

# **Extending the Siena Publish/Subscribe System**

Dennis Heimbigner  
([dennis@cs.colorado.edu](mailto:dennis@cs.colorado.edu))

CU-CS-946-03    January 10, 2003



University of Colorado at Boulder

Technical Report CU-CS-946-2003  
Department of Computer Science  
Campus Box 430  
University of Colorado  
Boulder, Colorado 80309-0430



# Extending The Siena Publish/Subscribe System

Dennis Heimburger  
(dennis.heimburger@colorado.edu)

31 March 2003

## Abstract

Siena is a form of peer-to-peer communication system based on the publish/subscribe paradigm. Siena messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. Currently, the set of supported types is bool (true or false), long (64-bit integer), double (128-bit floating point), and byte-string, which also subsumes the more traditional string type. These existing, built-in types are adequate for many kinds of applications. However, there are a number of situations where it would be desirable to support types that are rather more complicated than boolean, integers, floating-point, and strings. This report describes how the existing Siena has been extended to support user-defined types. It also demonstrates the use of that type extension system to add several types that support complex query mechanisms such as Site-Select, Query-Advertise, and Unification

## 1 Introduction

Siena is a form of peer-to-peer communication system based on the publish/subscribe paradigm. Siena messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. Currently, the set of supported types is bool (true or false), long (64-bit integer), double (128-bit floating point), and byte-string, which also subsumes the more traditional string type. These existing, built-in types are adequate for many kinds of applications. However, there are a number of situations where it would be desirable to support types that are rather more complicated than boolean, integers, floating-point, and strings. This report describes how the existing Siena has been extended to support user-defined types. It also demonstrates the use of that type extension system to add several types that support complex query mechanisms (Section 6).

## 2 Overview of Siena

Siena [1] is a form of peer-to-peer communication system based on the publish/subscribe [5] paradigm. In a publish/subscribe system, clients publish *event* (or *notification*) messages with highly structured content, and other clients make available a *filter* (a kind of pattern) specifying a *subscription*: the content of events to be received at that client. Event message distribution is handled by an underlying *content-based networking* [2] network, which is a set of server nodes interconnected into a peer-to-peer network. The content-based router is responsible for sending copies of event messages to all clients whose filters match that message.

Siena messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. Currently, the set of supported types is *bool* (true or

false), *long* (64-bit integer), *double* (128-bit floating point), and *byte-string*, which also subsumes the more traditional *string* type. An example message could be represented as the following set of tuples.

(author, "John Steinbeck") (title, "Grapes of Wrath") (edition,1)(instock,true)

A client establishes a subscription by specifying a filter pattern that specifies the kinds of messages it wishes to receive. A filter is a set of triples of (attribute, operator, value). The set of pre-defined operators is shown in Table 1.

Except for Equals and Not-Equals, these operators are not polymorphic in the traditional sense. Rather, each operator requires arguments of a specific type. Given values of other types, a pre-defined set of conversion operators is applied to convert the value to the required type.

In order for a message to match a filter, every attribute in the message must satisfy all corresponding filter triples when the message value is substituted and the operator applied. That is, given the message pair (x,5) and the filter tuple (x,>,0), the value associated with x in the pair (namely 5) is substituted for the name x in the filter tuple and the result is evaluated to either true or false. Thus in this case the evaluation is on the triple (5,>,0), which of course evaluates to true.

The set of filter triples may be considered to be logically "and"ed together so that application of a message to a filter requires that all substitutions evaluate to true. A logical "or" can be achieved by specifying multiple separate filters.

It is important to note that the attribute names used in messages and filters have no inherent semantic meaning. As with all such attribute-based systems, there must be some external agreement about their meaning, and all parties must adhere to that agreement.

