Paxtools User's Guide

Emek Demir, B. Arman Aksoy

March 18, 2008

1 Introduction

BioPAX is a collaborative effort to create a data exchange format for biological pathway data.

Paxtools is a library for accessing and manipulating BioPAX. Software tools that use BioPAX, such as exporters, importers, analysis algorithms or editors can use Paxtools as their core BioPAX API. Paxtools supports **BioPAX Level 2** [2], and can automatically convert **Level 1** [1] models to **Level 2**. The latest release candidate for **BioPAX Level 3** is fully supported, but it is still in beta as BioPAX Level 3 is not finalized [3].

Using Paxtools, a user can read a BioPAX model from a file or create a new model in memory. A model is a container for BioPAX elements, and has methods for querying and retrieving its contents. In a BioPAX model, elements are represented as plain java beans, with getter and setter methods for their BioPAX properties. A user can create and add new BioPAX elements to the model, remove existing elements or modify their properties. It is also possible for users to traverse the corresponding BioPAX graph to retrieve interesting subgraphs.

Paxtools provides a domain object model (DOM) with strong typing for domains and ranges of the properties. It checks whether cardinality constraints are observed. Rules that were documented but could not be represented formally in BioPAX OWL are also validated against. Paxtools is fail-fast, it immediately reports rule violations either with an exception or logging a warning. Additionally, Paxtools checks for best practices documented in the BioPAX documentation (L2 and L3 and logs potentially problemmatic usages.

Paxtools comes with a persistence and searching layer using Java Persistence API (JPA) and Lucene. This enables storing and querying particularly large models and concurrent editing of the same model by multiple users. Lucene allows efficient fulltext querying.

Paxtools has an experimental support for integrating two pathways from different sources by identifying interactions that are similar. It is often difficult to make an exact matching between entities in several cases- for example the same state of the protein can be represented as the *active* state in one pathway database and *doubly phosphorylated* in another one. An iterative two-step process– fuzzy-matching entities followed by finding interactions that have similar participants– however provides quite reasonable pathway alignments. This facility can be used for cross-validating pathways from different sources or as a first step for integrating overlapping pathways.

Paxtools has also several other useful functionalities such as reflection based access, traversal methods, and programmatic access to Pathway Commons [4] web service API.

2 Usage notes

2.1 Paxtools API

2.1.1 Reading a model

Paxtools is very straightforward to use. For example, in order to read an OWL model from an *inputStream* using *jenaIOHandler* you need to:

```
JenaIOHandler jenaIOHandler = new JenaIOHandler();
Model model = jenaIOHandler.convertFromOWL(inputStream);
```

or you can fetch a model directly from Pathway Commons Database [4] if you have its CPATH id[1]:

```
String id = "1";
PathwayCommonsIOHandler pcIOHandler
    = new PathwayCommonsIOHandler();
Model model = pcIOHandler.retrieveById(id);
```

2.1.2 Access to contents of a model

You can retrieve *model*'s contents by

```
model.getObjects();
```

or you can use the filtering feature in order to get only interactions:

```
model.getObjects(interaction.class);
```

When you have an interaction, getting its participants is as easy as:

```
Set<InteractionParticipant> participants
    = anInteraction.getPARTICIPANTS();
```

In order to create an object, you can use a factory through the default *BioPAXLevel* of the model:

```
BioPAXFactory factory = model.getLevel.getDefaultFactory();
biochemicalReaction rxn1
    = (biochemicalReaction) factory.reflectivelyCreate(biochemicalReaction.class);
```

[1]Please see Section 5.1 for further information

or you can ask model to create one for you :

```
biochemicalReaction rxn1
    = model.addNew(biochemicalReaction.class, "http://mydomain.org/rxn1");
```

The first parameter is the class of the object and the latter is the RDF id. The latter method also adds the newly created object to the model.

A *Model* object contains multiple BioPAX elements. Each element within a model has to have a unique id. A model can have multiple namespaces and these namespaces can be obtained as a map in which *key* is the prefix and *value* is the namespace:

```
Map<String, String> nspMap = model.getNameSpacePrefixMap();
```

The namespace with empty (“”) tag is used as the XML base when writing out the model as an OWL file.

