Systems Biology Format Converter

Developer Manual

by the SBFC team

7th March 2016

Contents

List of Figures

1 Introduction

This chapter illustrates how to develop new converters or add new features to SBFC. SBFC is free software; you can redistribute it and/or modify it under the terms of the *GNU Lesser General Public License* as published by the Free Software Foundation; either version 2.1 of the License, or any later version. If you wish to contribute with new converters or join the team, please let us know using the following e-mail address:

biomodels-net-support@lists.sourceforge.net

A mailing-list is also available for developers at sbfc-devel@googlegroups.com.

We welcome new contributors, and your contributions can be included in the official SBFC and become publicly available. Currently, SBFC is available in two branches:

1. main branch called SBFC core; and
2. the OSGi[1] branch called SBFC osgi.

This chapter shows how to install and develop with both these branches. The next sections describe the steps required for developing SBFC using Eclipse and getting the SBFC source code. A guide for creating new converters is also provided.

2 Requirements

To follow these directions you will need the following tools:

- Eclipse[2] with the Plug-in Development Environment (PDE) included. As indicated here, the PDE comes with the Classic, Java EE or RCP/Plugin versions of Eclipse;
- A tool for checking out a repository from subversion (e.g. Subclipse for Eclipse, Tortoise for Windows, or the typical SVN command line client for Linux and Mac OSX);
- MySQL (only required for SBFC Online);
- Tomcat version 5.5 (only required for SBFC Online).

3 Installation and Configuration for SBFC core

This section explains how to set up and configure SBFC core and Ant[3].

3.1 Check out the source code

As reference, SBFC_core will be the directory containing the project to check out.
Go into the directory where you want to check out the project.

- Command line check out with SVN[4]:

```
svn checkout svn://svn.code.sf.net/p/sbfc/code/trunk SBFC_core
```

- Eclipse. Check out source code with Subclipse SVN[5]:

[1] See http://www.osgi.org/
[2] See https://eclipse.org/
[3] See http://ant.apache.org/
[4] See https://subversion.apache.org/
[5] See http://subclipse.tigris.org/

```
File -> Import -> SVN -> Checkout Projects with SVN
URL= svn://svn.code.sf.net/p/sbfc/code/trunk
Create new Java Project -> Save the project as SBFC_core -> Finish
```

After checking out the project, you will need reference the SBFC_core libraries to the Java Build Path. In Eclipse:

```
Package Explorer -> SBFC_core -> Properties -> Java Build Path -> Add JARs
Expand SBFC_core -> Expand lib.
Select all the jar files in lib -> OK.
```

3.2 Set up and configure SBFC core with Ant

- Eclipse:

```
(Set up)
Package Explorer window -> open the project SBFC_core;
Select build.xml -> Run As -> External Tools Configurations;
In the window corresponding to SBFC_core build.xml, select compile in the tab
    Targets.
(Run)
Package Explorer window  -> open the project SBFC_core;
Select build.xml -> Run As -> Ant Build.
```

- Run using command line:

```
cd SBFC_core
ant jar
```

4 Installation and Configuration for SBFC online (Web server)

This section explains how to set up and configure SBFC Online, MySQL[6], Tomcat[7] and Ant.

4.1 Check out the source code

As reference, SBFC_Online will be the directory containing the project to check out. Go into the directory where you want to check out the project.

- Command line check out with SVN:

```
svn checkout svn://svn.code.sf.net/p/sbfc/code/sbfcOnline SBFC_Online
```

- Eclipse. Check out source code with Subclipse SVN:

```
File -> Import -> SVN -> Checkout Projects with SVN
URL= svn://svn.code.sf.net/p/sbfc/code/sbfcOnline
Create new Java Project -> Save the project as SBFC_Online -> Finish
```

[6]See http://www.mysql.com/
[7]See http://tomcat.apache.org/

After checking out the project, you will need reference the SBFC_Online libraries to the Java Build Path. In Eclipse:

```
Package Explorer -> SBFC_Online -> Properties -> Java Build Path -> Add JARs
Expand SBFC_Online -> Expand lib.
Select all the jar files in lib -> OK.
```

4.2 Install and configure MySQL

- Linux users: Install MySQL from the repository of your Linux distribution.
- Windows users: Download and install MySQL from:
  http://dev.mysql.com/downloads/windows/

