The graph2tab library, an introduction

This package is a generic implementation of a method for producing spreadsheets out of pipeline graph. It is based on the node splitting approach: all the nodes in the input pipeline are reduced to "chain nodes", i.e.: nodes that have at most one input and at most one output. This is done by taking those nodes having splittings or pooling, creating copies of them and distributing the excessive inputs or outputs over the copies. See the class ChainsBuilder for details about such procedure.

The package is designed in a way that allows to adapt a particular object model (eg.: ISATAB or MAGETAB) and to reuse classes like TableBuilder to produce spreadsheets describing a particular experimental pipeline (ie.:CSV/TSV files).

Summary

In a nutshell, what you've to do to use graph2tab, in most cases is:

- define your node wrappers and the attributes the yield
- define your node factory
- define your table builder
- invoke your table builder with the right set of nodes coming from your to-be-converted model (such as the sources). Collect the results as a matrix, the SDRF-like table.

Let's see the details below.

The basics

graph2tab can be used to convert a graph about an experimental pipeline to a tabular view, similar to the SDRF structure in the MAGETAB format, or the Sample/Assay files in the ISATAB format.

In order to do that, you must start from a Java object model that is used to represent the experimental pipeline (e.g.: the BII model, or the model defined in the Limpopo MAGETAB parser). For example, you'll have instances of classes like BioSource or Hybridization and these will have properties like factorValues, characteristics, inputs, outputs.

graph2tab is designed to be generic: apart from few basic elements, the graph/table conversion does not depend neither on the particular graph that is being converted, nor on the particular set of headers that the output consists of. The library ensures this generality by working with a few basic interfaces.

Node is one of the most important of them, it just defines inputs, outputs and attributes. The former point at other instances of the *Node* interface itself. The attributes are defined in in a tabular-oriented way, that is: every attribute is a pair of strings, the header and the value. These will correspond to headers/values in the spawn table format. The way this is implemented consists of two lists of strings, defined in TabValueGroup, one for the headers and one for the values.

uk/ac/ebi/tablib/export/graph_algorithm/Node.java

```
package uk.ac.ebi.tablib.export.graph_algorithm;

public interface Node extends Comparable<Node>
{
	public SortedSet<Node> getInputs ();
	public boolean addInput ( Node input );
	public boolean removeInput ( Node input );

	public SortedSet<Node> getOutputs ();
	public boolean addOutput ( Node output );
	public boolean removeOutput ( Node output );
```

```
        public List<TabValueGroup> getTabValues ();

        public Node createIsolatedClone ();
}
```

uk/ac/ebi/tablib/export/graph_algorithm/TabValueGroup.java

```
package uk.ac.ebi.tablib.export.graph_algorithm;

public interface TabValueGroup
{
        public List<String> getHeaders ();
        public List<String> getValues ();
}
```

An example of how these interface can be used is provided in the package *uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests* (test/ folder). Here, you've two examples of how a to-be-converted model can be adapted to graph2tab. One is very simple and it's in the package *dummy_graphs_tests*. Here nodes from the input model are directly implementing out *Node* interface. While this is so simple, it's not very realistic, because you will typically have your model and you will not want to derive it from classes in the *grap2tab* package. So, in such cases, what you need is something similar to the example in the package *simple_biomodel_tests*. Let's describe that in detail. Suppose that the model you've to convert has a node like *BioMaterial* in this model, which has this shape:

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/BioMaterial.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class BioMaterial extends ExperimentNode
{
        public BioMaterial ( String name )
        {
                super ( name );
        }

        /**
         * Biomaterial characteristics are managed by means of {@link Annotation}.
         */
        public void addCharacteristic ( String type, String value, String termAcc, String termSrc )
        {
                super.addAnnotation ( type, value, termAcc, termSrc );
        }

        /**
         * Biomaterial characteristics are managed by means of {@link Annotation}.
         */
        protected List<Annotation> getCharacteristics () {
                return super.getAnnotations ();
        }