2.1.3 Modifying the contents of a model

Having a new biochemical reaction (*rxn1*), now you can define the reaction’s *LEFT* and *RIGHT*:

```
rxn1.addLEFT(aSubstrate);
rxn1.addRIGHT(aProduct);
```

For functional properties (one-to-one or many-to-one properties) there are two methods: *getX* and *setX*, where *X* is the name of the property. For other properties (one-to-many or many-to-many) there are four methods: *getX*, *setX*, *addX*, and *removeX*. Regular users are strongly recommended to use *addX* and *removeX* methods for modifying the latter properties, and should not directly modify the list returned by the get methods, because Paxtools maintain super-properties automatically without a need for an extra effort for the user.

So in the above example if we get the set containing participants by the following method

```
rxn1.getPARTICIPANTS()
```

it will contain both *aSubstrate* and *aProduct*.

Additionally there are methods in the form of *isXOf* for the properties between major elements. These methods allow you to traverse the graph bidirectionally and are automatically maintained by Paxtools. So the following will evaluate to true:

```
Set<Process> participating = aSubstrate.isPARTICIPANTSOf();
participating.contains(rxn1);
```

2.1.4 Exporting a model

1. **OWL file:** Once you are done with your model you can write it out using one of the writers in the IO package. For example, in order to output the model as an OWL file, you need to:

```
SimpleExporter exporter = new SimpleExporter();
exporter.convertToOwl(model, anOutputStream);
```

2. **SIF file:** You can also export your *model* in simple interaction format (SIF) with the interaction rules you specify:

   ```
   SimpleInteractionConverter converter
       = new SimpleInteractionConverter(
           new ParticipantRule(),
           new ConsecutiveCatalysisRule() );
   converter.writeInteractionsInSIF(model, anOutputStream);
   ```

   In the example above, only *Participant* and *ConsecutiveCatalysisRule* were requested, but user can give a different list of rules. These rules are defined in package *org.biopax.paxtools.io.sif.level2*. Here is another example using different rules:

   ```
   SimpleInteractionConverter converter
       = new SimpleInteractionConverter(
           new ControlRule(),
           new ParticipantRule(),
           new ComponentRule() );
   converter.writeInteractionsInSIF(model, anOutputStream);
   ```

2.1.5 Merging models

If you want to merge two or more models, you can initiate the *Merger* as the following:

```
JenaEditorMap editorMap = new JenaEditorMap();
Merger merger = new Merger(editorMap);
```

Assuming you want to merge two models (*targetModel* and *sourceModel1*), in order to merge *sourceModel1* into *targetModel*, you need to:

```
merger.merge(targetModel, sourceModel1);
```

or if you want to merge more than two models, you can use:

```
merger.merge(targetModel, sourceModel1, sourceModel2,
                           sourceModel3, sourceModel4);
```

Note that in the end of a successful merging, *targetModel* will be the merged model.

2.1.6 Putting it all together

In Section 5.3, you can find a complete example in which some common methods–along with those mentioned above–are put together.

2.1.7 Exceptions

If at any point a BioPAX invariant is violated Paxtools will throw an *IllegalBioPAXArgument* exception without modifying the model.

2.2 Paxtools Command Line Tool

Paxtools has a command line accessible tool in order to enable users to perform basic operations. You can access the main class via *PaxtoolsMain* under *org.biopax.paxtools* package. It is also manifested as the standard main class, therefore the following usage will also invoke this command tool[2]:

```
$ java -jar /path/to/Paxtools.jar

Avaliable operations:
--merge file1 file2 output      merges file2 into file1 and writes
                                it into output
--to-sif file1 output           converts model to simple interaction
                                format
--validate file1                validates BioPAX model file1

--help                          prints this screen and exits
```

where the dolar sign ($) indicates the command line prompt. There are three basic functionalities that come with the comand line tool (CLT): validation, merging, and conversion to SIF. To validate a BioPAX OWL, you can use the following command:

```
$ java -jar /path/to/Paxtools.jar \
    --validate /another/path/to/biopax.owl

Model is valid
```

There are two possibilites for *validation*: First one is shown above, and the second is in the following format:

```
$ java -jar /path/to/Paxtools.jar \
    --validate /another/path/to/biopax.owl

Model is invalid:
Exception : Description
```

where *Exception* is the thrown exception type while reading the model, and *Description* is the relavent information regarding to the problem. Problem(s) may include but not limited to: RDF syntax errors or semantics of model.