After installing MySQL, you have to source the file converters.sql inside the folder db/. Assuming you use the user root:

```
cd SBFC_Online/db
mysql -u root
# and from MySQL command line, type:
create database converters;
connect converters;
source converters.sql;
# you should see the tables now:
show tables;
# you should see the new database now:
show databases;
exit;
```

Finally, you have to link SBFC_Online to the imported database. To do this you need to:

```
open the file: SBFC_Online/WebContent/META-INF/Context.xml
edit the line: url="jdbc:mysql://localhost/converters?autoReconnect=true"
```

If you want to specify a specific non-root user with read/write access to the database, you also have to edit the lines:

```
username="root"
password=""
```

If you use another database instead of MySQL, you also have to edit the lines:

```
name="jdbc/mydb"
driverClassName="com.mysql.jdbc.Driver"
```

4.3 Install and configure Tomcat

SBFC Online requires Tomcat v. 5.5 to run correctly.

- Linux users: Install tomcat from the repository of your Linux distribution.
- Windows users: Download and install Tomcat[8].

After installing Tomcat, you need to reference the jar file mail.jar to the Java Build Path for SBFC_Online. In Eclipse:

[8] See http://tomcat.apache.org/

```
Package Explorer -> SBFC_Online -> Properties -> Java Build Path -> Add External JARs
Search for APACHE_TOMCAT_FOLDER/common/lib/
Select the file mail.jar -> OK.
```

Then, add admin and manager permissions to the user tomcat:

```
# File: apache-tomcat-5.5.36/conf/tomcat-users.xml
<user username="tomcat" password="tomcat" roles="admin,manager,tomcat"/>
```

You will need to created the nested folder WWW/onlineConvert:

```
# If you want these folders to be located in your $HOME, then:
cd $HOME
mkdir -p WWW/onlineConvert
```

And then configure SBFC_Online with your parameters. To do this you need to edit the field value in the configuration file context.xml accordingly with your paths. In particular, you will have to replace $HOME with the path where the folder WWW/onlineConverter was created in the previous step, and $SBFC_FOLDER with the path where SBFC_core is located.

```
# File: SBFC_Online/WebContent/META-INF/context.xml
<Environment name="SBFC_WEBAPP_HOME" value="$HOME/WWW/onlineConvert" type="java.lang
    .String" override="false"/>
<Environment name="SBFC_WEBAPP_FILES" value="$HOME/WWW/onlineConvert/jobs/" type="
    java.lang.String" override="false"/>
<Environment name="SBFC_WEBAPP_ZIP" value="$HOME/WWW/onlineConvert/zip/" type="java.
    lang.String" override="false"/>
<Environment name="SBFC_WEBAPP_WS" value="$HOME/WWW/onlineConvert/ws/" type="java.
    lang.String" override="false"/>
<Environment name="SBFC_HOME" value="${SBFC_FOLDER}/SBFC_core" type="java.lang.
    String" override="false"/>
<Environment name="SBFC_COMMAND_EBI" value="bsub -o $HOME/WWW/onlineConvert/jobs/{0}.
    bsubout ${SBFC_FOLDER}/SBFC_core/sbfConverterOnline.sh {1} {2}
     $HOME/WWW/onlineConvert/jobs/{0}.input" type="java.lang.String" override="false
        "/>
<Environment name="SBFC_COMMAND" value="${SBFC_FOLDER}/SBFC_core/sbfConverterOnline.sh
     {1} {2} $HOME/WWW/onlineConvert/jobs/{0}.input"
     type="java.lang.String" override="false"/>
```

Finally, start tomcat by typing catalina.sh from a shell.

```
Test: start your web browser and visit the page http://localhost:8080/manager/html.
```

4.4 Set up and configure SBFC Online with Ant

Before proceeding, make sure that your Tomcat is running properly. In order to use your localhost, in the folder SBFC_Online you need to edit the deploy.url property name in the file build.xml:

```
<property name="deploy.url" value="http://localhost:8080/manager" />
```

- Eclipse:

```
(Set up)
Package Explorer window -> open the project SBFC_Online;
```

```
Select build.xml -> Run As -> External Tools Configurations;
In the window corresponding to SBFC_Online build.xml, select tomcat.deploy in
    the tab Targets.
(Run)
Package Explorer window  -> open the project SBFC_Online;
Select build.xml -> Run As -> Ant Build.
```

- Run using command line:

```
cd SBFC_Online
ant tomcat.deploy
```

Test (Tomcat and MySQL must be running): start your web browser and visit the page:
http://localhost:8080/converters/ .
If *ant tomcat.deploy* does not work, you can run *ant package* to generate the war file.