}
```

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/Annotation.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class Annotation
{
	...

	public Annotation ( String type, String value, String termAcc, String termSrc )
	{
	...
	}

	public Annotation ( String type, String value, OntoTerm ontoTerm ) {
	...
	}

	public Annotation ( String type, String value ) {
	...
	}

	public String getType () {
	...
	}

	public String getValue () {
	...
	}

	public OntoTerm getOntoTerm () {
	...
	}

}
```

As you can see, *Annotation* defines attributes for the input model. This is an example quite similar to the kind of attributes the nodes have in MAGE-OM, MAGETAB, ISATAB. The idea is that at the end of the day a node is converted into a list of such string pairs, so you'll have to convert the attributes from your model, whatever they are, to such string pairs, i.e.: you've to convert them to instances of *TabValueGroup*.

Model wrapping

The typical approach to implement such an adaptation from your model to the graph2tab interfaces is defining wrappers. Let's go straight to an example about the aforementioned *Biomaterial*. This derives from another class, written for this particular model, which is the abstract *ExpNodeWrapper*.

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/ExpNodeWrapper.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.node_wrappers;

...
public abstract class ExpNodeWrapper extends DefaultAbstractNode
{
	private ExperimentNode base;

	ExpNodeWrapper ( ExperimentNode base )
	{
		this.base = base;
	}
```

```
	/**
	 * This is used by the implementation of {@link Node#createIsolatedClone()}. Essentially, copies
	 * the wrapped node and makes empty input/output sets.
	 *
	 */
	protected ExpNodeWrapper ( ExpNodeWrapper original )
	{
		this.base = original.base;
		this.inputs = new TreeSet<Node> ();
		this.outputs = new TreeSet<Node> ();
	}

	/**
	 * In this case we're able to write a generic method that is customised by the descendants.
	 * The methods shows how to add an ontology term to a free text value. This is done by keeping
	 * all into the same {@link TabValueGroup}.
	 *
	 * @param nameHeader eg: "BioMaterial Name", "Protocol REF"
	 * @param annHeaderPrefix eg: "Characteristic", "Parameter Value", here headers will be built
	 * with the schema annHeaderPrefix [ type ], eg: Characteristic [ Organism ]
	 *
	 */
	protected List<TabValueGroup> getTabValues ( String nameHeader, String annHeaderPrefix )
	{
		List<TabValueGroup> result = new ArrayList<TabValueGroup> ();
		result.add ( new DefaultTabValueGroup ( nameHeader, base.getName () ) );

		for ( Annotation annotation: base.getAnnotations () )
		{
			DefaultTabValueGroup tbg = new DefaultTabValueGroup (
			  annHeaderPrefix + " [ " + annotation.getType () + " ]", annotation.getValue ()
			);
			OntoTerm ot = annotation.getOntoTerm ();
			if ( ot != null ) {
				tbg.add ( "Term Accession Number", ot.getAcc () );
				tbg.add ( "Term Source REF", ot.getSource () );
			}
			result.add ( tbg );
		}
		return result;
	}

	/**
	 * If it's not a clone produced by {@link Node#createIsolatedClone()},
	 * it uses {@link NodeFactory} to build wrappers
	 * for the input nodes of the base and to return them.
	 *
	 * This is the typical way this method is implemented by.
	 */
	@Override
	public SortedSet<Node> getInputs ()
	{
		if ( inputs != null ) return super.getInputs ();
		inputs = new TreeSet<Node> ();
		NodeFactory nodeFact = NodeFactory.getInstance ();
		for ( ExperimentNode in: base.getInputs () ) inputs.add ( nodeFact.getNode ( in ) );
		return super.getInputs ();
	}

