USING THE STRUCTURE OF XML TO GENERATE NOTIFICATIONS AND SUBSCRIPTIONS IN SIENA

by

Chris Corliss

A thesis submitted in partial fulfillment of
the requirements for the degree of

Masters of Science in Computer Science

University of Colorado, Boulder

2001

This thesis entitled:
Using the structure of XML to generate notifications and subscriptions in SIENA
written by Christopher Ray Corliss has been approved for the
Department of Computer Science

Approved by ____
Chairperson of Supervisory Committee

____

Date ____

University of Colorado, Boulder

Abstract

USING THE STRUCTURE OF XML TO GENERATE NOTIFICATIONS AND SUBSCRIPTIONS IN SIENA

by Christopher Ray Corliss

Chairperson of the Supervisory Committee: Professor Alex Wolf
Department of Computer Science

This thesis focuses on how to allow XML clients to interface with SIENA, an event notification architecture. There are two challenges to make this happen. The first one is how to take a client's XML notification and translate it into a SIENA notification, which has a different data model from XML. The other challenge is translating an XPath expression into a SIENA subscription. The approach taken to overcome these challenges is to use the structure of the XML notification as a way to guide the translation process. The structure of the XML notification is specified in a file using the XSchema language. A set of generic rules has been developed, which can then be applied to any XSchema that will result in the creation of a map mapping the XML notification tags into SIENA notification attribute names. This same mapping is also used to translate an XPath expression into a set of constraints used in the creation of the SIENA subscription.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to thank Professor Alex Wolf and Antonio Carzaniga for their patience and my wife whose love and support has given me the strength to write this thesis.

GLOSSARY

SIENA. Scalable Intranet Event Notification Architecture: architecture of an event notification system that can span heterogeneous networks and uses a publisher/subscriber protocol.

XML. Extensible Mark-up Language.

Notification. A data structure used by SIENA to transmit events to its clients.

Subscription. A set of filters that are used by a SIENA server to determine if a notification should be sent to a client.

XPath. A way of expressing a path through an XML documents using tree axes specifier, note tests, and predicate tests.

XSchema. A language used to describe the structure of an XML document by describing the elements, types, and ordering of the XML document.

Chapter 1

INTRODUCTION

Overview of Siena

SIENA is a software architecture for an event notification that has been developed at the University of Colorado. What makes it stand out is its ability to span different types of networks and its context sensitive routing of information. This last part allows the system to be more flexible compared to the standard network multicast or broadcast means of sending information to multiple clients because it allows individual clients to filter out information they do not want on the servers. The purpose behind the architecture is to have "an event notification service that we have designed to maximize both expressiveness and scalability." [1]. This is accomplished in several ways: By using an extension of the publisher/subscriber protocol; the ability to configure SIENA servers in a way that allows them to work the most effectively for the underlying network topology; and the use of a subscription language that is expressive. Only a brief overview of the publisher/subscriber protocol is given here followed by a description of the software elements that make up the SIENA architecture and its subscription language.

The publisher/subscriber protocol is a way of routing information only to the clients that are interested in the information and not to all clients that are connected to a server. The clients simply “subscribe” to information that the server is “publishing”. In SIENA this idea is extended in that the servers are not the computers publishing information but rather other clients connected to the servers. Clients are therefore able to both publish an event to a server and subscribe to different events. The task of the servers is to figure out a path from a client that is publishing the information to a client that has subscribed to it independent of where the clients are located. This is the context sensitive routing in which, based on content, the event is routed to different servers, which then send to other servers in the SIENA network or to the clients that are connected to it if they have subscribed to the information. This is different than a multicast socket or listserv in that there is a selection mechanism, a subscription, which allows the events to be filtered unlike a multicast socket or listserv environment where there is no ability to filter on the client side. This way, only the clients that want the particular event receive it while other clients do not even know of its existence. A key to making this work is the subscription language that will be examined after first looking at the data model used by SIENA.

The notification is the container for an event in SIENA. It is a type-less data structure containing attributes that contain the information of the event. Type-less means that the notification itself has no specific type but that attributes inside of the notification do have types associated with them. An attribute consists of a name, type, and value. The name of an attribute is represented as a string and has to be unique in the notification that contains it. The type of an attribute is chosen from a predefined set of types that are common to programming languages. These types also have a defined set of operations that can be used on them. The value is the part that holds the data for the attribute. The only constraint for the value is that it must be of the same type designated by the attribute. There is no limit to the number of attributes that can appear in a notification. The notification is referred to as being "flat" because there is no relational information between the attributes contained or structure in the notification. This will become a problem when dealing with translating a XML notification into a SIENA notification. These attributes are what the subscriptions for SIENA are based on.

The subscription submitted to a server contains a set of filters that are used to determine if a notification is "covered" by a subscription. When a subscription covers a notification it means that the attributes in the notification satisfy the requirements of all the filters in the subscription. A filter consists of four basic parts: The name of the attribute to evaluate; the type of the attribute; the operation to be applied to its value; and the target value for the operation. In order for an attribute to pass a filter it must meet the first two parts of the filter, name and type, and have the result after the operation is applied to its value to satisfy the requirement of the target value. In cases of equality this means the attribute value must be equal or the same as the target value. For inequalities, the value must be less than or greater than the target value based on the type of inequality used. Two filters for the same attribute is allowed. In this case the attribute has to satisfy both filters in order for the subscription to cover it. The following is an example of a notification and a subscription.

Attribute Type	Attribute Name		Value
String	class	=	/finance/exchantes/stock
Time	date	=	Mar 4 11:43:37/MST 1998
String	exchange	=	NYSE
String	symbol	=	DIS
Float	prior	=	105.25
Float	change	=	-4
Float	earn	=	2.04

Figure 1: Example of a Notification for a stock

String	class	>*	/finance/exchantes
String	exchange	=	NYSE
String	symbol	=	DIS
Float	change	>	0
Float	earn	=	2.04

Figure 2: Example of a Subscription

If a client used the example subscription in SIENA and another client published the notification then, in this case, a server would reject the notification because the subscription did not cover the notification since the attribute named “change” was a value that was not greater than zero.

This concludes the background information about the SIENA architecture. The next brief overview describes XML and its structure. The sections that follow will then discuss the problems and solutions to building a XML client for SIENA.

XML Overview

XML "is a method for putting structured data into a text file." [2]. XML is a text based markup language with the tags being used to delimit data. The basic element of an XML document is a tag composed of four parts: A name; optional attributes in the tag; optional value for the tag; and other tags contained with in it. The only required part of a tag is the tag name. There is no information about the interpretation of the tag parts within XML, such as the meaning of the tag name or the meaning of the value. The interpretation is left to the application that is reading the XML document.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
```

```
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```
Figure 3: Example of an XML document

The data model in XML is hierarchical in nature. Tags are arranged in such a way that a tag can have one enclosing tag but that may itself enclose multiple tags. This is basically a tree relationship of one parent/many children. A specification has been written that explains a structure that can be used to view the XML document in this tree form explicitly. The structure is the DOM, Document Object Model. The DOM is represented as a tree of nodes where each node specifies a particular part of the XML document such as a tag, an attribute, and their respective values. Because of this data model, tags in a XML document can have tags with the same name without any conflicts because the tags can be differentiated by their position in the tree. Besides the specification for the DOM, other specifications have been written to help in the processing of an XML document. The program heavily uses two of them: XSchema [4] and XPath.

XSchema is a specification that has been developed to define the structure of the DOM, and thus, the structure of a XML document. XSchema defines the structure of the DOM by specifying the valid tags that can be used in the XML document, the positions of the tags in the document, the type of values the tags can contain, any type of attributes that the tags support, the number of occurrences of a tag, and other structural elements.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">


 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:element name="comment" type="xsd:string"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
 </xsd:complexType>
```

Figure 4: Example XSchema document

As can be seen in the example shown in Figure 4 the XSchema does not directly specify the XML tags but rather uses a notation that defines the valid tags and the possible order in which they can be in. This XSchema describes the XML document in figure 3. All this information is stored as a XML document itself, with an .xsd extension, in a separate file from the XML document it describes. The XML document has a reference to the file that contains the XSchema it adheres to. A XML document that conforms to the XSchema specified is said to be a valid document.

The XPath [3] specification explains the format and rules of a path expression that can be used to navigate through a XML document. It describes that a path is composed of steps with each step specifying a set of constraints to be applied to the current context node. The basic structure of a step can be broken down into three parts: What axis of the DOM tree to look at; the name of the nodes to filter out; and then predicates to execute against these nodes for further filtering. The tree axis specifier simply states in what direction along the tree that the node test should be applied. These are specifically defined in the specification to cover different parts of the XML document. Some examples are:

- parent: This tells the XPath processor to test the parent of the context node.
- child: This tells the XPath processor to test the children of the context node.
- ancestor: This tells the XPath processor to test any of the ancestors of this node. An ancestor is simply a node that can be reached by walking up the tree using only the parent nodes.
- following-sibling: This tells the XPath processor to test the siblings of the context node that follow it.
- preceding: This tells the XPath processor to test *all* of the nodes that precede the context node in reverse document order.

At the end of each evaluation of a step a set of nodes is returned back to the XPath processing task, which then uses them in the next step as the context nodes. If a step does not return any nodes then the XPath fails. By the end of the evaluation of a path, a set of nodes is returned containing the final tree nodes that satisfy the XPath. The set of nodes can be empty, contain only a single node, or contain multiple nodes. These are all valid in XPath.

This concludes the overview of XML. The next chapter looks at the problems of integrating an XML client into the SIENA architecture followed by a description of the architecture of the client itself. For further information about SIENA or XML the following web sites are recommended:

- SIENA: http://www.cs.colorado.edu/~carzanig/siena/
- XML: http://www.w3c.org/xml

C h a p t e r 2

CHALLENGES

Differences in Structure

One of the challenges faced in creating a XML client for SIENA was resolving the differences in the two data models. Because only the clients are able to see and handle the XML notification and not the servers of SIENA, the XML notification has to be "flattened" into a SIENA notification. Flattening an XML notification is the process of removing the relational information, or in essence, removing the DOM from the document. This would not be a problem if it were not for the fact that XML tags can have the same name in an XML document but SIENA attributes have to have unique names. After removing the DOM from the document, these tags can no longer be distinguished. This could result in a situation where data is lost if a simple scheme of using the name of the XML tag as the name of the notification attribute is used to translate the XML notification into a SIENA notification. For some XML tags this is not a problem if the type of data in the tag was a string. These tags could be grouped together into one attribute with the name of the tag being the name of the notification attribute. Then the string operators of SIENA could be used to test to see if there are instances of particular strings in the attribute. However, for tags that contained integer values this was not an option since subscription filters can perform inequality on the values. Some method would be needed to differentiate these tags from each other in the notification.

Flattening introduced another problem: How much information should there be in the SIENA notification about the structure of the XML notification. The first try at a flattening scheme simply did the translation by brute force and included unnecessary information in the notification. Figure 5 shows an example of some of the unnecessary information generated.

