PAXTOOLS DEVELOPERS

Using Biological Pathway Data with Paxtools

A USER'S GUIDE

Contents

Introduction

Extraordinary advances in sequencing and other molecular technologies have led to a large increase in data about biological processes. The total volume of pathway data mapped by biologists and stored in databases has entered a rapid growth phase, with the number of online resources for pathways and molecular interactions increasing 70%, from 190 in 2006 to 325 in 2010 [1]. Unfortunately, most of these databases were originally developed to use their own pathway representation language, resulting in a heterogeneous set of resources that are extremely difficult to combine and use. It is therefore imperative to develop computational methods to cope with both the magnitude and fragmented nature of this rapidly expanding and exceedingly valuable pathway information.

A key component of this infrastructure is a common standard language. BioPAX (Biological Pathway Exchange) [2] is an OWL(Web Ontology Language) based community developed language for representing pathway data. Software developers can use BioPAX to maximize their application's access to pathway data from different sources. Similarly pathway database providers can reach a greater number of users through BioPAX exports.

BioPAX covers a large spectrum of pathway data, including signaling and metabolic pathways as well as genetic and protein-protein interactions. Developing software that produces or consumes BioPAX can be a daunting task at first because of the large number of classes and properties that BioPAX contains. Paxtools addresses this by offering a Java API that maps OWL classes and properties to Java beans and methods that can significantly reduce software development time.

Paxtools can help scientists to quickly access, analyze and manipulate pathway information. Developers of BioPAX exporters, visualization tools, editors or algorithms can use Paxtools to significantly reduce time spent on BioPAX. Developing applications with Paxtools also offers future flexibility and extensibility.

[1] Bader, G. D., Cary, M. P., & Sander, C. 2006, Pathguide: a pathway resource list., Nucleic acids research, 34, D504, D504

[2] Demir, E., Cary, M. P., Paley, S., et al. 2010, The BioPAX community standard for pathway data sharing, Nat Biotechnol, 28, 935, 935

Getting Started

Features of Paxtools

Historically, Paxtools was developed to provide an OWL[3] -Java mapping, but today it offers much more. There are utilities for quickly traversing or manipulating objects, fixing common problems or converting pathway models to different BioPAX levels or other standards and formats. There are also BioPAX JPA (Java Persistence API) mappings for persisting large BioPAX models into a database and advanced graph queries for finding biologically relevant paths between molecules. All of these are organized in a lightweight modular structure to help you embed only the portions you need within your application with minimum footprint. Here is a list of some features:

A COMPLETE AND CONSISTENT IMPLEMENTATION OF BIOPAX SPECIFICATION: BioPAX elements in Paxtools are plain Java beans which provide methods to access the properties described in BioPAX, and a model, acting as a container for all BioPAX elements, provides querying facilities for them. Users can either read a BioPAX model from a file or create an empty one from scratch. Methods to add new elements to a model and to remove elements from a model are also provided.

SUPPORT FOR OWL PROPERTIES AND ADDITIONAL INVERSE LINKS: OWL properties can be symmetric, transitive or subtyped into other properties. These semantics can not be represented directly in an object oriented programming language. Paxtools implements these additional semantics and automatically updates the fields of objects as needed.

SYNTACTIC VALIDATION: Each operation that modifies the model is internally validated by Paxtools to comply with BioPAX syntax, including RDF well-formedness, domain and range restrictions, bidirectional links, and redundancies.

Seamless handling of different BioPAX levels: Recently released BioPAX Level 3 introduced significant improvements to the naming and structure of the BioPAX at some cost of backwards compatibility. Paxtools supports all three BioPAX levels (1, 2 and 3) and provides facilities for upgrading older BioPAX models to Level 3, reducing the burden of working with different BioPAX levels for developers.

Converting to and from different formats: Paxtools can convert PSI-MI[4] models to BioPAX Level 3. In addition, BioPAX models can be exported back to OWL and several other useful formats, including SBGN(Systems Biology Graph Notation)[5], SIF (Simple Interaction Format) and GSEA (Gene Set Enrichment Analysis)[6].

Efficient traversal and editing via reflection: Paxtools allow tools to manipulate BioPAX models without actually hard coding property and class names. This pattern considerably simplifies development of BioPAX exporters and other tools and makes it easier to extend and update them to support future changes in the BioPAX specification. PathAccessors allow XPath like querying and traversal of the object graph.

Direct Access to PC: PC (Pathway Commons)[7] is a convenient point of access to biological pathways collected from public pathway databases. Pathway Commons aggregates, validates and normalizes pathway information significantly facilitating its use for biological applications. Paxtools have a built in client that allows you to access PC, run field and graph queries and obtain results as BioPAX models.

Advanced Graph Queries: Paxtools offers graph queries, such as "paths-between" or "common up/downstream" that are specifically customized based on BioPAX semantics for answering common biological questions.

Modular and lightweight structure: The Paxtools core module, which provides a complete implementation of BioPAX and read in/write out functionality, is a very compact library(400 kB) that can be extended as required with additional modules. Paxtools is distributed as a Maven project which allows developers to easily select just the parts of Paxtools they need in their application.

A platform for development of BioPAX software infrastructure: A growing BioPAX software infrastructure, available as a part of the Pathway Commons project, is directly built on top of

[4] Kerrien, S., Orchard, S., Montecchi-Palazzi, L., et al. 2007, Broadening the horizon–level 2.5 of the HUPO-PSI format for molecular interactions, BMC Biol, 5, 44, 44

[5] Le Novère, N., Hucka, M., Mi, H., et al. 2009, The Systems Biology Graphical Notation, Nat Biotechnol, 27, 735, 735

[6] Subramanian, A., Tamayo, P., Mootha, V. K., et al. 2005, Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles., Proceedings of the National Academy of Sciences of the United States of America, 102, 15545, 15545

[7] Cerami, E. G., Gross, B. E., Demir, E., et al. 2011, Pathway Commons, a web resource for biological pathway data, Nucleic Acids Res, 39, D685, D685

Paxtools, including a state-of-the-art persistence system using Java Persistence API integrated with the querying facilities, an advanced validator that allows checking complex rules and best practices using an extensible framework, and a pathway alignment tool. Paxtools is also currently being used by several software tools including CellDesigner[8], Cytoscape[9] and ChiBE[10].

[8] Mi, H., Muruganujan, A., Demir, E., et al. 2011, BioPAX support in CellDesigner., Bioinformatics (Oxford, England), 27, 3437, 3437

[9] Cline, M. S., Smoot, M., Cerami, E., et al. 2007, Integration of biological networks and gene expression data using Cytoscape., Nature protocols, 2, 2366, 2366

[10] Babur, O., Demir, E., Gönen, M., Sander, C., & Dogrusoz, U. 2010, Discovering modulators of gene expression, Nucleic Acids Res, 38, 5648, 5648

Modules and Package Structure

Here is a quick overview of the modules and packages they contain:

- Core: Core module contains the model interfaces and their default implementation as well as some utility classes. All other modules depend on the core. Some important packages are:

  - model: This package (and its subpackages) defines the model interfaces that maps to BioPAX classes. All applications should be written to these interfaces rather than the actual implementation classes.

  - impl: Default implementation package.

  - controller: This package contains classes that contain logic for traversing, copying and merging BioPAX elements as well as property editors.

- io: Simple IO package for reading/writing BioPAX files.

- converter: This is to convert (upgrade) BioPAX Level1, 2 to Level 3.

- jena IO: This module contains an alternative Jena based handler for reading/writing BioPAX files.

