The Axlog Guide

Bogdan-Eugen MARINOIU

July 22, 2009

This is a guide for a user or a developer of the AXML framework. In particular, this is a technical guide on *axlog*, a module of AXML that maintains views over active documents. The concepts of the view maintenance are explained in the papers [1] and [3]. Please refer to them if you encounter difficulties in understanding the concepts.

The first part, i.e. The User's Guide, is about how to configure the axlog module and how to define views on active documents. The second part, i.e. The Developer's Guide, contains a brief description of the different software components of the axlog module. Detailed information about the software has been documented automatically using the *javadoc* tool and is available next to the axlog's sources.

Chapter 1

The User's Guide

1.1 Defining views: the axlog subscriptions

An axlog subscription is usually made of four parts: the document, the tree-pattern query, the template for rendering results as trees and an optional dissemination section. For now, there is no XML Schema or DTD for typing an axlog subscription but it would be good to have one in the future.

On every AXML peer there is an axlog subscription service, named *AxlogService*. E.g. an endpoint URL for that service would be:
`http://localhost:6969/MyPeer/services/AxlogService`.

MyPeer is the name of the peer, *localhost* is the host and 6969 is the port number. The name of the method is *subscribe* and the namespace can be set to *n/a*. The root of the XML subscription has as prefix the prefix of the namespace and as local part *subscription*.

1.1.1 Active Document

There are several options for the specification of the active document. All need that a *document*-labeled XML element be child of the *subscription*-labeled XML element.

The first possibility is that the active document is on the current AXML peer, where the axlog module is situated. Then, the name of the active document has to appear as value of a *docID*-attribute of the *document*-element, e.g. `<document docID="doctest.xml"/>`.

The second possibility is that the active document be on a different peer. In this case, the a *peerURL* attribute should be set to the URL of the peer.

For both cases, an optional *where* attribute might be set to *peer*. If it is not provided, the *peer* value is implicitly considered.

A third possibility is that the active document be provided explicitly as subtree of the *document*-labeled element. There must be explicitly a *where* attribute valued *included* for the *document*-element to specify this. This possibility is mainly used for testing (since the document can be loaded directly in memory by the axlog module).

1.1.2 Tree-pattern Query

A tree-pattern query (TPQ) is provided as an XML element, child of *subscription*-labeled element. The root of this subtree is labeled *query*.

There are two parts for the query: one that captures the structure, subtree rooted *tree* and another one, labeled *conditions*, that captures the join/equality conditions and inequalities.

The *node*'s of the tree-pattern query are mapped to elements of the XML document. The *node*'s are descendants of the *tree*-labeled element. This has at least as attribute a *label* attribute and a *link* attribute. The *label* attribute has its value set to the label of the TPQ node, part of the label alphabet or $*$ in case of a variable. The *link* attribute specifies the relation of that node to its parrent, either a / or a //. There are two additional attributes that are optional: *output*, that is set to *true* if the node's valuation is needed in the output and *id* - an identifier of the node, also used for the output. For the extraction, one can specify either to extract the label of the element or the text content. This is done with an additional attribute *selectionStyle* of TPQ *node*'s that is valued either to *node* or *text*. If this attribute is missing, the implicit *selectionStyle* is *node*.

The *conditions*-labeled element has as children either *condition*-labeled children (one per equality condition), or *timecondition*-labeled children (for inequalities).

Examples:

1.
   ```
   <condition>
               <op>i1</op><relation>equ</relation><op>i4</op>
   </condition>
   ```

   stands for an equality condition between two TPQ nodes, one identified by `i1` and the other identified by `i4`.

2. `<timecondition>i5 - i1 lt 150000</timecondition>`

   stands for the inequation: `i5 - i1 < 150000`, where `i1` and `i5` are TPQ node identifiers. This condition means that the document data that matches the TPQ nodes identified by `i5` and `i1` has to be timestamped with timestamps that are separated by 150 seconds of delay. The timestamps correspond to time instants when the data has been inserted into the document. The accepted operators are: *, +, `lt` (for <), `gt` (for >), `lte` (for <=) and `gte` (for >=). Empty spaces are needed to separate identifiers and operators.

1.1.3 The template

The template is used to reformat the tuples obtained with the tree-pattern query. The template is XML data with variables. A variable is of form `$<id>` (do not confound with XML tags), where `<id>` is the identifier of a TPQ node. For instance, `<a><b>{$i1}</b><c>{$i2}</c></a>` is a legitimate template with two variables: one for `i1` and one for `i2`. For each valuation of the variables, a tree is produced by replacing the variables with their values. The result, since there may be several valuations, is a forest in general (a set of XML documents).