```
String purchaseOrder: billTo sendTo items
String sendTo: name street city state zip
…
Items: Item Item
```

Figure 5: Example of some of the useless information in the notification

The information was unnecessary because it only gave information about the structure of the XML notification that the clients already know from the XSchema. The clients know that the purchaseOrder tag will contain a “billTo”, “sendTo” and “items” tags. Therefore, the information does not need to be included in the notification. Having this useless information is not desirable as the clients subscribing to the notification would have to deal with this unnecessary information, while trying to pull the real information about the event they want. Going to the other side of the spectrum, by not including any information about the structure of the XML notification would not work either. Some information about the structure of the XML needs to be included when dealing with a XML notification that has tags with the same name. This information can be used to distinguish between tags that have the same name but are in different places in the DOM. The problem then is how much structural information should be put in the notification and when. A related problem to this was how to use the information in the XSchema to guide the process of flattening the XML notification. It was not clear on how the knowledge of the structure of the XML notification could be leveraged to help in making the decisions of what tags to include or how to differentiate tags with the same name in the SIENA notification.

Subscription Language

Another problem faced was how to handle the subscriptions from an XML client. One basic question that needed to be answered was how to represent a subscription in XML. XQuery [5] was looked at and deemed too difficult to implement as a subscription language for the first release of the software. The main reason for this was that XQuery was too powerful and allowed the existence of queries that would be too difficult, if not impossible, to translate into a SIENA subscription. An example of this would be a query that relied on the value of one attribute to determine the name or value of another attribute; i.e. attribute x has value y, which is then used to determine the name of another attribute, which contains the information to be queried. The other option was to use XPath. One of the problems encountered using XPath was that it used the DOM tree to navigate an XML document that no longer existed for the XML notification because it was flattened. This makes the translation of an XPath statement into a SIENA subscription not as straightforward as hoped for and in fact it may be necessary to represent the XPath statement as more than one SIENA subscriptions.

Now that the problems have been identified the next two chapters cover the architecture and implementation of a XML Client interface to SIENA that contains solutions to these problems.

Chapter 3

XML INTERFACE ARCHITECTURE

Overview

One of the driving ideas behind the architecture is to not limit what the XML clients are able to do. It would have been very easy to place requirements on the XML notification structure by providing an XSchema that their notifications would have to adhere to, or other constraints that would limit the power of the client and in some ways limit the power of XML itself. Instead, the only requirement of the architecture is placed on the publisher of the XML notification. The XSchema for the notification has to be made available to both the publishing and subscribing parts of the architecture. Besides this the clients are free to do what they want with the XML notification.

The high level architecture is shown in figure 6. Two main sections are shown in the diagram. To the left is the part of the architecture that deals with publishing the XML notifications and on to the right is the other part that handles the subscriptions to the XML notifications. The SIENA network of servers ties the two sections together. The dashed lines show data flow in the architecture while solid lines show control flow. The XML Client is on both sides of the architecture because it is a generic interface to the internal components that do the publishing and subscribing of XML notifications. There are also three boxes that do not lie within either section of the architecture. These boxes represent data sources that both sections need access to in order to handle the XML notification and XPath subscription correctly. Not shown is a common component that is used by both sections to generate the XML to SIENA map.

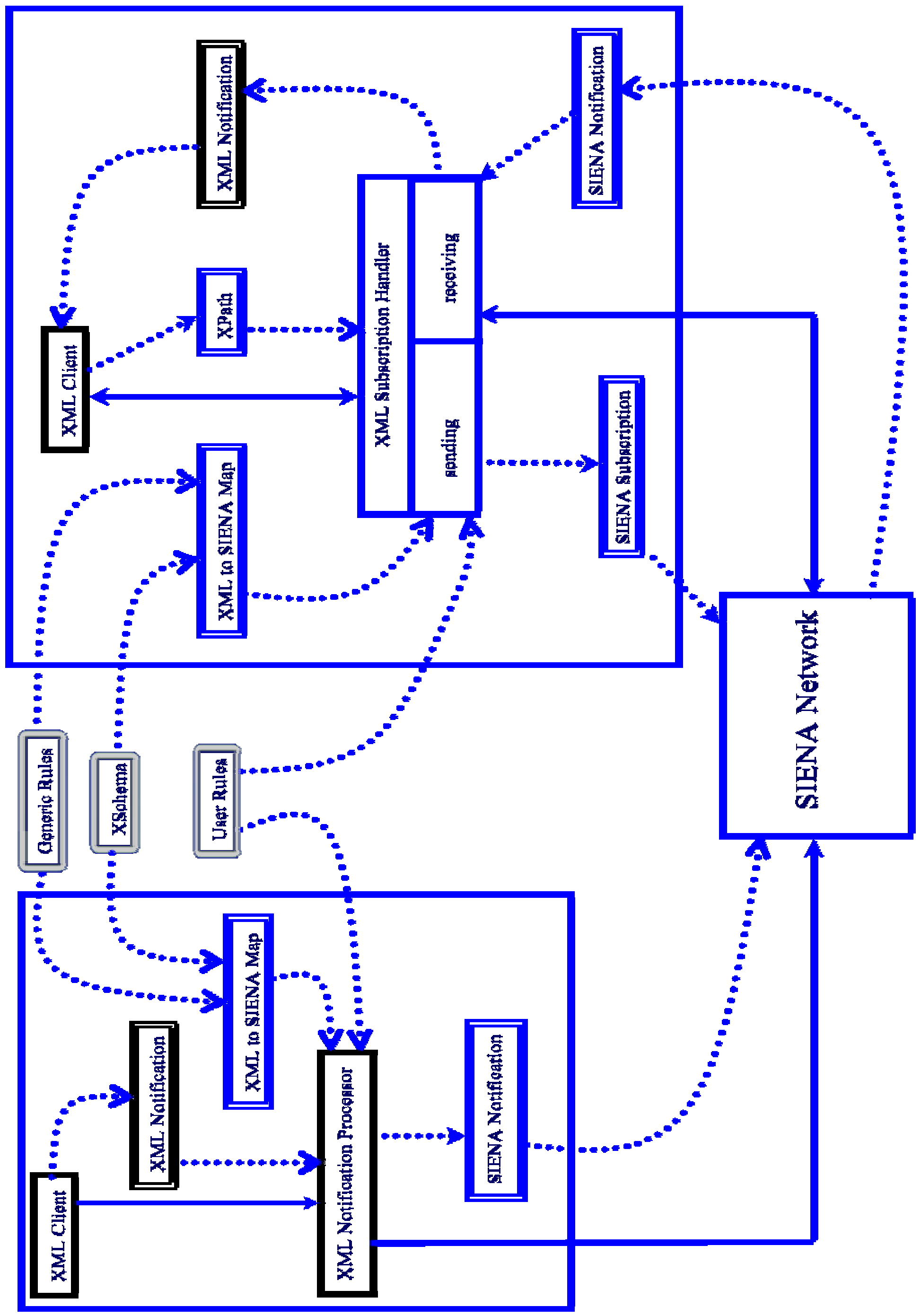


Figure 6: High Level Architecture

This component takes the generic rules and XSchema and produces the map that is then used by both sections to translate either the XML or XPath into a set of SIENA attributes. This component will be described in more depth later on. The user rules for either section are directly applied during the translation process with no preprocessing involved. It is not mandatory that the subscription section have access to the user rules, however, it is highly suggested that the user rules should be given to the subscription section so that it can correctly translate the XPath steps into the appropriate SIENA notification attributes.

XML Notifications

The focus of this next section is on the part of the architecture that translates a given XML notification into a SIENA notification. The architecture diagram shows that the XML Client transfers control to the XML Notification processor. When doing so it also sends the XML Notification to it. The processor will then take the notification and use the XML to SIENA map and user rules if they have been specified, to generate a SIENA notification. An optional step is to include the user rules in the translation process if they have been specified. Once the SIENA notification has been created it then simply gets published like any other notification. There are basically two steps in the translation process: Applying the map to the notification and applying the user rules to the notification. After these two steps have been taken, a SIENA notification is ready to be published. Before a more in-depth explanation of the two steps is given a description of the language that is used to specify the rules in is given.

An Introduction: the Rules Language.

The fundamental idea behind the language is to have a language that gives someone the ability to specify a set of parameters or conditions to select XML tags. Once a tag has been selected then a particular action is associated with the node that indicates how it should be treated when translating it into a SIENA attribute. These two parts, the condition and action, along with a tag specifier make up a rule in the language.

Every rule in the language begins with the word “rule” followed by a colon. This is an indication to the parser that a rule is about to begin. Following this is a tag specifier that indicates the XML tag, which the rule should be applied to. This is the first part of the filtering process. The rule will only be considered when the current XML tag being looked at matches the tag specifier in the rule. The tag specifier can be any valid identifier followed by an optional attribute identifier. An identifier is defined as starting with a letter or underscore, followed by more letters, numbers or underscores. There is no limit to the length of the tag specifier. Spaces and any other symbol besides the underscore cannot be used in the specifier. The optional attribute identifier is a ‘@’ followed by another identifier. The attribute identifier allows the rule to be applied to an attribute of an XML tag instead of the actual XML tag. There is a special key word, which can be used for the tag specifier. The “any” keyword indicates that any XML tag or any attribute is the target of the rule.

Once the specifier has been given then the condition is expressed. The condition is a test that the XML tag must pass before the action is applied to the tag. The condition can specify if the XML tag or attribute is present or compare the tag’s value to some other value. Some conditions have a special comparison value. A tree axis specifier can be used followed by a tag specifier. The tree axis specifier can be only one of two words: Parent or sibling. The tag specifier follows the same rules that the tag specifier used at the beginning of the rule definition. What this allows is the value in the XML tag or attribute to be compared to either its parent’s value or all of its sibling’s value. If the two values are equal then the rule is applied to the XML tag. An example is in the default rules:

rule: any@type = sibling any@type PATH

The tag specifier uses the special "any" keyword for the XML tag. This means that any XML tag will work. It also includes the attribute specifier that is "@type". So this rule specifies that the target XML tag is any tag that has an attribute whose name is "type". The condition is an equality condition with the tree axis specifying sibling. When the rule is being checked against an XML tag that has a type attribute the program will search through its siblings. The next part indicates the tag the program should look for in the current XML tag's siblings. This rule indicates any XML tag is valid if it also has the "type" attribute. If a XML tag is found that does, then the value of its type attribute is compared to the value in the current XML tags "type" attribute. Only if they are equal will the program then apply the action to the current XML tag. If "any" was not specified but some other tag name was given then the program would select only the siblings that had the XML tag specified.

The last part of the rule definition is the action that is to be taken. The action can simply be the name of the SIENA attribute in which to store the value of the XML tag or attribute into. If multiple tags have the same action then the values will be separated in the notification by comas.

The program comes with a set of default rules that are used to process a XSchema. After the XSchema has been processed a map is generated that maps a XML tag into a name of a SIENA attribute. This is the XML to SIENA map shown in the architecture. More detailed information on the rules language is given in a later section in this chapter and also in Appendix A and B. The next section covers the use of the generated map.

Applying the Map

The applying of the map is rather a straightforward task. The XML Notification Processor (processor) first parses the XML notification and creates the DOM for it. The Xerces XML parser is used to do the parsing, which returns a Document node that is the root of the DOM for the XML. Then using recursion, the processor walks through the DOM depth first checking to see if the current node it is on matches a user rule or has an entry in the map.

The processor checks to see if any of the user rules can be applied to the XML tag that is represented by the DOM node first. This allows a user to override any of the generic rules if they desired to do so. If there is a rule that should be applied then the processor will ask the user rule to apply itself to the node. The rule will then return a string that will be the name of the notification attribute that the value of this node should be placed in. If there are no user rules to be applied then the map is consulted.

The map maps either the name of a XML tag or the name of an attribute in a tag to the name of the SIENA notification attribute. The map's keys are the names of the XML tags and attributes without the namespace prefix. In the Xerces implementation of the DOM each node has a set of methods to get information about the node. One of the methods is called "getLocalName". This method returns the name of the XML tag that the node represents without the namespace prefix. This is what is used as a key into the map. If there is an entry in the map then the value returned is the name of the attribute of the notification that the value of this node should be placed in.

If neither the user rules or a mapping can be used to generate a name for a notification attribute then the processor will do nothing for the node and continue its transversal of the DOM tree.

One of the problems encountered when retrieving the value for an XML tag is that the DOM does not store the value of the tag in the node. Instead, the first child node of the current node is a text node that contains the value of the XML tag. Figure 7 shows the structure of a DOM based on some XML tags to express the problem in a pictorial way.

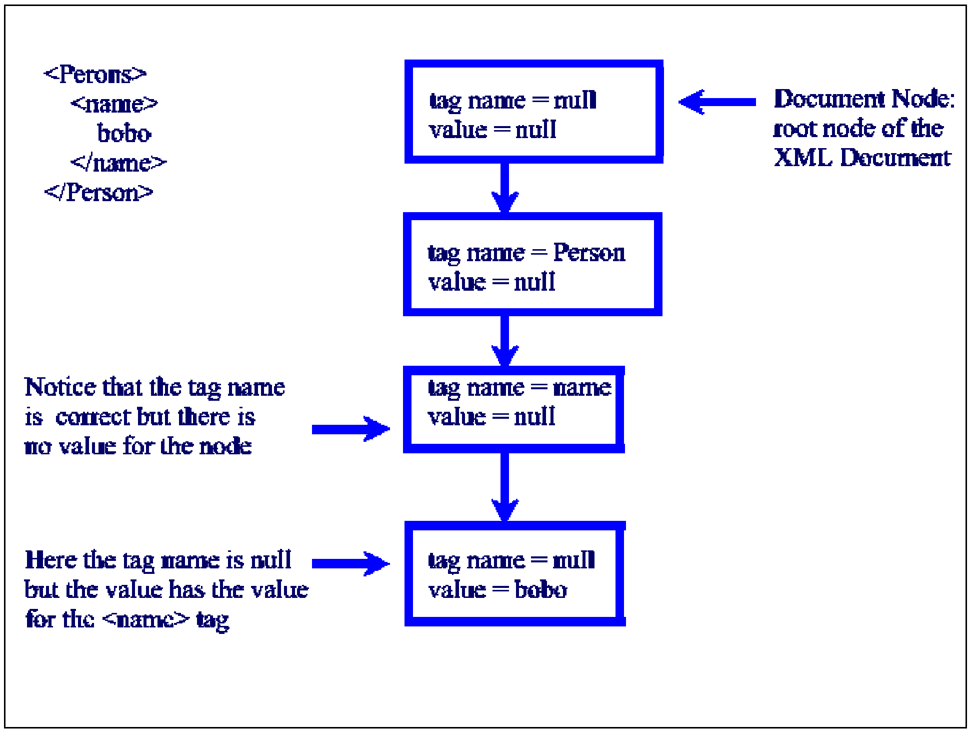

<Perons>
<name>
bobo
</name>
</Person>
tag name = null
value = null
Document Node: root node of the XML Document
tag name = Person
value = null
Notice that the tag name is correct but there is no value for the node
tag name = name
value = null
Here the tag name is null but the value has the value for the <name> tag
tag name = null
value = bobo

Figure 7: Example structure of a DOM tree

What was done to fix this problem was to process the child text node as if it was a regular node. This node has the unique property that it will not have a local name. So calling the getLocalName method will result in a null being returned. When this condition is encountered the parent node is retrieved and whatever the map returns for the parent would be applied to current text node.

The DOM's structure handles attributes of a XML tag differently from actual XML tags that are children tags. The attributes are stored as separate children from the actual XML children tags. Therefore, when walking down the tree, a check is made to see if there are any attribute nodes associated with the current node. If this check returns true then the attribute "children" are processed first before the real children of the node. Once all of the nodes have been processed a list of attributes and values has been generated. A Siena Notification object is then created populated with the information found in the list. The notification is then published.

There are times where the map will not return a name of an attribute but rather some special processing instruction. The processor checks for these cases each time it gets a value from the map for a node. There are three different types of processing instructions that are handled: "ignore", path and unique.

The "ignore" processing instruction simply informs the processor that this node and all of its children should be ignored by the processor. This is indicated by the value having the following string "IGNORE". The processor will simply skip over the node and its children and return back to the parent node to continue processing. This way a publisher can tell the processor that certain tags in the XML notification can be ignored and not put into the SIENA notification. This does not mean the subscription client would not receive these ignored tags, only that they are not included in the SIENA notification.

The path processing instruction is used to indicate the number of ancestor tag names that should be included in the attribute name. The value for this processing instruction is "PATH". The way this works is that the instruction indicates to the current node that it should append its name to the path expression. The time that the path expression is used is when the notification attribute name and its value are about to be stored in the list. The processor first appends the attribute name to the path to generate the correct notification attribute. This removes the task of finding out the names of a node's ancestors from the node itself and automatically generates the path for the children who need it.

The last processing instruction is the unique instruction. This instruction is executed whenever the attribute name returned from the map contains a dollar sign in it. If a dollar sign is found then the attribute name is broken into two parts: The part before the dollar sign, which is a name of a SIENA notification attribute; and the part after the dollar sign that is either another tag or, most likely, an attribute of the current tag, which is used to uniquely identify the tag. This processing instruction is used in the instances where there is more than one tag with the same name. This is one of the problems stated in chapter two. The problem is that the processor has to deal with the XML notification that has multiple tags with the same name in it in a way that no data is lost. What the processor does is append the first part with a slash followed by the value in the tag or attribute specified in the second part. This results in way to uniquely identify a tag in the XML notification from others that have the same name. An example of this occurring is shown.

```
<?xml version="1.0"?>
<purchaseOrder orderDate="1999-10-20">
    <shipTo country="US">
        <name>Alice Smith</name>
        <street>123 Maple Street</street>
        <city>Mill Valley</city>
        <state>CA</state>
        <zip>90952</zip>
    </shipTo>
    <billTo country="US">
        <name>Robert Smith</name>
        <street>8 Oak Avenue</street>
        <city>Old Town</city>
        <state>PA</state>
        <zip>95819</zip>
    </billTo>
    <comment>Hurry, my lawn is going wild!</comment>
    <items>
```

```
        <item partNum="872-AA">
            <productName>Lawnmower</productName>
            <quantity>1</quantity>
            <USPrice>148.95</USPrice>
            <comment>Confirm this is electric</comment>
        </item>
        <item partNum="926-AA">
            <productName>Baby Monitor</productName>
            <quantity>1</quantity>
            <USPrice>39.98</USPrice>
            <shipDate>1999-05-21</shipDate>
        </item>
    </items>
</purchaseOrder>
```

Figure 8: Purchase Order Notification with tags that have the same name.

The purchase order notification has two items that are being sent. Since the items have to be uniquely specified they will have the unique processing rule as the attribute name in the XML to SIENA map. In this case the attribute values look like this: item$partNum. The generation of this instruction will be covered later when the generic rules are being described. For now assume that somehow it was determined that the partNum attribute of the item tag works well for uniquely identifying the item. The processor would then extract the first part from the name, item, and the second part, partNum. To generate the name of the SIENA attribute the processor would concatenate the first part with a slash followed by the actual value contained in the partNum attribute of the tag. The end result would be for the first item: item/872-AA. The second one looks similar: item/926-AA. An examination of the purchase order would indicate that there are other tags with the same name yet they will not be treated the same and there are some cases where it does not matter that the XML tags have the same name. A more in-depth discussion of this will be given while describing the generic rules and how they work.

There is some special manipulation that the processor does when translating the XML notification. When a unique processing instruction is encountered the processor will create a notification attribute that contains the number of tags that have the same name for the particular instance it is working on. Using the previous example the processor would count the number of item tags that are in the notification under the same parent tag and store this value in an attribute. The name of the attribute is simply the name of the tag, in this case “item”, appended with a dollar sign and the half word “quant”: item$quant. The subscription handler when converting an XPath position predicate into a SIENA subscription uses this attribute as a test to see if the notification meets a minimum requirement of having at least *x* amount of tags, where *x* is the position specified in the predicate. This attribute will always be generated when the processor encounters a unique processing instruction, even if there is only one tag and not multiple tags in the XML notification that have the same name.

Another type of special processing that is done is when some type of numerical data is found. The processor keeps track of the maximum and minimum values for any XML tag that contains numerical data according to the XSchema. Again, using the previous example, the processor would keep the minimum and maximum values for the tags: zip, USPrice and quantity. The reason this is done is to allow XPath subscriptions that look for general information about a particular tag. For example, the XPath might request for all notifications that have items that are less than a hundred dollars in price to be returned. If the special processing was not done this type of XPath could not work, since in this example, the only way to see the USPrice of an item is to know its partNum. Then multiple constraints for the USPrice attribute would be generated for the subscription, one for each possible item. This would not work since *all* of the items would then have to have the USPrice under one hundred dollars instead of just one. With the special processing, a subscription can be generated that tests the USPrice$min attribute to see if it is less than 100.00. Item partNum is no longer needed and only one constraint is needed instead of multiple ones to handle this case.

The processor adds one last attribute to the notification that is not contained in the XML Notification itself. The name of the attribute is “_xml” and it contains the zipped text of the XML notification. This is used as a means to transmit the original XML notification to the clients subscribing to the event. Instead of forcing the subscription client to reconstruct the XML notification from the SIENA notification it was deemed easier and faster to simply gzip the XML text and store it in an attribute in the notification. Then when a subscription covers the notification the client can simply unzip this attribute and have the original XML notification in hand. It is because of this attribute that when a publisher indicates that a XML tag should be ignored that the tag is ignored only for the SIENA notification but the subscriber still gets the full, original, XML notification. Figure 9 contains a SIENA notification that was generated from the XML notification in figure 8.

```
USPrice$max=148.95
USPrice$min=39.98
_xml= … (the value is not displayed for sake of space)
billTo/city="Old Town"
billTo/name="Robert Smith"
billTo/state="PA"
billTo/street="8 Oak Avenue"
billTo/zip=95819
comment="Hurry, my lawn is going wild!, Confirm this is
electric"
item$quant=2
item/872.AA/USPrice=148.95
item/872.AA/productName="Lawnmower"
item/872.AA/quantity=1
item/962.AA/USPrice=39.98
item/962.AA/productName="Baby Monitor"
item/962.AA/quantity=1
item/962.AA/shipDate="1999-05-21"
orderDate="1999-10-20"
partNum="872-AA, 962-AA"
quantity$max=1
quantity$min=1
shipTo/city="Mill Valley"
shipTo/name="Alice Smith"
shipTo/state="CA"
shipTo/street="123 Maple Street"
shipTo/zip=90952
zip$max=95819
```

```
zip$min=90952}
```
Figure 9: SIENA Notification

XML Subscription

This section focuses on the right hand side of the architecture diagram. The diagram shows a XML Client submitting to a XML Subscription Handler (subscription handler) an XPath to use for a subscription to XML notifications. The subscription handler translates the XPath into one or more SIENA constraints using the XML to SIENA map in the process. If the user rules that the XML Notification processor used is available to the subscription client they are also applied to the XPath at this point. The subscription handler then issues a subscription that contains the constraints generated by the XPath to the SIENA servers. When a notification is covered by a subscription a SIENA notification is received by the subscription handler, which then extracts the original XML notification from it. The XPath submitted by the client is then executed against the notification to make sure that there is a true match. If the XPath processor returns at least one node then the XML notification is returned to the client, otherwise it is discarded. These processes will now be discussed in more depth.

When the subscription handler receives the XPath expression from the client it decomposes it into its individual steps. Each step of the path is then evaluated separately to determine if there is a need to generate an attribute constraint for that particular step. There may be a concern that by evaluating the steps individually information about the needed structure of the XML notification is lost. It is correct that the structure information is lost but it is not needed nor can it really be used in the creation of a SIENA subscription. The reason is that the SIENA subscription does not include any information about the structure of the XML notification except for the cases when multiple tags have the same name or when a “tag path” has been appended to an attribute name. And in these cases it is the XSchema that provides the needed information about the structure of the

XML notification and not the XPath statement. This allows each step to be handled on an individual level and not as a collective whole.

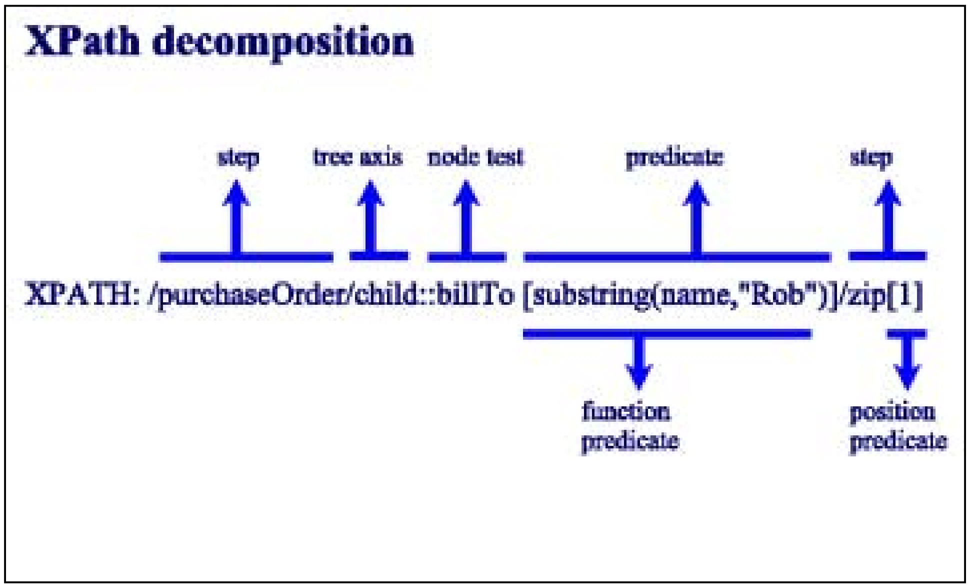


Figure 10: The different parts of an XPath

Each step is further divided into its separate parts: The tree axis specifier, the node test, and any predicates. The tree axis contains information about the structure of the XML notification that is not in the SIENA notification. Therefore, the subscription handler ignores this part completely. If the node test is not a function, such as text(), it will be the name of the XML tag. This name is used as a key to the XML to SIENA map. If a value exists for the key, the value returned would contain the name of the SIENA notification attribute that the value of the XML tag was stored in. This is the name that the attribute constraint will be constructed for. If there is not a predicate then no attribute constraint will be generated and the step ignored. Otherwise, the predicate will be used to determine the type of attribute constraint needed to satisfy the predicates. It is also at this stage that any rules that have been specified by the user are taken into account. If there are user rules, the subscription handler will check to see if a

XML tag has a rule there first before using the XML to SIENA map. Because the user rules are in a rules format some processing must be done to the rule before it can be used to create the notification attribute name. The result of the processing is a map between the current XML tag and the SIENA attribute. This is then treated as if the map was taken from the original XML to SIENA map.

Processing instructions must also be handled since they are stored in the values that are returned by the map. They are the same ones that the XML Notification Processor must deal with so only a brief description will be given here followed by the actions taken by the subscription handler.

The "ignore" processing instruction simply informs the subscription handler that this XML tag and all of its children should be ignored. This is indicated by the value having the following string "IGNORE" as its value. The subscription handler will simply move on to the next step and resume processing. The subscription handler cannot stop processing here as the next tree axis in the following step may refer to tags around this one, and not merely its children, that are not ignored. An example: If one of the generic rules stated that any comment tags should be ignored then the following XPath would result in the subscription handler having to deal with the ignore processing instruction.