- query: This module contains graph queries such as shortest-path.

- PSI-MI converter: This module converts PSI-MI 2.5 files to BioPAX Level 2 or 3.

- Sif, GSEA and SBGN converters: These modules export BioPAX models to SIF, GSEA and SBGN (work in progress) respectively.

Obtaining Paxtools

Release Bundles

You can download the "Fat" jars (with dependencies) from: `http://sourceforge.net/projects/biopax/files/paxtools/`
     This is a good choice to get things working now and worry about which modules to package with your application later.

Alternatively, individual modules (compiled JARs and POMs) can be obtained from the BioPAX maven repositories:

http://biopax.sourceforge.net/m2repo/snapshots and

http://biopax.sourceforge.net/m2repo/releases for snapshots and stable releases respectively.

Use modules to minimize the size of your application and use only the parts of Paxtools that you need. The "Core" module offers a complete implementation of BioPAX and read in/write out functionality in a very compact 400 kB package The fat jar can be as large as 16 MB most because of the Jena library and its dependencies.

Embedding Paxtools in a maven project

If you are already using Maven in your project, you can add Paxtools as a dependency. Add the following lines into your pom.xml, within the repositories section:

```xml
<repository>
  <id>biopax.releases</id>
  <name>BioPAX Repository at Sourceforge</name>
  <url>http://biopax.sourceforge.net/m2repo/releases/</url>
  <snapshots>
   <enabled>false</enabled>
  </snapshots>
</repository>
```

and, if you want to use the latest builds:

```xml
<repository>
  <id>biopax.snapshots</id>
  <name>BioPAX Snapshots Repository at Sourceforge</name>
  <url>http://biopax.sourceforge.net/m2repo/snapshots/</url>
  <releases>
   <enabled>false</enabled>
  </releases>
 </repository>
```

Then, add the modules of your choice in the <dependencies> element:

```xml
<dependency>
  <groupId>org.biopax.paxtools</groupId>
  <artifactId>paxtools-core</artifactId>
  <version>4.1.5</version>
</dependency>
```

Downloading sources

Checkout (read-only) using the Mercurial client:

```
hg clone http://biopax.hg.sourceforge.net:8000/hgroot/biopax/paxtools
```

More information on how to access to the source code can be found
here:
  http://biopax.sourceforge.net/paxtools/source-repository.
html

Documentation and Help

More information on Paxtools can be found at:
- Wiki: http://sourceforge.net/apps/mediawiki/biopax/index.php?title=Paxtools

- API: http://biopax.sourceforge.net/paxtools-<version>

- If you get stuck or have questions, use Paxtools support mailing list:
  https://lists.sourceforge.net/lists/listinfo/biopax-paxtools

- For general questions related to BioPAX, use BioPAX discuss mailing list:
  http://groups.google.com/group/biopax-discuss

Paxtools Basics

First Model

In order to create paxtools objects, we will need a factory. For most use cases, the default factory is a good choice. We can get a level 3 specific factory by[11]:

```
BioPAXFactory factory = BioPAXLevel.L3.getDefaultFactory();
```

BioPAXLevel is an enum that contains multiple level specific facilities and fields. These can be very useful if you want to implement level agnostic facilities. You can find more information about BioPAXLevel in the API documentation. A Paxtools model is a container for your objects. When you read, write or pass around BioPAX objects, these objects will be contained inside a model. This is useful for validation and querying purposes. Let's begin by creating a new model:

```
Model model = factory.createModel();
```

Set xml:base [12]:

```
model.setXmlBase("http://biopax.org/tutorial/");
```

Now we can create new BioPAX objects and insert them into the model:

```
Protein protein1 = model.addNew(Protein.class,
        "http://biopax.org/tutorial/test1");
```

This method will create a new instance of protein class and insert it into the model. The second parameter is a unique identifier for the protein state. [13] You can always access an object if you know their id by the model's getByID method:

```
Protein retrievedProtein = model.getByID(
        "http://biopax.org/tutorial/test1");
```

Modifying Objects

In the model package you will find interfaces for each class in BioPAX. All setter/getter methods for BioPAX properties in those interfaces follow the default java bean pattern - for single cardinality property TTT, there are two methods: setTTT/getTTT[14]. For multi-cardinality objects, Paxtools keeps a Set of objects. Two additional methods, ad-dTTT and removeTTT are provided for manipulating the content of these sets.

Let's modify the fields of our newly minted protein :

```
protein1.addName("Tutorial Example Some Transporter 1");
protein1.setDisplayName("TEST1");
```

Let's add a biochemical reaction and connect it to our protein:

```
BiochemicalReaction rxn1 = model.addNew(
        BiochemicalReaction.class,
         "http://biopax.org/tutorial/rxn1");
rxn1.addLeft(protein1);
```

You can keep adding different properties and classes to get a complete pathway. [15].

Reading and Writing BioPAX

Most of the time you, will not be creating your model from the scratch but rather reading it from a file. For reading files, Paxtools give you the BioPAXIOHandler interface and two alternative implementations, JenaIOHandler and SimpleIOHandler,

Jena IO

Jena is a full-fledged Java framework for semantic web applications. Paxtools can read OWL files using Jena's OWL API. This is a complete solution that can read most OWL encodings. On the negative side, it has a big footprint and can demand a significant amount of memory for some large BioPAX files. Jena is a good choice if you are already using it in your application or increasing the size of your application is not a big problem, do not know the owl encoding of the BioPAX files you need to read beforehand and you will only deal with small BioPAX files. You can initialize a Jena based reader by:

```
BioPAXIOHandler handler = new JenaIOHandler();
```

or for level 3:

[14] Unlike Object Oriented languages like Java, in OWL properties are first class citizens that can have inheritance and attributes like transitive and functional. Paxtools implements these additional semantics internally. For example, since property "standardName" is a subproperty of "name", updating the standardName of a protein will also update its list of names.

[15] In the BioPAX specification properties are unidirectional for brevity. For example, the "participant" property links interactions to physical entities. You should always add/removes properties in that "forward' direction. Paxtools also provides additional "inverse" links for key properties that allow efficient bidirectional navigation. These inverse links are also automatically updated when you modify the forward direction

```
BioPAXIOHandler handler = new JenaIOHandler(BioPAXLevel.L3);
```

Simple IO

Alternatively, you can use the SimpleIO class for a minimal footprint and high performance. The Simple reader can only read BioPAX encoded in RDF/XML and will not give you the flexibility of the Jena based reader, but it may be the optimal solution for use cases where you are processing large BioPAX files or do not want to increase the size of your application.

```
// auto-detects Level
BioPAXIOHandler handler = new SimpleIOHandler();
```

Once you have the handler you can import an OWL file into a model by simply:

```
Model model = handler.convertFromOWL(inputStreamFromFile);
```

Writing your model to a file is as easy as:

```
handler.convertToOWL(model, outputStream);
```

Exporting a sub-model to OWL

If you want to export only a set of BioPAX elements rather than the whole model itself, you can use the following call to the convert-ToOWL:

```
handler.convertToOWL(model, outputStream, id1, id2, id3);
```

ids is a vararg that you can use to declare the identifiers of elements you want to export from the model. This method will "auto-complete" meaning that dependents of the given objects will also be exported.