5 Installation and Configuration for SBFC osgi

Aside from the standard branch (trunk), SBFC core is also available in a branch, called SBFC osgi using the OSGi package architecture. The OSGi technology offers a substantial support for developing highly modular applications and is a design key for the software Cytoscape, a powerful visualisator and analyser of biological networks. By implementing a SBFC branch adopting OSGi, we hope to make easier to port SBFC into Cytoscape and other software adopting OSGi in the long term. This section explains how to set up and configure SBFC OSGi and Ant[9].

5.1 Check out the source code

As reference, SBFC_osgi will be the directory containing the project to check out.
Go into the directory where you want to check out the project.

- Command line check out with SVN[10]:

```
svn co https://svn.code.sf.net/p/sbfc/code/branches/osgi SBFC_osgi
```

- Eclipse. Check out source code with Subclipse SVN[11]:

```
File -> Import -> SVN -> Checkout Projects with SVN
URL= svn://svn.code.sf.net/p/sbfc/code/branches/osgi
Create new Java Project -> Save the project as SBFC_osgi -> Finish
```

After checking out the project, you will need reference the SBFC_osgi libraries to the Java Build Path.
In Eclipse:

```
Package Explorer -> SBFC_osgi -> Properties -> Java Build Path -> Add JARs
Expand SBFC_osgi -> Expand lib.
Select all the jar files in lib -> OK.
```

[9] See http://ant.apache.org/
[10] See https://subversion.apache.org/
[11] See http://subclipse.tigris.org/

5.2 Set up and configure SBFC osgi with Ant

- Eclipse:

```
(Set up)
Package Explorer window -> open the project SBFC_osgi;
Select build.xml -> Run As -> External Tools Configurations;
In the window corresponding to SBFC_osgi build.xml, select osgi in the tab
    Targets.
(Run)
Package Explorer window  -> open the project SBFC_osgi;
Select build.xml -> Run As -> Ant Build.
```

- Run using command line:

```
cd SBFC_osgi
ant osgi
```

6 Develop new SBFC converters

To facilitate the implementation of additional converters, SBFC was designed with an high degree of modularity. At the software core are:

- the interface *GeneralModel* in the package *org.sbfc.converter.models*;
- the abstract class *GeneralConverter* in the package *org.sbfc.converter*.

For the OSGi branch of SBFC both these two classes are located in the package *org.sbfc.api*. The interface *GeneralModel* is used for data exchange and describes the services that every input or output computational model object must implement to interact with SBFC.

```
/**
 * Interface defining the specifications that each Model must implement.
 */
public interface GeneralModel {
  /**
   * Set the Model from a file in the file system.
   * @param fileName path to the file containing the model
   * @throws ReadModelException
   */
  public void setModelFromFile(String fileName) throws ReadModelException;

  /**
   * Set the model from a String.
   * @param modelString Model
   * @throws ReadModelException
   */
  public void setModelFromString(String modelString) throws ReadModelException;

  /**
   * Write the Model into a new file.
   * @param fileName path at which the new file will be created
   * @throws WriteModelException
   */
  public void modelToFile(String fileName) throws WriteModelException;

  /**
   * Return the Model as a String.
   * @return Model
   * @throws WriteModelException
   */
```

```
  public String modelToString() throws WriteModelException;

  /**
   * Return an array of model file type extension (e.g.: [.xml, .sbml]
   * for SBML, [.owl] for BIOPAX)
   * The first is the preferred extension.
   *
   * @return file type extensions
   */
  public String[] getExtensions();

  /**
   * This method is used to distinguish between converters with the same file
   * extension.
   * For example, a file ending with .xml could be either an SBML model, or
   * some other XML file type.
   * <p>
   * Implementers should perform a quick heuristic, not a full validation. For
   * example XML file types may examine the root element to determine if it has
   * the correct name or namespace.
   * <p>
   * Implementers should <b>only</b> return false if they are <b>certain</b> that
   * the file type is wrong. If the correctness could not be determined for sure,
   * the method should always return true.
   * @return false if the file is not the correct type to be used with this
   * GeneralModel
   */
  public boolean isCorrectType(File f);

  /**
   * Return a URI for the model
   * e.g. MIME types: image/png, application/matlab, text/xpp
   * e.g. COMBINE spec ids: http://identifiers.org/combine.specifications/sbml
   *
   * @return the model URI
   */
  public String getURI();
}
```