```
/purhcaseOrder/comment/following-sibling::items/item
```

When the word "comment" is used as a key into the XML to SIENA map the value returned would be "IGNORE". The subscription handler would then skip over everything else in the step. If processing terminated here, then an error would be generated since according to the subscription handler there was not enough information in the XPath to create a subscription. This is not the case in this example since the steps that follow do have enough information to generate a subscription. The reason is that the next step uses the tree axis "following-sibling" to shift the focus of the XPath to tags that are not ignored. This is the reason that when an “ignore” processing instruction is encountered, the subscription handler evaluates the next step instead of terminating processing of the XPath.

The path processing instruction, indicated by the word “PATH” appearing in the value returned by the map, is used to indicate the number of ancestor tag names that should be included in the attribute name. When the subscription handler sees this instruction, it appends its name to the prefix string that is used to generate the notification attribute names. At the beginning of each step, if a tree axis is specified and its value is not child, then the last added name to the prefix must be removed, as the PATH instruction only effects the children tags of the current tag and no others.

The last processing instruction is the unique instruction. This instruction is executed whenever the attribute name returned from the map contains a dollar sign in it. When found, the attribute name is broken into two parts: The part before the dollar sign, which is a name of a SIENA notification attribute; and the part after the dollar sign that is either another tag or, most likely, an attribute of the current tag, which is used to uniquely identify the tag. What the subscription handler does is search the predicates in this step to see if the specified tag or attribute in the processing instruction exists as an operand in an equality expression. If so, then the name of the attribute is created by taking the tag name from the first part and concatenating it with a slash and the other operand in the equality expression. If the equality operation is not found then processing will resume on this step only if the other predicates deal with the position of the tag. Otherwise, the next step is processed in the XPath statement. Next is a discussion of predicates and how they are handled.

There are many different types of predicates that can be used in an XPath step. The predicates have been grouped together for ease of processing into the following categories:

- Function: This group of predicates use functions to generate results. A list of valid functions can be found in the XPath specification [5].
- Equality: These predicates test to see if two operands are equal in value. An operand can be a child node, an attribute, a String literal, or a variety of other types.
- Position: This group of predicates returns the XML node in the indicated. The position refers to the position the tag is in the set of children nodes of its parent.
- Comparison: These predicates uses some type of inequality expression, such as the less than or greater than operators, as a test.
- Compound: These predicates are made up of two or more other predicates that are combined together through the Boolean operators “and” and “or”.

The subscription handler first determines the group that the predicate specified in the step falls into. Then based on the group it creates the attribute constraint for the subscription.

When a predicate in the function group is processed the subscription handler first determines if the function can produce an attribute constraint. It does this by consulting a list containing the name of the functions that cannot be used to generate a constraint. If the function is not one of these then the arguments of the function are processed to determine what attribute or child tag is being used. If none can be found in the arguments then the function is ignored, as it does not contain any useful information that would help in constructing the subscription. Once the child or attribute name has been determined an attribute constraint based on the substring operator is created. The attribute name to be used for the constraint is the name found in the predicate appended to the prefix that was generated during the evaluation of the node test. The value to be used in the constraint is the other argument to the function. Currently only a subset of the string functions specified in the XPath specification are supported.

For the equality predicate the subscription handler determines if one of the operands is a XML tag name or an attribute name. If one of the operands is a tag or attribute the subscription handler uses the operand as the name of the SIENA attribute to apply the constraint to, after appending it to the prefix generated in the examination of the node test in the step. A constraint is created using the equals operation and the value of the other operand in the equality expression. If neither of the operands are tags or attributes then the predicate is ignored and processing continues.

There are three different ways that a position predicate can be created. The first two are by methods, the "last" method and the "position" method. The "last" method simply wants the last tag specified in the node test in the set of children of the previous tag specified in the last step. For example the XPath /purchasOrder/items/item[last()} requests that the last item in the items list should be returned. The "position" method returns the current position of the tag. It is usually combined with an equality to select a particular tag at a particular position. To get the second item in the items list an XPath would look something like this: /purchaseOrder/items/item[position() = 2]. The third method of specifying a position predicate is actually a short hand of the position method call. A simple number is used to indicate the desired position of the tag. Using the previous XPath example, the shorthand version of it would be: /purchaseOrder/items/item[2]. When the subscription handler encounters a position predicate created by the last two methods it determines the desired location for the tag first. Then the attribute name it creates for the attribute constraint is simply the tag name, as specified in the node test, append with a dot and the word “quant”. The constraint is then built using the “<=” operand with the value being the position specified in the predicate. If the predicate happens to be the method “last” then the value for the constraint is –1. The reason for this is that if predicate is applied to a XML document that contains only one tag then the “last” method should return this tag. By setting the constraint to –1 this insures that even if the XML notification contained only one of the desired tags the filter would work since the value of the “$quant” attribute would be one and hence greater than the -1.

A comparison predicate is handled in the same way as the equality predicates. The only difference is that the attribute constraint operation is not always constant but rather changes based on the predicate. This is simply handled by passing in the actual text that is in the predicate for the operator to the attribute constraint and allowing it to parse the text and determine the operation for its self. Besides this, the comparison predicate is handled the same way.

The compound predicate is found when either one of the compound statements, “and” or “or”, are found in the text of the predicate. The subscription handler handles this predicate by simply examining the joined predicates individually, generating constraints for each one. However, this only works when the compound statement is “and”. For “or” a different tactic must be taken since only one of a possible combinations need to be true. In this case the constraints that are built use the “any” operation. What this does is simply test for the existence of the attributes in the notification and not their values. When the XPath is executed against the original XML notification then the actual predicates are used to insure that the XML notification does match the subscription.

After each step the generated attribute constraints are stored in a filter. At the end of processing the complete XPath the filter is then used as a subscription by submitting the filter to a SIENA servers. If the filter is empty though, i.e. no constraints were created based on the XPath submitted, the subscription handler will return an exception to the client stating that it did not have enough information to generate a subscription based on the XPath it was given. Figure 11 contains the subscription that would be generated from the XPath shown in Figure 10 to show the different processing strategies at work.

```
filter{ billTo/name *"Rob" zip any}
```

Figure 11: SIENA subscription

When a SIENA notification is received by the subscription handler it cannot assume that the XML notification contained within the SIENA notification is a match to the XPath specified by the client. The reason is that not all of the information in an XPath expression, such as tag position or tree axis information, was used directly in the subscription. As explained previously there were things done to try and filter on some of these elements but others just could not be used. The subscription handler first retrieves the XML notification from the SIENA notification by getting the value of the _xml attribute and unzipping it. The result is the original XML notification that the publisher sent. The XPath is then evaluated using the XML notification. If the evaluation returns at least one node then the notification matches the XPath and should be returned to the client, otherwise it is not a match and will be discarded by the subscription handler. Now that the two main architectural sections have been discussed the next section deals with the common component between the two, the generic and user rules.

Rules

A special language has been developed to specify rules that are to be applied to XSchemas or XML notifications. This part will discuss the language in-depth and the different components used to process and apply the rules to schemas and notifications.

In the language a rule is defined as having four parts: The string literal “rule:”, the target, the condition, and the action. The target simply states what XML tag the rule applies to. The value for the target can be anything that is also a valid XML tag according to the XML specification. The target can also have an additional part of a ‘@’ sign followed by a name appended to the XML tag name. This indicates that the target of the rule is an attribute of the XML tag and not the tag itself. There is also a special value “any” that specifies that any XML tag or attribute is the target of the rule. Following are some examples of the possible valid targets for a rule:

- Annotation : Matches only annotation XML tags.
- item@partNum : Matches item XML tags that have an attribute named partNum. If an item XML tag exists that does not have this attribute the rule would not be applied to it since it is not the target of the rule.
- any : Matches any tag.
- any@type : Matches any XML tag that has an attribute named type.
- any@any : Matches any attribute in any tag.

The condition specifies constraints that the XML tag needs to meet in order for the rule to be applied to it. The condition is composed of an operator and a possible value to be used in the operation. The valid operations are:

- [‘<=’ | ‘>=’ | ‘<’ | ’>’] | ‘=’] *value*: These conditions check to see if the value of the XML tag or attribute return true when the specified operator is applied to them using the value as the other operand in the expression.

- present:: This condition simply checks to see if the XML tag or attribute is present in the XML document.
- notpresent: This condition checks to see if the XML tag or attribute is *not* present in the XML document.

The values for the comparison operators need to be numerical. However, the "=" and the "!=" operators can have a strings as the value as well as a name of another XML tag or attribute that have a value associated with them. When specifying another tag a tree axis should be used to indicate where the tag is located in the XML document. The valid tree axes are:

- parent: The parent tag of the current XML tag.
- sibling: Any sibling of the current XML tag.
- child: Any of the children of the current XML tag.

The condition also allows the Boolean operators "and" and "or". By using these operators a rule can specify multiple attribute constraints that the XML tag needs to meet. When using the Boolean operators only attributes can be specified and not other tags for the conditions. This prevents a constraint that tests the value of the XML tag and the value of one of its attributes. The "and" operator is used when all of the attribute constraints need to be met before the rule can be applied to the XML tag. The "or" operator specifies that one or more of the constraints need to be true in order for the rule to be applied to the XML tag. Examples of valid conditions are listed below:

- comment present: The condition is for the comment XML tag to be present.
- any@type = "xsd:string" : The condition is an equality that checks to see if the type attribute in any XML tag has the value "xsd:string".

- any@type = sibling any@type : This condition checks to see if the type attribute in any XML tag is equal to the type attribute of any of the current XML tag's siblings. In essence, do two XML tags with the same parent have the same type?
- any and { @maxOccurs = 1 @minOccurs = 1 @fixed notpresent @use = "required"} : This condition checks multiple attributes on any XML tag. All of the conditions inside have to be true before the rule will be applied to the XML tag being processed. In this case the attributes "minOccurs" and "maxOccurs" must equal one, the attribute "fixed" should *not* be present and the attribute "use" should have the value required.

The action states what should be done to the XML tag. The following actions are currently valid:

- ignore Ignore the tag and not include it in the notification.
- group:*name* This states that multiple instances of this tag should be grouped together under the attribute with the name *name*. Note: this is exactly what the *name* action does. This is included to help human readers understand the rule.
- path This action indicates that the XML tag should have its named added to a prefix that is used to generate the names of the SIENA attributes.
- unique:*target* This specifies that the current XML tag or attribute can be used to uniquely identify the tag specified by *target*.
- *name* This is the name of the notification attribute that the value of this tag should be stored in. This is generally the action most rules will have. For example, this rule states that the value of the XML tag should be stored in a notification attribute called names:

```
rule: name present names.
```

There is another special key word that can be used in actions: currentTag. This keyword informs the action that what ever the current XML tag is when the rules is invoked should have its name substituted for currentTag when processing the action. This allows the delaying of the naming of a notification attribute for an XML tag to the time that an actual XML notification is being processed and not during the earlier time when the XSchema is being processed. Generally the "any" rules, rules that have the "any" keyword as the rule target, use this keyword and not the other rules since the XML tag is already know when the other rules are being executed.

Figure 9 shows the generic rules that are applied to any XSchema that is given to the XML Client interface. Most of the rules make heavy use of the "any" target because the exact structure and tag names are unknown to these rules when they are applied to an XSchema. Figure 10 shows some rules that a user could submit to the XML Client to prevent any XML comment tags being placed in the SIENA notification or to specify where to store the value of an attribute.

```
rule: any@type = "xsd:string" group:currentTag
rule: any@type = "xsd:date" group:currentTag
rule: any@type = sibling any@type path:1
rule: any@fixed present ignore
rule: any present group:currentTag
rule: annotation present ignore
rule: attribute and {
  @maxOccurs = 1,
  @minOccurs = 1,
  @fixed notpresent,
  @use = "required"
} unique:currentTag
rule: attribute and {
  @maxOccurs notpresent,
  @minOccurs notpresent,
  @fixed notpresent,
  @use = "required"
} unique:currentTag
```

Figure 12: The definition of generic rules to be used to process any XSchema

```
rule:productName present group:productName
rule:comment present ignore
```

Figure 13: Example of possible rules that a user of the client could submit

Appendix A contains the grammar for the rules language and also the valid tokens for the language. The descriptions are written in EBNF. Appendix B is a short guide on how to write rule definitions.

Once the rules have been defined the next task is to generate the XML to SIENA map. A component called the XSchema Processor is responsible for this task. This component is located on both sides of the architecture. When built, the XSchema Processor (processor) tries to load the rules from a file. The file name is a static property of the processor and is currently set to “default.rules”. To load the rules a parser built from the rules grammar is created to read in the contents of the file and return a hash map that contains a mapping of a target name to one or more rules that have the same target. The key does not include the name of the attribute, only the name of the XML tag. It is left up to the rule to check for the optional attribute target. Once loaded the hash map is returned to the processor, which then waits for an XSchema file to be submitted.

The “default.rules” file contains rules that are to be used as basic guidelines in generating the XML to SIENA map. Basically, the rules specified check for the following:

1) If a tag has a type of string or date, then multiple instances of this tag can be grouped under an attribute with the same name as the tag name.

2) If a tag has a sibling with the same type as it, then its children will need to include its name as part of the SIENA attribute name.

3) Ignore any XSchema annotation tags and its children.

4) If an attribute is designated as fix, it can be ignored.

5) If an attribute is designated as required and can only appear one time this attribute should be considered as a way to make a tag unique.

These are only basic guidelines in attempt to allow broad range of XML notifications to be translated into SIENA notifications. Because they are only basic guidelines there may be instances where more detailed rules need to be also used. The user rules fill this role, which will be talked about later.

When the processor is given an XSchema file it then uses a XML parser to parse the file and create a DOM for it. The processor walks through the DOM depth first, using the local names of the node as a key into the map returned by the rules parser. If there is a match in the map for the key then the processor takes the value returned and extracts one or more rules from it. Then for each rule the processor sees if the rule should be applied to the XML tag by asking the rule to check its conditions against the node. If the rule returns true then the processor knows that it should apply the rule to the node. The rule is asked to apply itself to the node and when it is finished return a string. This string will contain one of two things: A notification attribute name or a processing instruction. The rule determines what is returned by what action was specified in the rule.

If the rule has a group action, the tag specified after the group key word is returned. For ignore actions the string "IGNORE" is returned. The unique action will return a string that is the combination of the tag specified after the unique key word appended with a dollar sign and the target for the rule. For the path action the string "PATH" is returned. Otherwise the value in the action string is returned.

Once the rule has returned a string then an entry into the XML to SIENA map is made consisting of the key, which is the local name of the current node, and a value, which is the string returned by the rule after it applied itself to the node. By the time the processor finishes waling through the DOM tree a complete map will have been created that takes an XML tag and returns either a processing instruction or notification attribute name. At this point the processor is finished and stores the map internally. Figure 14 displays a map that would be generated from processing the XSchema in figure 4, duplicated here for ease of reading.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:annotation>
  <xsd:documentation xml:lang="en">
   Purchase order schema for Example.com.
   Copyright 2000 Example.com. All rights reserved.
  </xsd:documentation>
 </xsd:annotation>

 <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

 <xsd:element name="comment" type="xsd:string"/>

 <xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
   <xsd:element name="shipTo" type="USAddress"/>
   <xsd:element name="billTo" type="USAddress"/>
   <xsd:element ref="comment" minOccurs="0"/>
   <xsd:element name="items"  type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
 </xsd:complexType>

 <xsd:complexType name="USAddress">
  <xsd:sequence>
   <xsd:element name="name"   type="xsd:string"/>
   <xsd:element name="street" type="xsd:string"/>
   <xsd:element name="city"   type="xsd:string"/>
   <xsd:element name="state"  type="xsd:string"/>
   <xsd:element name="zip"    type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"
     fixed="US"/>
 </xsd:complexType>

 <xsd:complexType name="Items">
  <xsd:sequence>
   <xsd:element name="item" minOccurs="0"
maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
```

```
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
         <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
      <xsd:element ref="comment"   minOccurs="0"/>
      <xsd:element name="shipDate" type="xsd:date"
minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU" use="required"/>
    </xsd:complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:complexType>

 <!-- Stock Keeping Unit, a code for identifying products -->
 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
 </xsd:simpleType>

</xsd:schema>

The map generated using the generic rules:

USPrice=USPrice, comment=comment, USAddress=USAddress,
shipTo=PATH, orderDate=orderDate, state=state,
productName=productName, shipDate=shipDate, quantity=quantity,
street=street, items=items, SKU=SKU,
purchaseOrder=purchaseOrder, country=IGNORE, city=city,
billTo=PATH, PurchaseOrderType=PurchaseOrderType, Items=Items,
zip=zip, item=item/$partNum, name=name
```

Figure 14: XML to SEINA map generated by an XSchema and generic rules

Chapter 4

IMPLEMENTATION

Core Package

The implementation of the interface is contained in the base package siena.xml. There are two other sub-packages under this one that contain specific components that are used by the main classes in the base package. There is a third package that contains a GUI interface used as an example of how to use the XMLClient object. The base package will be examined first followed by the two sub-packages. The gui sub-package will not be explained as it dose not contribute to the XML client interface.

