This tutorial explains how to use SBOL 2 to represent the function of a state-of-art design, namely a CRISPR-based repression module Kiana et al. [1] using the Java library. Before attempting this tutorial, users may find the corresponding tutorial on Maven instructions for using libSBOLj to be useful. The detailed Maven instructions can be accessed in the Maven Setup tutorial.

A brief description of the CRISPR circuit using SBOL 2 data model

We first give a brief description of the CRISPR-based repression module. We use bold font in the following text and figure captions to mark available data model in SBOL 2. Detailed description of properties of the data model is available in the Specification (Data Model 2).

First, consider the CRISPR-based Repression Template ModuleDefinition shown in the center of Figure 1. It provides a generic description of CRISPR-based repression behavior. Namely, it includes generic Cas9, guide RNA (gRNA), and target DNA FunctionalComponent instances. It also includes a genetic production Interaction that expresses a generic target gene product. Finally, it includes a non-covalent binding Interaction that forms the Cas9/gRNA complex (shown as dashed arrows), which in turn participates in an inhibition Interaction to repress the target gene product production (shown with a tee-headed arrow). The CRISPR-based Repression Template is then instantiated to test a particular CRISPR-based repression device, CRPb, by the outer CRPb Characterization Circuit ModuleDefinition. This outer characterization circuit includes gene FunctionalComponents to produce specific products (i.e., mKate, Gal4VP16, cas9m_BFP, gRNA_b, and EYFP), as well as FunctionalComponents for the products themselves. Next, it includes genetic production Interactions connecting the genes to their products, and it has a stimulation Interaction that indicates that Gal4VP16 stimulates production of EYFP. Finally, it uses MapsTo objects (shown as dashed lines) to connect the generic FunctionalComponents in the template to the specific objects in the outer ModuleDefinition. For example, the outer module indicates that the target protein is EYFP, while the cas9_gRNA complex is cas9m_BFP_gRNA_b.
Figure 1: Illustration of a hierarchical CRISPR-based repression module represented in SBOL 2 (adapted from Figure 1a in [1]). The CRISPR-based Repression Template ModuleDefinition describes a generic CRISPR repression circuit that combines a Cas9 protein with a gRNA to form a complex (represented by the dashed arrows) that represses a target gene (represented by the arrow with the tee arrowhead). These relationships between these Functional-Components (instances of ComponentDefinitions) are represented in SBOL 2 using Interactions. This Module is instantiated in the outer CRPb Characterization Circuit ModuleDefinition in order to specify the precise (including Sequences when provided) FunctionalComponents used for each generic FunctionalComponent. The undirected dashed lines going into the template Module represent MapsTo objects that specify how specific FunctionalComponents replace the generic ones.

Modeling CRISPR repression using libSBOLj 2

Setting up the CRISPR repression model

Now we are ready to create the CRISPR model to our project. When the “crispr” project was created, two template Java class files “App.java” and “AppTest.java” were automatically created as shown below.

We can modify “App.java” to build the CRISPR model. First, remove all lines in “App.java” except the first line, which contains its package information. Then, rename “App.java” to “RepressionModel.java”. You may now copy the import statements shown below to this file.

```java
import java.net.URI SyntaxException;
import javax.xml.stream.FactoryConfigurationException;
import javax.xml.stream.XMLStreamException;
import org.sbolstandard.core2.AccessType;
import org.sbolstandard.core2.ComponentDefinition;
import org.sbolstandard.core2.DirectionType;
import org.sbolstandard.core2.Interaction;
import org.sbolstandard.core2.Module;
import org.sbolstandard.core2.ModuleDefinition;
import org.sbolstandard.core2.RefinementType;
import org.sbolstandard.core2.RestrictionType;
import org.sbolstandard.core2.SBOLDocument;
import org.sbolstandard.core2.SBOLWriter;
import org.sbolstandard.core2.Sequence;
import org.sbolstandard.core2.SequenceOntology;
import uk.ac.ncl.intbio.core.io.CoreIOException;
```
Creating SBOL Document

For simplicity, we put all of our code in the `main` method of the `RepressionModel` class. All SBOL data objects are organized within an `SBOLDocument` object. The `SBOLDocument` provides a rich set of methods to create, access, update, and delete each type of `TopLevel` object (i.e., `Collection`, `ModuleDefinition`, `ComponentDefinition`, `Sequence`, `Model`, or `GenericTopLevel`). Every SBOL object has a `uniform resource identifier` (URI) and consists of properties that may refer to other objects, including non-`TopLevel` objects such as `SequenceConstraint` and `Interaction` objects. `libSBOLj 2` organizes the URI collections to enable efficient access, and validation of uniqueness. We first create an `SBOLDocument` object by calling its constructor as shown below. One thing to mention, if users receive compiler errors for unhandled exceptions, then add the exception to the list of possible thrown exceptions as shown below.

```java
public class RepressionModel {
    public static void main(String[] args) throws URISyntaxException,
            SBOLValidationException, SBOLConversionException {
        SBOLDocument doc = new SBOLDocument();
        doc.setDefaultURIprefix("http://sbols.org/CRISPR_Example/");
        doc.setComplete(true);
        doc.setCreateDefaults(true);
    }
}
```