Accessing and Manipulating Pathway Elements

Basic Traversal

Once you have loaded a BioPAX file into a model, you are ready to start traversing it. The simplest option to start exploring your model is to call getObjects(). This will return a set of all BioPAX Elements in your model. You can then inspect each one. For example:

```java
// Load a sample test BioPAX File via Simple IO Handler
FileInputStream fin = new FileInputStream("test.owl");
BioPAXIOHandler handler = new SimpleIOHandler();
Model model = handler.convertFromOWL(fin);

// Iterate through all BioPAX Elements and print basic info
Set<BioPAXElement> elementSet = model.getObjects();
for (BioPAXElement currentElement : elementSet)
{
 String rdfId = currentElement.getRDFId();
 String className =
 currentElement.getClass().getName();
 System.out.println("Element: " + rdfId + ": " + className);
}
```

Alternatively, you can call getObjects() with a BioPAX class name and extract only the matching elements. For example, the following code extracts all proteins in your model, and outputs their name and display name:

```java
// Get Proteins Only
Set<Protein> proteinSet = model.getObjects(Protein.class);
for (Protein currentProtein : proteinSet)
{
        System.out.println(currentProtein.getName() +
         ": " + currentProtein.getDisplayName());
}
```

Path Accessors

Path accessors provide you with XPath-like access to the model. This allows you to explicitly state the paths you would like to access and can significantly reduce boiler plate code, especially when you need to traverse multiple cardinality properties. For example, consider the simple case of extracting the UnificationXrefs that belongs to protein references for each pathway in your model. To do that you need to access the processes that belong to a particular pathway, then get their participants that are Proteins, access their ProteinReferences and then access unification XRefs of all entityReferences and then test each XRef for whether it is a unification XRef. This is further complicated by the fact that pathways can be nested. Without Path accessors, this leads to some complicated code:

```
// Iterate through all pathways in the model
for (Pathway aPathway : model.getObjects(Pathway.class))
  extractProteinUrefsFromPathway(aPathway);
```

The following method will dig into the Processes in the pathway: [16]

```
extractProteinUrefsFromPathway(Pathway aPathway)
{
 for(Process aProcess: aPathway.getPathwayComponents())
 {
  if(aProcess instance of Pathway)
  { //Dig into the nested structure recursively
   extractProteinUrefsFromPathway((Pathway)aProcess);
  }  else
  { //It must be an Interaction
   extractAndPrintProteinUrefs((Interaction)aProcess));
  }}}
```

Finally we have reached the Proteins:

```
public void extractAndPrintProteinUrefs(Interaction anInteraction)
{
 for(Entity participant:anInteraction.getParticipants())
 {
  if(participant instanceof Protein)
  {
   ProteinReference entityReference=
     ((Protein)participant).getEntityReference();
```

[16] Note that "Process" is not an official BioPAX class. There are several anonymous union classes in the OWL specification of BioPAX. Paxtools creates special classes for these cases as anonymous superclasses are not supported in Java. In this case a Pathway might have "an Interaction or another Pathway" as a component. This union superclass maps to the Process interface in Paxtools. For more details please consult the API docs.

```
   if (entityReference != null)
   {
   Set<Xref> xrefSet = entityReference.getXref();
   for (Xref currentRef : xrefSet)
    {
     if (currentRef instanceof UnificationXref)
     {
      System.out.println(
        "Unification XRef: " + currentRef.getDb() + ": "
        + currentRef.getId());
}}}}
```

That's a lot of coding for a relatively common and straightforward task. The same code, rewritten with a Path Accessor:

```
// Set up the Path Accessor
PathAccessor pathAccessor = new PathAccessor(
 "Pathway/participants*:Protein/entityReference/xref:UnificationXref");
// Iterate through all proteins in the model
for (Pathway currentPathway : model.getObjects(Pathway.class))
{
 System.out.println("Pathway:"+currentPathway.getName());
 Set<Xref> unificationXrefs = pathAccessor.getValueFromBean(currentPathway);
 for (Xref currentRef : unificationXrefs)
 {
  System.out.println(
  "Unification XRef: " + currentRef.getDb() + ": " + currentRef.getId());
 }}
```

The format of the path query is in the form:

[Initial Class]/[property1]:[classRestriction(optional)]/[property2]...

A "*" sign after the property instructs path accessor to transitively traverse that property. For example, the following path accessor will traverse through all physical entity components within a complex:

```
PathAccessor accessor = new PathAccessor(
 "Complex/component*/entityReference/xref:UnificationXref");
```

The optional classRestriction will allow you to restrict the returned values of a property to a certain subclass of the range of the property. In the example above, this is used to get only the Unification Xrefs.

Using property editors

There are many cases where you need to handle BioPAX classes
and properties, but don't want to hard-code their names into your
code. For example you might be writing an "Inspector" for your
visualization tool –a tabled display of the selected object's properties–
or you might be writing a tool that allows users to export BioPAX
into a customizable spreadsheet. If you find yourself hard-coding
lists of properties into your code, you can probably simplify it using
property editors. Let's implement a simple "Inspector" that returns
the list of properties and values of a given BioPAX object.

```java
public static String[][] listProperties(BioPAXElement bpe)
{
  // In order to use properties we first need an EditorMap
  EditorMap editorMap = SimpleEditorMap.L3;
  // And then get all the editors for our biopax element
  Set<PropertyEditor> editors = editorMap.getEditorsOf(bpe);
  // Let's prepare a table to return values
  String value[][] = new String[editors.size()][2];
  int row = 0;
  // For each property
  for (PropertyEditor editor : editors)
  {
    // First column is the name of the property, e.g. "Name"
    value[row][0] = editor.getProperty();
    // Second column is the value e.g. "p53"
    value[row][1] = editor.getValueFromBean(bpe).toString();
    // increase the row index
    row++;
  }
  return value;
}
```

That was easy! Editors also provide you with a generic way to
set a value to the property, access its cardinality and other useful
methods. They are also used extensively for traversing and cloning
which we will cover in the next part of the tutorial.

Advanced Controls:Traversing, cloning and merging

Traversing

Traversing is an efficient way to perform operations that affects multiple connected entities using the visitor pattern. Given a model and a starting element, Traverser will follow the dependency relations and it will be calling the Visitor.visit() method for each triplet of a property editor, its domain (a BioPAX Element) and its range (a BioPAX Element, String, Float, and etc.) it traverses. A use case where a traverser is extremely useful is getting dependents of an object. For example, in order to extract a single reaction from a BioPAX model, one may be tempted to write:

```
//read initial model
Model model1 = handler.convertFromOWL(inputStream);
//create an empty model with the same level
Model model2 =
 model1.getLevel().getDefaultFactory().createModel();
//extract reaction
model2.add(model1.getByID("The_reaction_id"));
//write it out
handler.convertToOWL(model2, outputStream);
```

This is not a good solution though. The problem with this approach is that the add() method inserts only the reaction element (but not the participants and etc.) to our new model. Hence, the exporter will throw an error when trying to write this model out to a file. We need a way where we can recursively get an element and its dependencies, and then add all these elements into the new model[17] . Here is a simple implementation using Traverser[18]:

[17] The repair() method in the Model class implements the functionality given in the example. It will find and add child elements that are not currently contained in the model.
[18] You can also extend AbstractTraverser instead and override its public abstract visit method. See the Fetcher class for an example implementation.

```java
/**
 * A controller that excises/extracts an element and all the elements it is
 * dependent on from a model and adds them into a new model.
 */
class Excisor implements Visitor
{
 private Traverser traverser;
 private EditorMap editorMap;
 private Model targetModel;

 public Excisor(EditorMap editorMap)
 {
  this.editorMap = editorMap;
  this.traverser = new Traverser(editorMap, this);
 }


 //The visitor will add all elements that are reached into the new model,
 // and recursively traverse it
 public void visit(BioPAXElement domain, Object range, Model model,
                   PropertyEditor editor)
 {
  // We are only interested in the BioPAXElements since
  // primitive fields are always copied by value
  if (range != null && range instanceof BioPAXElement)
  {
   BioPAXElement bpe = (BioPAXElement) range;

   if (!targetModel.contains(bpe))
   {
    targetModel.add(bpe);
    traverser.traverse(bpe, model);
   }}}
```