The abstract class *GeneralConverter* represents the generic algorithm for converting one model into another.

```
/**
 * Abstract class defining the specifications that each Converter must implement.
 */
public abstract class GeneralConverter {
  /**
   * The input model to be converted.
   */
  protected GeneralModel inputModel = null;

  /**
   * The options for the converter. Each option is defined as a pair (name,
   * value). For instance, for the converter SBML2SBML, one option is
   * ("sbml.target.level", "3").
   */
  protected Map<String, String> options;

  /**
   * Method to convert a GeneralModel into another.
   * @param model
   * @return GeneralModel
   */
  public abstract GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException;

  /**
   * Return the extension of the Result file.
```

```
   * @return String
   */
  public abstract String getResultExtension();

  /**
   * Set the converter options.
   * @param options
   */
  public void setOptions(Map<String, String> options) {
    this.options = options;
  }

  /**
   * Return the input model.
   * @return the input model
   */
  public GeneralModel getInputModel() {
    return inputModel;
  }

  /**
   * Return the converter name as it should be displayed.
   * @return the name
   */
  public abstract String getName();

  /**
   * Return the converter description.
   * @return the description
   */
  public abstract String getDescription();

  /**
   * Return the converter description in HTML format.
   * @return the HTML description
   */
  public abstract String getHtmlDescription();
}
```

To add a new converter, a developer needs to extend the class *GeneralConverter* and implement the method *GeneralModel convert(GeneralModel model)*, where the parameter *model* is the model to convert. The model type is an implementation of the interface *GeneralModel*. UML class diagrams for the *GeneralModel* and *GeneralConverter* class hierarchies are shown in Figures 1 and 2, illustrating the model types and the converters currently implemented in SBFC.

Figure 1: UML class diagram for the *GeneralModel* class hierarchy.

