 and 1 are swapped. BDDs can also "or-ed" and "and-ed" together.

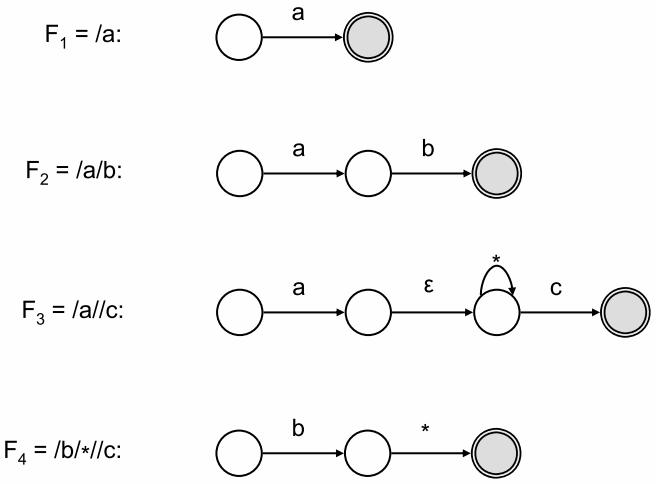

Fig. 3.18. XPath Queries and their corresponding finite state automaton

3.2.5 Efficient XML Matching

As XML becomes more popular, using XML as a data model for publish/-subscribe systems is also gaining increased attention. In the area of XML processing, XPath [398] is often used to select parts of an XML document that match a *path expression*. This approach can also be used to test whether a document contains a matching part. A path expression searches for elements and attributes in an XML document that satisfy the given condition. Because XPath allows for very complex queries, implementing efficient matching for XPath filters is challenging. In the literature, XFilter and YFilter have been proposed to facilitate XPath for matching XML documentsr. Both approaches are based on *finite state machines (FSMs)*. Recent approaches [183] are based on a constructing a *deterministic finite automaton (DFA)* from the given NFA. In the following, we give an overview of XFilter and its successor YFilter. Altinel and Franklin [12] have proposed XFilter, which was the first FSM-based approach. XFilter translates each XPath query into a separate FSM (Fig. 3.18) and uses a novel indexing mechanism to allow all of the FSMs to be executed simultaneously during the processing of a document. When a document arrives, it is processed by an event-based XML parser (e.g., based on the SAX interface). The events raised (e.g., an element is opened or an element is closed) during parsing are used to drive the FSMs through their various transitions. A query is said to match a document if during parsing, an accepting state for that query is reached. The approach of XFilter to use one FSM per XPath query has the disadvantage that commonalities among queries are not exploited.

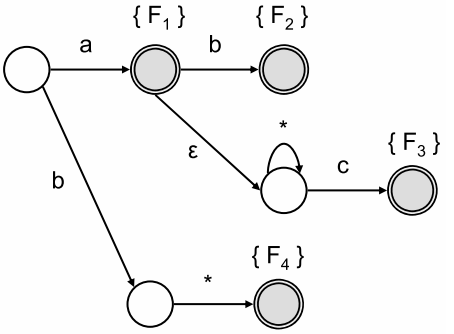

Fig. 3.19. Combined nondeterministic finite state automaton

YFilter, which can be seen as the successor of XFilter, was proposed by Diao et al. [110, 111, 112]. YFilter combines all path expressions into a single *nondeterministic finite automaton (NFA)* (Fig. 3.19), where the common prefixes among path expressions are shared, i.e., represented only once. This NFA-based approach can be extended to also process predicates attached to path expressions. The authors have developed two alternatives to combining the NFA execution and predicate evaluation. One approach evaluates predicates as early as their addressed elements are matched, while the other delays predicate evaluation until the corresponding path expression has been entirely matched.

3.3 Further Reading

Approximate Matching

In this chapter we assumed the Boolean filter model [404]. Either the notification exactly matches the filter or it does not match the filter. An alternative to exact matching is *approximate matching*. Liu and Jacobsen presented A-ToPSS [240], a publish/subscribe prototype with approximate matching. Yan and Garcia-Molina [403] discussed index structure for information filtering under the vector space model.

Matching Algorithms

Fabre et al. [132] and Pereira et al. [305] present matching algorithms which exploit similarities among predicates. In a first step the satisfied predicates are computed, and after that the number of predicates satisfied by a subscription are counted using an association table. Two variants of this algorithm are described that incorporate special treatment of equality tests and of constraints having only inequality tests.

A predicate matching algorithm for database rule systems is presented by Hanson et al. [187] that indexes the most selective predicate that is determined by the query optimizer. They use a special indexing data structure called interval binary search tree to support the efficient evaluation of interval tests.

Gough and Smith [181] present a matching algorithm that is based on automata theory. They show how a set of conjunctions of predicates, each dependent on exactly one attribute, can be transformed to a deterministic finite state automaton. In the paper different types of test predicates are considered and complexity results are obtained. Their algorithm is very efficient, but its worst-case space complexity is exponential. The proposed solution is also not suited for dynamic environments as the automaton has to be newly constructed from scratch if subscriptions change.

Pu et al. [241, 372] present indexing strategies for continual queries based on trigger patterns. In particular, a strategy which uses an index on the most selective predicate is described. More complex indexing strategies exploit similarities among trigger patterns to reduce the processing costs. They restrict optimizations to constraints which place a constraint on a single attribute involving at most one constant.

Gryphon uses the content-based matching algorithm presented by Aguilera et al. [6]. This algorithm traverses a parallel search tree, where nonleaf nodes correspond to simple tests and edges from nonleaf nodes represent results. Leafnodes are associated with matched subscriptions. Banavar et al. [26] present a multicast routing algorithm that executes the matching algorithm at each broker. The algorithm presented is limited to equality tests.