1.1.4 Optional: the BY clause

The *BY* clause serves for dissemination purposes. If the subscription is "sent" through a Web service call from an AXML document, this clause is not required. The implicit behaviour of axlog in this case is to send results to the AXML document from where the call to the Axlog service has been fired.

The *BY* clause, if present, will override the default behaviour. For now, axlog can only upload results to Web services, but other dissemination means may be implemented, fon instance: send e-mails, write in a RSS feed etc..

A typical *BY* clause accepted by the current implementation is:

```
<by type="serviceCall">
        <ns2:address xmlns:ns2="http://futurs.inria.fr/gemo/axml/service/Algebra"
endpoint="http://localhost:6969/MyPeer/services/ReceiveOperator">
             <ns2:currentID>
                <ns2:peerID>MyPeer</ns2:peerID>
                <ns2:docID>docsub.xml</ns2:docID>
                <ns2:nodeID>task1</ns2:nodeID>
              </ns2:currentID>
       </ns2:address>
  </by>
```

1.1.5 An example

Let us consider the document, the query and the template in Figure 1.1. The query imposes a join/equality condition on the labels of two nodes (n_1 and n_4 are labeled with the same variable) and an explicit inequality condition on timestamps for other two nodes (n_1 and n_5). It also extracts (see the `+` prefix) the labels of four nodes: n_1, n_2, n_3 and n_6.

I present below how all these are encoded in XML.

First, the AXML document to monitor called *doctest.xml* has the following content:

```
  <a xmlns:axml="http://futurs.inria.fr/gemo/axml/">
    <b>
      <axml:sc axml:id="f">
         <axml:return>
```

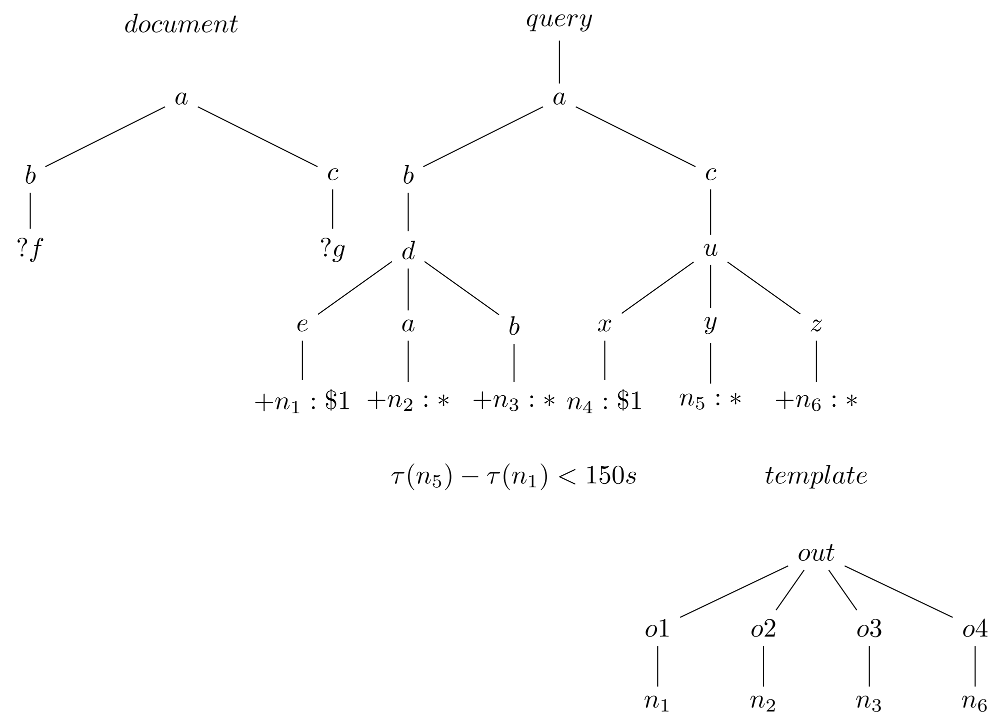

Figure 1.1: Document, query and template

```
          <axml:append/>
        </axml:return>
        <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/DummyStreamService">
          <s:streamToMe xmlns:s="n/a">
             <s:max-timeout>5</s:max-timeout>
             <s:query>for $i in doc('/db/sourcef.xml')/*/* return $i</s:query>
          </s:streamToMe>
        </axml:ws-soap>
      </axml:sc>
  </b>
  <c>
    <axml:sc axml:id="g">
      <axml:return>
         <axml:append/>
      </axml:return>
      <axml:ws-soap endpoint="http://localhost:6969/MyPeer/services/DummyStreamService">
        <s:streamToMe xmlns:s="n/a">
           <s:max-timeout>10</s:max-timeout>
           <s:query>for $i in doc('/db/sourceg.xml')/*/* return $i</s:query>
        </s:streamToMe>
     </axml:ws-soap>
   </axml:sc>
  </c>
 </a>
```