The following method uses the traverser to "excise" the objects from the source model and adds them to the target model.

```java
public Model excise(Model sourceModel, String... ids)
{
 // Create a new model that will contain the element(s) of interest
 this.targetModel = editorMap.getLevel().getDefaultFactory().createModel();

 for (String id : ids)
 {
  // Get the BioPAX element
  BioPAXElement bpe = sourceModel.getByID(id);
  // Add it to the model
  targetModel.add(bpe);
  // Add the elements that bpe is dependent on
  traverser.traverse(bpe, sourceModel);
 }

 return targetModel;
 }
}
```

Optionally, Traverser uses property filters. It will not visit a property if the corresponding property editor fails to pass all the specified filters. One can define the filters by implementing org.biopax.paxtools.util.Filter interface, and add them to the constructor.

```java
public Excisor(EditorMap editorMap, boolean filtering)
{
 this.editorMap = editorMap;
 if (filtering)
 //We will filter nextStep property, as Reactome pathways leads
 //outside the current pathway. Step processes are listed in the
 //pathwayComponent property as well so this does not affect the fetcher.
 {
  final Filter<PropertyEditor> nextStepFilter = new Filter<PropertyEditor>()
  {
   public boolean filter(PropertyEditor editor)
   {
    return !editor.getProperty().equals("nextStep");
   }
  };
  this.traverser = new Traverser(editorMap, this, nextStepFilter);
 }
 else this.traverser =  new Traverser(editorMap, this);
}
```

Merging

One of the major goals of BioPAX is to integrate pathway information that were curated in a distributed fashion, similar to putting pieces of a puzzle together. Integration can be very complicated due to incomplete information, multiple levels of details and finally curation differences and errors [19]

EQUALITY is based on Java object identity. Paxtools do not override equals() and hashCode methods to avoid potential confusions with Java frameworks such as Hibernate.

RDF EQUALITY is based on the unique RDF-IDs of objects and its semantics are relatively straightforward. If two different objects have the same RDF-ID then: they can not be added to the same model at the same time.

EQUIVALENCE is based on the BioPAX properties of the objects and have much more complicated semantics.

In general two BioPAX elements are equivalent if they are equal OR if they are defined to be semantically equivalent which considers its properties.

In general, UtilityClasses that only have primitive properties are equivalent if the values of these properties are all known AND equal. Unknown/undefined values are considered unequal. For example, two sequenceSites with undefined positions are not considered equivalent.

EntityReferences equivalence is typically defined externally, e.g. by UniProt or Entrez Gene xrefs. Paxtools do not define additional equivalence semantics other than being equal for EntityReferences.

PhysicalEntities are defined to be equivalent iff their EntityReference, CellularLocation, set of Features and set of NotFeatures are all equivalent.

Complexes are defined to be equivalent iff their participant sets are equivalent.

Interactions are defined to be equivalent iff their participant sets are equivalent. For subclasses of interactions with different types of participants e.g. catalysts each subset of participant should be equivalent to the corresponding subset.

[19] We differentiate between 4 different types of matching between objects: Equality, RDF Equality, Equivalence and External Equivalence. Paxtools handles the first three, the fourth is handled by integration data warehouses such as Pathway Commons. There are other object matching/alignment relationships including Similarity, Subsumption and Containment. These are currently open problems.

Pathway definition has no strong semantic definition and is often demarcated arbitrarily. Paxtools do not define additional semantic equivalence for Pathways.

Features are equivalent iff their EntityReferences, FeatureTypes and Locations are equivalent.

ControlledVocabularies are equivalent iff their CV source and terms are equivalent.

Two stoichiometries are equivalent iff their PhysicalEntity and stoichiometric number is equivalent.

If A is equivalent to B then: Both A.isEquivalent(B) and B.isEquivalent(A) method returns true; they can be added to the same model and their equivalenceCode() are the same.

Merging based on Equivalence

If you want to merge multiple BioPAX models into a single model, the Merger class under the paxtools-core module can help you to do so:

```
Merger merger = new Merger(editorMap);
  merger.merge(targetModel, srcModel1, srcModel2);
```

The merger identifies equivalent elements and merges them[20]. The method modifies the target model (but not the source models) so that the target becomes the merged model. If you do not want to modify the target model, then you might want to use a new model as the target:

```
Model targetModel =
  editorMap.getLevel().getDefaultFactory().createModel();
Merger merger = new Merger(editorMap);
//vararg method âĂŞ accepts multiple models
  merger.merge(targetModel, srcModel1, srcModel2);
```

[20] If one element in the source model is matched to a target element, then for BioPAX properties with multiple cardinality, the values of the source element are added to the target. For single cardinality properties, the source value overwrites the target value.

Simple Merging

If you don't want to infer equivalence for interactions and participants and want to simply merge based on RDF-IDs, you can use SimpleMerger. SimpleMerger will migrate elements from the source model to the target, only if target does not contain another element with the same id. While migrating objects, SimpleMerger updates (re-wires) object property values of source elements to make sure they now refer to objects in the target model. SimpleMerger is a good

choice if the models you would like to merge are coming from same sources or are already normalized by some service, such as Pathway Commons. An example use case might be "stitching" together multiple query results from Pathway Commons. In a special case, to merge a single source model to the target, you can use merge(source) method of the Model class, which internally translates to the Simple-Merger merge method call with two Model parameters.

Exporting BioPAX to and from other formats

Paxtools also provides several converters to other standards and common formats.

Exporting to Simple Interaction Format (SIF)

Simple Interaction Format (SIF) was originally created for use with Cytoscape, the open source bioinformatics software platform for visualizing molecular interaction networks. SIF is simple to parse, and easy to load into Cytoscape and other third-party applications.

Relations in a SIF file are formatted as:

```
source relationship type target
```

where source and target are a valid primary id and relationship type is one of the interaction inference rules specified in 1.

As its name suggests SIF, is a substantially simpler format compared to BioPAX. A BioPAX formatted network is capable of storing rich biological semantics, including multi-participant relationships, states and locations of entities, complexes and complex control relationships. A SIF network, on the other hand, is only capable of storing pairwise interactions between entity references.

Inference Rules

Paxtools reduces BioPAX to pairwise SIF relationships using a set of rules. Although this translation is lossy, SIF network remains useful for certain types of bioinformatics applications that require pairwise interaction input. The table below outlines these rules. To export a BioPAX model to SIF, use the SimpleInteractionConverter class:

```
SimpleInteractionConverter converter =
new SimpleInteractionConverter(new ControlRule());
converter.writeInteractionsInSIF(level2, out);
```

Note the ControlRule object passed to the constructor. The constructor accepts a comma separated list of inference rules that are

applied when binary interactions are requested. If not specified, all default binary interaction rules will be applied. You can also write your own rules by implementing the InteractionRule interface.