Figure 15 is a class diagram of the classes contained in the base package. There are three main classes that implement the XML interface to SIENA. The three classes are:

- XMLProcessor: This class handles the translation of a notification into a SEINA notification.
- XMLSubscriptionHandler: This class deals with the XPath translation into a SEINA subscription and the testing of the XPath against the XML notification that is received from a server.
- XMLSchemaProcessor: This file handles the loading of the generic rules and creating the XML to SIENA map used in the translation processes.

The XMLClient class is an extension to the siena.ThinClient class that can be used as the interface to SIENA. It adds two methods that allow a client to publish XML Notifications without having to know about the translation process and to subscribe to events using XPath as the subscription language. The other class, DOMParserWrapper, is a class wrapper around the Xerces DOM parser. This class is used to parser a XML document and creates a DOM tree from it.

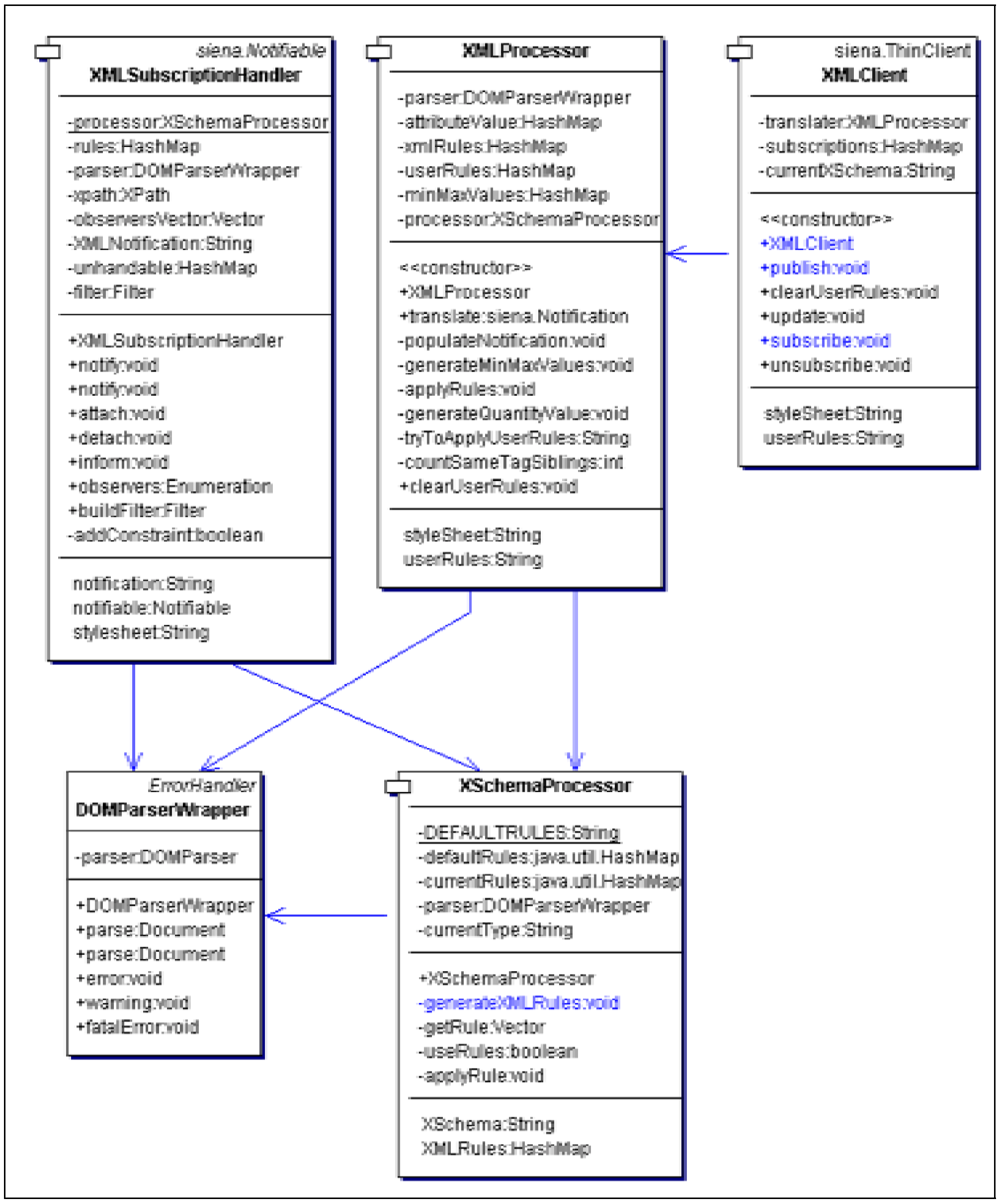


Figure 15: Class diagram of the classes in the base package.

The XMLProcessor has five public methods for the use of external programs. The methods that have to be used in order for an XML notification to be correctly translated into a SIENA notification are: “setStyleSheet” and “translate”. The “setStyleSheet” method tells the processor the location of the file that contains XSchema of the XML notification. Not calling this method will result in an exception being thrown by the translate method with the error message of: “XSchema has not been specified. Processing of the XML notification can not be done without it.” Once this method has been called the XMLProcessor will tell the XSchemaProcessor the location of the XSchema. The XSchemaProcessor will then generate a HashMap that maps an XML tag to a SIENA attribute name or a processing instruction. The XMLProcessor will store this table internally and use it while executing the “applyRules” method. The translation method is used to translate the XML notification, sent in as a String object, into a SIENA notification, which is returned. Figure 16 contains a condensed version of a sequence diagram of this method.

The method first calls the method applyRules to apply the map generated by the XSchemaProcessor to the XML tags and also to apply any user rules that may have been set by the method setUserRules.

The applyRules is a recursive method that works on the nodes of the DOM tree. The arguments to the method are a node to be processed and a prefix that should be used when creating the final name of the notification attribute. This method works on all of the nodes in the DOM, including the attribute nodes and the special text nodes. The first thing the method does is check to see if a user rule can be applied to the node that was passed in. In the method that handles the processing of the user rules a check is made to see if two user rules can be applied to the same node but produce different mappings. If this happens the method will print out to a log file that XML tag the rules can be applied to and the rules that could be applied. The last rule found in the user rules file is the one that will be applied to the node.

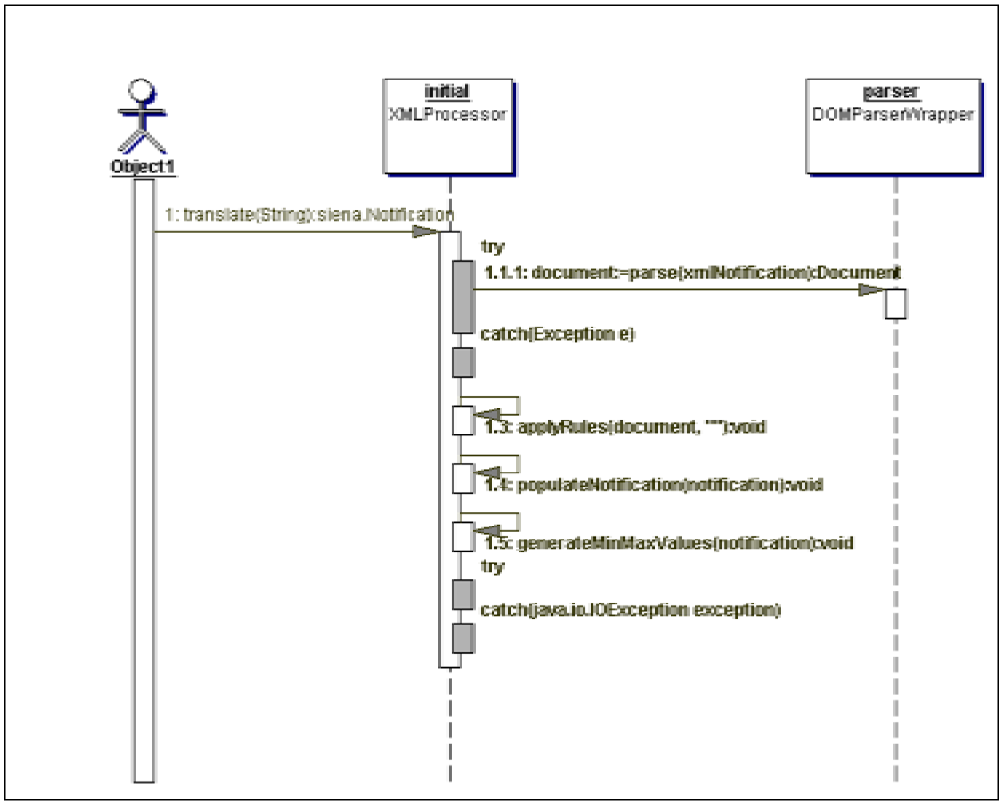

Object1
initial
XMLProcessor
parser
DOMParserWrapper
1: translate(String):siena.Notification
try
1.1.1: document:=parse(xmlNotification):Document
catch(Exception e)
1.3: applyRules(document, ""):void
1.4: populateNotification(notification):void
1.5: generateMinMaxValues(notification):void
try
catch(java.io.IOException exception)

Figure 16: Sequence diagram for the translate method

If so, the XMLProcessor has the rule apply itself to the node and the return value is used as the name of the notification attribute. The map is still consulted for an attribute name even if a user rule was found that did apply to the node. This is done to check to see if a user rule defines a different mapping from an XML tag to a SEINA name attribute than a default rule. If this is the case then the method will print out a warning to a log file stating that the user rule over rides a default rule. The user rule will always be used in this case. If a value was found for the XML tag in the map or the user rules then it is checked to see if it is one of the special processing instructions. If so the XMLProcessor deals with the processing instruction accordingly. If the node has a value then the value of the node is then associated with the notification attribute name and stored in the attributeValue HashMap. This HashMap associates a Vector of values with a String that is the name of the SIENA notification attribute. If there is not a value for the node then the method checks to see if the node and its children are to be ignored. If so the method is done and returns otherwise the attribute nodes and the children nodes are processed by calling the applyRules method on them.

The next method called after the applyRules method is finished is populateNotification. This method takes in a notification as an argument and proceeds to fill in the notification with the names and values of the notification attributes contained in the attributeValue map. After this the generateMinMaxValues method is called to store in the notification the minimum and maximum values of the XML tags that have numeric data for values. After these two methods have been called the SIENA notification has been created and is almost ready to be sent. The last thing that the translation method does is to build an ObjectOutputStream on top of a GZIPOutputStream on top of a ByteArrayOutputStream. Then the XML notification String is written to the object output stream that results in the notification being compressed and stored into a byte array. This byte array is then stored in the SIENA notification under the "_xml" attribute name. After this the notification is returned back to the caller of the method.

The XMLSubscriptionHandler is the class that handles the translation of the subscriptions and checking to see if a notification received from SIENA matches the XPath supplied by the client. The class was designed so that there could be multiple instances of the class running at the same time. Each instance would be in charge of a single subscription. In order to intercept the SIENA notifications the XMLSubscriptionHandler registers itself with the SIENA servers as the object to be notified. When notified the XMLSubscriptionHandler will then check the XML notification against the XPath to see if it returns at least one node. If so the XMLSubscriptionHandler informs the client about the XML notification. The observer pattern is used with the XMLSubscriptionHandler class being the subject. The observers are the clients who are submitting the XPath subscriptions to the XMLSubscriptionHandler. The client must register itself with the XMLSubscriptionHandler by calling the "attach" method, passing itself in. When a XML notification is received the XMLSubscriptionHandler calls the "inform" method that calls the "update" method on all of its observers. The "update" method takes a single parameter of type XMLSubscriptionHandler. The XMLSubscriptionHandler will pass itself as the parameter to the update method. This allows multiple clients to be informed about the same subscription.

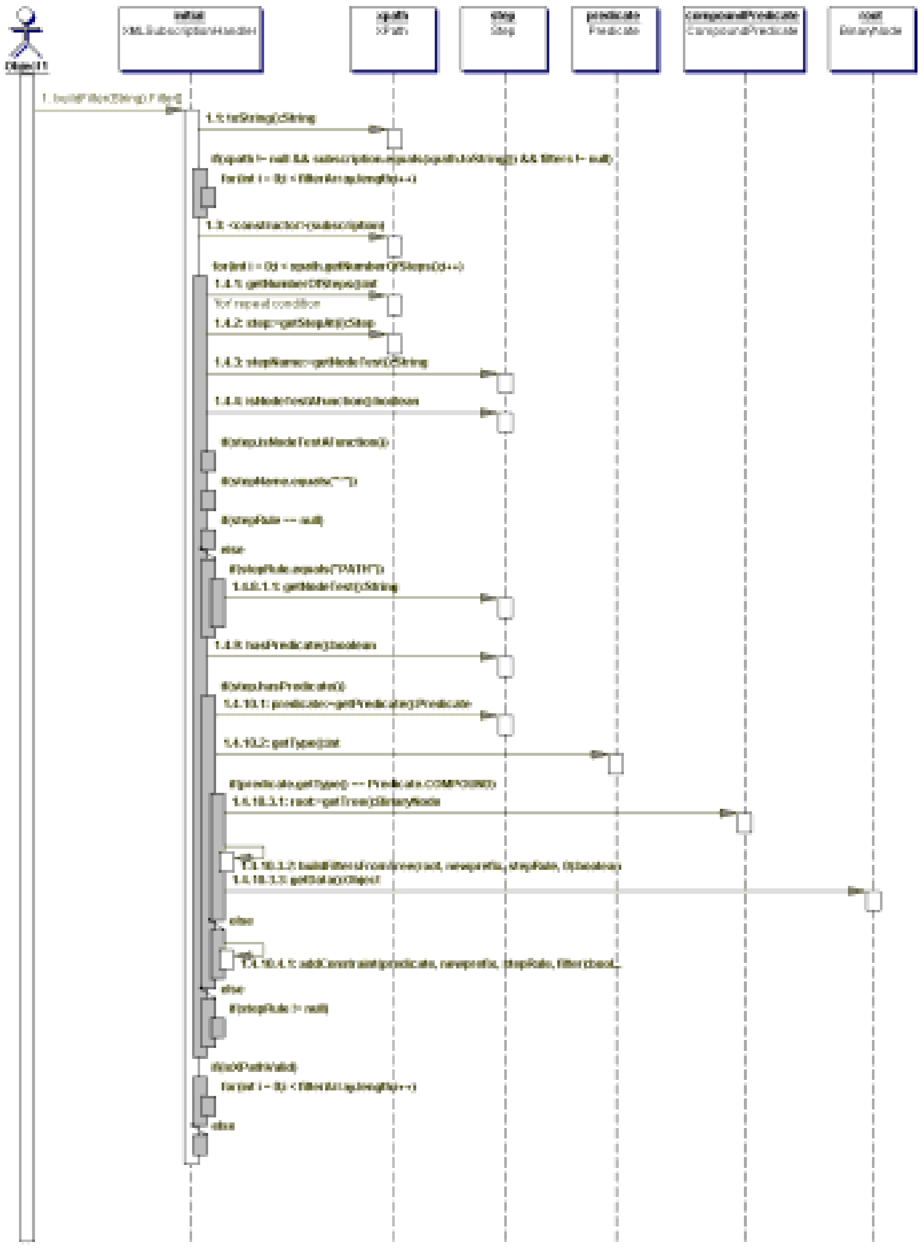

Object1
initial
XMLSubscriptionHandler
xpath
XPath
step
Step
predicate
Predicate
compoundPredicate
CompoundPredicate
root
BinaryNode
1: buildFilter(String):Filter[]
1.1: toString():String
1.3: <constructor>(subscription)
1.4.1: getNumberOfSteps():int
'for' repeat condition
1.4.2: step:=getStepAt(i):Step
1.4.3: stepName:=getNodeTest():String
1.4.4: isNodeTestAFunction():boolean
if(stepRule == null)
else
if(stepRule.equals("PATH"))
1.4.8.1.1: getNodeTest():String
1.4.9: hasPredicate():boolean
if(step.hasPredicate())
1.4.10.1: predicate:=getPredicate():Predicate
1.4.10.2: getType():int
if(predicate.getType() == Predicate.COMPOUND)
1.4.10.3.1: root:=getTree():BinaryNode
1.4.10.3.3: getData():Object
else
else
if(stepRule != null)
if(isXPathValid)
else