Siena adopts a peer-to-peer architecture where arbitrary Siena servers connect to form a specific topology. In the simplest case, a client connects to a server and establishes a subscription. The server then forwards the subscription filter to all of its peers. Each peer notes where the subscription came from, and forwards it to its peers. Later, when some other client connects to a server and generates an event message, the local copy of the filter can be applied at that server to determine the next server to whom the message should be forwarded. Note that if a message is

Table 1. Pre-Defined Operators in Standard Siena

Operator	Argument Type
Equals (=)	bool, long, double, byte-string
Not-Equals (!=)	bool, long, double, byte-string
Less-Than (<)	long
Greater-Than (>)	long
Greater-Equals (>=)	long
Less-Equals (<=)	long
Prefix (>*)	byte-string
Suffix (*<)	byte-string
Contains (*)	byte-string
Any (any)	N.A.

generated for which no filter matches at the local server, then it will not be forwarded at all and so will generate no inter-server traffic. This kind of *content-based routing* is analogous to IP routing in the Internet, but instead of specific IP addresses, the content of messages determines the destination (or destinations) for the message.

### 3 The Covers Relation

In order to understand the complexities of adding types to

Siena, it is important to understand the *Covers* relation. Siena is specifically designed to scale well to wide-area networks. One important way this is achieved is by utilizing an optimization that can reduce the number of filters that a given server must maintain. Key to this optimization is the *Covers* relation over filters. At a given server, for any two filters, F1 and F2, say, it can be determined if (F1 Covers F2) or (F2 Covers F1), or neither. The relation (F1 Covers F2) holds if any message that matches F2 also matches F1; F1 is more general than F2 in that the set of matching messages is larger than the set of messages matching F2.

Since a filter is composed of triples of the form  $(x, op, a)$ , F1 covers F2 if every matching triple satisfies the Covers relation in the following sense.

1.  $\forall$  triples  $t2=(x2,op2,b) \in F2$  ( $\exists$  triple  $t1=(x1,op1,a) \in F1$  s.t.  $x2 = x1$ ) (i.e., every attribute name that occurs in F2 also occurs in F1)
2.  $\forall (x,op1,a) \in F1 \ \& \ (x,op2,b) \in F2$  ( $\exists z (z,op2,b) = true \Rightarrow (z,op1,a) = true$ ) (i.e., the set of values satisfying a triple from F2 is a subset of the set of values satisfying any similarly named triple from F1).

Using this relationship, a forest of partial order trees can be constructed over all filters. Siena servers need only propagate the filters that are at the root of each Covers ordering. As we shall see, support for the Covers relation will be an important issue when adding new types to Siena.

## 4 Requirements for an Extended Type

To be precise, adding a type to Siena using the mechanism in this paper is really a matter of adding new operators to the Siena run-time system; the set of visible types in Siena is left unchanged.

These operators expect that their arguments are of a certain type. In the examples described in Section 7, the operators assume that their arguments have been serialized into the form of byte-strings, and these operators convert their arguments to byte-strings and then de-serialize the byte-string to construct an instance of the actual type on which they operate.

With this in mind, specifying a type requires the specification of a set of operators that can be used in filters. Each operator must specify the following items.

- Name – the printable representation of the operator. The bit-set type in Section 7.1, for example, defines the operators “set<=”, “set>=”, and “set~” (set intersection).
- Index – a unique integer for use in the Siena interface when the operator is specified by integer value rather than by its name; this integer is actually assigned by Siena.
- Apply function – when a filter is matched against a notification message, it must be possible to evaluate each triple of the form (name, operator, value) for specific values taken from the notification message. Thus, each operator must have an associated function to evaluate the operator when given two argument values.
- Covers function – this function takes two filter triples  $t1=(x,op1,a)$  and  $t2=(x,op2,b)$ , for example, and returns true if  $t1$  Covers  $t2$  as defined in Section 3. It is important to note that this operation is purely an optimization and that Siena would work correctly without it, albeit not as efficiently as with it. Constructing this function can be challenging for non-transitive operators. This will be discussed further in Section 7.

```

package siena;

public interface ExtendedOp
{
    public String getName();
    public int getIndex();
    public void setIndex(int index);
    public ExtendedType getType();
    public boolean apply_operator(AttributeValue x,AttributeValue y);
    public boolean covers(AttributeConstraint af1,AttributeConstraint af2);
}

```

Figure 2. ExtendedOp.java Interface.