SIFNX Format

SIFNX is an extensible SIF-based format that allows users to customize the information extracted from the BioPAX model while keeping the simple binary network structure of SIF. You can use the same SimpleInteractionConverter class to export to SIFNX. For example, if you want to get the name of entities, their class, their external references and publications associated with interactions you can use:

```
converter.writeInteractionsInSIFNX(model,
    out, out, Arrays.asList("Entity/name","Entity/xref"),
    Arrays.asList("Entity/xref:PublicationXref"), true);
```

Parameters in this method represents the flexibility of SIFNX exporter. Model is the source model, and the two output streams are for writing out nodes and edges respectively. In this case, we opted to write both of them to the same stream. Next, two lists of strings define the path accessors to extract information for nodes and edges of the SIF. Finally, the last boolean parameter instructs converter to write the class of entity nodes to the export.

PSI-MI (Proteomics Standards Initiative - Molecular Interaction)

The PSI-MI format is a data exchange format for molecular interactions. With help from the PSI-MI Developer Tools Library, the PSI-MI converter has the ability to create a BioPAX-L2 or BioPAX-L3 model from a PSI-MI Level 2.5 file. For each EntrySet in the PSI-MI file, a BioPAX model is created. For each Entry in the EntrySet, the converter creates a physicalInteraction (BioPAX-L2) or a Molecular-Interaction (BioPAX-L3) for each interaction in the Entry. Currently, genetic interactions are ignored. For each Participant in the Interaction, the converter will create a sequenceParticipant (BioPAX-L2) or a SimplePhysicalEntity (BioPAX-L3). The ParticipantâĂŹ Interactor determines the physical entity underlying the sequenceParticipant or SimplePhysicalEntity. The converter will create a smallMolecule, dna, rna or protein class (BioPAX-L2) or SmallMolecule, Dna, Rna, or Protein class (BioPAX-L3) based on the type of Interactor. This converter is also available using Paxtools from the command line.

Gene Set Enrichment Analysis(GSEA)

Paxtools can create a Gene Matrix Transposed (*.gmt) file from a
BioPAX-L3 model. For each Pathway object in the model, the GSEA
converter traverses all Protein objects in the Pathway and adds
the RDF ID belonging to the first Xref returned by a call to Pro-
tein.getXref() to the gene set. An optional external database name
can be supplied to the converter, which would limit gene set mem-
bership to only those Protein objects that have an Xref to the external
database. The GSEA converter also has the ability to limit gene set
membership to only those Protein objects that belong to the same
species as the Pathway. This converter is also available using Paxtools
from the command line.

Systems Biology Graph Notation - Process Description(SBGN-PD)

Paxtools use libSBGN library to create SBGN -PD exports in SBGN-
ML for BioPAX-L3 models. BioPAX-L3 to SBGN-PD conversion is
an approximation because BioPAX-L3 is not always fully convertible
to SBGN-PD. The main point of conflict is PhysicalEntity objects in
BioPAX-L3 can define overlapping sets of pool of molecules, while
EntityPoolNode (EPN) of SBGN-PD have to be disjoint. Paxtools
creates an EPN for each PhysicalEntity as if they represent disjoint
pools. The following table is a summary of the conversion process.

BioPAX L3 component	Corresponding SBGN-PD glyph
SimplePhysicalEntity	Glyph types "macromolecule", "simple chemical", "nucleic acid feature", and "unspecified entity" are used according to the type of the PhysicalEntity.
Conversion	A "process" glyph is created for each direction of the Conversion, i.e. two glyphs are created if the Conversion is reversible. "consumption" and "production" arcs are used to associate left and right participants, according to the direction.
Control	Controllers are linked with "catalysis", "stimulation", and "inhibition" arcs to the related process. When there are multiple Controller of the Control, they are connected with a logical "and" glyph. When the control has another control on itself, controls are connected with "and" glyph, and a "not" glyph is used after the second controller if the second control is negative. So the Control tree in BioPAX-L3 is represented with an "and" and "not" tree in SBGN-PD.
TemplateReaction	Similar to Conversion, a "process" glyph is created, but without the consumed participants. Instead, a "source and sink" glyph is created and linked with a "consumption" arc.

The SBGN creation facility is implemented under the package
org.biopax.paxtools.io.sbgn, and used as following:

```
//Create SBGN-PD and write to the given file
L3ToSBGNPDConverter.writeSBGN(level3Model, "filename.sbgn");
```

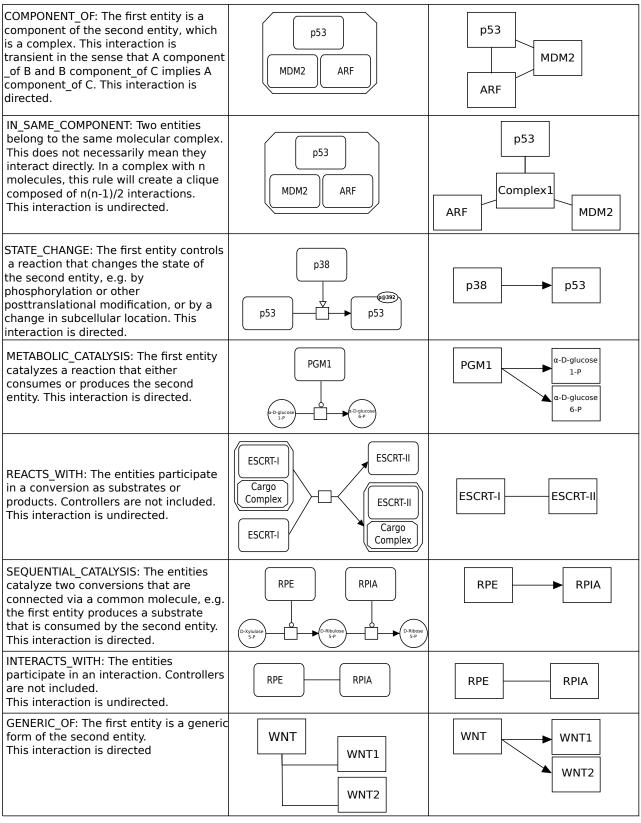|		
COMPONENT_OF: The first entity is a component of the second entity, which is a complex. This interaction is transient in the sense that A component_of B and B component_of C implies A component_of C. This interaction is directed.	p53 / MDM2 / ARF	p53 — MDM2 / ARF
IN_SAME_COMPONENT: Two entities belong to the same molecular complex. This does not necessarily mean they interact directly. In a complex with n molecules, this rule will create a clique composed of n(n-1)/2 interactions. This interaction is undirected.	p53 / MDM2 / ARF	p53 / Complex1 / ARF MDM2
STATE_CHANGE: The first entity controls a reaction that changes the state of the second entity, e.g. by phosphorylation or other posttranslational modification, or by a change in subcellular location. This interaction is directed.	p38 / p53 → p53 p@392	p38 → p53
METABOLIC_CATALYSIS: The first entity catalyzes a reaction that either consumes or produces the second entity. This interaction is directed.	PGM1 / α-D-glucose 1-P → α-D-glucose 6-P	PGM1 → α-D-glucose 1-P / α-D-glucose 6-P
REACTS_WITH: The entities participate in a conversion as substrates or products. Controllers are not included. This interaction is undirected.	ESCRT-I / Cargo Complex / ESCRT-I → ESCRT-II / ESCRT-II / Cargo Complex	ESCRT-I — ESCRT-II
SEQUENTIAL_CATALYSIS: The entities catalyze two conversions that are connected via a common molecule, e.g. the first entity produces a substrate that is consumed by the second entity. This interaction is directed.	RPE RPIA / D-Xylulose 5-P → D-Ribulose 5-P → D-Ribose 5-P	RPE → RPIA
INTERACTS_WITH: The entities participate in an interaction. Controllers are not included. This interaction is undirected.	RPE — RPIA	RPE — RPIA
GENERIC_OF: The first entity is a generic form of the second entity. This interaction is directed	WNT / WNT1 / WNT2	WNT → WNT1 / → WNT2

Figure 1: SIF rules: The left column is the recognized BioPAX pattern drawn in SBGN-PD, the right column is the resulting inferred SIF relationship(s).