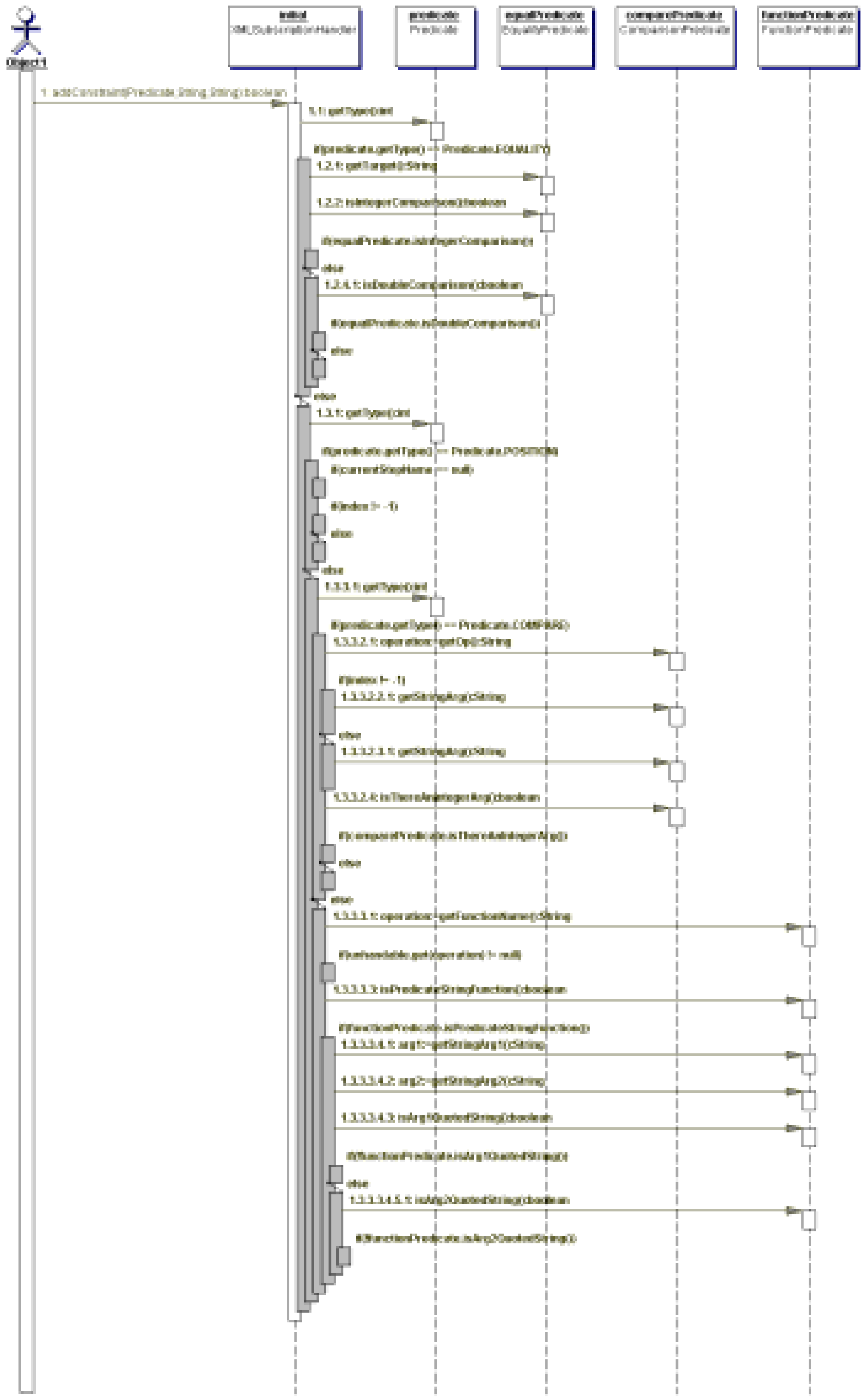
Figure 17: Sequence diagram for the buildFilter method

The method that is invoked to build the subscription is the buildFilter method. A sequence diagram of this method is show in figure 17. The method takes a String containing an XPath statement as an argument and returns a filter based on the XPath. A SienaException can be thrown by the method.

The method first checks to see if the last filter created could be used for the XPath passed in. This is a simple check to see if the XPath passed in as an argument is the same stored in the class and if the filter is not equal to null. If these conditions are met then the filter is returned and the method finishes otherwise an XPath object is created from the argument passed in. Then for each step in the XPath the node test is retrieved and checked for any processing instructions. For the path instruction the method appends the name of the step to the prefix string. After the processing instructions are handled a check is made to see if there are any predicates for the step. If so the method checks to see if the predicate is either a compound predicate or a regular predicate. The compound predicate has to be handled differently because it has multiple predicates that need to be translated into attribute constraints for the subscription.

To handle the compound predicate the method invokes another method called buildFiltersFromTree. This method takes as one of its arguments a BinaryNode object. Using this object as the root of a tree the method will walk the tree depth first. When the method hits a leaf it simply converts the predicate into a filter calling the addConstrint method. It then stores this new filter into a vector in the node's user data section. When the method is processing a node in the tree then it takes two different types of actions based on the type of node. If the node represents an "or" Boolean operation then the method will take the filters from the children of this node and simply add them together. The new set of filters is then stored in the current node's user data section. In this way filters are propagated from bottom of the tree to the top. If the node represents an "and" Boolean operation then all possible combinations of the children filters is produced. The new set of filters is then stored in the current node's user data section. When the method finishes the node that was passed in contains a vector that contains all of the filters that should be issued to the SIENA server in order to cover all the possible combinations that the compound predicate covers.

The translation of a predicate happens in the "addConstraint" method for which the sequence diagram is shown in figure 18. This method will return true only if a constraint was added to the filter. The return value is combined with a result Boolean using a logical "or" operation. After all the steps for the XPath have been examined the "buildFilter" method checks to see if the result Boolean is true or not. If the value is true then the method returns an array of filters. If the value is false then an exception is thrown stating that there was not enough information in the XPath subscription to generate a SIENA subscription.



Object1
initial
XMLSubscriptionHandler
predicate
Predicate
equalPredicate
EqualityPredicate
comparePredicate
ComparisonPredicate
functionPredicate
FunctionPredicate
1: addConstraint(Predicate,String,String):boolean
1.1: getType():int
if(predicate.getType() == Predicate.EQUALITY)
1.2.1: getTarget():String
1.2.2: isIntegerComparison():boolean
if(equalPredicate.isIntegerComparison())
else
1.2.4.1: isDoubleComparison():boolean
if(equalPredicate.isDoubleComparison())
else
else
1.3.1: getType():int
if(predicate.getType() == Predicate.POSITION)
if(currentStepName == null)
if(index != -1)
else
else
1.3.3.1: getType():int
if(predicate.getType() == Predicate.COMPARE)
1.3.3.2.1: operation:=getOp():String
if(index != -1)
1.3.3.2.2.1: getStringArg():String
else
1.3.3.2.3.1: getStringArg():String
1.3.3.2.4: isThereAnIntegerArg():boolean
if(comparePredicate.isThereAnIntegerArg())
else
else
1.3.3.3.1: operation:=getFunctionName():String
if(functiontable.get(operation) != null)
1.3.3.3.3: isPredicateStringFunction():boolean
if(functionPredicate.isPredicateStringFunction())
1.3.3.3.4.1: arg1:=getStringArg1():String
1.3.3.3.4.2: arg2:=getStringArg2():String
1.3.3.3.4.3: isArg1QuotedString():boolean
if(functionPredicate.isArg1QuotedString())
else
1.3.3.3.4.5.1: isArg2QuotedString():boolean
if(functionPredicate.isArg2QuotedString())

Figure 18: addConstraint sequence diagram

The “addConstraint” method takes as arguments a Predicate object, a String containing a prefix to be applied to the notification attribute name, and a String containing the name of the current step, which is the node test. The method first determines the type of predicate being processed using the “getType” method in the Predicate class and the predefined types. Once the type of predicate has been determined the Predicate object passed in as one of the arguments is cast into its subtype for ease of processing.

For equality predicates the method builds a new constraint based on the type of equality. The type is can be a double, or an integer, or a string and is based on what used as operands in the expression. The name of the target tag or attribute is retrieved from the EqualityPredicate object and used as the name of the SIENA notification attribute to apply the constraint to. If the tag returned is a unique processing instruction then the base part, everything before the dollar sign, is extracted and used as the name of the SIENA attribute after it is appended with the string “$quant”.

When a position predicate is found the method checks to see if the current step name is empty. If so the method returns false because the position predicate needs to know the current step name to generate the name of the notification attribute to build the constraint for it. If the step name contains a dollar sign then the method will generate the attribute name by taking the name before the dollar sign and appending it with the string “$quant”. An attribute constraint is built using the “greater than or equal to” operation and setting the value to the position specified in the predicate. If the step dose not contain the dollar sign then the attribute constraint built uses the “any” operation and inserts the string “any” as the value to check for. This value is ignored because of the “any” operator that is being used. The name of the notification attribute is set to be the same as the current step name.

The comparison predicate is handled by first checking to see if the comparison is a greater then or lesser than operation. If it is a greater than operation then the attribute name is built from the string argument retrieved from the comparison predicate appended with the string “$min”. For the lesser than operation the attribute name is appended with the string “$max”. The minimum value for an attribute is checked for the greater than operation because if the lower limits of the range of values for the notification attributes is greater than the value supplied in the predicate then all of the values of any of the XML tags being searched for will be greater than the value submitted. The predicate then will never be satisfied by the XML notification. The same logic applies to the less than operation being compared with the maximum value of the attribute. Once the notification attribute name has been created the method determines if a comparison to an integer is being made. If so the constraint is built using the operation and the integer argument from the predicate. Otherwise the double argument is used instead of the integer argument. The prefix string is cleared at the end of the method because the min/max attributes generated by the XMLProcessor consist only of the name of the XML tag and no prefixes. If the prefix was not cleared then it would be applied to the notification attribute name generated by this method and create an attribute name that would possible not exist in the SIENA notification.

For the function predicates the method checks to see if the function is a string function. If so, the method creates an attribute constraint using the substring operator. The method checks to see which argument was the literal string, referred to as the quoted string in the code, and which one is the name of the tag or attribute the predicate is working on. The literal string is supplied to the constraint as the value to look for in the substring expression while the other argument is used to create the name of the notification attribute. For other functions no attribute constraint is generated because it may not make sense or be possible to convert them into a constraint.

At the end of the addConstraint method the generated notification name and attribute constraint are added to the filter and a true value is returned unless the notification name is null. In this case false is returned.

In the case where a step does not have a predicate the buildFilters method will see check to see if the step has a rule/mapping associated with it. If so then a filter is created where the name of the SIENA attribute is whatever was mapped to the XML tag and the value looked for is set to “any”. This allows paths that do not have predicates to still work as subscriptions.

The only other method of interest in the XMLSubscriptionHanlder is the “notify” method. This method receives a SEINA notification as an argument that matches the subscription generated by the “buildFilter” method. The notify method extracts the “_xml” attribute and proceeds to unzip the contents of the attribute to generate the original XML notification. Then using the Xalan XML engine by Apache[9] the method checks to see if the XML Notification satisfies the requirements specified in the XPath which was used in the creation of the subscription. If so, the XMLSubscriptionHandler invokes the “inform” method to inform all of its observers about the notification. Otherwise the notification is discarded and the method finishes.

The other main class in the core package is the XSchemaProcessor. There are only three public methods in the class: the constructor, setXSchema, and getXMLRules methods. When the constructor is invoked the XSchemaProcessor loads the generic rules by creating a siena.xml.rules.RulesGenerator object and passing it a FileStreamInput that is connected to the file specified in the static attribute DEFAULTRULES. The RulesGenerator object will generate a HashMap that will contain a map between a target XML tag and a vector of siena.xml.rules.Rule objects that can be applied to the target. The XSchemaProcessor stores the HashMap for latter use. The constructor also loads a set of rules that are used to identify the types being defined by the XSchema. The RulesGenerator object is built again but is loaded with a different set of rules that are located in the file named in the VALUERULES attribute. The hash map returned is also stored for later use. The setXSchema method takes the name of the file that contains the XSchema and builds an org.sax.InputSource object that is connected to the XSchema file. The method then invokes the parse method in the DOMParserWrapper object to parse the XML and generate the DOM for it. The method returns the root node of the DOM. The method generateXMLRules is called passing it the root node of the XSchema.

The generateValueInformation method is used to determine what types will *not* have a value associated with them. Simply checking to see if the type is defined as a complex Type does this. However, if the definition of the complex type has the mixed attribute set to true in it, then it will hold values and will not be included in the list of types that do not have values. This list is then used in the generateXMLRules method to determine if the XML tag to SIENA name attribute mapping should be stored.

The generateXMLRules method is a recursive method that has the following arguments: a Node to examine to see if a rule can be applied to it and a string that contains the name of current type being processed. The method will walk through the DOM depth first in search of nodes that have rules that can be applied to them. The first thing done is to determine the current type being processed. This is based on the name of the of XSchema tag “complexType”. The name attribute of the tag is used as the name of the current type. There is a case when a complexType tag will appear after an element tag. The complexType will not have a name attribute in this situation. To get the current name of the type the parent element tag’s name attribute is used instead. The method does this by checking all of the element tags and storing the value of the name attribute into a String called newCurrentType. Then the method checks for the complexType tag and try to set the newCurrentType value to the value of the attribute name. By doing the processing this way the method is able to know the name of the parent element if the complexType dose not have a name attribute.

Once the type is determined the method then checks a list to see if the XML tag being process will have a value or not. If it will not have a value then no more processing is done and the children of the DOM tree node are processed. Otherwise the method will try to retrieve a rule for the node. The method “getRule” is invoked, which takes the current node as an argument. The method will then call the “getLocalName” method of the node and use it as a key into the map that contains the generic rules. The “getRule” method returns whatever the map returns. This can be a vector object or null.

If a vector was returned then the “generateXMLRules” method will call the method “useRules” to apply the rule to the node. The “useRules” method takes the vector and node as arguments. In the “userRules” method each element, which is a Rule object, is extracted from the vector using a for loop. The rule object’s method “doesRuleApplyToNode” is invoked, passing it the node. If this method returns true then the “applyRule” method is called in the XSchemaProcessor and a true value is returned. Otherwise the loop starts its next iteration. If the for loop is able to finish then the method returns false to indicate that no rules in the vector could be applied to the node.

If the “useRules” method returns false the “generateXMLRules” method retrieves the set of rules that can be applied to any tag using the string “any” as a key into the generic rules map. This will return a vector containing the rules that have been designated as applying to any tag. The method then calls the “useRules” method again, passing in the any rules vector. After this the “generateXMLRules” method retrieves all the children from the current node and calls it self one time for each child node, passing in the child node as an argument.

The “applyRule” method main purpose is to handle the unique processing instruction. First the rule’s “apply” method is invoked, passing it the node. The valued return from this method is stored in the variable attributeName. If this variable is null, the method returns without doing anything else, otherwise it checks for the index of the dollar sign. If the index is not –1 then the substring from index 0 to the index of the dollar sign is extracted and stored in a variable called xmlTagName. If the value of the substring is equal to “currentTag” then the method substitutes the value in the xmlTagName with the value in the currentType attribute. Then a “put” operation is used on the currentRules HashMap with the xmlTagName as the key and the value the concatenation of the xmlTagName, a slash, and the substring of the attributeName starting at the dollar sign index to the end of the string.

If the index was –1, then the “applyRule” method attempts to get the value of the attribute name from the node. If the name attribute exists then it is used as the key for the “put” operation to the currentRules. The value that is associated with the key is the value of the attributeName variable. If a name attribute does not exist then the method simply returns.

By the time the generateXMLRules method finishes with all of its recursion the currentRules HashMap will contain a mapping between a XML tag and a SIENA attribute. This is the map shown in the architecture with the name XML to SIENA map.

Rules Subpackage

This sub-package contains the classes that define what a rule is. The main class is the Rule class with the other classes taking the roles of supporting or utility classes. Figure 19 shows the class diagram for this package. Here are the classes and a brief description of each

- Rule: This class defines what a rule is and how it works
- Condition: An interface that contains a method to check to see if a Node object meets a condition.
- ExistsCondition: This class is a condition that checks for the existence, or nonexistence of a XML tag or attribute.
- EqualityCondition: This class is a condition that checks to see if two strings are equal.
- NumberEqualityCondition: This is a condition class that checks to see if two numerical operands are equal.
- InequalityCondition: This compares two numbers and returns true depending on the operation and the values of the numbers. The two valid operations are: less than and greater than.
- RulesGenerator: This class uses the RulesParser to parse an input stream and produce a HashTable of rules. The HashTable maps the target of a rule to the rule object. This class is generated with the Antlr utility.
- RulesParser: A parser generated from the rules grammar (see Appendix A) by the Antlr[10] utility.

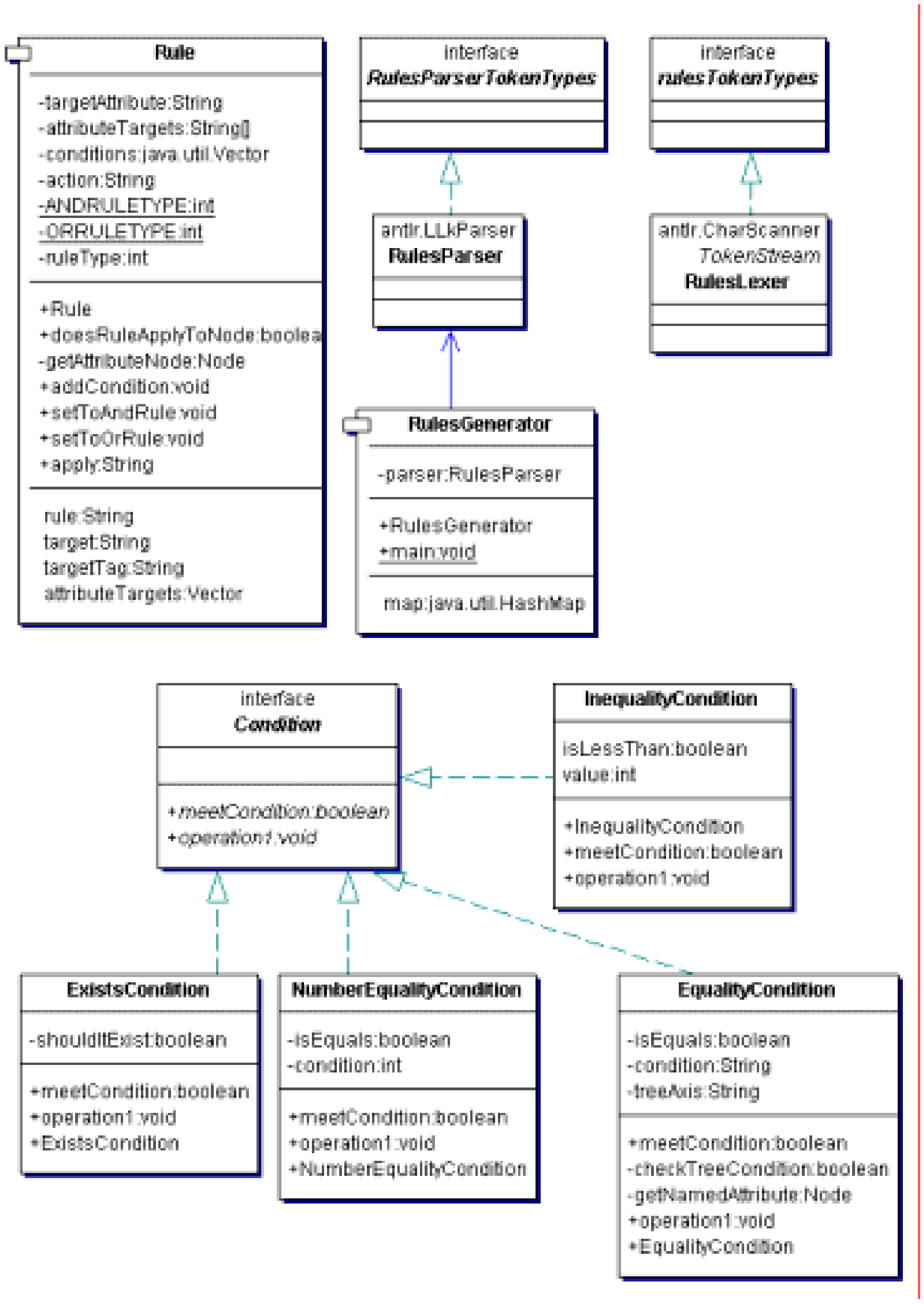


Figure 19: Class diagram for the rules sub-package.