## 5 Adding Type Extensions to Siena

The process for adding a new type to Siena is relatively simple. Basically, you need to define a Java class, *SetType*, for example that implements the *ExtendedType* interface shown in Figure 1. To simplify this, an abstract super type called *AbstractExtendedType* is defined that handles most of the details. Associated with the type object will be one or more classes that implement the *ExtendedOp* interface (Figure 2). Again, the class *AbstractExtendedOp* can simplify that process. This interface defines the requirements listed in the previous section.

Once the type and operator objects are defined, then it is possible to insert them into Siena by creating an instance of the type class (e.g., *SetType*) and passing it to the static method *siena.Extension.addtype()*.

The details of *SetType* are described in Section 7.1. Appendix 9 shows the actual code for *SetType*. More details can be seen by examining the notifier and subscriber examples in the tests cases provided with the type extension code.

## 6 Applications of an Extended Siena Type System

Two specific examples that could benefit from additional types are Site-Select Addressing [4] and Query-Advertise [3]. The nature of these two application is described in the following subsections. The specific example types described in Section 7 relate back to these example applications.

### 6.1 Site-Select Addressing

Site-Select addressing is an approach to using Siena to carry out intensional (content-based) command against a dynamically changing set of targets. Each potential target exports a

```

package siena;

public interface ExtendedType
{
    public String getName(); // typename
    public boolean define(); // insert type's operators into Siena
    public ExtendedOp[] getOperators(); // return set of operators
                                     // associated with type
}

```

Figure 1. ExtendedType.java Interface.

subscription in the form of a bit-set describing its characteristics. Commands are published that include a bit-set describing the characteristics of targets for whom the command is intended. Standard Siena does not support this efficiently because it does not have the proper bit-set operators defined. The Set Type in Section 7.1 can, however, efficiently support Site-Select.

## 6.2 Query-Advertise

In our prior Query-Advertise work, the goal was to publish queries and have them efficiently directed to subscribing sites that had the potential to provide data that satisfied the published query. One way to do this required that it be possible to determine if two queries,  $q_1$  and  $q_2$ , potentially had a common solution. That is,  $q_1$  was a subscription by a provider of information, where  $q_1$  “described” the data it could provide. If  $q_2$  intersected  $q_1$ , then it made sense to direct  $q_2$  to the  $q_1$  site because there was a reasonable probability that the  $q_1$  site could provide an answer satisfying  $q_2$ . The problem then became one of defining queries such that this kind of intersection could be computed. Again, standard Siena cannot provide this, but both the Unification operations (Section 7.2) and the FilterIntersection operation (Section 7.3) can be used for this purpose.

# 7 Example Type Extensions

## 7.1 Bit-Set Type

The *bit-set* (or just *set*) treats the Siena byte-string type as representing a set of bits. The type provides three operators over these sets. We will use the notation  $b[i]$  to indicate the  $i$ 'th bit of the set. The three operators are as follows.

1. Set-Less-Equal – Given two sets  $S$  and  $T$ ,  $S \text{ set? } T$  if  $(\forall i (S[i] \text{ ? } T[i]))$ ; in other words, every bit that is set in  $S$  is also set in  $T$ . Of course,  $T$  may have additional bits set.
2. Set-Greater-Equal – Given two sets  $S$  and  $T$ ,  $S \text{ set? } T$  if  $(\forall i (T[i] \text{ ? } S[i] ))$ ; in other words, every bit that is set in  $T$  is also set in  $S$ .  $S$  may have additional bits set.
3. Set-Intersection – Given two sets  $S$  and  $T$ ,  $S \text{ set} \sim T$  if  $(\exists i (T[i] \& S[i]))$ ; in other words, at least one bit is set in the same position in  $S$  and  $T$ .