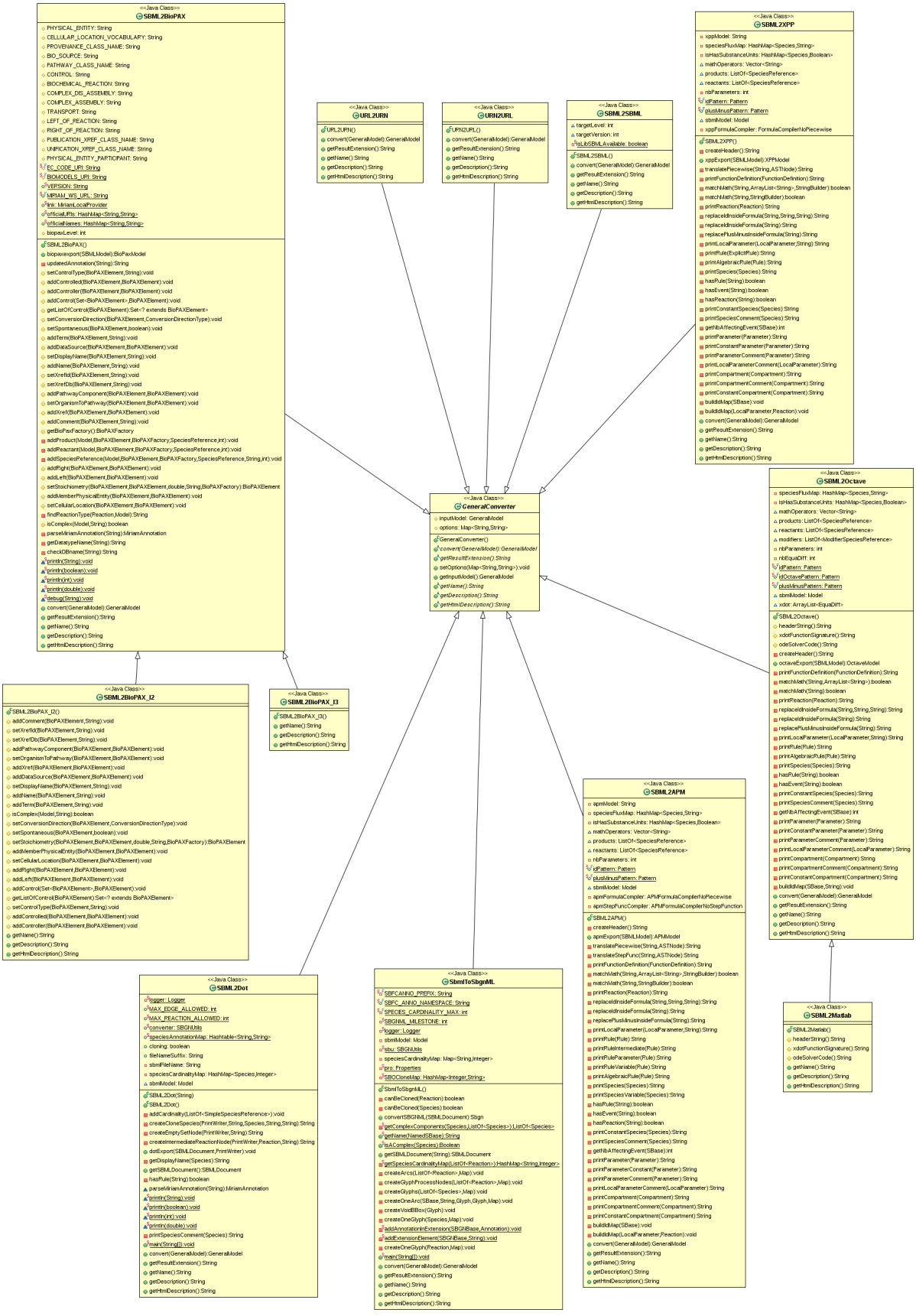
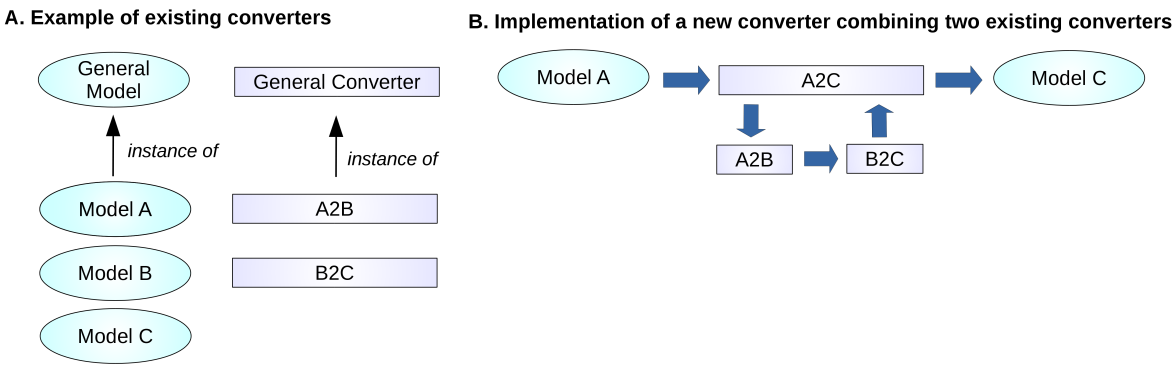


Figure 2: UML class diagram for the *GeneralConverter* class hierarchy.



C. Implementation of the method convert() for the converter A2C

```
public class A2C extends GeneralConverter {
  ..
  /** Convert model format A to model format C. */
  @Override
  public GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException {
    // assume model is of type AModel
    try {
       A2B a2b = new A2B();
       B2C b2c = new B2C();
       // concatenate as a transitive relationship
       return b2c.convert(a2b.convert((AModel)model));
    }
    catch (ReadModelException e) { throw e; }
    catch (ConversionException e) { throw e; }
  }
  ..
}
```

Figure 3: Creation of complex converters. (A) In this scenario, three existing SBFC models (A, B, and C) and two converters (A2B and B2C) are considered. Each of these models represents a model format and implements the interface GeneralModel. The current converters extend the abstract class GeneralConverter and translate from Model A to Model B, and from Model B to Model C. (B) A new converter A2C to translate from Model A to Model C, can be added without effort by invoking the method convert() implemented in the converters A2B and B2C. (C) Java source code illustrating the implementation of the method convert() for the converter class A2C.