- RulesLexer: The lexer for the rules language. Generated by Antlr [10]using the rules grammar (ass Appendix A).
- rulesTokenTypes: Support class for the RulesLexer class that contains the tokens for the rules language. Generated by Antlr[10].
- rulesParserTokenTypes: A support class for the RulesParser class. Generated by Antlr[10].

Five of the classes in this package where automatically generated by the tool Antlr[10]. Antlr is a software package that generates parsers and lexers defined by a grammar. In this case the grammar is for the rules language. The only generated class that is used in the interface is the RulesGenerator class. This class hides the details of the parser and lexer and provides a convenient way of getting a HashMap of rules from a file.

There are two important methods in the Rules class: the "doesRuleApplyToNode" method and "applyRule" method. These two methods control what nodes in the DOM the rule is applied to and how the rule is applied to the node. Figure 20 shows the sequence diagram for the "doesRuleApplyToNode" method.

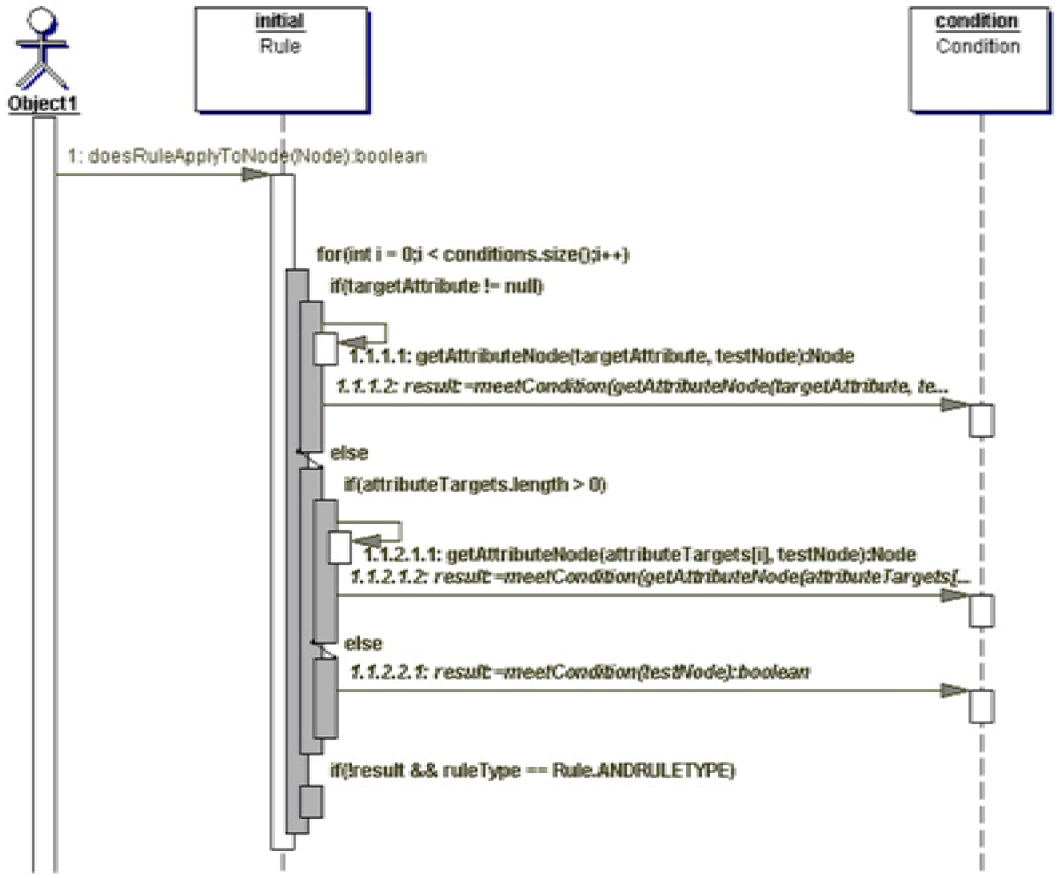

Object1
initial
Rule
condition
Condition
1: doesRuleApplyToNode(Node):boolean
for(int i = 0;i < conditions.size();i++)
if(targetAttribute != null)
1.1.1.1: getAttributeNode(targetAttribute, testNode):Node
1.1.1.2: result =meetCondition(getAttributeNode(targetAttribute, te...
else
if(attributeTargets.length > 0)
1.1.2.1.1: getAttributeNode(attributeTargets[i], testNode):Node
1.1.2.1.2: result =meetCondition(getAttributeNode(attributeTargets[...
else
1.1.2.2.1: result =meetCondition(testNode):boolean
if(!result && ruleType == Rule.ANDRULETYPE)

Figure 20: Sequence diagram for the doesRuleApplyToNode method

The “doesRuleApplyToNode” method starts off by first checking to see if the local name of the node is the same as the target for the rule or if the target is “any” for the rule. If either case is true the method continues on to check to see if the node meets the conditions for the rule. Otherwise false is returned to the caller of the method.

The Rule object has two attributes that effect the execution of the condition. One attribute is called the targetAttribute and the other one is called attributeTargets. Only one of these attributes can be set to a value for a rule. The first attribute is set if the target of the rule specified an attribute. The rule definition would look like the following:

rule: any@type = “xsd:string” group:CurrentTag

In this case the targetAttribute would be set to “type” and the target would be set to “any”. This is done in the RulesParser object. When the “doesRuleApplyToNode” method is evaluating a condition and the targetAttribute is not null then the method retrieves the attribute node that has the name specified in the targetAttribute attribute. The attribute node then is passed in to the “meetCondition” method instead of the original node that was passed in.

The attributeTargets attribute is an array of string that contains the names of all the attributes that were listed in a compound condition. A compound condition specified in a rule looks like the following:

rule: any and { @maxOccurs = 1 @minOccurs = 1 @fixed notpressent } ….

The attributeTargets attribute would then have the following values: maxOccurs in element zero, minOccurs in element one, and fixed in element two. If this attribute is not null during the processing of the conditions then the method will get the attribute node that has the same name of the element in the array indexed by the for loop count variable. This is done so that the correct condition is matched with the correct attribute it targets. Using the rule above as an example the conditions would be evaluated in the following order: “= 1”, “= 1”, and “notpresent”. The names of the attribute in the array are in the exact same order as the conditions, which allows the for loop counter work as the index for both the conditions and the attribute names.

If neither of the two attributes, targetAttribute or attributeTargets, are set then the node that was passed in is used as the argument to the “meetCondition” method. At the end of the “for” loop there is an “if” statement which checks to see if the result is false and if the type of rule is an “ANDRULETYPE”. If this is true the method will return false since the rule type indicates that all of the conditions must be meet in order for the rule to apply to the node. At the end of the method it simply returns true.

The attribute ruleType is set during the parsing of the rules file. The default value for it is “ANDRULETYPE” indicating that all the conditions have to be met before the doesRuleApplyToNode return true. When a compound condition is encountered in the rules file, the condition’s type sets the ruleType attribute. If the compound condition is “and” then the ruleType attribute is not changed. If it is “or” though, the ruleType attribute is set to “ORRULETYPE”. The “ORRULETYPE” indicates that only one of the conditions of the rule has to be true in order for the “doesRuleApplyToNode” method return true.

The other method in the Rule class worth looking at is the “apply” method. This method is called when the rule should be applied to a node. What this means is that the action of the rule will be examined to determine the name of the notification attribute the node’s value should be stored in or if a processing instruction needs to be returned. The method first checks to see if the action is “ignore”. If so it returns the string “IGNORE” back to the caller. If the action is not ignored then the action is checked to see if it equals “path”. If the action is “path” then the method returns the string “path”. If the method is still executing the next step it takes is to determine what the name of the notification attribute the action specifies is. This is the name of the location in the SIENA notification to store the value of the node. Once this is determined then an attempt to extract the name of the node is made. This is done by either checking for an attribute that has the name “name” or by getting the attribute node specified in the targetAttribute attribute of the Rule object. This last case is used for the rules specified by the user because they are not being applied to the XSchema but to the XML notification. It is possible that the name of the node is not attainable. In this case the variable that holds the name of the node is set to null. The method checks to see if the node has any text node children that would contain the value of the node. If the node does not have a text node child or any attributes then the rule is unable to apply itself to the node and null is returned to the caller. Otherwise this method continues processing by checking to see if the action is a group action and if the group name is not "currentTag". When this happens the method simply returns the attribute name that was specified after the colon in the action. After this a check is to see if the action specified was unique is made. If the check is true the unique processing instruction is created by concatenating the name of the notification attribute specified in the action after the colon, a dollar sign, and then the name of the current node. This new value is then returned to the caller. The last thing the method executes is a return statement containing the name of the node that was passed in, if it is not null, or the rule's action variable.

The Condition classes execute their respective operators against the Node object passed in during the execution of the "meetsCondition" method. If the operation returns true after being applied to the node then the method also returns true. Otherwise false is returned.

This ends the look description of the rules sub-package. The last package that is going to be looked at is the xpath package that contains the classes to handle and manipulate an XPath.

XPath Sub-package

Figure 21 is the class diagram for the xpath sub-package. The following are the classes in this package an brief description of each:

- XPath: This is the main class that is used to represent the XPath.
- Step: This class represents a single step in the XPath statement.

- Predicate: An abstract class that is also a factory used to build the correct type of predicate based on the string argument passed in to it.

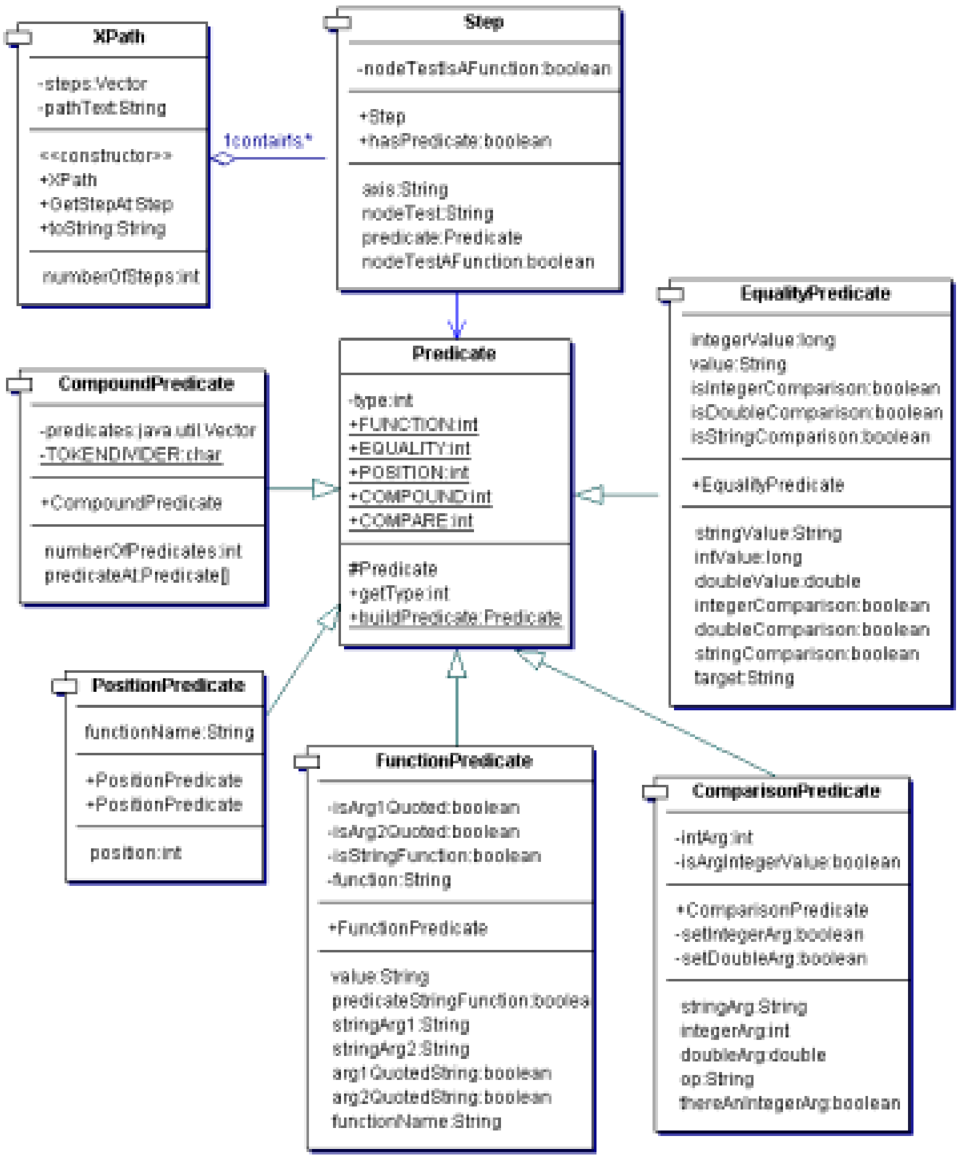


figure 21: Class diagram for the xpath sub-package.

- ComparisonPredicate: This predicate handles any of the expressions that deal with comparing values with an inequality operator, such as greater than.

- CompoundPredicate: This predicate is a container for two or more predicates that are joined together by a Boolean operation.
- EqualityPredicate: This predicate represents the expressions that check to see if two values are the same or not.
- FunctionPredicate: When a function is specified in a predicate statement this class is used to represent it. The only functions that do not use this predicate are position and last, which are covered by the PositionPredicate class.
- PositionPredicate: This is used when a position of a tag is specified in a predicate.

When an XPath is passed into the XMLSubscriptionHanlder it creates a new XPath object, passing it in the String object that contains the XPath expression. The XPath object tokenizes the string using the slash as the separator. For each token the XPath object creates a new Step object, passing it the current token, and stores it in a Vector. All of this is done in the constructor for the XPath object.

In the Step object's constructor it determines if a tree axis specifier is used by checking for the index of a double colon. If the double colon exists then a tree axis specifier is being used in the step. The constructor saves this information into the axis attribute of the class. The next piece of information extracted from the string is the existence of a predicate. By checking a bracket the constructor is able to determine if a predicate exists and the location in the string the definition of the predicate begins. The predicate is removed from the argument string using the substring method and passed in as an argument to the Predicate static method "buildPredicate". This method will return a reference to a Predicate object that represents the predicate defined by the string passed into it. The Step constructor saves this object in the predicate attribute.

After the predicate has been dealt with the Step constructor determines what the node test is from the argument passed in to it and saves it in the nodeTest attribute. If there is a parenthesis in the node test name then the Boolean function is set to true to indicate that the node test is a function of some sore instead of a name of a node to test for.

The “buildPredicate” method in the Predicate class is the factory method used to create predicates based on the information passed in through the String object parameter. First thing to be done is strip the string parameter of the braces that enclose the predicate. After that a check is made for different values in the string to determine the type of predicate that should build. The first thing checked for is for the Boolean operators “and” and “or”. If one of these is present then a CompoundPredicate object is created, passing the constructor of the CompoundPredicate the string parameter passed into this method.

The CompoundPredicate uses a predicate parser to generate a tree where each node is a Boolean operator, either AND or OR, and each leaf is a predicate. By building a tree in this fashion it is easier to create the all possible combinations that the OR’s and AND’s can produce. This is done when creating filters for this predicate.

The “buildPredicate” method then checks to see if it can convert the string argument into an integer. If this works or the string starts with the words “last” or “position” then a PositionPredicate is created. The next check is for the equality sign. If the index for the equality sign is not –1 then it is determined if there is an exclamation point in front of it. In either case a EqualityPredicate is built and then returned to the caller of the method. There is a special case that is checked for when building the EqualityPredicate. If one of the values of the expression happens to be a function then a CompoundPredicate is built instead. This allows for functions to be correctly translated into SINEA subscriptions by the XMLSubscriptionHandler. If this were not done then the information contained in the function would be lost, as the EqualityPredicate would not correctly process it. The next check is to see if the string contains a '>' or a '<' character. If so then a ComparisonPredicate is built otherwise the default is the creation of a FunctionPredicate object.

This ends the description of the implementation of the architecture. The next section details a testing plan for the implementation followed by the next chapter on the evaluation of the implementation.

Testing plan

There are two basic methods of unit testing that are going to be done on the implementation: white box and black box testing. The reason for using both is that for some of the more complex classes white box testing will be able to test the class more thoroughly because it can test the private methods individual and make sure they are generating the correct output which black box testing can not do. Black box testing on the other hand can test the consistency of class by checking to see if the public methods, which call the private methods, consistently return the desired results. If they don't and the white box testing indicates that the private method is working correctly then there is a problem with the public method calling the private methods correctly. Not all classes need to have both white and black box testing applied to them as some are very basic in nature and black box testing will adequately test the class. The only classes that will use both types of testing will be the XMLProcessor class, XMLSubscriptionHanlder class, and the XSchemaProcessor class.

Black box testing will be implemented using the JUnit library. JUnit is a set of classes in java that create a "simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks."[6]. JUnit is developed by Erich Gamma and Kent Beck and more information on it can be found on the JUnit website [6]. All of the classes will have separate JUnit test classes for black box testing that were automatically generated by the Netbeans IDE [7].