For these operators, the Apply function is straightforward. Figure 3 shows the Java code for this function (referred to in that code as “apply\_operator”) for the set? operator. The computation converts each argument to a byte-string and then compares each string byte-by-byte. If the two strings are different length, then one of two things happens. If the argument 2 is shorter, then it is extended with zero bytes for comparison purposes. If argument 1 is shorter, then argument 2 (the potentially larger set) is truncated because padding argument 1 (the potentially smaller set) with zeros will never change the result computed on the truncated value. The apply\_operator functions for the set? and set~ operators are similar and are not included here.

```
boolean apply_operator(AttributeValue n,
                      AttributeValue f)
{return setLessEqual(n,f);}

boolean setLessEqual(AttributeValue n,
                    AttributeValue f)
{
    byte[] na = toBytes(n);
    byte[] fa = toBytes(f);
    // Need to worry about different length byte arrays.
    // Basically, we will zero extend, which means
    // (if you analyze) that we only need to worry
    // about the filter value;
    // extending the notify value by zeros
    // will not affect the result (because (0 set<= x)
    // is always true).
    int ln = na.length;
    int lf = fa.length;
    boolean match = true;
    // check that for all i: n[i] & f[i] == n[i]
    for(int i=0;i<ln;i++) {
        byte bn = na[i];
        byte bf = (i<lf?fa[i]:0);
        if((bn & bf) != bn) {match = false; break;}
    }
    return match;
}
```

Figure 3. Bit-Set Apply Function

```

boolean covers(AttributeConstraint af1,
               AttributeConstraint af2)
{
    boolean covers = false; //default
    if(af1.op == index && (af2.op == index || af2.op == Op.EQ)) {
        covers = setLessEqual(af2.value,af1.value);
    } else if(af1.op == Op.NE && af2.op == index) {
        covers = ! setLessEqual(af1.value,af2.value);
    }
    return covers;
}

```

Figure 4. Bit-Set Covers Function

Computing the Covers relationship is also straightforward for the  $\text{set?}$  and  $\text{set?}$  operators because they are transitive. Recall from Section 3, that we apply the covers relationship on a per-triple basis. Simplifying somewhat, this requires determining if the following holds.

$$\text{? } z(z, \text{set?}, b) = \text{true} \text{ ? } (z, \text{set?}, a) = \text{true}$$

Note that this is true if the following holds.

$$(b \text{ set? } a) = \text{true}$$

This is basically because the  $\text{set?}$  operator is transitive.

Each operator is required to implement a *covers()* function that is given two triples and returns true if the first triple covers the second, and returns false if the first does not cover the second, or equally important, it cannot determine if the first covers the second. Figure 2 shows the code for the covers relationship for the  $\text{set?}$  operator. Note that actually, the function is not given two triples, but rather only the last two elements of each triple since the attribute name part is assumed to be the same and its specific value is irrelevant. This pair, (op,value), is represented by the class *AttributeConstraint* in Siena.

The code in Figure 4 implements the covers relation for the  $\text{set?}$  operator. It assumes that the operator used in each *AttributeConstraint* argument is the  $\text{set?}$  operator or conforms to one of two special cases. There is no guarantee that only the  $\text{set?}$  operator will be used with a given attribute name. In fact, any other operator could be used. It turns out that the two following special cases can be handled.

$$(b, \text{set?}, a) = \text{true} \text{ ? } (x, \text{set?}, a) \text{ Covers } (x, =, b)$$

$$(b, \text{set?}, a) = \text{false} \text{ ? } (x, !=, a) \text{ Covers } (x, \text{set?}, b)$$

However, aside from the equals and not-equals operators, determining the covers relationship may be impossible in general for the  $\text{set?}$ , operator, hence the code returns false in all those cases.

Computing the Covers relation for the  $\text{set?}$  operator proceeds in a similar fashion as with  $\text{set?}$ . Since it too is transitive, we can show the following.

$$(b, \text{set?}, a) = \text{true} \text{ ? } (x, \text{set?}, a) \text{ Covers } (x, \text{set?}, b)$$

Computing the Covers relation for set intersection (set~) is more difficult because it is not a transitive operator; that is, given that (x set~ y) and (y set~ z) are both true, we cannot infer that (x set~ z). The key is to find some other relationship/operation R such that

$$b R a \quad ? \quad (x, \text{set~}, a) \text{ Covers } (x, \text{set~}, b)$$

Note that this inference need not be “perfect” in the sense that we do not require that

$$(x, \text{set~}, a) \text{ Covers } (x, \text{set~}, b) \quad ? \quad b R a$$

Remember that this is an optimization, and so failure to determine all instances of Covers will only lead to inefficiency, not failure.