- `setDefaultURIprefix` sets the default URI prefix to the string “http://sbols.org/CRISPR_Example/”. All data objects created following this statement carry this default URI prefix.
- `doc.setComplete(true)` statement sets the “complete” flag to `true` for the given `doc`. It means that the URI for each data object in the current `doc` can be dereferenced to a valid object in this `SBOLDocument` object. Otherwise, the library throws an exception.
- `doc.setCreateDefaults(true)` statement sets the “createDefaults” flag to `true` for the given `doc`. It means that when an object that must reference either a `Component` or `FunctionalComponent` object and cannot find the component, but it can find a `ComponentDefinition` object with the same `displayId`, it creates a default `Component` or `FunctionalComponent` object with this `displayId` and references it.

Adding CRISPR-based Repression Template module

Creating TopLevel objects

A point reiterated is all code present in this tutorial should be pasted into the `main` method of a java project. We first create the CRISPR-based Repression Template module shown in Figure 1. In this template, we include definitions for generic Cas9, guide RNA (gRNA), and target DNA `FunctionalComponent` instances. They are encoded as `ComponentDefinition` objects. Creation of the generic Cas9 (line 3) `ComponentDefinition` is done by passing its `displayId` “cas9_generic”, `version` specified by the `version` string, and `type` to the `createComponentDefinition` method. Every `ComponentDefinition` must contain one or more types, each of which is specified by a URI. A type specifies the component’s category of biochemical or physical entity (for example DNA, protein, or small molecule). The generic Cas9’s type is `PROTEIN`, which is defined as the BioPAX ontology term for protein (http://www.biopax.org/release/biopax-level3.owl#Protein). Other `ComponentDefinition` objects shown below are created in the same way. A `ComponentDefinition` object can optionally have one or more roles, also in the form of URIs. The gRNA_generic has a role of `SGRNA` (line 11 below), defined as the `Sequence Ontology` (SO) term “SO:0001998” (http://identifiers.org/so/SO:0001998) in the library. Similarly, the target_gene on line 21 below has a role of `PROMOTER`, defined as SO term “SO:0000167” (http://identifiers.org/so/SO:0000167). We then create the `ModuleDefinition` template by calling the `createModuleDefinition` method with the `displayId` “CRISPR_Template” and `version`.
Each of create methods in the library creates a compliant URI with the following form:

```
http://⟨prefix⟩/⟨displayId⟩/⟨version⟩
```

using the default URI prefix and provided displayId and version. The ⟨prefix⟩ represents a URI for a namespace (for example, www.sbols.org/CRISPR). The author of a TopLevel object, such as the ModuleDefinition object we just created, should use a URI prefix that either they own or an organization of which they are a member owns. When using compliant URIs, the owner of a prefix must ensure that the URI of any unique TopLevel object that contains the prefix also contains a unique ⟨displayId⟩ or ⟨version⟩ portion. Multiple versions of an SBOL object can exist and would have compliant URIs that contain identical prefixes and displayIds, but each of these URIs would need to end with a unique version. Lastly, the compliant URI of a non-TopLevel object is identical to that of its parent object, except that its displayId is inserted between its parent’s displayId and version. This form of compliant URIs is chosen to be easy to read, facilitate debugging, and support a more efficient means of looking up objects and checking URI uniqueness.

### Specifying Interactions

We are now ready to specify the interactions in the repression template. The first one is the complex formation interaction for cas9_generic and gRNA_generic. We first create an Interaction object Cas9Complex_Formation in CRISPR_Template, with the displayId “cas9_complex_formation” and a non-covalent binding type (line 1 to 4). It is recommended that terms from the Systems Biology Ontology (SBO) [2] are used specify the types for interactions. Table 11 of the Specification (Data Model 2) document provides a list of possible SBO terms for the types property and their corresponding URIs.

Next, we create three participants to this interaction object. Each participant is modeled as a Participation object and it refers to the corresponding FunctionalComponent object. For example, the participant_cas9_generic Participation object refers to the cas9_generic FunctionalComponent object. Note that such an object does not exist yet, but since there exists a cas9_generic ComponentDefinition object, and the “createDefaults” of this SBOLDocument is set to true, a FunctionalComponent with the same displayId “cas9_generic” is automatically
created and it connects the reference from the participant_cas9_generic Participation to the cas9_generic ComponentDefinition. We also assign a REACTANT role to it. The remaining two participants are created in a similar fashion.

```java
Interaction Cas9Complex_Formation = CRISPR_Template.createInteraction(
    "cas9_complex_formation",
    SystemsBiologyOntology.NON_COVALENT_BINDING
);
Cas9Complex_Formation.createParticipation(
    "participant_cas9_generic",
    "cas9_generic",
    SystemsBiologyOntology.REACTANT
);
Cas9Complex_Formation.createParticipation(
    "participant_gRNA_generic",
    "gRNA_generic",
    SystemsBiologyOntology.REACTANT
);
Cas9Complex_Formation.createParticipation(
    "participant_cas9_gRNA_complex",
    "cas9_gRNA_complex",
    SystemsBiologyOntology.PRODUCT
);
```

