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.

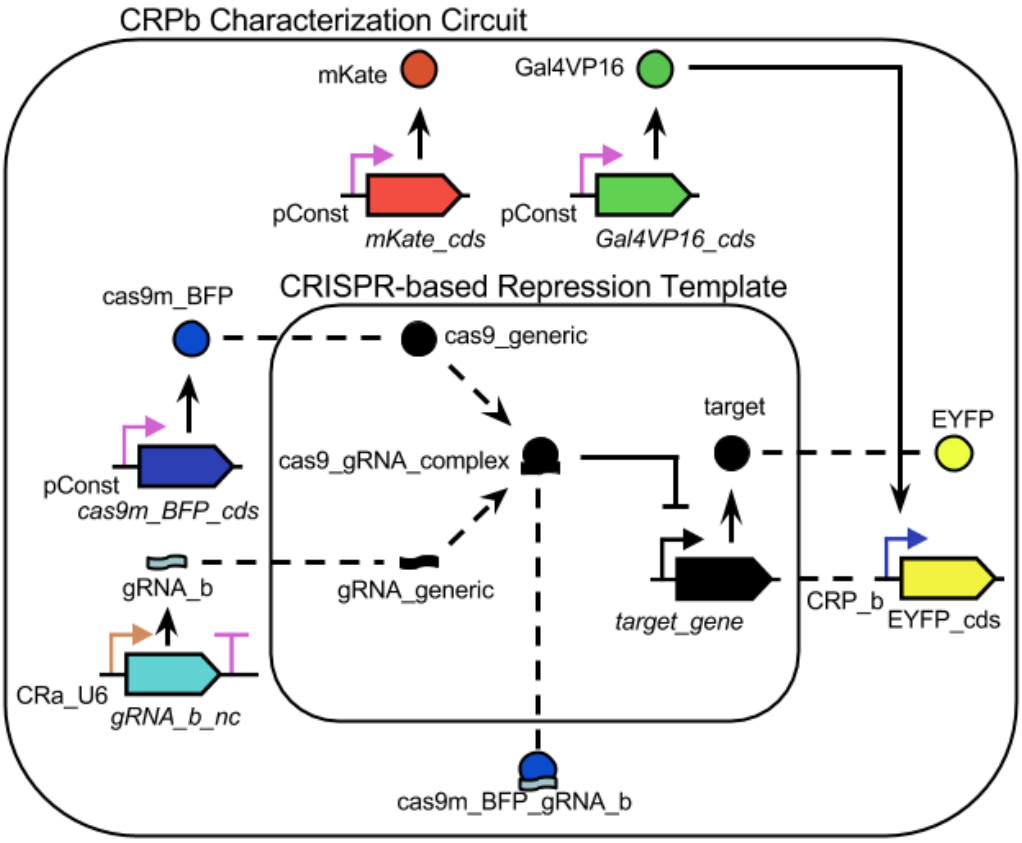
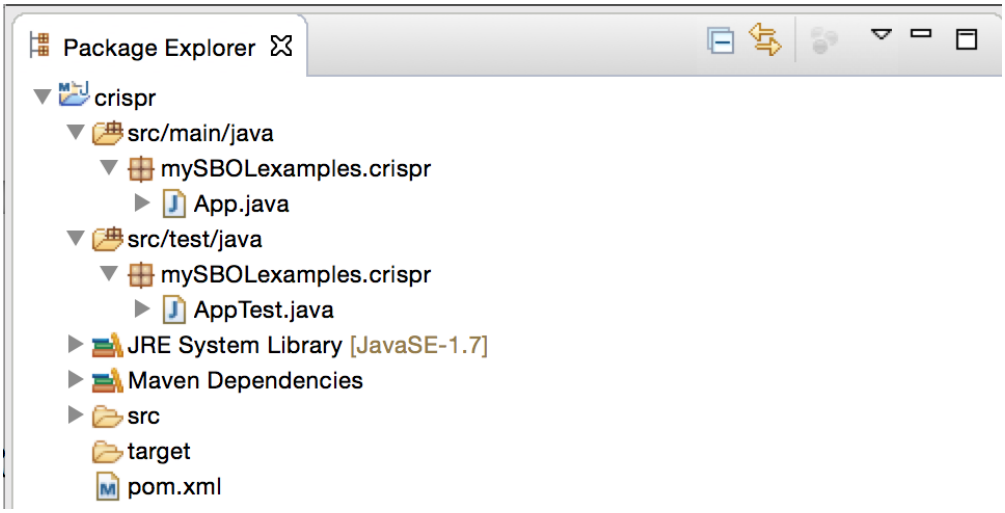