Running Graph Queries

Paxtools provides several graph queries such as neighborhood, shortest + k paths, paths-in-between, common upstream, and common downstream, in the package org.biopax.paxtools.query. The algorithms used in the these queries are based on Dogrusoz et al.[21]. Each query takes a source set (and sometimes also a target set) of BioPAX objects, the model, and several query parameters as input in their constructor. The run() method returns the result set of BioPAX objects. To run graph queries, you can use the static methods of the class QueryExecuter:

[21] Dogrusoz, U., Cetintas, A., Demir, E., & Babur, O. 2009, Algorithms for effective querying of compound graph-based pathway databases, BMC Bioinformatics, 10, 376, 376

```java
Set<BioPAXElement> source = new HashSet<BioPAXElement>();
// Add the related source PhysicalEntity
//(or children) objects to the  source set
Collections.addAll(source, entity1, entity2, entity3);
int limit = 2;
// Direction can be upstream, downstream, or bothstream.
Set<BioPAXElement> result = QueryExecuter.runNeighborhood(
  sourceSet, model,limit, Direction.BOTHSTREAM);
```

The query result will contain only the objects on the resulting paths. It will be a subgraph of the model, but it may not be a semantically consistent subgraph. For instance, consider the model : A, B, X are instances of PhysicalEntity, C is a Conversion, A is at the left of C, B is at the right of C, direction of C is left-to-right, and X is the positive Controller of C. A neighborhood query with source X, limit 1, and towards downstream will retrieve the path X->C->B; however, this subgraph is not semantically complete. It contains a Conversion with a missing "left1" (in this case, an input). When the query result is detached from the model, for example if the query is being performed in a server, and the resulting graph is being sent to a visualization client, omitting A in the view will lead semantically incomplete view. In that case the user needs to use the Completer class to complete query result:

```
Completer c = new Completer(
 SimpleEditorMap.get(BioPAXLevel.L3));
result = c.complete(result, model);
```

Now, the result set contains A because of C. This set can be safely cloned to get a valid BioPAX model.

QueryExecuter provides several methods for common graph queries:

- Neighborhood

  Searches directed paths from and/or to the given source set of entities, in the specified search limit.

  Parameters

  - sourceSet: Set of source physical entities

  - model: Current model

  - limit: Search limit

  - direction: Enum for search direction. Can be upstream, downstream, or bothstream.

- GOI

  GOI stands for graph-of-interest. This query retrieves the directed paths between the objects in the given source set. User gives a set of PhysicalEntity object and this query retrieves the graph connecting these entities.

  Parameters

  - sourceSet: Set of source physical entities

  - model: Current model

  - limit: Path length limit for the search

- POI

  POI stands for paths-of-interest. Similar to GOI, but paths are searched from a source set, towards a target set. There are two types of search limit in this case. The first one is the normal limit, where you limit the length of the search paths by a value. The second is a shortest + k type limit, where the user specifies the value of k, and the actual limit becomes the length of the shortest path between sets plus k. The latter is useful when the user wants minimal connections between sets but does not know the length of the shortest path.

  Parameters

  – sourceSet: Source set of physical entities to start search

  – targetSet: Target set of objects to reach during the search

  – model: Current model

  – limitType: Whether to use a constant path length limit or a shortest+k limit

  – limit: Value of the limit. If limit type is shortest+k, then this represents k

- PathsBetween

  This query is similar to GOI âĂŞ it finds paths between a given source set of objects. The source set may contain Xref, EntityReference, and/or PhysicalEntity objects. They are automatically converted to sets of related PhysicalEntitiy objects to use as source in the query. The major difference here is that it avoids finding paths between PhysicalEntity objects that are associated with the same EntityReference, unless the user provides them explicitly in the source set. For instance, when the source set contains EntityReference of P53 protein, the query automatically finds all different modified versions of P53 in the model to use as seed in the query, but does not return paths between them. However, if a user provides different modification states of P53 in the source set, the query finds paths between them. The major use case of this query is the user has a set of entity references and wants to find paths connecting them.

  Parameters

  – sourceSet: Source set of xrefs, entity references, or physical entities

  – model: Current model

  – limit: Search limit

- CommonStream

  This query searches for the common upstream (common regulators) or common downstream (common targets) objects of the given source set. The source set should contain at least 2 elements.

  Parameters

  – sourceSet: Source set of physical entities

  – model: Current model

  – direction: Upstream or downstream

  – limit: Search limit

- CommonStreamWithPOI

  Common-stream query returns only the objects in the common
  upstream or downstream, but it does not include the paths leading
  to them. If a connected graph that contains both common objects
  and source objects is desired, then a POI query should be run
  between common objects and source objects. This method does it
  automatically.

  Parameters

  – sourceSet: Source set of physical entities

  – model: Current model

  – direction: Upstream or downstream

  – limit: Search limit

Resources and Non-Java Access

Accessing Pathway Commons

Pathway Commons (PC) is a convenient point of access to biological pathway information collected from BioPAX supporting public pathway databases. You can programmatically access PC using the pathwaycommons-client module. Currently there are two separate Pathway Commons clients implemented as part of the Paxtools under the org.biopax.paxtools.io.pathwayCommons package: PathwayCommonsIOHandler which interacts with the older PC WEB API and supports BioPAX Level 2; and PathwayCommons2Client which makes use of the newer PC WEB API and supports BioPAX Level 3.

Using Pathway Commons Client you can search Pathway Commons, run graph queries in a bigger and more complete BioPAX graph or normalize your custom pathway using Pathway Commons services.

For example, here is a code snipplet which searches Pathway Commons 2 for the term "BRC*" restricting the results only to the BioPAX type Control and to the organism Homo sapiens; and then obtains these control reactions as a Paxtools model:

```java
PathwayCommons2Client pc2 = new PathwayCommons2Client();
pc2.setOrganisms(Collections.singleton("homo sapiens"));
pc2.setType("Control");
//Let's search for anything that starts with BRC
SearchResponseType result = pc2.find("BRC*");
HashSet<String> uris = new HashSet<String>();
// For each search hit we got, get the uri for the
// resource at PC and add it to the set
for (SearchHitType hit : result.getSearchHit())
{
 uris.add(hit.getUri());
}

//Create a model from this set of resources
Model model = pc2.get(uris);
```

Using Paxtools from the command line

Paxtools can be run as command line application. Command line options are summarized below:

```
merge file1 file2 output
toSif file1 output
toSifnx file1 outEdges outNodes prop1,prop2,...
validate path out [xml|html|biopax] [auto-fix] [normalize] [only-errors] [maxerrors=n]
integrate file1 file2 output
toLevel3 file1 output
fromPsimi level file1 output
toGSEA file1 output database crossSpeciesCheck
fetch file1 id1,id2,.. output
getNeighbors file1 id1,id2,.. output
```

If you have obtained Paxtools as a ("fat") JAR archive (e.g., a paxtools-YYYYMMDD.jar), then you can use the command line interface as the following:

```
java -jar paxtools.jar --merge file1.owl file2.owl output.owl
```

For very large files you might want to increase jvm's memory:

```
java -Xmx2048M -jar paxtools.jar --merge file1.owl file2.owl output.owl
```