If you want to merge two BioPAX OWL models, *model1.owl* and *model2.owl*), and want to write it as *merged_model.owl*, you need to:

```
$ java -jar /path/to/Paxtools.jar \
    --merge model1.owl model2.owl merged_model.owl
```

or you may want to convert an OWL model, *model.owl* into a SIF[3], *converted_model.sif*, you can use:

```
$ java -jar /path/to/Paxtools.jar \
    --to-sif model.owl converted_model.sif
```

[2]Assuming you have the final JAR release of Paxtools under **/path/to/Paxtools.jar**.

[3]Please see Section 5.2 for further information

Programmers who are going to use this CLT as a part of their program or script should note that *PaxtoolsMain* returns 0 in success, and a *non-zero* integer in case of error:

```
$ java -jar /path/to/Paxtools.jar \
    --validate /another/path/to/biopax.owl

Model is valid
$ echo $?
0

$ java -jar /path/to/Paxtools.jar \
    --validate /another/path/to/biopax.owl

Model is invalid
Exception : Description
$ echo $?
255
```

Users who want to alter parameters or require a more dedicated usage of these basic methods are suggested to use *Paxtools API*.

3 Package Structure

3.1 org.biopax.paxtools.util

This package contains several utility classes that are used throughout the project, including exceptions and special collection classes.

3.2 org.biopax.paxtools.model

Model package contains the interfaces for BioPAX objects. Software applications using paxtools should code to these interfaces. Actual implementations are initialized through factories decoupling the applications from the actual implementations.

3.3 org.biopax.paxtools.impl

This is the default implementation of *model* package. It contains most of the domain logic as well as validation functionality.

3.4 org.biopax.paxtools.proxy

This is the implementation layer used for persistence. Proxy pattern is used to redirect all domain and validation logic to an underlying model layer, while handling persistence related issues.

3.5 org.biopax.paxtools.controller

This package contains classes used for manipulating model objects: *PropertyEditors* for reflection based access and modification of model objects, a *traverser* and a *visitor* for generic graph traversal, and a *merger* for merging two biopax models based on graph identity. There is also an experimental semantic integration class for comparing two models and finding similar interactions.

3.6 org.biopax.paxtools.io

This package contains classes used for reading and writing BioPAX models in different formats.

3.7 org.biopax.paxtools.commander

This package contains classes for editing BioPAX models. It has commands for several edit operations, undo/redo stack, and command line code completion facility.

3.8 org.biopax.paxtools.examples

This package contains several examples that uses Paxtools.

4 Dependencies

4.1 Production

4.1.1 General

- collections-generic-4.01.jar
- commons-logging.jar
- xercesImpl.jar[4]
- xml-apis.jar[5]

4.1.2 jena

Only required if *JenaIOHandler* is used.

- concurrent.jar
- icu4j_3_4.jar
- iri.jar
- jakarta-oro.jar

[4]included in jvm 1.5 or higher
[5]included in jvm 1.5 or higher

- jena.jar

4.1.3 Persistence

Only required if database persistence is used.

- antlr-2.7.6.jar
- asm.jar
- cglib-2.1.3.jar
- commons-collections-2.1.1.jar
- dom4j-1.6.1.jar
- ehcache-1.2.3.jar
- ejb3-persistence.jar
- hibernate-annotations.jar
- hibernate-commons-annotations.jar
- hibernate-entitymanager.jar
- hibernate-search.jar
- hibernate3.jar
- javassist.jar
- jboss-archive-browsing.jar
- jdbc2_0-stdext.jar
- jta.jar
- lucene-core-2.1.0.jar

4.1.4 DBMS driver

- mysql-connector-java-5.0.7-bin.jar[6]
- postgresql-8.2-504.jdbc3.jar[7]

4.2 Development

Not required on run-time.