	...
	@Override
	public SortedSet<Node> getOutputs ()
	{
		...
	}
}
```

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/BioMaterialWrapper.java

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.node_wrappers;
...
public class BioMaterialWrapper extends ExpNodeWrapper
{
        BioMaterialWrapper ( BioMaterial base ) {
                super ( base );
        }

        private BioMaterialWrapper ( ExpNodeWrapper original ) {
                super ( original );
        }

        public List<TabValueGroup> getTabValues () {
                return getTabValues ( "Biomaterial Name", "Characteristic" );
        }

        public Node createIsolatedClone () {
                return new BioMaterialWrapper ( this );
        }

}
```

Details

You're free to directly implement the Node interface for defining your wrappers. However, for common cases, extending *DefaultAbstractNode* is probably quite easier. The latter defines common functionality, such as the collection class members for maintaining input/output nodes.

Differently than *DefaultAbstractNode*, which is a general class and part of the main package, *ExpNodeWrapper* makes sense in this example only (it's in the test/ folder too) and define a few common elements that apply to the simple model used for this particular case.

Every node type in your original model has a corresponding wrapper and the two are linked by the base object received by the constructor.

getInputs(), *getOutputs()*. Your custom implementation will typically behave like (for *getInput()*): if the inputs collection != null returns it, else initialise it by getting wrappers for each node that is input of the base node for the current wrapper. Wrappers are usually created via a custom wrapper factory (more later on that). In short, what we're doing is creating a graph of wrappers that is isomorphic to the graph of nodes from your to-be converted model.

getTabValues(). In this particular model (yours may be more complicated), there is only one type of attribute that can be attached to nodes: the *Annotation* class. So the *getTabValues()* method converts the attribute into the required list of headers/values. In this foo example, each header is typed, ie.: it is like "*Characteristics[Organism]*", every attribute has a string value and can have an optional ontology term, which, in turn is made of an accession and a source. So, the wrapper defines the node type and name as first *TableValueGroup* (eg.: "BioMaterial"/"source1"), followed by one *TableGroup* per *Annotation*, which contains the annotation type, its value and, optionally, additional pairs like *"Term Accession Number"/"123"*, *"Term Source REF"/"MO"*.

createIsolatedClone () and *ExpNodeWrapper (ExpNodeWrapper original)*. The conversion algorithm needs to duplicate nodes. That is, given a node, it needs to create another wrapper node that

has the same base (points to the same node in your original model), but different edges (ie: different inputs/outputs), taken from the original node. For instance, a 2-input/2-output node is splitted into two nodes, the second node will get one input and one output from the original, which will be left with the other two edges. The algorithm decides which edges to remove/add, so the duplicated node is initially isolated (no inputs/outputs). Now, *createIsolatedClone ()* provides such a duplicate of the current node (ie: of *this*). The typical implementation of such method is something like "*return ExpNodeWrapper (this)*". So the constructor of this type, as you can see, actually creates the isolated duplicate. Indeed, it's quite simple: the new wrapper gets the base from the original one and initialise inputs/outputs with empty sets.

More notes

- The order the attributes are defined with in the wrappers is reflected in the final output table (this also depends on how nodes of the same types are merged, see below).

- Those header/value pairs that go together have to be put in the same *TableGroup* object, so that they are never split apart during the conversion. For instance, if you have a factor value with an ontology term, you have to put the *"Factor Value"*, *"Term Accession"*, *"Term Source"* headers into the same *TableGroup* (and add the corresponding values of course, which may be "", but *getValues().size()* must always be the same of *getHeaders()*).

- Two nodes of the same type (eg: two sample nodes) may have different lists of attributes, eg: one has a *"characterisc[cell line]"* attribute and the other doesn't, you just don't define any cell line or value for the second one. The converter takes care automatically of "merging" the two (or more) samples, creating the right set of columns and populating the lines with proper values or empty cells. **However**, do not play with attribute groups of the same type (having the same initial header), for example, if you define *"Factor Value [X]"* + ontology accession/source headers once, just redefine the same three headers every time you need to export the factor value of type X. Fill missing accessions or sources with "" if needed (should the factor value be blank, then accession and source will be empty too => you don't need this *TabValueGroup* at all in such a case).