The subscription is part of another AXML document, called *docsub.xml*. It specifies the name of the document on which the view is defined, i.e. *doctest.xml*, a TPQ, a template and a *BY* clause.

```
<trace xmlns:axml="http://futurs.inria.fr/gemo/axml/">
  <monitoringTask>
    <axml:sc axml:id="task1">
     <axml:return>
      <axml:append/>
     </axml:return>
     <axml:ws-soap
       endpoint="http://localhost:6969/MyPeer/services/AxlogService">
       <s:subscribe xmlns:s="n/a">
          <s:subscription>
            <document docID="doctest.xml"/>
            <query>
               <tree>
                 <node label="a" link="/">
                   <node label="b" link="/">
                     <node label="d" link="/">
                        <node label="e" link="/">
```

```
                    <node id="i1" label="*" link="/" output="true"/>
                  </node>
                  <node label="a" link="/">
                    <node id="i2" label="*" link="/" output="true"/>
                  </node>
                  <node label="b" link="/">
                      <node id="i3" label="*" link="/" output="true"/>
                  </node>
               </node>
             </node>
             <node label="c" link="/">
                <node label="u" link="/">
                   <node label="x" link="/">
                     <node id="i4" label="*" link="/"/>
                   </node>
                   <node label="y" link="/">
                     <node id="i5" label="*" link="/"/>
                   </node>
                   <node label="z" link="/">
                      <node id="i6" label="*" link="/" output="true"/>
                   </node>
                </node>
             </node>
           </node>
           <conditions>
              <condition>
                 <op>i1</op><relation>equ</relation><op>i4</op>
              </condition>
              <timecondition> i5 - i1 lt 150000 </timecondition>
           </conditions>
       </tree>
      </query>
      <template><out><o1>{$i1}</o1><o2>{$i2}</o2><o3>{$i3}</o3><o4>{$i6}</o4></out>
      </template>
      <by type="serviceCall">
        <ns2:address xmlns:ns2="http://futurs.inria.fr/gemo/axml/service/Algebra"
          endpoint="http://localhost:6969/MyPeer/services/ReceiveOperator">
          <ns2:currentID>
            <ns2:peerID>MyPeer</ns2:peerID>
            <ns2:docID>docsub.xml</ns2:docID>
            <ns2:nodeID>task1</ns2:nodeID>
         </ns2:currentID>
       </ns2:address>
     </by>
</s:subscription>
 </s:subscribe>
```

```
   </axml:ws-soap>
  </axml:sc>
 </monitoringTask>
</trace>
```

1.2 Configuration

The whole configuration is done in the *web.xml* file of the AXML peer.

Several things can be configured:

- *host_name* this is the name of the host of AXML peer, e.g. *localhost* or an IP address
- *peer_url* this is the URL of the AXML peer, e.g. `http://localhost:6969/MyPeer`
- *exist_url* this is the URL of the eXist database that AXML (and axlog) uses, e.g. `xmldb:exist://localhost:6969/exist/xmlrpc/db`
- *port* this is the port number where the alerter publishes observations, e.g. 9900
- *timestamping* this is a delay, in seconds, between two timestamping sessions for a document, e.g., 100 (for 100 seconds)
- *reevaluation* the views are recompiled periodically, this is the delay in seconds between two recompilation sesions, e.g., 1000 (for 1000 seconds)
- *persistence* this is set to *yes* if the views are persistent, i.e. they are stored in the eXist DB and reloaded when the peer restarts
- *test* set to *yes* if the default battery of axlog tests is to be run when the peer starts
- *storage* set to *yes* if the memory is cleaned between updates handled by axlog, otherwise axlog data is memory-resident
- *debug* set to *yes* if axlog functions in verbose mode

1.3 Installing AXML with Axlog and testing it

Remark. A distribution has been compiled for Windows, but you will still need to download sources for checking that everything works fine (the sources contain also test files) or recompile the distribution.