- junit-4.1.jar
- log4j-1.2.12.jar

[6]In case MySQL is used for the persistence
[7]In case PostgreSQL is used for the persistence

5 Appendices

5.1 Pathway Commons Integration

Users can access the following Pathway Commons web service API commands via the corresponding methods under *org.biopax.paxtools.io* package: *get_record_by_cpath_id* via *retrieveByID*[8], *get_neighbors* via *getNeighbors*[9], and *get_pathways* via *getPathways*[10].

As documented in the help of the web service [5], some commands support multiple IDs: "a comma separated list of internal or external identifiers (IDs), used to identify the physical entities of interest. For example, look up two distinct proteins by their UniProt IDs using the following query: O14763, P55957. To prevent system overload, clients are currently restricted to a maximum of 25 IDs." Therefore, like the use in Section 2.1.1 the following use of the method is also valid:

```
String id = "1,2,3,4,5,6,7,8,9";
PathwayCommonsIOHandler pcIOHandler
    = new PathwayCommonsIOHandler();
Model model = pcIOHandler.retrieveById(id);
```

Moreover, Pathway Commons interface supports different ID types which can be indicated by *setInputIdType* method before fetching the model from the database.

5.2 Simple Interaction Format (SIF)

A SIF file can be defined as:

"*... a simple text file that lists all the interactions in the following simple interaction format: **physical_entity_id <relationship type> physical_entity_id**, where physical_entity_id is a valid CPATH_ID...*" [5]

where <**relationship type**> is one of the followings: COMPONENT_OF, COMPONENT_IN_SAME, CO_CONTROL_DEPENDENT_SIMILAR, CO_CONTROL_DEPENDENT_ANTI, CO_CONTROL_INDEPENDENT_SIMILAR, CO_CONTROL_INDEPENDENT_ANTI, SEQUENTIAL_CATALYSIS, CONTROLS_METABOLIC_CHANGE, CONTROLS_STATE_CHANGE, PARTICIPATES_CONVERSION, or PARTICIPATES_INTERACTION.

SIF files are also recognized by **Cytoscape** [6], and along with other network types a detailed SIF description can also be found Cytoscape User Manual.