The white box testing will be done using inner classes that will use the JUnit framework. Inner classes have the unique ability to access the private members and methods of its outside class, which allows private methods to be tested and private members to be checked to see if they contain the correct values before and after the execution of the methods. The inner classes are declared to be static and all of them have the main method in them to execute the tests. There are individual public methods in each class that tests a particular method or control path in the code. There is at least one test for each private method in the outside class with more test methods for the more complicated methods.

All of the test classes are stored in a separate directory under the xml directory called tests. A class called TestAll can be used to run all of the tests and print out the results onto the screen. The test classes will not be included in the default make of the XML client interface.

Chapter 5

EVALUATION

To evaluate the interface two tests were created to see how well it handled the translation of a XML notification into a SIENA notification. One test examined the time it took to generate the notification with the DOM tree for both the XSchema and the XML Notification being very broad while the other test examined the time for the notification to be generated with the DOM trees very deep.

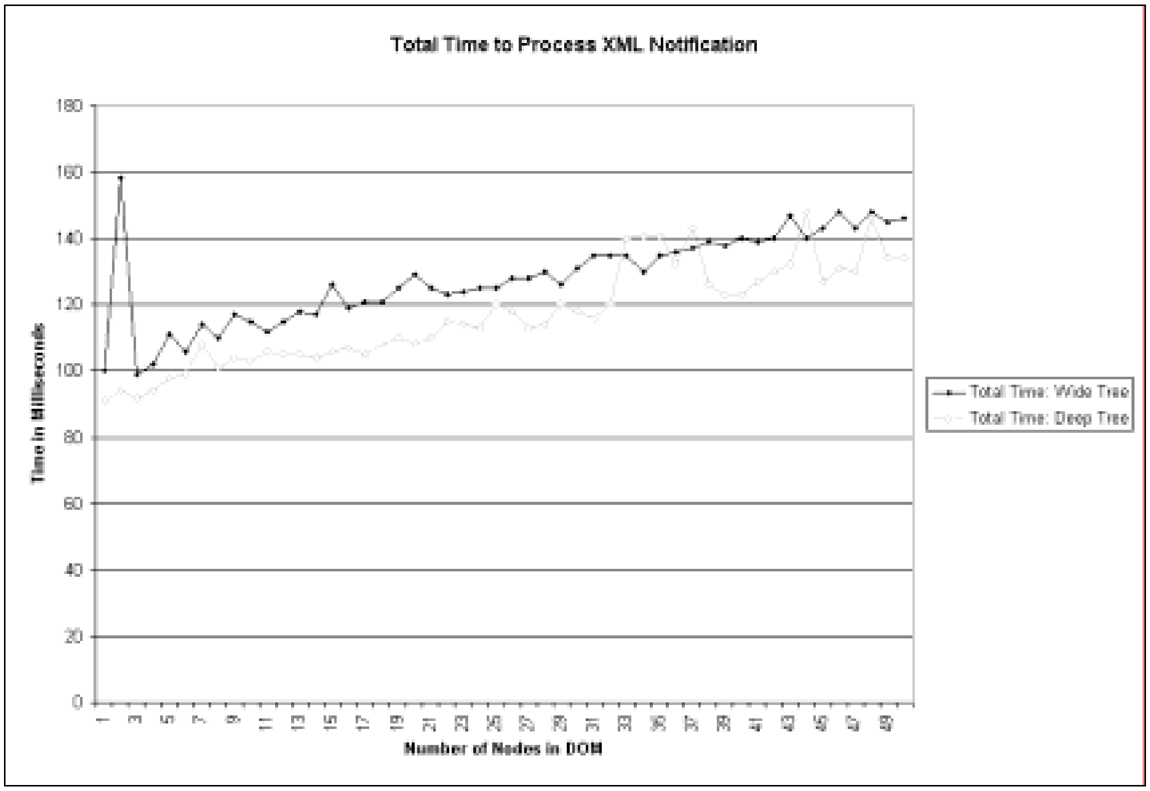


Figure 22: Graph of Total Time to Process the XML Notification

Figure 22 shows the graph comparing the number of nodes in the DOM tree to the total amount of time to process the XML notification. The spikes in the graph are anomalies produced by increased usage of the computer running the tests. The layout of the data points suggest a linear relationship between the number of nodes in the DOM and the time needed to translate an XML notification into a SIENA notification. The translation process performs a little better when dealing the deeper trees. A possible explanation for this is that in the "applyRules" method there is a "for" loop near the end of it which loops until all of the children of the current node are processed. With the DOM tree being skewed in a way so that there was a single child per node the "for" loop only executed once instead of many times.

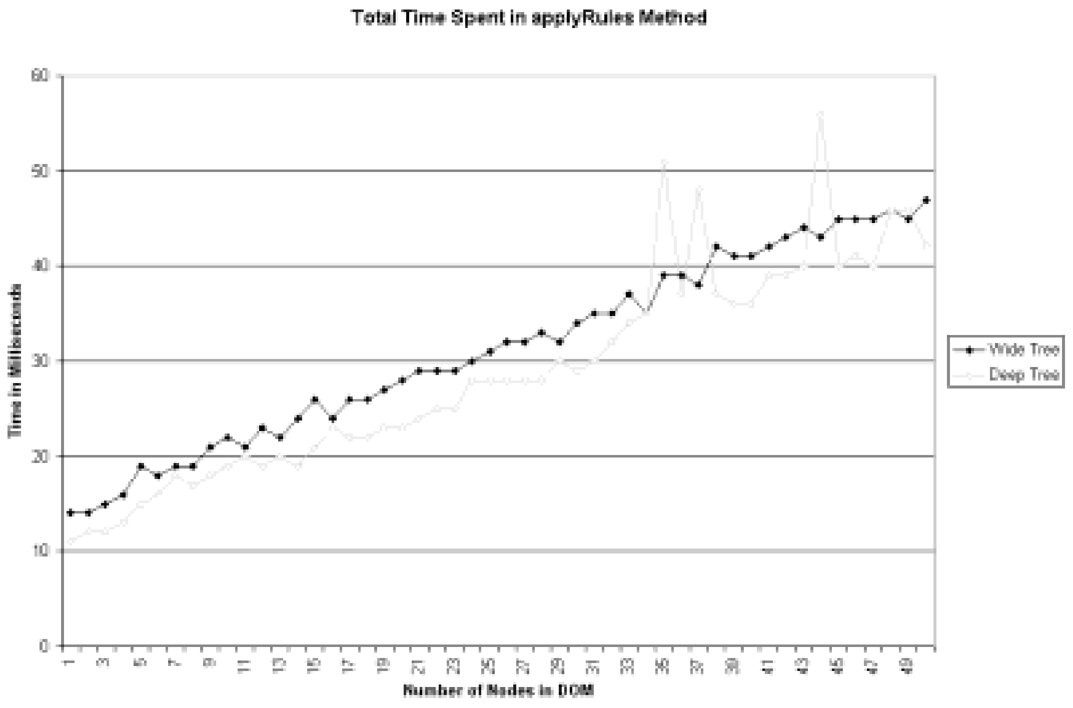


Figure 23: Graph showing the total amount of time spend in the applyRules method

In figure 23 the graph showing the total amount of time spent in the "applyRules" method compared to the number of nodes in the DOM. The three spikes in the graph are anomalies that were generated during high use of the computer running the test. In general though the translation process faired better with the deep DOM tree than the shallow DOM tree. This is to be expected since the graph in figure 22 showed the same result. The time spent in the "applyRule" method is a fraction of the time spent overall in translating the notification, though it increases at a greater rate than the time needed to translate the notification in general. The reason for this is that the translation time includes the time to serialize a String object and compress it. This part of the translation takes up a significant portion of the processing time.

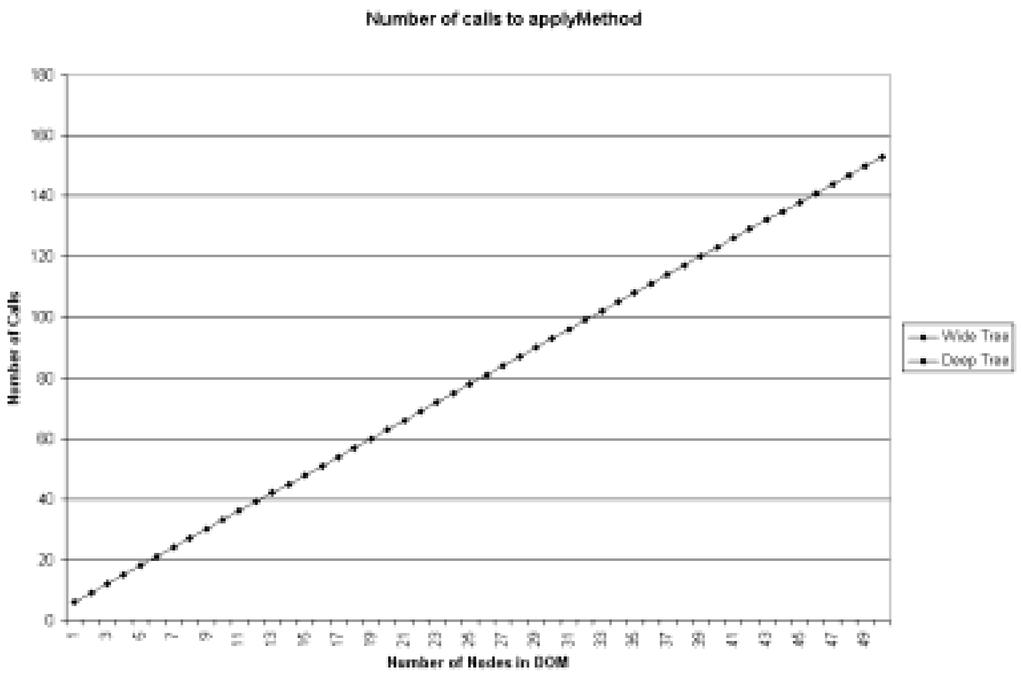


Figure 24: Graph showing the number of calls to the applyMethod

The graph in figure 24 shows the number of calls to the “applyMethod”. A very linear relationship can be seen in the data. This is to be expected since the “applyMethod” is called for every node in the DOM tree. It also shows that the shape of the DOM tree has no bearing on the number of calls that are made. The only thing that affects the number of calls is the number of nodes in the DOM.

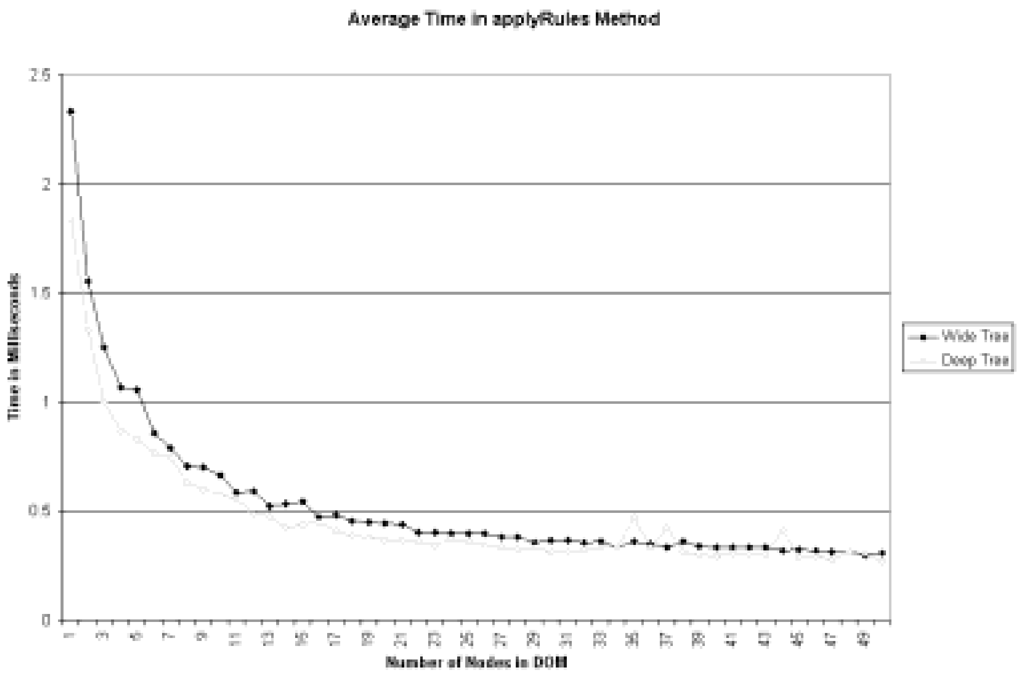


Figure 25: Graph showing the average time spent in the applyRules method

The graph in figure 25 is the most interesting graph. It shows that as the number of nodes increase in the trees, the less amount of time was spent in the “applyRules” method. The three spikes in the deep tree data near the tail end of the graph are the anomalies explained previously for figure 23. The layout of the data suggests that there is a lower limit to the amount of time spent in the “applyRules” method. This is reasonable as there is a limit to how fast a single instruction can be executed on the computer. The somewhat exponential decay of time though is what is counter -intuitive. Logic would suggest that the average amount of time spent in the “applyRules” method would increase as the number of nodes to process increases. A possible explanation for this is that the method runs rather quickly without any outside influences. However, it is possible that the JVM at different times executes the garbage collector when the method is being executed. When the method is only executed once then the additional time incurred by the method can perhaps be traced to the JVM doing something else in the background. This would result in the time in the method being larger than what it truly is. As more calls are made to the “applyRules” method then the time impact from the JVM is reduced which could produce the pattern of the data in the graph.

Chapter 6

CONCLUSSIONS

Using the structure of an XML document provides useful information in the translation process of a XML notification to a SIENA notification. The information gathered from a XSchema document guides the translation process by indicating elements that do not need to be included in the notification, by specifying elements that would need a unique way to identify them, and removed the need to include any type of structural information in the SIENA notification because the client also has access to the XSchema which it can use to reconstruct the XML notification from the SIENA notification.

A nice side effect from the use of the structure is that the same information generated from the XSchema could also be used directly in the client that is subscribing to the XML notification. The XPath expression is translated into SIENA attribute constraints using the same set of guidelines that were used to translate the XML notification. This allows client applications using the XML interface to SEINA to be separated from knowing about how SIENA works. Instead the client applications simply publish XML notifications and subscribe to the notifications using XPath expressions.

The main source of difficulty in the implementation of the interface was the development of the rules language and generating a basic set of rules that could be used with any XSchema to translate an XML notification. The reason is that the rules language had to be powerful enough to express some rather complex ideas yet be simple enough that some one new to the system could understand the language fairly easily and starting using it right away. The generic rules were difficult because they had to cover a wide range of cases and possibilities that could exist in an XML document and provide the correct type of processing to guarantee that a XML notification could be transformed into a SIENA subscription. No proof or data has been generated to prove or disprove if the current rule set meets this goal.

In closing it has been determined that the interface is a success because it does not reduce the power of the clients using XML but instead empowers them to use a power event notification system without needing to know how it works.

Chapter 7

FUTURE WORK

From this point there are some different possibilities to explore. The next logical step would be an effort to push the XML interface into the SIENA servers. This would reduce the amount of client side processing because some of it would go into the servers and also some of it would disappear completely, such as the translation process of the XML notification into SIENA notifications. Some difficulties in doing so though would include the re-architecture of the notification system so that the native format would be XML. This would effect any of the optimizations the servers use to route the notifications. The implementation of the routing algorithms would have to change also to take advantage of the properties of XML.

Just recently a draft of the specification for XPath 2.0, the next version of XPath, was released. This newer version has been updated to meet the requirements that the XQuery specification has. This new specification could be evaluated tp see if the new enhancements to it allow it to work as a better subscription language than version 1.0.

Another expansion would be the use of XQuery as the subscription language. Because XQuery is more complex and powerful than XPath more effort would have to be made in converting the query into a SIENA subscription. If the XML interface was pushed back into the SIENA servers then it would also require changing the subscription language for SIENA which could be a major piece of software engineering work.

The use of an XSchema to guide the translation process of a XML notification into a SIENA notification could be furthered researched. One direction would be to look at the generic rules and the application of them to the XSchema. What other pieces of information are available in the XSchema which can be exploited by the proper set of rules? How can this technique be used in other applications that transmit XML data across networks?

BIBLIOGRAPHY

[1] A. Carzaniga, D.S. Rosenblum, A.L. Wolf, *Interfaces and Algorithms for a Wide-Area Event Notification System*, Technical Report CU-CS-888-99, Department of Compute Science, University of Colorado, 1999. (Revised 2000)

[2] Bert Bos, *XML in 10 Points*, http://www.w3.org/XML/1999/XML-in-10-points, 1999.

[3] James Clark, Steve DeRose, editors, *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/xpath, 1999.

[4] David C. Fallside, editor, *XML Schema Part 0: Primer*, http://www.w3.org/TR/xmlschema-0/, 2001.

[5] Don Chamberlin, editor, *XQuery 1.0: An XML Query Language*, http://www.w3.org/TR/xquery/, 2001.