It turns out that the set? operator can be used as our R in the above equation. In other words,

$$b \text{ set? } a \quad ? \quad (x, \text{set~}, a) \text{ Covers } (x, \text{set~}, b)$$

The reason for this should be clear. If (b set? a) is true, then it means that a has at least the same set of bits set as are set in b, and possibly more. Thus, any value that intersects b will also intersect a, and hence the set of values that satisfies (x, set~, b) is a subset of the values that satisfy (x, set~, a), and hence

$$(x, \text{set~}, a) \text{ Covers } (x, \text{set~}, b)$$

This method of inferring the Covers relation from some related transitive operator (set?, in this case) is quite general, and will be used in the subsequent examples where non-transitive operators exist.

## 7.2 Unification and the Expression Type

The second type extension example involves a unification operation over functional expression with variables. This is patterned after PROLOG expressions and unification.

Figure 5 gives a YACC-style grammar for expressions. Expressions take the usual form of a nested tree of terms (n-ary functions) with constants or variables or 0-ary functions as the leaves of the nested expression tree.

The Unification operation (unifies) takes two expressions and returns true if there is some assignment of values to the variables such that the two expressions can be made to match when the values are substituted for the variables in each expression. Multiple occurrences of the same variable in an expression must be assigned the same value. Variable names are local to the expression, thus “\_X” in one expression need not have the same assignment as “\_X” in the other expression.

```

unifyexpr : constant | unifyterm ;
unifyterm : ID | ID '(' arguments ')' ;
arguments : argument | argument ',' arguments ;
argument  : unifyterm | constant ;
constant  : NUMBER | STRING | VARIABLE ;

Lexical Tokens:
ID : any string of characters not containing '(', ')', ',', '!', '"',
    and not beginning with a digit or underscore ('_').
VARIABLE: underscore followed by an ID.
NUMBER:  sequence of decimal digits possibly
        preceded by '+' or '-'.
STRING : any sequence of characters enclosed in
        double quotes (""). Occurrences of the
        double quote or backslash ('\') in the string
        are represented as \" or \\ respectively.

```

Figure 5. Unification Expression Grammar.

Table 2. Examples of Unification

Expression 1	Expression 2	Unifiable?	Variable Assignment
f(g(y,5))	f(g(_X,_Y))	true	_X=y _Y=5
_X	_Y	true	_X=_Y _Y=_X
a(_X,b(c(_X)))	a(g("h"),b(c(g(_Y))))	true	_X=g("h") _Y="h"
a(_X,_X)	a(f,g)	false	

Table 2 shows some examples of unification. When the unification is possible (unifies() returns true), then the third column shows the associated assignments of values to variables.

Adding the unification operation to Siena requires providing some form of representation for the expressions. The “external” representation was in the form of an expression tree whose nodes were instances of a general UnifyTerm object class. In order to insert such expression trees into a Siena Filter or notification message, a serialization function was provided that converted an expression tree into an instance of the standard Siena byte-string by doing a pre-order walk of the expression. Similarly, a de-serialization function was provided.

Figure 6 shows the code for the apply\_operator function for the unifies operator. It operates by first converting its two arguments to Strings and then using the de-serialization function (i.e., decode()), to produce an expression tree for each argument. It then attempts to unify the two trees and returns the boolean result. It is worth noting that this procedure is not fast, so there is a definite cost to using unification in a Siena filter.

As with the set intersection operator (set~), unifies is not a transitive operation, and so computation of the covers relation is not simple. As with the set~ operator, the approach is to find a related transitive operation that can be used to compute the covers relation.

The transitive operator used here is *one-way* unification. One-way unification is a special form of the general unification operation that succeeds under the following conditions.

1. E1 unifies E2 is true for expressions E1 and E2.
2. The unification of E1 and E2 only assigns values to the variables in E1, except that E2’s variables may be assigned the value of a variable in E1. This last point is referred to as variable equivalencing and is show in row

```

boolean apply_operator(AttributeValue n,
                       AttributeValue f)
{
    String sn = n.stringValue();
    String sf = f.stringValue();
    UnifyExpr un = new UnifyExpr();
    if(!un.decode(sn)) return false;
    UnifyExpr uf = new UnifyExpr();
    if(!uf.decode(sf)) return false;
    boolean match = Unifier.unify(un,uf);
    return match;
}

```

Figure 6. Unification Apply Function

2 of Table 2.