[8]http://www.pathwaycommons.org/pc/webservice.do?cmd=help#get_by_cpath_id

[9]http://www.pathwaycommons.org/pc/webservice.do?cmd=help#get_neighbors

[10]http://www.pathwaycommons.org/pc/webservice.do?cmd=help#get_pathways

5.3 A case study: *Converting GO Unification XREFs to Relationship XREFs*

```
package org.biopax.paxtools.examples;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.biopax.paxtools.impl.level2.Level2FactoryImpl;
import org.biopax.paxtools.io.jena.JenaIOHandler;
import org.biopax.paxtools.model.BioPAXLevel;
import org.biopax.paxtools.model.Model;
import org.biopax.paxtools.model.level2.Level2Factory;
import org.biopax.paxtools.model.level2.XReferrable;
import org.biopax.paxtools.model.level2.relationshipXref;
import org.biopax.paxtools.model.level2.unificationXref;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.lang.reflect.InvocationTargetException;
import java.util.HashSet;
import java.util.Set;

/**
 * User: Emek Demir Date: Jan 18, 2007 Time: 4:56:53 PM
 *
 * In this example we get all the unification xrefs in the model
 * and check if they point to the Gene Ontology. If this is the case
 * we convert them to relationship xrefs.
 */
public class GOUnificationXREFtoRelationshipXREFConverter
{
private static Log log = LogFactory.getLog(
  GOUnificationXREFtoRelationshipXREFConverter.class);

    static Level2Factory factory = new Level2FactoryImpl();

static JenaIOHandler jenaIOHandler =
  new JenaIOHandler(factory, BioPAXLevel.L2);

    /**
      *
      * @param args a space seperated list of owl files to be processed
      * @throws IllegalAccessException
      * @throws InvocationTargetException
```

```
        */
      public static void main(String[] args)
   throws IllegalAccessException,  InvocationTargetException
{
            // Process all the args
            for (String arg :  args)
            {
                log.info(arg);
                if (arg.toLowerCase().endsWith("owl"))
                {
                    try
                    {
                        processXrefs(arg);
                    }
                    catch (FileNotFoundException e)
                    {
                        e.printStackTrace();
                    }
                }

            }
}

      /**
        * Main conversion method. Demonstrates how to read and write a BioPAX
        * model and accessing its objects.
        * @param arg file name to be processed
        * @throws FileNotFoundException
        * @throws IllegalAccessException
        * @throws InvocationTargetException
        */
      private static void processXrefs(String arg)  throws
   FileNotFoundException,
   IllegalAccessException,
   InvocationTargetException
{
   //Read in the model
            FileInputStream in =
     new FileInputStream(new File(arg));
   Model level2 =
     jenaIOHandler.convertFromOWL(in);

            //Get all unification xrefs.
            Set<unificationXref> unis =
     level2.getObjects(unificationXref.class);
   //Create another set for avoiding concurrent modifications
```

```
        Set<unificationXref> gos = new HashSet<unificationXref>();

        //Process all uni. refs
        for (unificationXref uni :  unis)
  {
   log.trace(uni.getDB());
   //Assuming DB is represented as "GO"
            if (uni.getDB().equalsIgnoreCase("GO"))
   {
    //this it to avoid concurrent modification.
    log.info("scheduling "+ uni.getRDFId());
    gos.add(uni);

   }
  }
        //Now we have a list of xrefs to be converted. Let's do it.
        for (unificationXref go :  gos)
  {
   convert(go, level2);
  }
  //And finally write out the file. We are done !
        jenaIOHandler.convertToOWL(level2, new FileOutputStream(
   arg.substring(0, arg.lastIndexOf('.'))  +
    "-converted.owl"));
}

    /**
     * This method converts the given unification xref to a relationship xref
     * @param uni xref to be converted
     * @param level2 model containing the xref
     */
    private static void convert(unificationXref uni, Model level2)
{
  //We can not simply convert a class, so we need to remove the
        //uni and insert a new relationship xref

        //First get all the objects that refers to this uni
        Set<XReferrable> referrables =
   new HashSet<XReferrable>(uni.isXREFof());

        //Create the new relationship xref.
        relationshipXref relationshipXref =
   factory.createRelationshipXref();

        //Copy the fields from uni
        relationshipXref.setRDFId(uni.getRDFId());
  relationshipXref.setCOMMENT(uni.getCOMMENT());
  relationshipXref.setDB(uni.getDB());
  relationshipXref.setDB_VERSION(uni.getDB_VERSION());
  relationshipXref.setID(uni.getID());
  relationshipXref.setID_VERSION(uni.getID_VERSION());
  relationshipXref.setRELATIONSHIP_TYPE(
   "http://www.biopax.org/paxtools/convertedGOUnificationXREF");

            //Add it to the model.
            level2.add(relationshipXref);

            //Create a link to the new xref from all the owners.
            for (XReferrable referrable : referrables)
  {
   referrable.addXREF(relationshipXref);
  }

            //Remove the references to the old uni
            for (XReferrable referrable : referrables)
            {
                referrable.removeXREF(uni);
            }
            //Now remove it from the model.
            level2.remove(uni);

            //We are done!
    }
}
```

References

[1] BioPAX - Biological Pathways Exchange Language *Level 1* (http://www.biopax.org/release/biopax-level1-documentation.pdf)

[2] BioPAX - Biological Pathways Exchange Language *Level 2* (http://www.biopax.org/release/biopax-level2-documentation.pdf)

[3] BioPAX - Biological Pathways Exchange Language *Level 3* (http://www.biopax.org/release/biopax-level3-documentation.pdf)

[4] Pathway Commons (http://www.pathwaycommons.org/)

[5] Pathway Commons Web Service Help (http://www.pathwaycommons.org/pc/webservice.do?cmd=help)

[6] Cytoscape (http://www.cytoscape.org)