The remaining two interactions, namely the genetic production of the target protein from the target_gene and the inhibition of the target protein by the cas9_gRNA_complex, are specified using the same method calls.

```java
// Production of target from target gene
Interaction EYFP_production = CRISPR_Template.createInteraction(
    "target_production",
    SystemsBiologyOntology.GENETIC_PRODUCTION
);
EYFP_production.createParticipation(
    "participant_target_gene",
    "target_gene",
    SystemsBiologyOntology.PROMOTER
);
EYFP_production.createParticipation(
    "participant_target",
    "target",
    SystemsBiologyOntology.PRODUCT
);

// Inhibition of target by cas9m_BFP_gRNA
Interaction target_generic_gene_inhibition = CRISPR_Template.createInteraction(
    "target_gene_inhibition",
    SystemsBiologyOntology.INHIBITION
);
target_generic_gene_inhibition.createParticipation(
    "participant_cas9_gRNA_complex",
    "cas9_gRNA_complex",
    SystemsBiologyOntology.INHIBITOR
);
target_generic_gene_inhibition.createParticipation(
    "participant_target_gene",
    "target_gene",
    SystemsBiologyOntology.PROMOTER
);
```
Creating CRPb Characterization Circuit

So far, we have completed the repression template. In order to build the CRPb Characterization Circuit, we need to add precise (including Sequences when provided) FunctionalComponents used for each generic FunctionalComponent in the template. We first create Sequence objects for those provided in [1]\(^1\) as shown in the code below. For example, to create the sequence for the CRP\(_b\) promoter, we call the createSequence method, as shown on line 57, with the displayId “CRP\(_b\) seq”, version, the sequence specified by CRP\(_b\) seq \_elements, and the IUPAC encoding for DNA, which is defined as a URI in the Sequence class, referencing http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html.

\(^1\)Unfortunately, as usual, not all sequences are provided in the paper.
// Create sequence for CRa_U6 promoter

String CRa_U6_seq_elements = "GGTTTACCGAGCTCTTATTGGTTTTCAAACTTCATTGACTGTGCC" +
"AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT" +
"TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT" +
"TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA" +
"AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT" +
"AAAATGGACTATCATATGCTTACCGTAACTTGAAATATAGAACCG" +
"ATCCTCCCATTGGTATATATTATAGAACCGATCCTCCCATTGGCT" +
"TGTGGAAAGGACGAAACACCGTACCTCATCAGGAACATGTGTTTA" +
"AGAGCTATGCTGAAACAGCAGAAATAGCAAGTTTAAATAAGGCT" +
"AGTCCGTTATCAACTCGTTAAAAGTGCCAGAGCTGCTTTTTTT" +
"TTGGTGCGTTTTTATGCTTGTAGTATTGTATAATGTTTTT;"

doc.createSequence("CRa_U6_seq", version, CRa_U6_seq_elements, Sequence.IUPAC_DNA);

// Create sequence for gRNA_b coding sequence

String gRNA_b_elements = "AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT" +
"TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT" +
"TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA" +
"AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT" +
"AAAATGGACTATCATATGCTTACCGTAACTTGAAAGTATTTCGAT" +
"TTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACCGTACCT" +
"CATCAGACGATCGATTGAAGCATGCGAAGACAACGAGAAT" +
"ACAGGTTGAAATTAAAGGCGTACTGCTTATCAACTCTGAAAGTAGG" +
"CAGGAAGGTCAGTTTTTATGCTTGTAGTATTGTATAATGTTTTT;"

doc.createSequence("gRNA_b_seq", version, gRNA_b_elements, Sequence.IUPAC_DNA);

// Create sequence for mKate

String mKate_seq_elements = "TCTAAGGGCGAAGAGCTGATTAAGGAGAACATGCACATGAAGCTG" +
"TACATGGAGGGCACCGTGAACAACCACCACTTCAAGTGCACATCC" +
"GAGGGCGAAGGCAAGCCCTACGAGGGCACCCAGACCATGAGAATC" +
"AAGGTGGTCGAGGGCGGCCCTCTCCCCTTCGCCTTCGACATCCTG" +
"GCTACCAGCTTCATGTACGGCAGCAAAACCTTCATCAACCACACC" +
"CAGGGCATCCCCGACTTCTTTAAGCAGTCCTTCCCTGAGGTAAGT" +
"GGTCCTACCTCATCAGGAACATGTGTTTTAGAGCTAGAAATAGCA" +
"AGTTAAAATAAGCAGTCTCCCTATACCTGAAAGATGGCCACC" +
"GATCTCGGTCTCATAAATCCTCAGGTCCTTCTTCTCTTTTCTACAG" +
"GGCTTCACATGGGAGAGAGTCAGGCACCTTCCTTCTTCTTCTTCACCCG" +
"CTGACCGTTCACCCAGAAGAGACCTCAGAGCGGTGCTCTTACT" +
"TACAACGTGAAATGAGGGAATCTCCATCATCGACAGGGCCT" +
"GTGACGAAAGAAACACTGCGGTCGGAGCTCCGACAGATGAGT" +
"CGTCAGCCGCCGAGGACCTCAACTGCTGAGAAATGGGCTGAG" +
"CGTCAGGCTTGGGGGGGCCACCTGACTGCTGAATCTGAAGACCG" +
"ACATTACAGATCAGAAACAGCGCTGGAAGCAGCAGGTGCCCCG" +
"GGCTTCATGTGGGAGAGAGTCAGGCACCTTCCTTCTTCTTCTTCACCCG" +
"AAAGGACACATCTACGCGAGCAGCAGGTTGCGTGCGCCAGATAAC" +
"TGCG;"