Figure 1: Illustration of a hierarchical CRISPR-based repression module represented in SBOL 2 (adapted from Figure 1a in [1]). The CRISPR-based Repression Template **ModuleDefinition** describes a generic CRISPR repression circuit that combines a Cas9 protein with a gRNA to form a complex (represented by the dashed arrows) that represses a target gene (represented by the arrow with the tee arrowhead). These relationships between these **FunctionalComponents** (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.

```
1  import java.net.URISyntaxException;
2  import javax.xml.stream.FactoryConfigurationError;
3  import javax.xml.stream.XMLStreamException;
4  import org.sbolstandard.core2.AccessType;
5  import org.sbolstandard.core2.ComponentDefinition;
6  import org.sbolstandard.core2.DirectionType;
7  import org.sbolstandard.core2.Interaction;
8  import org.sbolstandard.core2.Module;
9  import org.sbolstandard.core2.ModuleDefinition;
10 import org.sbolstandard.core2.RefinementType;
11 import org.sbolstandard.core2.RestrictionType;
12 import org.sbolstandard.core2.SBOLDocument;
13 import org.sbolstandard.core2.SBOLWriter;
14 import org.sbolstandard.core2.Sequence;
15 import org.sbolstandard.core2.SequenceOntology;
16 import org.sbolstandard.core2.SystemsBiologyOntology;
17 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.

```
1 public class RepressionModel {
2   public static void main(String[] args) throws URISyntaxException,
        SBOLValidationException, SBOLConversionException {
3     SBOLDocument doc = new SBOLDocument();
4     doc.setDefaultURIprefix("http://sbols.org/CRISPR_Example/");
5     doc.setComplete(true);
6     doc.setCreateDefaults(true);
7   }
8 }
```

- `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 **SBOLDocuemnt** 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`.

```
String version = "1.0";
doc.createComponentDefinition(
          "cas9_generic",
          version,
          ComponentDefinition.PROTEIN
          );
doc.createComponentDefinition(
         "gRNA_generic",
         version,
         ComponentDefinition.RNA
         ).addRole(SequenceOntology.SGRNA);
doc.createComponentDefinition(
         "cas9_gRNA_complex",
         version,
         ComponentDefinition.COMPLEX
         );
doc.createComponentDefinition(
         "target_gene",
         version,
         ComponentDefinition.DNA
         ).addRole(SequenceOntology.PROMOTER);
doc.createComponentDefinition(
         "target",
         version,
         ComponentDefinition.PROTEIN);
ModuleDefinition  CRISPR_Template = doc.createModuleDefinition(
         "CRISPR_Template",
         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_Example`). 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 **SBOLDocuemnt** 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.

```
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.

```
// 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" +
                "AGAGCTATGCTGGAAACAGCAGAAATAGCAAGTTTAAATAAGGCT" +
                "AGTCCGTTATCAACTTGAAAAAGTGGCACCGAGTCGGTGCTTTTT" +
                "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" +
                "CATCAGGAACATGTGTTTAAGAGCTATGCTGGAAACAGCAGAAAT" +
                "AGCAAGTTTAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGG" +
                "CACCGAGTCGGTGCTTTTTTT";
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" +
                "AGTTAAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGGCACC" +
                "GAGTCGGTGCTACTAACTCTCGAGTCTTCTTTTTTTTTTCACAG" +
                "GGCTTCACATGGGAGAGAGTCACCACATACGAAGACGGGGGCGTG" +
                "CTGACCGCTACCCAGGACACCAGCCTCCAGGACGGCTGCCTCATC" +
                "TACAACGTCAAGATCAGAGGGGTGAACTTCCCATCCAACGGCCCT" +
                "GTGATGCAGAAGAAAACACTCGGCTGGGAGGCCTCCACCGAGATG" +
                "CTGTACCCCGCTGACGGCGGCCTGGAAGGCAGAAGCGACATGGCC" +
                "CTGAAGCTCGTGGGCGGGGGCCACCTGATCTGCAACTTGAAGACC" +
                "ACATACAGATCCAAGAAACCCGCTAAGAACCTCAAGATGCCCGGC" +
                "GTCTACTATGTGGACAGAAGACTGGAAAGAATCAAGGAGGCCGAC" +
                "AAAGAGACCTACGTCGAGCAGCACGAGGTGGCTGTGGCCAGATAC" +
                "TGCG";
doc.createSequence("mKate_seq", version, mKate_seq_elements, Sequence.IUPAC_DNA);