1.3.1 Install Binaries

1. download a `AXMLaxlogDistrib1_0.zip`, `exist.war` and `demo.zip` from the *Files* section of the `ICDE2009proj` project. The address of the project is `https://gforge.inria.fr/frs/?group_id=1356`

2. unzip `AXMLaxlogDistrib1_0.zip`, this is actually a Tomcat server that has two AXML peers as Web applications, i.e. `MyPeer` and `peer1`

3. add `exist.war` to the webapps directory of the Tomcat and a brand new eXist webapp manager of a database will be created when you start Tomcat for the first time

4. make sure that `CATALINA_HOME` is set to the path of the Tomcat installation directory

1.3.2 Obtain the sources and create a development environment

1. make sure that the *ant* tool is installed, so that the `build.xml` scripts can be run using the *ant* tool

2. download into a directory that will become the `<AXLOG_HOME>` directory the sources from the SVN of the project ICDE2009proj available at

   `https://gforge.inria.fr/scm/?group_id=1356`

3. set the `AXLOG_HOME` variable of your environement to the path of the `<AXLOG_HOME>` directory

4. if you want to compile the sources and install the resulted binaries, run the `build.xml` scripts from the root of the `<AXLOG_HOME>` directory and from `<AXLOG_HOME>/services/subscription`

1.3.3 Demonstration

Let's consider that you want to use the peer *MyPeer*. You can check the installation by setting the parameter *test* to the *yes*-value in the web.xml of the peer *MyPeer*. When the peer starts (this happens when you start Tomcat), it will automatically run the tests.

If you want to do play with the distribution, unzip demo.zip. It contains 4 files: *docsub.xml*, *doctest.xml*, *sourcef.xml* and *sourceg.xml*. The first two files will need to be loaded in the eXist collection corresponding to the *MyPeer* peer, e.g. `/db/MyPeer`, while the three *source*.xml* files will be loaded into the `/db/` collection. You can do this by using the graphical interface of eXist.

You can use the AXML peer's interface to activate service calls, e.g., for MyPeer an example of URL would be `http://localhost:6969/MyPeer`. The *docsub.xml* has a *task*1 service call that sends a subscription to the axlog service of the same peer, asking for the monitoring of the *doctest.xml* document. You can first activate this service call. Then, you could activate the service calls of the document *doctest.xml*, for instance first activate the f service call (the service will stream data from *sourcef.xml*) and then the g service call (the service will stream data from *sourceg.xml*). The view data will be uploaded in the document *docsub.xml*.

Chapter 2

The Developer's Guide

The current version of axlog supports : *active updates*, *end-of-stream* and *time queries*. The nodes of the document are timestamped. TPQs are enriched with systems of inequality constraints. Axlog does satifiability evaluation for tuples with equality constraints only. A module for solving systems of constraints could be added for computing satisfiablity with inequalities, e.g. JaCoP `http://jacop.osolpro.com/`. Axlog does not support non-monotone features like negative queries or deletes.

Timestamping of the axml documents' nodes for now is performed periodically for all the nodes in a document. A more "incremental" way of doing it should be adopted in the future. The timestamping is exact: the roots of the inserted subtrees are already timestamped by the core module of AXML, so the timestamping done by axlog consists of propagating the timestamp value of the subtree's root to all the nodes of the subtree.

2.1 Model (axlog.model.query)

This package holds the classes that model the tree-pattern queries. The most important are *QueryTree* - that represents the tree-pattern, and *QueryNode* - that represents a node of this tree.

A QueryTree typically has a QueryNode as root, a set of equality and inequality constraints and a set of output nodes. A QueryNode is part of a QueryTree,has a label, has a father node (which is null in case this node is the root), with which is in either a "/" or "//" relation and might appear in the output, or might be needed to solve join constraints. It has also a set of needed descendants, meaning it has to propagate their matches upwards. There is also a *Constraint* class hierarchy for modeling (equality or inequality) constraints. The equality constraints form equivalence classes (sets of QueryNode's) registered with a QueryTree.

2.2 Model to XQuery (axlog.querytranslator)

The XQueryGenerator class in the *querytranslator* package has methods for building XQuery queries for the persistent XML data, either for looking for satisfiable tuples and for satisfied ones. For instance, the methods *generateXQuery** are used to generate XQuery queries that extract tuples that are satisfied, as well as a *plan*-an XML document. This *plan* is the support of the datalog program that maintains the view and for the computation of scenarios, see the *Datalog* section. The other methods build the XQuery queries that correspond to sub-patterns of the tree-pattern query and are used to extract the satisfied tuples only.

2.3 Engine (axlog.engine)

The engine consists of three components: a *ViewCompiler*, a *Maintainer* and a *Tuner*. Subpackages and classes have been defined for each of them.