7 Examples: how to create new converters

7.1 Example: how to create a new converter from existing converters

This section presents a Java example illustrating the implementation from scratch of new converters between trivial model formats and how to create a converter from existing converters. It is assumed that all these new models classes are located in the SBFC Java package: *org.sbfc.converter.models*, whereas the converter classes are located in the package: *org.sbfc.converter.example*. A graphic diagram for this example is provided in Figure 3.

First of all, we create three new classes, each representing a new model format. Let's call these classes: AModel, BModel, and CModel. For simplicity, each of these classes extends the abstract class StringModel located in the package *org.sbfc.converter.models*. This abstract class implements the methods for reading and writing a file or string model, as declared in the interface GeneralModel.

../../examples/org/sbfc/converter/models/AModel.java

```
package org.sbfc.converter.models;

import java.io.File;
import org.sbfc.converter.models.StringModel;

public class AModel extends StringModel {

  @Override
  public String[] getExtensions() { return new String[] { ".a" }; }
```

```
  @Override
  public boolean isCorrectType(File f) { return true; }

  @Override
  public String getURI() { return "text/a"; }
}
```

../../examples/org/sbfc/converter/models/BModel.java

```
package org.sbfc.converter.models;

import java.io.File;
import org.sbfc.converter.models.StringModel;

public class BModel extends StringModel {

  @Override
  public String[] getExtensions() { return new String[] { ".b" }; }

  @Override
  public boolean isCorrectType(File f) { return true; }

  @Override
  public String getURI() { return "text/b"; }
}
```

../../examples/org/sbfc/converter/models/CModel.java

```
package org.sbfc.converter.models;

import java.io.File;
import org.sbfc.converter.models.StringModel;

public class CModel extends StringModel {

  @Override
  public String[] getExtensions() { return new String[] { ".c" }; }

  @Override
  public boolean isCorrectType(File f) { return true; }

  @Override
  public String getURI() { return "text/c"; }
}
```

Now, we would like to create two new converters: one from AModel to BModel, called A2B, and another from BModel to CModel, called B2C. As expected, both these converters extends the abstract class GeneralConverter. The structure is the same and the two converters would differ for the way data is extracted from the input model and translated into the output model.

../../examples/org/sbfc/converter/example/A2B.java

```
package org.sbfc.converter.example;

import org.sbfc.converter.exceptions.ConversionException;
import org.sbfc.converter.exceptions.ReadModelException;
import org.sbfc.converter.GeneralConverter;
import org.sbfc.converter.models.AModel;
import org.sbfc.converter.models.BModel;
import org.sbfc.converter.models.GeneralModel;

/** Convert model format A to model format B. */
public class A2B extends GeneralConverter {

  /** Constructor A2B. */
  public A2B() { super(); }

  /** Convert model format A to model format B. */
```

```
  public BModel bExport(AModel aModel)
  throws ReadModelException, ConversionException {
    String bStringModel = "I am a B model!";
    // extract species, reactions, parameters, compartment, rules,
    // events, and functionDefinitions from aModel
    // and store the converted values in bStringModel

    // if aModel cannot be parsed, throw ReadModelException
    // if aModel cannot be converted to bStringModel, throw ConversionException
    BModel bModel = new BModel();
    bModel.setModelFromString(bStringModel);
    return bModel;
  }

  @Override
  public GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException {
    // assume model is of type AModel
    try { return bExport((AModel)model); }
    catch (ReadModelException e) { throw e; }
    catch (ConversionException e) { throw e; }
  }

  @Override
  public String getResultExtension() { return ".b"; }

  @Override
  public String getName() { return "A2B"; }

  @Override
  public String getDescription() {
    return "It converts a model format from A to B";
  }

  @Override
  public String getHtmlDescription() {
    return "It converts a model format from A to B";
  }
}
```