In effect, if one-way unification succeeds, then E1 is “more general” than E2. Figure 7 indicates how one-way unification operates and why it is transitive. One-way unification of A and B effectively overlays A on top of B starting at the root of each. Thus, the leaves of A form a frontier on B such that the variables in B are either below the frontier or on the frontier in the case of equivalencing. Similarly, if B one-way-unifies (a.k.a. unifies1) C, then C’s variables are below B’s frontier. Obviously, that also means that C’s variables are below A’s frontier, hence we can see that

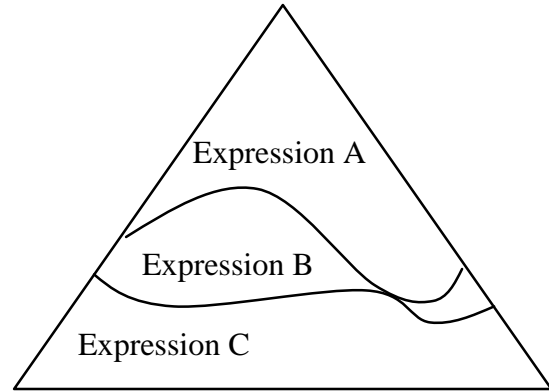


Figure 7. Unify1 Overlays.

$$(A \text{ unifies1 } B) \text{ ? } (B \text{ unifies1 } C) \text{ ? } (A \text{ unifies1 } C)$$

which means that unifies 1 is transitive.

The unifies1 operator can be used as our transitive operator from whom we can compute a subset of the Covers relation; that is, the following is true.

$$(a \text{ unifies1 } b) \text{ ? } (x, \text{unifies}, a) \text{ Covers } (x, \text{unifies}, b) \quad (1)$$

The reason for this is that if x unifies with b, then it will unify with a because a is “more general” than b (because (a unifies1 b)). Note that equation (1) only allows the computation of a subset of Covers because there may be special cases where the Covers relation holds, but (a unifies1 b) does not hold. But since Covers is an optimization, this is acceptable, and it just means that filter covering will not always operate as efficiently as possible.

### 7.3 Filters as an Extended Type

The third example of adding an extended type to Siena involves adding Siena Filters as the types with filter intersection as the operation. The filter intersection operation will be designated using “&&”. This particular example can be confusing because it seems somewhat recursive.

The basic idea is that we want a filter triple (x,&&,f) to match a notification message pair (x,g) if (g intersects f), where f and g are instances of Siena filters. Two filters are defined to intersect if the following is true.

$$\text{? } m (f(m) = \text{true}) \text{ ? } g(m) = \text{true}$$

In other words, two filters intersect if there is some possible message that matches both filters.

Two filters F1 and F2 intersect if their triples can be shown to intersect. This requires the following condition is satisfied.

$$\text{? } (x, \text{op1}, a) \text{ ? } F1 \text{ and } (x, \text{op2}, b) \text{ ? } F2 \text{ (? } z (z, \text{op2}, b) = \text{true and } (z, \text{op1}, a) = \text{true)}$$

Note that we do not require every attribute name in one filter to occur in the other. This is because a missing attribute name is equivalent to the triple (y,any,\_) and so the following is trivially true.

$$\text{? } (y, \text{op1}, a) \text{ ? } F1 \text{ and } (y, \text{any}, \_) \text{ ? } F2 \text{ (? } z (z, \text{op2}, b) = \text{true and } (z, \text{any}, \_) = \text{true)}$$

Thus, our apply function essentially must enumerate triples and for each pair of triples (one from each filter), it must examine the combinations of operations and determine if there is a potential common solution. Figures 8a and 8b (following pages) show the code for the core of the intersection computation. It takes two AttributeConstraint objects and determines if they can intersect or not.