You can also directly call the PaxtoolsMain class by:

```
java -cp paxtools.jar org.biopax.paxtools.PaxtoolsMain --validate file.owl
```

Using Paxtools from Python

You can use JPype to control Paxtools from Python. You can download and install JPype from http://jpype.sourceforge.net/. Following example demonstrates how you can create a simple BioPAX model and write it out to a file using Python.

```python
from jpype import *
#call this to initialize use of Java
startJVM(getDefaultJVMPath(), "-ea", "-Xmx1g", "-Djava.class.path=paxtools.jar")

#get the paxtools root package as a shortcut
paxPkg = JPackage("org.biopax.paxtools")

#create a new BioPAX L3 factory
l3Factory = paxPkg.model.BioPAXLevel.L3.getDefaultFactory()

#create a new empty BioPAX model
model = l3Factory.createModel()

#define and set the xml base (URI prefix for elements we create)
xmlBase = "http://biopax.org/examples/pythonPaxtools#"
model.setXmlBase(xmlBase)

#get BioPAX classes (model interfaces); in this example:
#Protein, CellularLocationVocabulary, UnificationXref:
proteinClass = java.lang.Class.forName(
 "org.biopax.paxtools.model.level3.Protein",
 True, java.lang.ClassLoader.getSystemClassLoader())
cellularLocationCvClass = java.lang.Class.forName(
 "org.biopax.paxtools.model.level3.CellularLocationVocabulary",
 True, java.lang.ClassLoader.getSystemClassLoader())
unificationXrefClass = java.lang.Class.forName(
 "org.biopax.paxtools.model.level3.UnificationXref",
 True, java.lang.ClassLoader.getSystemClassLoader())

#create/add a couple of elements to the model
#Let's create a simple protein state using the factory method and unique identifier (URI)

protein = l3Factory.create(proteinClass, xmlBase + "protein1")
protein.addComment("python: created " + protein.getRDFId())

#And add it to the model
model.add(protein)

#step 3: set properties
protein.addAvailability("availability text")
cellLoc = l3Factory.create(
 cellularLocationCvClass, "http://identifiers.org/obo.go/GO:0005737")
model.add(cellLoc)
cellLoc.addComment("python: created " + cellLoc.getRDFId())
cellLoc.addTerm("cytoplasm")
protein.setCellularLocation(cellLoc)

#alternatively, one can create, set the id (URI), and add the element in one step
protein2 = model.addNew(proteinClass, xmlBase + "protein2")
protein2.addComment("created " + protein2.getRDFId())
```

```python
# let's add a unification xref to the CV
ux = model.addNew(unificationXrefClass, xmlBase + "XREF_GO_0005737")
ux.setDb("GO")
ux.setId("GO:0005737")
cellLoc.addXref(ux)

#export the model to a BioPAX OWL file
javaIO = JPackage("java.io")
io = paxPkg.io.SimpleIOHandler(paxPkg.model.BioPAXLevel.L3)
fileOS = javaIO.FileOutputStream("test.owl")
io.convertToOWL(model, fileOS)
fileOS.close()

#import a BioPAX model from the file
fileIS = javaIO.FileInputStream("test.owl")
model2 = io.convertFromOWL(fileIS)
fileIS.close()

#output to console
io.convertToOWL(model, java.lang.System.out)

#end use of jpype - docs say you can only do this once,
# so all java must be run before calling this
shutdownJVM()
```

Putting it all together

Here is an example application that takes two or more protein names and queries/saves the neighborhood of each protein from Pathway Commons. It also queries the paths between these proteins, converts the resulting model to SIF and lists the publications that support the each interaction in the final model.

```java
public class ProteinAnalyzer
{
 protected final static String outputPath = "files/";

 public static void main(String[] arg) throws IOException
 {
  /* Expects two or more protein names */
  if (arg.length < 2)
  {
   System.err.println(
     "Usage: ProteinAnalyzer protein1 protein2 [protein3 [protein4 [...]]]");
   System.exit(-1);
  }

  // This IO Handler will be used to export pathways in BioPAX format
  SimpleIOHandler ioHandler = new SimpleIOHandler();

  // Create the Pathway Commons client and configure it
  PathwayCommons2Client pc2 = new PathwayCommons2Client();
  // Search only for Proteins
  pc2.setType("Protein");
  // Restrict results to H. sapiens
  pc2.getOrganisms().add("homo sapiens");
  // Expand the graph query limit to get more results
  pc2.setGraphQueryLimit(2);

  // General set to collect ids of all matching protein
  Set<String> allProteinIds = new HashSet<String>();
```

```java
for (String protein : arg)
{
 // Search PC2 for the given protein name
 SearchResponseType searchResponse = pc2.find(protein);
 if (searchResponse.getTotalNumHits() < 1)
 {
  System.err.println("No results for protein:" + protein);
  System.exit(-1);
 }

 // Collect all ids associated to this search
 Set<String> ids = new HashSet<String>();
 for (SearchHitType searchHit : searchResponse.getSearchHit())
 {
  ids.add(searchHit.getUri());
 }

 // Also add all these ids to overall set
 allProteinIds.addAll(ids);

 // Query the neighborhood of this protein
 Model proteinNeighborhood = pc2.getNeighborhood(ids);
 // And export the resultant network in BioPAX format
 FileOutputStream fileStream = new FileOutputStream(
   outputPath + protein + ".owl");
 ioHandler.convertToOWL(proteinNeighborhood, fileStream);
 fileStream.close();
}

// Query the paths between all given proteins
Model pathsBtwModel = pc2.getPathsBetween(allProteinIds);
// Save the model in BioPAX format
FileOutputStream pathsBtwFile = new FileOutputStream(
  outputPath + "pathBetween.owl");
ioHandler.convertToOWL(pathsBtwModel, pathsBtwFile);
pathsBtwFile.close();

// Also save the model in Simple Interaction Format with all default rules
SimpleInteractionConverter sifConverter = new SimpleInteractionConverter();
FileOutputStream pathsBtwSIFFile = new FileOutputStream(
  outputPath + "pathBetween.sif");
sifConverter.writeInteractionsInSIF(pathsBtwModel, pathsBtwSIFFile);
pathsBtwSIFFile.close();

// Now, get all interactions in the models to extract publication information
for (Interaction interaction : pathsBtwModel.getObjects(Interaction.class))
{
 // Print the name of the interaction
 System.out.println("* " + interaction.getDisplayName());
```

```java
// Get all external references
for (Xref xref : interaction.getXref())
{
 String db = xref.getDb(),
    id = xref.getId(),
    url = "";

 // Convert PubMed ids to URLs for easy access
 if (db.equalsIgnoreCase("PubMed"))
  url = " (http://www.ncbi.nlm.nih.gov/pubmed/" + id + ")";

 // Print the external reference information
 System.out.println("\t" + xref.getDb() + ":" + xref.getId() + url);
}}}}
```

Frequently Asked Questions

Q. When I convert BioPAX to simple interaction format file many relations contains unresolvable ids starting with http://biopax.org/generated/group/.... How do I map this generated link to a known ID such as protein name or entrezID?

A. Unresolvable URIs like this are used to represent groups that are either complexes, generics or generic complexes. Generics are typically families of proteins (think WNT) that pathway databases captured as a group. A generic complex would be WNT-FRZ complex that maps to combinatorially many instances. Since in a SIF conversion nodes are entityReferences and complexes and some generics in BioPAX do not have entity references, without groups SIF conversion would be lossy or inaccurate.