doc.createSequence("mKate_seq", version, mKate_seq_elements, Sequence.IUPAC_DNA);

// Create sequence for CRP_b promoter

String CRP_b_seq_elements = "GCTCCGAATTTCTCGACAGATCTCATGTGATTACGCCAAGCTACG" +
"GGCGGAGTACTGTCCTCAGAGCGAGTACTGTCCTCAGAGCGAG" +
"TACTGTCTCCAGAGCGAGTACTGTCCTCAGAGCGAGTACTGTC" +
"TCTCCAGGGCAGACTCCTAGTACTACCTACGAGAAGATTGAAAT" +
"TCTAGGCTCGATGTGGGAGGCCTTATAAGCAGACGGCTGTTTA" +
"GTGACCGTGACAGGACTCTCAGACATCAGGAAAGAGGCAATGCAG" +
"ATCAATCCAGCACAG";

doc.createSequence("CRP_b_seq", version, CRP_b_seq_elements, Sequence.IUPAC_DNA);

Next, we specify ComponentDefinitions for all FunctionalComponents in the CRPb Characterization Circuit. The code snippet below first creates a ComponentDefinition of DNA type for the CRP_b promoter (lines 1-7). Note that by using compliant URIs, the sequence can be looked up using its displayId, i.e. CRP_b_seq that was added previously, and since no version is provided, it is referenced by its persistentIdentity URI (line 7). It is simply its URI without the version when using compliant URIs. The purpose of a persistent identity is to allow an object to refer to the latest version of another object using this URI. The latest version of an object is determined using semantic versioning conventions (c.f., http://semver.org/). Then, we create two ComponentDefinition objects, one for the
EYFP coding sequence (CDS), and one for the EYFP gene (lines 8-17). We use a SequenceConstraint object (lines 18-23) to indicate that the CRP_b promoter precedes the EYFP_cds, because the sequence for the CDS has not been provided and thus cannot be given an exact Range. Note that since the “createDefaults” flag is set to true, FunctionalComponents are automatically created for CRP_b and EYFP_cds ComponentDefinition objects, completing the reference from EYFP_gene_constraint, via the CRP_b FunctionalComponent, to the CRP_b ComponentDefinition, and the reference from EYFP_gene_constraint, via the EYFP_cds FunctionalComponent, to the EYFP_cds ComponentDefinition.

Other ComponentDefinition objects can be created using the same set of method calls. As an exercise, the reader should next create the ComponentDefinitions specified in Table 1 and 2. Please note that the code will not run correctly unless these ComponentDefinitions are created. You can use the same version for all of the ComponentDefinition objects. Entries “type” column in the table are defined as constants in the ComponentDefinition class. Roles are SO terms defined as URI constants in the SequenceOntology class. URIs for these terms are described in Table 3 of the Specification (Data Model 2) document.

```java
ComponentDefinition CRP_b = doc.createComponentDefinition(  "CRP_b",
version,
ComponentDefinition.DNA
);  
CRP_b.addRole(SequenceOntology.PROMOTER);  
CRP_b.addSequence("CRP_b_seq");  
doc.createComponentDefinition(  "EYFP_cds",
version,
ComponentDefinition.DNA
).addRole(SequenceOntology.CDS);  
ComponentDefinition EYFP_gene = doc.createComponentDefinition(  "EYFP_gene",
version,
ComponentDefinition.DNA
);  
EYFP_gene.createSequenceConstraint(  "EYFP_gene_constraint",
RestrictionType.PRECEDES,
"CRP_b",
"EYFP_cds"
);  
```

We are now ready to create all remaining FunctionalComponent objects. We first create a ModuleDefinition object for the CRPb Characterization Circuit as shown below (lines 1-4). Then we create a FunctionalComponent EYFP that has a PRIVATE access type and refers to the EYFP ComponentDefinition object. The private access type means that this FunctionalComponent must not be referred to by remote MapsTo objects. Its direction property is NONE, indicating that it serves neither input or output with regards to the CPRb_circuit. Other FunctionalComponent objects, namely cas9m_BFP, cas9m_BFP_gene, gRNA_b, gRNA_b_gene, mKate, mKate_gene, Gal4VP16, Gal4VP16_gene, EYFP_gene, and cas9m_BFP_gRNA_b, can be created similarly with the same access type, version, and direction type.