## **8 Summary**

We have shown how to extend the Siena run-time system with new types (really new operators). We have also shown examples of extended types that are especially relevant to the problem of using Siena to perform distributed query as represented by Site-Select and Query-Advertise.

```

boolean constraintIntersect(AttributeConstraint c1,
                           AttributeConstraint c2)
{
    // Special case the situation when c1.op == Op.any or c2.op == Op.any
    if(c1.op == Op.ANY || c2.op == Op.ANY) return true;
    // Special case the situation when c1.op == c2.op
    if(c1.op == c2.op) {
        switch (c1.op) { // following cases always true
            // because of transitivity
            case Op.LT: // x<a && y<b == true (e.g. any x <= min(a,b))
            case Op.GT: // x>a && y>b if a>b
            case Op.GE: // x>=a && y>=b == true
            case Op.LE: // x<=a && y<=b == true
            case Op.NE: // x!=a && y!=b == true
                return true;
            default: break; // need to look more closely
        }
    }
    // Canonicalize the order since intersection is commutative
    // Assume the ordering in Op.java
    if(c1.op > c2.op) {AttributeConstraint c = c1; c1 = c2; c2 = c;}
    switch (c1.op) {
    case Op.EQ: // x=a && y op b if a op b
        return Covering.apply_operator(c2.op,c1.value,c2.value);
    case Op.LT: switch (c2.op) {
        case Op.GT: // x<a && y>b if b<a
        case Op.GE: // x<a && y>=b if b<a
        case Op.PF: // x<a && y prefix b if b<a
        case Op.SF: // x<a && y suffix b if b<a
        case Op.SS: // x<a && y contains b if b<a
            return Covering.apply_operator(c1.op,c2.value,c1.value);
        case Op.LE: // x<a && y<=b == true
        case Op.NE: // x<a && y!=b == true
            return true;
        default: return false;
    }
    case Op.GT: switch (c2.op) {
        case Op.LE: // x>a && y<=b if b>a
        case Op.PF: // x>a && y prefix b if b>a
        case Op.SF: // x>a && y suffix b if b>a
        case Op.SS: // x>a && y contains b if b>a
            return Covering.apply_operator(c1.op,c2.value,c1.value);
        case Op.GE: // x>a && y>=b == true
        case Op.NE: // x>a && y!=b == true
            return true;
        default: return false;
    }
    }
}

```

Figure 8a. Filter Intersection Function

```

case Op.GE: switch (c2.op) {
  case Op.LE: // x>=a && y<=b if b>=a
  case Op.PF: // x>=a && y prefix b if b>=a
  case Op.SF: // x>=a && y suffix b if b>=a
  case Op.SS: // x>=a && y contains b if b>=a
    return Covering.apply_operator(c1.op,c2.value,c1.value);
  case Op.NE: // x>=a && y!=b == true
    return true;
  default: return false;
}
case Op.LE: switch (c2.op) {
  case Op.PF: // x<=a && y prefix b if b<=a
  case Op.SF: // x<=a && y suffix b if b<=a
  case Op.SS: // x<=a && y contains b if b<=a
    return Covering.apply_operator(c1.op,c2.value,c1.value);
  case Op.NE: // x<=a && y!=b == true
    return true;
  default: return false;
}
case Op.PF: switch (c2.op) {
  case Op.PF: // x prefix a && y prefix b if a prefix b | b prefix a
    return Covering.apply_operator(c1.op,c1.value,c2.value)
      | Covering.apply_operator(c1.op,c2.value,c1.value);
  case Op.SF: // x prefix a && y suffix b == true (i.e. a concat b)
  case Op.SS: // x prefix a && y contains b == true (i.e. a concat b)
  case Op.NE: // x prefix a && y!=b == true
    return true;
  default: return false;
}
case Op.SF: switch (c2.op) {
  case Op.SF: // x suffix a && y suffix b if a suffix b | b suffix a
    return Covering.apply_operator(c1.op,c1.value,c2.value)
      | Covering.apply_operator(c1.op,c2.value,c1.value);
  case Op.SS: // x suffix a && y contains b == true (i.e. b concat a)
  case Op.NE: // x suffix a && y!=b == true
    return true;
  default: return false;
}
case Op.SS: switch (c2.op) {
  case Op.SS: // x contains a && y contains b == true
  case Op.NE: // x contains a && y!=b == true
    return true;
  default: return false;
}
default: return false;
}
}