[6] Erich Gamma and Kent Beck, *JUnit*, http://www.junit.org, 2001

[7] Sun Microsystems, *Netbeans*, http://netbeans.org

[8] TogetherSoft, *Together ControlCenter*, http://www.togethersoft.com

[9] Apache foundation, http://xml.apache.org/

[10] Antlr parser generator, http://www.antlr.org

APPENDIX A

Lexer grammar

```
tokens {
    "rule"; "group"; "ignore"; "unique"; "path"; "and"; "or";
"present";
"notpresent"; "notunique"; "sibling"; "child"; "parent";
"novalue";
}

WS  :   (' '
    |   '\t'
    |   '\n'    {newline();}
    |   '\r')
        { _ttype = Token.SKIP; }
    ;


SL_COMMENT :
    "//"
    (~'\n')* '\n'
    { _ttype = Token.SKIP; newline(); }
    ;

NOT:    '!'
    ;

LCURLY: '{'
    ;

RCURLY: '}'
    ;

LESS:   '<'
    ;

GREATER:    '>'
    ;

EQUAL
    :   '='
    ;

COLON:  ':'
    ;
```

```
COMA:   ','
    ;

SLASH
    :   '/'
    ;

CHAR_LITERAL
    :   '\'' (ESC|~'\'') '\''
    ;

STRING_LITERAL
    :   '"' (ESC|~'"')* '"'
    ;

protected
ESC :   '\\'
        (   'n'
        |   'r'
        |   't'
        |   'b'
        |   'f'
        |   '"'
        |   '\''
        |   '\\'
        |   '0'..'3'
            (
                options {
                    warnWhenFollowAmbig = false;
                }
            :   DIGIT
                (
                    options {
                        warnWhenFollowAmbig = false;
                    }
                :   DIGIT
                )?
            )?
        |   '4'..'7'
            (
                options {
                    warnWhenFollowAmbig = false;
                }
            :   DIGIT
            )?
        )
    ;

protected
DIGIT
```

```
    :   '0'..'9'
    ;

DOT: '.' ;

INT: (DIGIT)+;

ID
options {
    testLiterals = true;
    paraphrase = "an identifier";
}
    :   ('a'..'z'|'A'..'Z'|'_')
('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
    ;

ATTRIBUTE : '@' ID
    ;
```

Rules grammar

```
program =
    :   (rule)* EOF
    ;

rule
    :   TK_rule COLON target conditions action
    ;

target
    : (tag (ATTRIBUTE)?)
    | STRING_LITERAL
    ;

tag
    :   (path)* ID
    ;

path
    :   ID SLASH
    ;

conditions
    : expr
    | compound LCURLY ATTRIBUTE expr (COMA ATTRIBUTE expr )*
RCURLY
    ;
```

```
compound
returns [String type]
{type = null;}
    : TK_and
    | TK_or
    ;


action
    :   TK_group COLON ID
    |   TK_unique COLON (tag | treeAxis)
    |   TK_notunique
    |   TK_path
    |   TK_ignore
    |   TK_novalue
    ;

treeAxis
    :   TK_sibling
    |   TK_child
    |   TK_parent
    ;

exprTarget
    : (treeAxis)? target
    ;

expr
    : assignExpr
    | notequalExpr
    | presentExpr
    | notpresentExpr
    | lessThanExpr
    | greaterThanExpr
    ;

assignExpr
    :   EQUAL (exprTarget | INT)
    |   EQUAL INT DOT INT
    ;

notequalExpr
    :   NOT EQUAL (exprTarget | INT )
    |   NOT EQUAL INT DOT INT
    ;

presentExpr:    TK_present ;

notpresentExpr: TK_notpresent ;
```

```
lessThanExpr
    :   LESS INT
    |   LESSINT DOT INT
    ;

greaterThanExpr
    :   GREATER INT
    |   GREATER INT DOT INT
    ;

atom
    :   ID
    |   INT
    |   CHAR_LITERAL
    |   STRING_LITERAL
    ;
```

APPENDIX B

RULES

Creating a Rule

Rule definitions are stored in files on the local file system to the client. The files can contain any number of rules as long as they follow the basic structure of a rule. The basic structure of a rule consists of four parts: The declaration, the tag specifier, the condition, and the action.

rule: *tag-specifier condition action*

The declaration is simply the word "rule" followed by a colon. This indicates the beginning of a new rule. The tag specifier indicates the XML tag the rule is to be applied to. The condition specifies any constraints that the XML tag must meet before the rule is applied to it. The action indicates what is to be done with the XML tag. Each of these parts is discussed in further detail in the following sections.

Tag Specifiers

The tag specifier is an identifier followed by an optional attribute identifier. An identifier is a string that starts with a character or underscore and is followed by zero or more characters, numbers, or underscores. Here are some examples of a tag specifier:

- annotation: Matches only annotation XML tags.

- item@partNum: Matches item XML tags that have an attribute named partNum. If an item XML tag exists that does not have this attribute the rule would not be applied to it since it is not the target of the rule.
- any: Matches any tag.
- any@type: Matches any XML tag that has an attribute named type.
- any@any: Matches any attribute in any tag.

The word “any” can be used in the tag specifier to indicate that any XML tag or any attribute would work for the rule. As seen in the examples the “any” keyword can be used as the tag identifier, the attribute identifier, or as both.

Conditions

The condition is a test to be applied to the XML tag that matches the name specified in the tag specifier. The condition has a set of operators that are applied to the XML tag, such as numerical comparisons and tests to see if the XML tag or attribute is present or not. A list of the operators and a brief description of what they do is below. Only when all of the conditions are met is the rule applied to the XML tag.

= *value* – This condition compares the value of the XML node to *value*. Only if the XML Tag value and the value specified are equal does the action get applied to the XML tag. The value can be a quoted string, a number, or a tree axis followed by a tag specifier.

!= *value* - This condition compares the value of the XML node to *value*. Only if the XML Tag value and the value specified are not equal does the action get applied to the XML tag. The value can be a quoted string, a number, or a tree axis followed by a tag specifier.

[<,<=] *value* – This condition compares the value of the XML tag to *value*. If the tag value is less than or less than or equal to the *value* then action is applied to the XML tag. The *value* can only be some type of numerical value.

[>,>=] *value* - This condition compares the value of the XML tag to *value*. If the tag value is greater than or greater than or equal to the *value* then action is applied to the XML tag. The *value* can only be some type of numerical value.

present – This checks to see if the XML tag or attribute is present. If so the action of the rules is applied to the XML tag.

notpresent – This condition checks to see if the XML tag or attribute is not present. If it is not, then the rule is applied to the XML Tag. This condition really only works with attributes and checking to see if a XML tag does not have a particular attribute.

If the condition requests a test to be done on something that does not exist, i.e. testing the value of the name attribute except the current XML tag does not have a name attribute, the condition will always fail. This is shown in figure 26.

Conditions can be grouped together in a rule by using a Boolean operator. The Boolean operations supported are “and” and “or”. When using a Boolean operator the conditions that it works upon must be enclosed in braces. All of the sub-conditions can only be applied to the attributes of the tag and nothing else. Therefore it is not possible to check to see if a XML tag has a specific value and if it has an attribute that contains another value. The “and” operator imposes the condition that all of the sub-conditions in the rule must be satisfied by the XML tag before the action is applied to it. The “or” condition simply states that one or more of the sub-conditions must be satisfied before the action is applied to the XML tag. Examples of the Boolean operators are given in figure 26.

```
rule: attribute and {
  @maxOccurs = 1,
  @minOccurs = 1,
  @fixed notpresent,
  @use = "required"
} unique:currentTag

This rule would apply to this tag:
<example minOccurs=”1” maxOccurs=”1” use=”requied”>
but not this one:
<example minOccurs=”1” use=”required”>
because the attribute maxOccurs is not present and therefore
untestable by the rule.

rule: attribute and {
  @maxOccurs notpresent,
  @minOccurs notpresent,
  @fixed notpresent,
  @use = "required"
} unique:currentTag

rule: any@type or {
  @type = “xsd:string”
  @type = “xsd:date”
} IGNORE

The following XML tags would satisfy the rule:
<example type=”xsd:string”>
<example type=”xsd:date”>
This tag would not though
<example type=”myType”>
```

Figure 26: Example of rules using the Boolean operators.

Actions

The last part of a rule definition that must be specified is the action. An action is applied only when an XML tag satisfies all the conditions of the rule. The action states where to store the value of the XML tag, or attribute, in the SIENA notification or a processing instruction that guides the program when translating the XML notification into a SIENA notification. When specifying the name of a SIENA attribute a valid identifier is all that is needed. If multiple tags have the same action then the values are stored in a coma-separated list in the SIENA notification attribute. The following list contains all of the valid processing instructions along with a description of what they do.

- ignore this will ignore the tag and not include it in the notification.
- group:*name* this states that multiple instances of this tag should be grouped together under the attribute named *name*.
- path this action indicates that the XML tag should have its named added to a prefix that is used to generate the names of the SIENA attributes.
- unique:target this specifies that this tag or attribute can be used to uniquely identify the tag specified by the target. Target is a tag specifier.

When the program finds a rule that contains the processing instruction “ignore” it will skip over the XML tag any of its children tags. For attributes the program simply skips over the attribute and continues to process the remaining attributes.

```
rule: ignoredTag present ignore
<example>
<ignoredTag attr=”nothing”>
<child1/>
<child2/>
</ignoredTag>
<nexTagInLine/>
</example>
```

Figure 27: XML Example and ignore action

In figure 27 an example XML notification is given with a rule containing an ignore processing instruction. When the program applies the rule to the ignoredTag in the XML it will skip the processing of the attributes of the tag and its children and resume processing with the nexTagInLine XML tag. The XML tags child1 and child2 are not processed nor the attribute “attr” in the ignoredTag.

The group instruction specifies that the value in the XML tag should be grouped together with other values in the attribute with the name *name.* When there is more than one value to be stored in the SIENA attribute then each value is separated by a coma.

The path instruction will cause the program to append the name of the XML Tag that matches the rule to the name of the SIENA attribute. Figure 28 shows an example of an XML notification along with the rule containing the path processing instruction and the resulting SIENA notification.

```
RULE:
rule: pathTag present path

XML NOTIFICATION:
<example>
<pathTag>
<child1> child1 </child1>
<child2> child2 </child2>
</pathTag>
<otherTag>
<child1>child1</child1>
</otherTag>
</example>

SIENA NOTIFICATION:
pathTag/child1=”child1”
pathTag/child2=”child2”
child1=”child1”
```

Figure 28: path processing instruction example

The unique processing instruction informs the program how to uniquely identify an XML tag from its siblings that have the same tag name in the XML notification. The program will append the value of the tag, or attribute, to the SIENA attribute, similar to what the path processing instruction does. Figure 29 shows an example of the unique processing instruction and the resulting SIENA notification.

```
RULE:
rule: tag@uniqueAttr present unique:tag


XML NOTIFICATION:
<example>
<tag uniqueAttr=”u1”>
<child1> child1 </child1>
<child2> child2 </child2>
</tag>
<tag uniqueAttr=”u2”>
<child1> child1 </child1>
<child2> child2 </child2>
</tag>
</otherTag>
</example>



SIENA NOTIFICATION:
tag/u1/child1=”child1”
tag/u1/child2=”child2”
tag/u2/child1=”child1”
tag/u2/child2=”child2”
```

Figure 29: unique processing instruction example

In the action the keyword “currentTag” can be used to indicate to the program that whatever the current name of the XML tag it is examining should replace the word “currentTag” in the action. When processing an XSchema file the program treats the “currentTag” keyword a little different. During the processing of the file the program keeps track of the element and complex definitions so that when the “currentTag” keyword is used the XSchema tag is not used but instead the current XML tag being defined by the XSchema. An example of this is in the default rules:

rule: any present group: currentTag

When processing an XSchema file with this rule the program will determine what the current XML tag being defined is and use the name of the XML tag as the name of the SIENA attribute to store its value in. The only real use for this keyword is in rules that specify “any” as the tag specifier. Once the action has been specified then the rule definition is finished and another rule can be started.

Submitting Rules to the Program

There are two ways that the program is able to get rules. The first method is through the set of default rules. These rules are meant to be very generic so that they can adapt to any situation and provide a basic framework to translate the XML notification into a SIENA notification. It is recommended that these rules not be altered. The second method of giving the program a set of rules is by using the user rules interface. This interface allows an application to give the program a set of rules, defined in a file, to be used when translating the XML notification. If there is a conflict between a default rule and a “user” rule then a warning message will be logged stating the conflict. The program will then use the “user” rule instead of the default rule. More information about the user rules interface is contained in a later portion of this document.

Rule Conflicts

It is permissible for rules to have the same tag specifier, same actions and even the same conditions. When there are multiple rules that have the same tag specifier there is a possibility that an XML tag will satisfy the conditions for all the rules that have the same tag specifier. The type of rule that is being processed determines the outcome of this situation. If the rule is one of the default rules then the *first* rule to match is the one used for the XML tag. If the rule is one that was specified by a user then a warning message would be logged listing the rules that could match the XML tag and the *last* rule to match is applied to the XML tag. The warning message is only logged if the matching rules produce different actions or results when applied to the XML tag.

APPENDIX C

SIENA'S XML INTERFACE

Introduction

There are two ways to use the XML client for SIENA. One way is to use the XMLClient object as the main interface to SIENA. The second way is to use the XML client package classes directly, such as the XMLProcessor and the XMLSubscriptionHandler. The second way of using the client requires more knowledge of SIENA than the first and should be used only by applications that may need to monitor the data flowing between the SIENA client and server closely. The first method is described first.

Using the XMLClient Class

The XMLClient object provides a XML interface to the SIENA servers. The application client only needs to know XML and very little about the underlying SIENA data structures. When an XMLClient object is created the name of a SIENA server is passed to it in the form of "senp: host.name: port". The host name is the name of the server where the SIENA server resides. The port is the port the SIENA server is listening on. The constructor will create a connection to the SIENA server if it can. If a server cannot be connected to then the constructor will throw a siena.InvalidSenderException.

The next step of setting up the XMLClient is to tell it where the file containing the XSchema for the XML notification is located. Before sending or receiving XML notifications this must be done otherwise the XML Client will not be able to correctly translate the notifications and subscriptions into their SIENA equivalents. The setStyleSheet method is used for this purpose. The method takes a single string argument which should contain the name, and if necessary the path, to the XSchema file. The method will then process the contents of the file. If there is an error during processing a siena.SienaException is thrown containing information about the error that occurred.

To publish a XML notification to SIENA the publish method in the XMLClient object is invoked. This method takes a single parameter, a string containing the XML notification. The method will then translate the XML notification and send it to the SIENA server for publishing. If an error occurs during this process a siena.SienaException is thrown containing the error message. This is all that needs to be done to publish a XML notification.

To subscribe to XML notifications the method subscribe is called in the XMLClient object. The subscribe method has two parameters: a string containing the XPath expression that is the subscription language for the XML notification and a siena.Notifiable object. The interface siena.Notifiable has two methods that are called when a XML notification matches a subscription. The application client needs to fill in these methods with the code to process the XML notifications. The notify methods both receive a siean.Notification object. This object will have a single attribute in it called “xml”. This attribute will be a string containing the complete XML Notification. See figure 5 at the end of this appendix for an example of how to extract the XML notification from the SIENA notification. This is all that needs to be done to subscribe to XML notifications. Multiple subscriptions can be issued using the same method. The XMLClient keeps track of the different subscriptions and objects that are to be notified when a particular subscription is fulfilled.

If there are user rules that the program should know about then the method setUserRules is invoked, passing it the name of the file that contains the rule definitions. If the rules cannot be processed then a siena.SienaException is thrown with the message containing information about the error that occurred. Otherwise the user rules will be used on the subsequent translation method calls. Note that this method can be called multiple times and the user rules will be combined together into one set. If two rules have the same tag specifier then the rule that was added last is the one that is used. To clear the user rules the method clearUserRules can be called. This will clear all user rules that have been given to the program.

Using the XMLProcessor and XMLSubscriptionHandler Classes

The second way of using the XML client is to directly use the XMLProcessor and XMLSubscriptionHandler objects. The application client must handle the registering of a SIENA client with the SIENA server. It must also handle the actual publishing and subscribing of the SIENA notifications and filters. In order for the XMLProcessor and XMLSubscriptionHandler objects to correctly handle the XML notifications they need to have the setStyleSheet method called. This method, like the one for the XMLClient object, accepts a single parameter that is the name of the file containing the XSchema. The objects will process the file and build the internal map they use for translations. Only after this method has been invoked will the translation methods work correctly. If an error happens during the processing of the file the method will throw a siena.SienaException with a message containing information about the error.

If any user rules are to be used in the processing of the XML notification the setUserRules method should also be called at this time, though it is not necessary to do so. This method takes in the name of the file that contains the user rules. A siena.SienaException is thrown if there are any troubles processing the rules. If this method is called more than once then the rules in the newer file will be appended to the current set of rules. For rules that have the same tag specifier then the new rule will replace the current rule.

To publish a subscription the translate method on the XMLProcessor is called, passing it a string containing the XML notification. The method will return a siena.Notification object that contains the original XML notification translated into a SIENA notification. It is up to the application client to send the notification to the SIENA server. There are no methods on the XMLProcessor that do this for the application client.

Creating a subscription for a XML notification is more complex than sending a XML notification. The XMLSubscriptionHandler constructor requires a siena.Notifiable object to be passed in to it. The object passed in will be the one that will be notified when an XML notification has met the requirements of the XPath statement. After the XMLSubscriptionHandler is built then the setStyleSheet method is invoked as describe in a previous paragraph. The XMLSubscriptionHandler will then be ready to build filters for a subscription.

To build a filter for the subscription the buildFilter method is invoked with a single parameter. The parameter is the string containing the XPath expression to be used as the subscription. This method will return an array of siena.Filter objects that are based on the XPath passed in. The application client is responsible for submitting the Filter objects to a SIENA server as a subscription. When the application client does this the XMLSubscriptionHandler should be the object given to the server as the object to be notified. The reason for this is that the XMLSubscriptionHandler does some post processing on the SIENA notification in order to validate the XPath against the XML notification received. The application client must also call the attach method, passing in an Observer object. If this is not done then the application client will never receive a XML notification, as the XMLSubscriptionHandler has no way to inform the application client that a XML notification has been received. When a XML notification is validated, i.e. the XPath statement returns at least one node, the XMLSubscriptionHandler will notify the application client buy invoking the update method on the Observer object, passing itself into the method. The application client is responsible for getting the siena.Notifiable object from the XMLSubscriptionHandler, given to it during its construction, and call the notify method on it, passing it the siena.Notification object stored in the XMLSubscriptionHandler. The siena.Notification object can be retrieved from the XMLSubscriptionHandler by calling the getNotification method. The XML notification must be extracted by the Notifiable object from the siena.Notification object using the getAttribute method with the string parameter equal to "xml". Calling toString on the returned object will return a String object containing the complete XML notification sent by another client. This process is shown in figure 30.

```
Public void notify(siena.Notification notification) {
    String xmlNotification =
        notification.getAttribute("xml").toString();
}
```

Figure 30: Extracting a XML Notification

It is possible for the XMLSubscriptionHandler not to be the object notified by the SIENA servers when a SIENA notification matches a subscription but the application client would have to reconstruct the XML notification and then validate it against the XPath expression. This is duplicate work since the XMLSubscriptionHandler already does all of this.

Using either method to use the XML client interface to SIENA works. In fact, the XMLClient uses the second method itself to implement the first method. In the end though the same results occur, the use of XML as a notification language in the SIENA environment.