Note: * in Table 1 and 2 denote that the code has been provided in this tutorial for those objects. Users will have to create the remaining components mentioned in the tables in order to create a fully functional model.
Table 1: ComponentDefinition objects

<table>
<thead>
<tr>
<th>component definition</th>
<th>type</th>
<th>role</th>
<th>sequence</th>
<th>sequence constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>pConst</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cas9m_BFP_cds</td>
<td>DNA</td>
<td>CDS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cas9m_BFP_gene</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>cas9m_BFP_gene_constraint</td>
</tr>
<tr>
<td>cas9m_BFP</td>
<td>PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>CRa_U6</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>CRa_U6_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b_nc</td>
<td>DNA</td>
<td>CDS</td>
<td>gRNA_b_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b terminator</td>
<td>DNA</td>
<td>TERMINATOR</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b_gene</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>gRNA_b_gene_constraint1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>n/a</td>
<td>gRNA_b_gene_constraint2</td>
</tr>
<tr>
<td>cas9m_BFP_gRNA_b</td>
<td>RNA</td>
<td>SGRNA</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>mKate_cds</td>
<td>DNA</td>
<td>CDS</td>
<td>mKate_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>mKate_gene</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>mKate_gene_constraint</td>
</tr>
<tr>
<td>mKate</td>
<td>PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Gal4VP16_cds</td>
<td>DNA</td>
<td>CDS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Gal4VP16_gene</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>GAL4VP16_gene_constraint</td>
</tr>
<tr>
<td></td>
<td>PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>CRP_b*</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>CRP_b_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>EYFP_cds*</td>
<td>DNA</td>
<td>CDS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>EYFP_gene*</td>
<td>DNA</td>
<td>PROMOTER</td>
<td>n/a</td>
<td>EYFP_gene_constraint</td>
</tr>
<tr>
<td></td>
<td>PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 2: SequenceConstraint objects

<table>
<thead>
<tr>
<th>displayId</th>
<th>restriction type</th>
<th>subject</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>cas9m_BFP_gene_constraint</td>
<td>PRECEDES</td>
<td>pConst</td>
<td>cas9m_BFP_cds</td>
</tr>
<tr>
<td>gRNA_b_gene_constraint1</td>
<td>PRECEDES</td>
<td>CRa_U6</td>
<td>gRNA_b_nc</td>
</tr>
<tr>
<td>gRNA_b_gene_constraint2</td>
<td>PRECEDES</td>
<td>gRNA_b_nc</td>
<td>gRNA_b_terminator</td>
</tr>
<tr>
<td>mKate_gene_constraint</td>
<td>PRECEDES</td>
<td>pConst</td>
<td>mKate_cds</td>
</tr>
<tr>
<td>Gal4VP16_gene_constraint</td>
<td>PRECEDES</td>
<td>pConst</td>
<td>Gal4VP16_cds</td>
</tr>
<tr>
<td>EYFP_gene_constraint*</td>
<td>PRECEDES</td>
<td>pConst</td>
<td>EYFP_cds</td>
</tr>
</tbody>
</table>

1
2 ModuleDefinition CRPb_circuit = doc.createModuleDefinition(
3   "CRPb_characterization_circuit",
4   version
5 );
6 CRPb_circuit.createFunctionalComponent(
7   "EYFP",
8  AccessType.PRIVATE,
9   "EYFP",
10  version,
11  DirectionType.NONE);

Next, we need to specify all interactions for the CRPb Characterization Circuit. Following the same procedure for creating Interactions before, we can create those specified in Table 3. Remember to use the CRPb circuit module definition.