- *getInputs()* and *getOutputs()* return *SortedSet*(s). This has a few implications. First, it's a bonus that allows you to establish some order on the output lines. Typically, nodes are compared on the basis of attributes like sample name or hybridization name. Second, you need to implement proper versions of *compareTo()*, *equals()* and *hashCode()*. *DefaultAbstractNode* has sensible default implementations for these methods that take the value of the first element in the first TableGroup returned by getTabValues(), ie:things like Source Name or Sample Name. This is usually what you're happy with. The implementation in *DefaultAbstractNode* also ensures a fundamental property: that two physically different node wrappers are considered different (eg: by collection-related methods), even when they are duplicated and point at the same base (the same node from your to-be-converted model). This is really important because unwanted behaviours would be observed otherwise. For example, without such a behaviour, duplicated nodes would be merged together when you attempted to store them in the same *Set* structure. So, the message is extend *DefaultAbstractNode* or, should you have particular needs why this implementation doesn't suit you well, at least give a look to the source of this class.

Node Factory

If you adapt mage2tab to your model with the wrappers approach, you typically will use an extension of *AbstractNodeFactory* either. This just keeps a map of base nodes (nodes from your model) => wrappers. It uses it in the method *NodeWrapper getNode (Base node)*, which creates a new wrapper and links it to the the given base, but only if this is not created yet and already inside the node/wrapper map. Obviously, it stores the new wrapper in the map, to return it in subsequent calls that have the same base as parameter (instead of creating a new wrapper). This ensures that you create a 1-1 graph of wrappers from the original graph is based on your to-be-converted model. Anyway, you don't need to care about

these details, you only need to implement *createNewNode()*, which will be like this:

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/NodeFactory.java

```
public class NodeFactory extends AbstractNodeFactory<ExpNodeWrapper, ExperimentNode>
{
	private NodeFactory() {}

	private static final NodeFactory instance = new NodeFactory ();

	public static NodeFactory getInstance () {
		return instance;
	}

	@Override
	protected ExpNodeWrapper createNewNode ( ExperimentNode base )
	{
		if ( base instanceof BioMaterial) return new BioMaterialWrapper ( (BioMaterial) base );
		if ( base instanceof Data ) return new DataWrapper ( (Data) base );
		if ( base instanceof ProtocolRef) return new ProtocolRefWrapper ( (ProtocolRef) base );
		throw new IllegalArgumentException (
			"Node of type " + base.getClass ().getSimpleName () + " not supported"
		);
	}
}
```

This method is what the *getNode()* above will call, when it decides it needs a new wrapper.

Table Builder

The *TableBuilder* class coordinates the conversion job. We suggest you extend it and define a method that receives nodes from your input graph, as it is done in the constructor *SimpleModelTableBuilder (Set<ExperimentNode> nodes)*:

uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/SimpleModelTableBuilder.java

```
public class SimpleModelTableBuilder extends TableBuilder
{
	public SimpleModelTableBuilder ( Set<ExperimentNode> nodes )
	{
		this.nodes = new HashSet<Node> ();
		NodeFactory nodeFact = NodeFactory.getInstance ();
		for ( ExperimentNode node: nodes ) this.nodes.add ( nodeFact.getNode ( node ) );
	}

}
```

As you can see, this method just populates the nodes class member, containing node wrappers and does it by using your node factory. Which nodes should be passed to this method? A good choice is the graph sources (e.g. *BioSource*) or the sinks (e.g.: *DataMatrix*). In general, any set of nodes that allows to reach

all the graph sources does the trick, graph2tab finds the sources walking back from such nodes. Of course the most efficient option is passing the sources straight.

Go!

Once all is in place, you can go with the conversion, basically invoking *getTable()* from your table builder. An invocation example can be found in *SimpleBioModelTest*.