../../examples/org/sbfc/converter/example/B2C.java

```
package org.sbfc.converter.example;

import org.sbfc.converter.exceptions.ConversionException;
import org.sbfc.converter.exceptions.ReadModelException;
import org.sbfc.converter.GeneralConverter;
import org.sbfc.converter.models.BModel;
import org.sbfc.converter.models.CModel;
import org.sbfc.converter.models.GeneralModel;

/** Convert model format B to model format C. */
public class B2C extends GeneralConverter {

  /** Constructor B2C. */
  public B2C() { super(); }

  /** Convert model format B to model format C. */
  public CModel cExport(BModel bModel)
  throws ReadModelException, ConversionException {
    String cStringModel = "I am a C model!";
    // extract species, reactions, parameters, compartment, rules,
    // events, and functionDefinitions from aModel
    // and store the converted values in cStringModel

    // if aModel cannot be parsed, throw ReadModelException
    // if aModel cannot be converted to cStringModel, throw ConversionException
    CModel cModel = new CModel();
    cModel.setModelFromString(cStringModel);
    return cModel;
```

```
  }

  @Override
  public GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException {
    // assume model is of type BModel
    try { return cExport((BModel)model); }
    catch (ReadModelException e) { throw e; }
    catch (ConversionException e) { throw e; }
  }

  @Override
  public String getResultExtension() { return ".c"; }

  @Override
  public String getName() { return "B2C"; }

  @Override
  public String getDescription() {
    return "It converts a model format from B to C";
  }

  @Override
  public String getHtmlDescription() {
    return "It converts a model format from B to C";
  }
}
```

At this point we could quickly implement a converter translating an AModel to a CModel, called A2C, by simply reusing the previously implemented converters. In order to do this, we simply invoke the converters A2B and B2C as a simple pipeline. The implementation of this new converter created from existing converters is therefore straightforward and illustrated as below.

../../examples/org/sbfc/converter/example/A2C.java

```
package org.sbfc.converter.example;

import org.sbfc.converter.exceptions.ConversionException;
import org.sbfc.converter.exceptions.ReadModelException;
import org.sbfc.converter.GeneralConverter;
import org.sbfc.converter.models.AModel;
import org.sbfc.converter.models.CModel;
import org.sbfc.converter.models.GeneralModel;
import org.sbfc.converter.example.A2B;
import org.sbfc.converter.example.B2C;

/** Convert model format A to model format C re-using the converter
 *  A2B and B2C.
 */
public class A2C extends GeneralConverter {

  /** Constructor A2C. */
  public A2C() { super(); }

  /** Convert model format A to model format C. */
  public CModel cExport(AModel aModel)
  throws ReadModelException, ConversionException {
    A2B a2b = new A2B();
    B2C b2c = new B2C();
    // concatenate the conversion as a transitive relationship
    return (CModel) b2c.convert(a2b.convert(aModel));
  }

  @Override
  public GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException {
    // assume model is of type AModel
    try { return cExport((AModel)model); }
    catch (ReadModelException e) { throw e; }
    catch (ConversionException e) { throw e; }
```

```
  }

  @Override
  public String getResultExtension() { return ".c"; }

  @Override
  public String getName() { return "A2C"; }

  @Override
  public String getDescription() {
    return "It converts a model format from A to C";
  }

  @Override
  public String getHtmlDescription() {
    return "It converts a model format from A to C";
  }
}
```

Another way to do this is to simply use the static method *String Convert.convertFromString(String inputModelType, String converterType, String modelString)* from the class *Convert* in the package *org.sbfc.converter* to convert models as string. This is particularly useful if one does not need to implement a new converter but simply wants to create a quick pipeline from existing converters.

```
String aModel;
// ... build the string aModel with data using the format for a
// model of type AModel.
String bModel = Convert.convertFromString(AModel, A2B, aModel);
String cModel = Convert.convertFromString(BModel, B2C, bModel);
// we now have a conversion from model a model of type AModel to
// a model of type CModel.
```

The same can be achieved with the method *String Convert.convertFromFile(String inputModelType, String converterType, String inputFileName)*.
A third and final way to achieve the same result with no Java programming, is to use the script *sbfConverter.sh* (or *sbfConverter.bat* on Windows machines), and run:

```
./sbfConverter.sh AModel A2B [file.a | folder containing files with extension .a]
./sbfConverter.sh BModel B2C [file.b | folder containing files with extension .b]
```

This will generate a file.c (or a set of files with extension .c) of type CModel.

7.2 Example: how to create a new converter invoking non-Java external programs

This example shows how to create a new converter which invokes an external program. This can be particularly useful when a converter programmed in a language different from Java should be used. In such a case, we could define a converter that works as a bridge between our code and this external existing converter.
First of all, we create a class representing a CellMLModel.