Now, the CRISPR-based Repression Template Module is instantiated and connected to the CRPb Characterization Circuit using MapsTo objects. The code below shows these mappings. For example, a MapsTo object is used to indicate that the target gene in the template should be refined to be the EYFP gene specified in the CRPb circuit (lines 30-35).
<table>
<thead>
<tr>
<th>interaction</th>
<th>type</th>
<th>participant</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>mKate_production</td>
<td>GENETIC_PRODUCTION</td>
<td>mKate_gene</td>
<td>mKate</td>
</tr>
<tr>
<td>Gal4VP16_production</td>
<td>GENETIC_PRODUCTION</td>
<td>Gal4VP16_gene</td>
<td>Gal4VP16</td>
</tr>
<tr>
<td>cas9m_BFP_production</td>
<td>GENETIC_PRODUCTION</td>
<td>cas9m_BFP_gene</td>
<td>cas9m_BFP</td>
</tr>
<tr>
<td>gRNA_b_production</td>
<td>GENETIC_PRODUCTION</td>
<td>gRNA_b_gene</td>
<td>gRNA_b</td>
</tr>
<tr>
<td>EYFP_activation</td>
<td>STIMULATION</td>
<td>EYFP_gene</td>
<td>EYFP</td>
</tr>
<tr>
<td>mKate_deg</td>
<td>DEGRADATION</td>
<td>mKate</td>
<td>REACTANT</td>
</tr>
<tr>
<td>GAL4VP16_deg</td>
<td>DEGRADATION</td>
<td>GAL4VP16</td>
<td>REACTANT</td>
</tr>
<tr>
<td>cas9m_BFP_deg</td>
<td>DEGRADATION</td>
<td>cas9m_BFP</td>
<td>REACTANT</td>
</tr>
<tr>
<td>gRNA_b_deg</td>
<td>DEGRADATION</td>
<td>gRNA_b</td>
<td>REACTANT</td>
</tr>
<tr>
<td>EYFP_deg</td>
<td>DEGRADATION</td>
<td>EYFP</td>
<td>REACTANT</td>
</tr>
<tr>
<td>cas9m_BFP_gRNA_b_deg</td>
<td>DEGRADATION</td>
<td>cas9m_BFP_gRNA_b</td>
<td>cas9m_BFP</td>
</tr>
</tbody>
</table>

Module Template_Module = CRPb_circuit.createModule("CRISPR_Template", version);
Template_Module.createMapsTo("cas9m_BFP_map", RefinementType.USELOCAL, "cas9m_BFP", "cas9_generic");
Template_Module.createMapsTo("gRNA_b_map", RefinementType.USELOCAL, "gRNA_b", "gRNA_generic");
Template_Module.createMapsTo("cas9m_BFP_gRNA_b_map", RefinementType.USELOCAL, "cas9m_BFP_gRNA_b", "cas9_gRNA_complex");
Template_Module.createMapsTo("EYFP_map", RefinementType.USELOCAL, "EYFP", "target");
Template_Module.createMapsTo("EYFP_gene_map", RefinementType.USELOCAL, "EYFP_gene", "target_gene");
At this point, we have completed the CRISPR circuit model. One final step is to serialize the complete model to produce an RDF/XML output. This can be done by adding the code below.

```java
try {
    SBOLWriter.write(doc, System.out);
} catch (XMLStreamException | FactoryConfigurationError | CoreIoException e) {
    e.printStackTrace();
}
```

The complete repression model is described in the “RepressionModel.java” under the libSBOLj examples directory. This example is self-contained in that you can run it to generate the RDF/XML output. Note that SBOL does not provide the specification of a mathematical model directly. It is possible, however, to generate a mathematical model using SBML [3] and the procedure described in [4]. Then, the SBOL document can reference this generated SBML model.

Other Features of libSBOLj

So far, we have demonstrated how one can build the CRISPR-based repression module [1] using libSBOLj. In this section, we present other major methods in the library’s API.

Retrieving an Existing Object

Often, we need getter methods to retrieve a previously created object. You can easily do this by calling a “get” method in the `SBOLDocument` class to get back the `TopLevel` object you are looking for. For example, if we want to get the `cas9_generic` protein `ComponentDefinition` object, we can use the `getComponentDefinition` method (lines 1-4) shown below by providing the display ID and version of the object. If we want to get the latest version of a `TopLevel` object, we can pass `null` to its version field (lines 5-8), and the getter method will retrieve the object with the latest version. This is an implementation of the persistent identity feature. In our case, since we only have one version of the `cas9_generic` object, the two retrieved objects, namely `cas9_generic1` and `cas9_generic2`, are identical, which can checked by calling the `equals` method (lines 9-11). In fact, they refer to the same object, i.e. `cas9_generic`. The library provides `equals` method for every data model class, as well as the `SBOLDocument` class. To retrieve a non-`TopLevel` object, the getter method needs to be called on its immediate parent object. For example, line 12 below retrieves the `gRNA_b_gene_constraint1` from its parent `ComponentDefinition` `gRNA_b_gene`. There is no need to provide the version here, because the child object’s version is always the same as its immediate parent’s, under our compliant URI scheme.

```java
ComponentDefinition cas9_generic1 = doc.getComponentDefinition(
    "cas9_generic",
    version
);
ComponentDefinition cas9_generic2 = doc.getComponentDefinition(
    "cas9_generic",
    null
);
if (cas9_generic1.equals(cas9_generic2)) {
    System.out.println("Two Cas9 generic protein objects are equal.");
}
gRNA_b_gene.getSequenceConstraint("gRNA_b_gene_constraint1");
```