You can try toSifnx command to get external references as:

```
java -Xmx1g -jar paxtools.jar toSifnx your_biopax_level3.owl
 edges.txt nodes.txt  "EntityReference/xref:UnificationXref,
EntityReference/xref:RelationshipXref,Entity/xref:UnificationXref,
Entity/xref:RelationshipXref"
"Interaction/dataSource/name,Interaction/xref:PublicationXref"
```

Q. I have encountered a conflict between jars and I get the following exception when I am reading BioPAX

```
Exception in thread "main" org.biopax.paxtools.util.BioPaxIOException:
Unexpected element at start: 6
    at org.biopax.paxtools.io.SimpleIOHandler.readNameSpaces
```

A. This is a known problem caused by the StaX factory auto-discovery in JDK. You probably have two different stax implementations in your classpath. This problem also might happen under certain resource frameworks such as OSGi. For some discussion on this issue please see: http://code.cytoscape.org/redmine/issues/829. Paxtools, in the future might allow different discovery solutions for different needs. As a temporary solution you can use the Jena based IO-handler.

Q. While reading a new BioPAX Level 3 model I am getting an error

```
INFO Detected biopax namespace for level 2
INFO Using level: 2
ERROR org.biopax.paxtools.util.IllegalBioPAXArgumentException
        : No creation methods for name: SmallMolecule
```

A. This sometimes happens with BioPAX files that contains both L2 and L3 namespaces confusing the auto level detection of the reader. Initialize your reader with the level explicitly:

```
JenaIOHandler jenaIOHandler =
 new JenaIOHandler(
  new Level3FactoryImpl(), BioPAXLevel.L3);
```

Q. I am getting a lot of errors from the reader while reading BioPAX in the form of: ERROR org.biopax.paxtools.controller.PropertyEditor - Failed to set value:.

A. This can happen either due to non-standard usage or extensions of the BioPAX that is not in the specification. We constantly work with data providers to reduce such cases but you might still encounter them with an unvalidated BioPAX file. Examine the properties reported in the error log to determine what kind of information is being lost. In some cases these are not important for your usecase and you can ignore them. Otherwise try contacting the data provider to fix them. Paxtools will provide support for extensions in the future releases.

Q. Is it possible –probably with loss of information– to convert Level 3 files into Level 2?

A. L3 contains significant changes compared to L2 and such a conversion would be quite lossy and problematic. Unfortunately, Paxtools can only convert L2 to L3.

Q. I am executing a paths-between query but the result is a BioPAX graph. I want to access the result paths separately. Where are the paths?

A. All graph queries return a graph which is the union of the queried paths, but they do not create the resultgraph by iterating over each result path. In most cases, there would be combinatorially many paths that are infeasible to enumerate.

Q. Why does Paxtools SBGN converter generates two process nodes for a reversible Conversion in the model when SBGN-PD can represent a reversible reaction with a single process node.

A. This is due to a subtle semantic mismatch between SBGN-PD and BioPAX. In BioPAX a reversible BiochemicalReaction (subclass of Conversion) can have a directed Catalysis (left-to-right or right-to-left) based on the physiological context. If we use a single process node for the reversible reaction, we cannot show the direction of the

Catalysis. If we use two process nodes, one for each direction, then we can associate the Catalysis only with the related process node. However, creation of two process nodes is only the default behavior and can be changed to single node using the "setUseTwoGlyphsFor-ReversibleConversion" method if this is the user preference.

Q. I want to write an application that can handle both Level 2 and Level 3. Do I need to handle them separately?

A. This is a side effect of lack of backward compatibility between BioPAX Level 2 and 3 specification and it is not something we could directly address at the Paxtools level.. BioPAX Level 3 was a major refactoring of the specification and it was impossible to maintain a common class hierarchy between 2 and 3. We have indeed a single class-base for Level 1 and Level 2 which are mostly backward compatible.,

To partially amend that situation we suggest all tool developers to upgrade the Level 2 models to Level 3 using the LevelUpgrader class and program to the Level 3 interfaces.

The other option is to use the PropertyEditors and EditorMaps extensively. For example both BioPAX readers can read both level 2 and level 3. This requires a little bit more programming than the previous option but arguably offers a more robust solution.

Q. I am confused by the statement "This syntactic validation does not include more detailed checking of semantics that is performed by the BioPAX validator." Which validation does what? Why is there a difference? Is the BioPAX validator not using Paxtools?

A. The validator is built on Paxtools but is a separate project that includes a sophisticated rule defining and reporting system. Paxtools limits itself to so-called *invariants* of the BioPAX specification, namely, RDF well-formedness, domain and range restrictions, bidirectional links, and redundancies. When these constraints are violated Paxtools fails-fast by throwing an exception. Checking additional semantics and best practices such as using proper controlled vocabularies or ensuring that a transport reaction has at least one substrate that changes its subcellular location as result of the reaction is left to the validator. There are no technical reasons for not including validation rules within Paxtools - in fact historically some of the current validator rules were a part of Paxtools. We quickly found out however that this was severely restricting our user base as they were using Paxtools and BioPAX in ways that we have not imagined. As a result we moved all of these optional constraints to the validator and built a system where users can activate and deactivate which rules are to be checked and how they are going to be reported(e.g. warnings vs. errors).

Q. Why are level2 model interfaces are not in CamelCase?

A.Class names of model interfaces strictly mirrors BioPAX specification and level 2 was specified using the LISP syntax. We decided to copy the case directly from the specification to avoid any confusion. In Level 3, BioPAX specification was migrated to CamelCase to amend that based on our input. All other classes in Paxtools strictly follow CamelCase and Java naming conventions.

Q. What happens to the additional properties that are not a part of BioPAX specification? If I define new properties or classes, can Paxtools read and write them?

A. The short answer is no - they are lost when you read them with Paxtools. The long answer is this is an important feature that we wanted to add but it turns out to be surprisingly difficult. Adding a simple custom property can be done relatively easily by adding a HashMap<key,Set> to every object but things get complicated when trying to implement more advanced OWL properties such as sub-properties, symmetric and transitive properties. We felt that it would be inconsistent to offer a facility to read arbitrary extensions of the BioPAX specification in the OWL file but limit ourselves to only simple properties. We have an early alpha implementation that did not make the cut for the latest release. We will include this facility in the future releases.

On the other hand, we believe that extending the library in the traditional sense by extending the classes is in fact very easy. In fact Paxtools were designed right from the start to enable such an extension by implementing a factory pattern for object creation and PropertyEditors for reading/writing properties.

Q. While exporting to SBGN-ML, I wonder why two process glyphs are created when a Conversion is reversible as it can be presented with one process glyph only as illustrated in `http: // www. sbgn. org/ Symbols/ consumption_ production_ reversible` *?*

A. This choice is made because of the BioPAX structure. In BioPAX, a reversible BiochemicalReaction can have a Catalysis with a specific direction. If we draw one process glyph for the reversible BiochemicalReaction, then we cannot show the direction of the Catalysis. In the two process glyphs case, the Catalysis will point to the related process glyph only. However, we realize that some users may opt for a reversible process glyph even though they can not visualize the Catalysis direction, and we this is now an option in the exporter.

A few last words...

Acknowledgements

Get Involved

Paxtools is open source software and we welcome all contributions. Get involved!

- Use Paxtools and report issues and feature requests. See: `http://biopax.sourceforge.net/paxtools/issue-tracking.html`

- Respond to questions by other users at biopax-paxtools

- Try your hand at fixing bugs and implementing requests

- Improve this documentation.

- Spread the word. Let other people know about BioPAX and Paxtools.