../../examples/org/sbfc/converter/models/CellMLModel.java

```
package org.sbfc.converter.models;

import java.io.File;

/**
 * A {@link StringModel} representing a CellML model (http://www.cellml.org).
 *
 * @author rodrigue
 *
 */
```

```
public class CellMLModel extends StringModel {

  @Override
  public String[] getExtensions() {
    return new String[] { ".cellml", ".xml" };
  }

  @Override
  public boolean isCorrectType(File f) {
    return true;
  }

  @Override
  public String getURI() {
    return "http://identifiers.org/combine.specifications/cellml";
  }

}
```

Then we can proceed with the creation of a new converter called CellML2SBML.

../../examples/org/sbfc/converter/example/CellML2SBML.java

```
package org.sbfc.converter.example;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;

import org.sbfc.converter.GeneralConverter;
import org.sbfc.converter.exceptions.ConversionException;
import org.sbfc.converter.exceptions.ReadModelException;
import org.sbfc.converter.exceptions.WriteModelException;
import org.sbfc.converter.models.GeneralModel;
import org.sbfc.converter.models.SBMLModel;

/**
 * This converter will use an external program to
 * do the conversion. As an example, we will convert from CellML to SBML using
 * antimony (http://antimony.sourceforge.net/).
 *
 * <p>By using this template, you can easily add a new converter that
 * call an external program.
 */
public class CellML2SBML extends GeneralConverter {

  /**
   * Path to the antimony sbtranslate binary.
   */
  public final static String PROGRAM = "/path/to/antimony/bin/sbtranslate";

  /** Creates a new instance. */
  public CellML2SBML() { super(); }

  @Override
  public GeneralModel convert(GeneralModel model)
  throws ConversionException, ReadModelException
  {
    try {
      //
      // 1. Dealing with the input format
      //
      // We first need to save the model into a temporary file so that the
      // external program can read it.
      // GeneralModel does not store the original file location and it can
      // also be initialize directly from String.

      // We could check here that the input format correspond to the CellML URI.

      // creating the temporary file
```

```
    File inputFile = File.createTempFile("cellml-", ".xml");

    // writing the model to the temporary file
    model.modelToFile(inputFile.getAbsolutePath());

    //
    // 2. Running the external program
    //

    // creating a second temporary file for the output
    File outputFile = File.createTempFile("sbml-", ".xml");

    // using the Runtime exec method to run the external program:
    Process p = Runtime.getRuntime().exec(PROGRAM + " -o sbml-comp -outfile "
        + outputFile.getAbsolutePath() + " " + inputFile.getAbsolutePath());
    // waiting for the program to finish.
    p.waitFor();

    // read the output messages from the command
    BufferedReader stdInput = new BufferedReader(new
        InputStreamReader(p.getInputStream()));
    String line;

    while ((line = stdInput.readLine()) != null) {
      // You might want to read the process output to check that the conversion went
          fine
      // and if not, you can throw an exception with an appropriate error message.
      System.out.println("Output: " + line);
    }

    // read the error messages from the command
    BufferedReader stdError = new BufferedReader(new
        InputStreamReader(p.getErrorStream()));

    while ((line = stdError.readLine()) != null) {
      // You might want to read the process error output to check that the conversion
          went fine
      // and if not, you can throw an exception with an appropriate error message.
      System.out.println("Errors: " + line);
      if (line.startsWith("Segmentation fault")) {
        throw new ConversionException("Encountered a Segmentation fault while running
            antomony !!");
      }
    }

    //
    // 3. Returning the output format
    //

    // creating a new empty SBMLModel
    GeneralModel outputModel = new SBMLModel();

    // reading the output file into the SBMLModel
    outputModel.setModelFromFile(outputFile.getAbsolutePath());

    return outputModel;
  }
  catch (IOException e) {
    throw new ReadModelException(e);
  } catch (WriteModelException e) {
    throw new ReadModelException(e);
  } catch (InterruptedException e) {
    throw new ReadModelException(e);
  }
}

@Override
public String getResultExtension() { return ".xml"; }

@Override
```

```
  public String getName() { return "CellML2SBML"; }

  @Override
  public String getDescription() {
            return "Convert a CellML model to SBML";
  }

  @Override
  public String getHtmlDescription() {
    return "Convert a CellML model to SBML";
  }
}
```

In the code above, the converter uses the Runtime method exec to start a new process.
To use this new converter the command will be:

```
./sbfConverter.sh CellMLModel CellML2SBML [file.cellml | file.xml | folder containing
    files with extension .cellml or .xml]
```