Manipulating Optional Fields

For any optional field that is not a set or list, the library provides methods to set its value, unset its value to `null`, and check its value. The only exceptions where these methods are not available are the following three fields in the
Identified class: persistentIdentity, displayId, and version. These fields cannot be edited, since they are crucial to maintaining compliant SBOL objects (see Section 11.2 “Compliant SBOL Objects” of the Specification (Data Model 2) for more details). The example code below first sets the name of the CRISPR_Template ModuleDefinition with some garbage characters, it then unsets the name and rename it with a clear one. The code then sets the description of this object to the source of its bibliographic information.

```java
CRISPR_Template.setName("C˜R*I!S@P#R-based Repression Template");
if (CRISPR_Template.isSetName()) {
    // always true in this case
    CRISPR_Template.unsetName();
    CRISPR_Template.setName("CRISPR-based Repression Template");
}
CRISPR_Template.setDescription(
    "Authors: S. Kiani, J. Beal, M. Ebrahimkhani, J. Huh, R. Hall, Z. Xie, Y. Li, and R. Weiss," +
    "Title: Crispr transcriptional repression devices and layered circuits in mammalian cells," +
);
```

For an optional field that is either a list or a set, the library provides methods for adding, removing, and checking if an element is contained in the list or set. One example we have seen several times so far is the call to the addRole method. Previously, we added the PROMOTER role, which is defined as http://identifiers.org/so/SO:0000167, to gRNA_b_gene. The code below adds a second role gRNA_b_gene_role2 to it first, it then checks the containment of this role before removing it.

```java
URI gRNA_b_gene_role2 = URI.create("http://identifiers.org/so/SO:0000613");
gRNA_b_gene.addRole(gRNA_b_gene_role2);
if (gRNA_b_gene.containsRole(gRNA_b_gene_role2)) {
    gRNA_b_gene.removeRole(gRNA_b_gene_role2);
}
```

Fields that are a list or set of objects also include operations to clear, get, and set them. The example code below removes all roles at once by calling clearRoles(), gets the set of roles for the gRNA_b_gene, and checks if it is empty. Finally, it sets the entire set of roles (replacing any existing set) by calling setRoles(Set<URI> roles). At this point, gRNA_b_gene should only contain one role, which is PROMOTER.

```java
gRNA_b_gene.clearRoles();
if (!gRNA_b_gene.getRoles().isEmpty()) {
    System.out.println("gRNA_b_gene set is not empty.");
}
gRNA_b_gene.setRoles(new HashSet<URI>(
    Arrays.asList(
        SequenceOntology.PROMOTER))
);
```

Creating and Editing References

TopLevel objects can refer to other TopLevel objects. For example, a ComponentDefinition object can refer to one or more sequences. This reference is created by calling the addSequence(URI) method. Methods available for manipulating references are similar to those for the optional fields. Previously, we added the CRP_b_seq Sequence to the CRP_b ComponentDefinition. The code below first clears all sequences associated with CRP_b, then adds the CRP_b_seq Sequence back. The last three lines will cause an exception to be thrown indicating that the sequence with the specified URI cannot be found in the doc SBOLDocument object. Note that since the complete flag is set to true, the library verifies that all objects referenced are present in it.
Creating Annotations

In order to allow representation of data that cannot currently be represented by the SBOL data model or data that are outside the scope of SBOL, SBOL offers developers the ability to embed custom data as annotations of SBOL objects and as generic top-level objects. These data are exchanged unmodified between software tools that adopt libSBOLj 2.

Each object in SBOL 2 can be annotated by having any number of Annotation objects that store data in the form of name/value property pairs. The name of an annotation must be a QName object, which is composed of a namespace, a local name, and an optional prefix. The value of an annotation must contain a literal (i.e., a String, int, double, boolean), URI, or NestedAnnotations object. The code snippet below creates an annotation for the pConst promoter.

First, a new namespace is added to document. It creates a short name “pr” for the prURI previously defined as the default URI prefix for this SBOL document. This annotation is named “pr:experience”, and is composed of the prURI namespace, a local name “experience” and its prefix “pr”. It contains a URI that can be resolved to the information web page on the Parts Registry for the constitutive promoter “J23119”.

```java
String prURI = "http://partsregistry.org";
String prPrefix = "pr";
doc.addNamespace(URI.create(prURI) , prPrefix);
ComponentDefinition pConst = doc.getComponentDefinition("pConst", version);
pConst.createAnnotation(
    new QName(prURI, "experience", prPrefix),
    URI.create("http://parts.igem.org/Part:BBa_J23119:Experience"));
```

Creating Generic TopLevel Object

To embed custom data directly in an SBOL document, we can store them using GenericTopLevel objects. The example code below first creates such an object datasheet, whose display ID is “datasheet” and version is “1.1”. Its required RDF type property is a QName object named “myersLab:datasheet”. Then we set its name to “Datasheet for Custom Parameters”. Custom data are encoded as annotations of this generic top-level object. The first one is the characterization data with a URI value that can be resolved to a location where measurement data can be found. The next annotation stores the value of the transcription rate, which is 0.75. The last three lines create an annotation for TetR_promoter that refers to the datasheet object.

```java
String prURI = "http://partsregistry.org";
String prPrefix = "pr";
doc.addNamespace(URI.create(prURI) , prPrefix);
ComponentDefinition pConst = doc.getComponentDefinition("pConst", version);
pConst.createAnnotation(
    new QName(prURI, "experience", prPrefix),
    URI.create("http://parts.igem.org/Part:BBa_J23119:Experience"));
```
Creating and Editing Child Objects

Certain classes in the SBOL 2 data model are allowed to own non-`TopLevel` child objects that are also part of the data model. For example, all `SequenceConstraints` we created previously are child objects of their corresponding parent `ComponentDefinitions`, and all `MapsTo` objects we created are child objects of their corresponding parent `Modules`. Note that a child object can not exist on its own, it therefore can only be created from its immediate parent object. The library provides similar methods for retrieving, removing and checking containment of child objects. Adding a child object to a parent object is not directly available, due to our effort in maintaining URI persistance between them. This, however, can be done by calling the create method on its immediate parent, which adds the child object after its creation.