A ViewCompiler typically builds a view from an AXML document and a tree-pattern query. Several constructors have been defined, depending whether the document is provided in memory or is in an AXML peer's DB and whether a QueryTree object (for the tree-pattern query) has been already built or needs to be extracted from an XML fragment. Typically, when recompiling a view, one wants to reuse the same QueryTree object, because the pattern nodes have been given identifiers that need to remain the same.

A Maintainer typically searches for the right relation (filter) in the right datalog program in order to launch an incremental evaluation of the program with the inserted data as input.

A Tuner will do a scenario analysis, will generate the appropriate filters and will update the index structure of the ViewManager, by adding filters for the relevant functions.

2.4 View (axlog.view)

This package contains two classes: the class *View* that models views on AXML documents and a class *ManagerViews* that indexes all the views registered with the system.

A View object contains all the informations related to the maintenance of a view on an active document. A View object has mainly three entries, for a (View)Compiler, for a Maintainer and for a Tuner. The system supports full recompilation for views, performed periodically, and incremental recompilation, that is performed whenever there is a new AXML tree (an active update) received by the document.

The Manager of views indexes the view objects with respect to the functions that need to be monitored for them: the key is a function, the value is a set of view objects. Besides methods for registering/unregistering a view, this class has three methods that are called depending on the type of the update detected:

maintain for standard XML trees, *activeUpdate* for AXML trees, and *eos* for end-of-stream messages.

2.5 Utility classes (axlog.utility)

This package contains very important classes: *AxlogServlet* and *Utility.*

AxlogServlet is the entry point of the module and the place where most of the configurations take place. That is because the class is a servlet, it is started when the AXML peer starts and has access to all the configuration parameters available in the web.xml file. If it is configured properly, the AxlogServlet starts predefined tests and pre-loads axlog subscriptions that are persistent in the eXist database.

The Utility class contains several static methods that are of much help for translating between several data models (DOM, StAX etc.), or for transfering data between memory and the eXist database.

2.6 Datalog (axlog.datalog)

These are the classes that implement datalog for View Maintenance over AXML documents. Operators (FilterOnFunction, Join, Unions, Operations) and Tuples and Relations are all Entity classes. The transfers between the memory and the database are done at the level of the Relation class. The tuples are produced by the FilterOnFunction classes (that correspond to Filter/Projectors associated to a sub-pattern). They climb upwards the object hierarchy and are transformed by the other operators. In the end, they reach the *Plan* root that registers them with the View.

2.7 The Subscription Parser (axlog.languageparser) and the Input/Output (axlog.environment)

This package contains only one class, the ViewSubscriptionParser, that parses an axlog subscription expressed in XML and builds a View for it, registers it with the Manager of Views and adds environment elements to the view: a template (*ReformatTuple*) and a dissemination mechanism (*Sender*). The template builds an XML document out of a tuple and the sender sends a set of tuples to a particular Web service, using the information extracted by the ViewSubscriptionParser. The parser is also in charge with storing the subscription definition in the eXist space of axlog. Every tuple that is added to the view is also stored in this subscription space on eXist, so that when re-loaded it is avoided resending data that has already been sent.

The connection with the alerter (on Web services) is done by COMAlerterAXML (client to the alerter - "server of updates") and by WSAlerterIn that processes the incoming messages for ReceiveService (the inserts in the AXML documents).

2.8 Periodic Tasks (axlog.timedtasks)

There are two kinds of tasks that are run periodically: the timestamping of the monitored documents and the reevaluation of satisfiability and satisfaction. Both types of tasks are implemented as Java TimerTask objects associated with a Timer object.

The first one is necessary because all the data is extracted by tree-pattern queries with the timestamps. But AXML by default puts timestamps only at the roots of the inserted subtrees. These tasks propagate the timestamps downwards to all the elements in the subtrees.

The second type of tasks redo the scenario analysis and retune the system. When the reevaluation happens, the set of tuples newly obtained is compared with the ones obtained previously. Only the tuples that where not previously in the view will be considered for reformatting with the template and for sending to the proper recipient.

The delays between two tasks are configurable in the web.xml file. Read the User's Guide to see how.

Bibliography

[1] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Efficient Maintenance Techniques for Views over Active Documents. International Conference on Extending Database Technology, Saint-Petersburg, Russia, 2009.

[2] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Distributed Monitoring of Peer to Peer Systems (demo). International Conference on Data Engineering, Cancun, Mexico, 2008.

[3] Serge Abiteboul, Pierre Bourhis and Bogdan Marinoiu. Satisfiability and Relevance of Queries for Active Documents. In *Proc. PODS*, Providence, Rhode Island, USA, June 2009.