```

Figure 8b. Filter Intersection Function

## 9 Appendix. SetType.java (and SetLessThan.java)

```
package sienatypes;

import siena.*;
import java.io.ByteArrayOutputStream;
import java.io.ObjectOutputStream;

public class SetType extends AbstractExtendedType
{
    static final int operatorcount = 3;

    public SetType()
    {
        super("Set",operatorcount);
        operators[0] = new SetLessEqual();
        operators[1] = new SetGreaterEqual();
        operators[2] = new SetIntersect();
    }
}

class SetLessEqual extends AbstractExtendedOp
{
    public SetLessEqual() {super("set<=");}

    // During matches, n is from the notification, f from the filter

    public boolean apply_operator(AttributeValue n, AttributeValue f)
    {
        byte[] na = toBytes(n);
        byte[] fa = toBytes(f);
        int ln = na.length;
        int lf = fa.length;
        boolean match = true;
        // check that for all i: n[i] & f[i] == n[i]
        for(int i=0;i<ln;i++) {
            byte bn = na[i];
            byte bf = (i<lf?fa[i]:0);
            if((bn & bf) != bn) {match = false; break;}
        }
        return match;
    }

    public boolean covers(AttributeConstraint af1, AttributeConstraint af2)
    {
        boolean covers = false; //default
        if(af1.op == index && (af2.op == index || af2.op == Op.EQ)) {
            covers = setLessEqual(af2.value,af1.value);
        } else if(af1.op == Op.NE && af2.op == index) {
            covers = ! setLessEqual(af1.value,af2.value);
        }
        return covers;
    }
}
```

Figure 9. SetType and SetLessEqual

## 10 Appendix. ExtendedStartServer.java

The final component added to Siena to support extended types is a replacement for the StartServer component. StartServer contains a main(), and is used to start up a standalone Siena router. It takes a variety of arguments as described at this URL.

<http://www.cs.colorado.edu/~carzanig/siena>

The replacement component is called ExtendedStartServer, and can be used as a drop-in replacement for StartServer. The primary change in ExtendedStartServer is to add a new parameter called “-extension”. An example command using ExtendedStartServer is shown in Figure 10. The first command establishes the CLASSPATH, which tells java where to look for specific classes. Note that the first file in the CLASSPATH is “extension.jar”, which contains the extension code, including modified versions of some standard Siena classes. It is important that the extension.jar file appear before the Siena jar file (siena-1.4.3.jar, in this case) so that the modified Siena classes will come from the extension jar file.

The second command line uses the “-extension” parameter to establish the set of new types that are to be loaded into the server. The argument to this parameter (“sienatypes.SetType”, for example) gives the class name for the class that defines the type. There is nothing special about the “sienatypes” prefix, and the class name could be anything. During startup of the server, reflection is used to find and load the specified extension classes. Thus, whatever they are named, these classes must be accessible through the class path.

The third command starts the ExtendedServer, and is exactly the same as starting the normal Siena StartServer, except that it includes the extension parameters.

```
CLASSPATH=extension.jar:siena-1.4.3.jar
EXTENSIONS=-extension sienatypes.SetType -extension sienatypes.UnifyType
Java -cp "$CLASSPATH" siena.ExtendedStartServer -port 2001 $EXTENSIONS
```

Figure 10. Using the ExtendedStartServer

## 11 References

- [1] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. “Design and Evaluation of a Wide-Area Event Notification Service”. *ACM Transactions on Computer Systems*. 10(3): 332–383 (Aug. 2001).
- [2] Antonio Carzaniga and Alexander L. Wolf. “Content-based Networking: A New Communication Infrastructure”. *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*. Oct. 2001. Scottsdale, AZ.
- [3] Dennis Heimbigner. “Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality”. *2001 ACM Symposium on Applied Computing (SAC 2001): Special Track on Coordination Models, Languages and Applications*. pp. 176–181, 11-14 March 2001, Las Vegas, NV.
- [4] Jonathan Hill. “Site-Select Messaging for Distributed Systems”. *University of Virginia Department of Computer Science Technical Report CS-2002-06*, April 1, 2002.
- [5] Steven P. Reiss. “Connecting Tools Using Message Passing in the Field Environment”. *IEEE Software*, July 1990, pp. 57–67.