Copying Objects

The library can make copies of `TopLevel` objects using the `createCopy` methods. There are several variations of this method. The `createCopy(TopLevel)` method makes an identical copy of its given `TopLevel` object. Note that this will cause an exception if it is copied into the same `SBOLDocument`, since it will not have a unique identity. Therefore, this method is meant to be used only when one wants to copy an object from one SBOL document to another. The `createCopy(TopLevel, String)` takes a `TopLevel` object to be copied and a new display ID, and it returns an identical copy with the display ID, and its own and its descendant’s URI identities updated accordingly. There is a copy method that also takes a version field, and it returns a copy with the version, and its own and its descendant’s URI identities updated accordingly. Finally, a new URI prefix can also be provided, once again resulting in updated identities. The example below makes a copy of the `pConst` promoter by calling `createCopy` with a new display ID, `pConst_alt`. The identity URI for `pConst_alt` is changed to include the new display ID, and this change also percolates through the identity URIs for any of its descendent objects. This example gives the new copy a sequence, but it keeps all other properties the same.
Validation

Since version 2.1.0 of libSBOLj, validation rules defined in the SBOL Version 2.1.0 specification that are at least partially machine-checkable have been encoded in the library. This enables users of the library to automatically validate an SBOL file on reading and/or during construction of SBOL objects using the library’s API call. The code lines [1-7] below validates the complete models we created in this tutorial. Once the user has created the Repression Model, they can use lines [10-11] to check against a completed version of this tutorial to ensure correctness.

```java
1 SBOLValidate.validateSBOL(doc, true, true, true);
2 if (SBOLValidate.getNumErrors() > 0) {
3     for (String error : SBOLValidate.getErrors()) {
4         System.out.println(error);
5     }
6     return;
7 }
8 // TODO: uncomment to compare with a "golden" version of the RepressionModel.xml
9 SBOLDocument doc2 = SBOLReader.read(<path of RepressionModel.xml>);
10 SBOLValidate.compareDocuments("Mine", doc, "Solution", doc2);
```

The method SBOLValidate.validateSBOL(SBOLDocument sbolDocument, boolean complete, boolean compliant, boolean bestPractice) is called to validate the SBOL document we created in the beginning of this tutorial. Errors encountered during validation either throw exceptions or, if not fatal, are added to the list of errors that can be later accessed using the getErrors() method. Setting the complete flag to true enables the validation routine to check if all references to SBOL objects in the given document can dereference to objects that exist in the same document. A true value for the compliant flag means that all SBOL objects’ identity URIs in the given SBOL document are compliant. Lastly, setting bestPractice to true enables the validator to check rules with the RECOMMENDED condition in the SBOL Version 2.1.0 specification.

Serialization

The library supports reading and writing data encoded in RDF/XML format. We can produce a serialization output by calling various write methods in the SBOLWriter class. These methods write to either an output stream in the form of a Java OutputStream object or a file, some of which are demonstrated below. The first method call produces an output stream and the second stores the output to file “RepressionModel.rdf”. Note that these two method calls do not specify the output file type, and the library’s default serialization output format is RDF/XML.

```java
1 SBOLWriter.write(doc,(System.out));
2 SBOLWriter.write(doc, "RepressionModel.rdf");
```

Reading of a file or an input stream in the form of a Java InputStream object is supported by similar read methods in the SBOLReader class. The default input format for any of these read methods is also RDF/XML. The method below first writes the SBOLDocument “doc” to a Java ByteArrayOutputStream, and then reads it back to a new SBOLDocument object.

```java
public static SBOLDocument writeThenRead(SBOLDocument doc)
1    throws SBOLValidationException, IOException,
2        FactoryConfigurationError
3    {
4        ByteArrayOutputStream out = new ByteArrayOutputStream();
5        SBOLWriter.write(doc, out);
6        return SBOLReader.read(new ByteArrayInputStream(out.toByteArray()));
7    }
```

The complete repression model described in this tutorial is provided as “RepressionModel.java” under the libSBOLj examples directory. This example is self-contained in that you can run it to generate the RDF/XML output. Note that SBOL does not provide the specification of a mathematical model directly. It is possible, however, to generate a math-
ematical model using SBML [3] and the procedure described in [4]. Then, the SBOL document can reference this generated SBML model.

References