// Create Sequence for CRP_b promoter
String CRP_b_seq_elements =  "GCTCCGAATTTCTCGACAGATCTCATGTGATTACGCCAAGCTACG" +
                "GGCGGAGTACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAG" +
                "TACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAGTTCTGTC" +
                "CTCCGAGCGGAGACTCTAGATACCTCATCAGGAACATGTTGGAAT" +
                "TCTAGGCGTGTACGGTGGGAGGCCTATATAAGCAGAGCTCGTTTA" +
                "GTGAACCGTCAGATCGCCTCGAGTACCTCATCAGGAACATGTTGG" +
                "ATCCAATTCGACC";
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 **ComonentDefinition** 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.

```
1  ComponentDefinition CRP_b = doc.createComponentDefinition(
2          "CRP_b",
3          version,
4          ComponentDefinition.DNA
5          );
6  CRP_b.addRole(SequenceOntology.PROMOTER);
7  CRP_b.addSequence("CRP_b_seq");
8  doc.createComponentDefinition(
9          "EYFP_cds",
10         version,
11         ComponentDefinition.DNA
12         ).addRole(SequenceOntology.CDS);
13 ComponentDefinition EYFP_gene = doc.createComponentDefinition(
14         "EYFP_gene",
15         version,
16         ComponentDefinition.DNA
17         );
18 EYFP_gene.createSequenceConstraint(
19         "EYFP_gene_constraint",
20         RestrictionType.PRECEDES,
21         "CRP_b",
22         "EYFP_cds"
23         );
```

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

component definition	type	role	sequence	sequence constraint
pConst	DNA	PROMOTER	n/a	n/a
cas9m_BFP_cds	DNA	CDS	n/a	n/a
cas9m_BFP_gene	DNA	PROMOTER	n/a	cas9m_BFP_gene_constraint
cas9m_BFP	PROTEIN	n/a	n/a	n/a
CRa_U6	DNA	PROMOTER	CRa_U6_seq	n/a
gRNA_b_nc	DNA	CDS	gRNA_b_seq	n/a
gRNA_b_terminator	DNA	TERMINATOR	n/a	n/a
gRNA_b_gene	DNA	PROMOTER	n/a	gRNA_b_gene_constraint1 gRNA_b_gene_constraint2
gRNA_b	RNA	SGRNA	n/a	n/a
cas9m_BFP_gRNA_b	COMPLEX	n/a	n/a	n/a
mKate_cds	DNA	CDS	mKate_seq	n/a
mKate_gene	DNA	PROMOTER	n/a	mKate_gene_constraint
mKate	PROTEIN	n/a	n/a	n/a
Gal4VP16_cds	DNA	CDS	n/a	n/a
Gal4VP16_gene	DNA	PROMOTER	n/a	GAL4VP16_gene_constraint
Gal4VP16	PROTEIN	n/a	n/a	n/a
CRP_b*	DNA	PROMOTER	CRP_b_seq	n/a
EYFP_cds*	DNA	CDS	n/a	n/a
EYFP_gene*	DNA	PROMOTER	n/a	EYFP_gene_constraint
EYFP	PROTEIN	n/a	n/a	n/a

Table 2: **SequenceConstraint** objects

displayId	restriction type	subject	object
cas9m_BFP_gene_constraint	PRECEDES	pConst	cas9m_BFP_cds
gRNA_b_gene_constraint1	PRECEDES	CRa_U6	gRNA_b_nc
gRNA_b_gene_constraint2	PRECEDES	gRNA_b_nc	gRNA_b_terminator
mKate_gene_constraint	PRECEDES	pConst	mKate_cds
Gal4VP16_gene_constraint	PRECEDES	pConst	Gal4VP16_cds
EYFP_gene_constraint*	PRECEDES	CRP_b	EYFP_cds

```
1  ModuleDefinition CRPb_circuit = doc.createModuleDefinition(
2          "CRPb_characterization_circuit",
3          version
4          );
5  CRPb_circuit.createFunctionalComponent(
6          "EYFP",
7          AccessType.PRIVATE,
8          "EYFP",
9          version,
10         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 3: **Interaction** objects

interaction	type	participant	role
mKate_production	GENETIC_PRODUCTION	mKate_gene mKate	PROMOTER PRODUCT
Gal4VP16_production	GENETIC_PRODUCTION	Gal4VP16_gene Gal4VP16	PROMOTER PRODUCT
cas9m_BFP_production	GENETIC_PRODUCTION	cas9m_BFP_gene cas9m_BFP	PROMOTER PRODUCT
gRNA_b_production	GENETIC_PRODUCTION	gRNA_b_gene gRNA_b	PROMOTER PRODUCT
EYFP_activation	STIMULATION	EYFP_gene Gal4VP16	PROMOTER STIMULATOR
mKate_deg	DEGRADATION	mKate	REACTANT
GAL4VP16_deg	DEGRADATION	GAL4VP16	REACTANT
cas9m_BFP_deg	DEGRADATION	cas9m_BFP	REACTANT
gRNA_b_deg	DEGRADATION	gRNA_b	REACTANT
EYFP_deg	DEGRADATION	EYFP	REACTANT
cas9m_BFP_gRNA_b_deg	DEGRADATION	cas9m_BFP_gRNA_b	REACTANT

```
Module Template_Module = CRPb_circuit.createModule(
        "CRISPR_Template",
        "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_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.

```
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 `libSBOj`

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.

```
1  ComponentDefinition cas9_generic1 = doc.getComponentDefinition(
2          "cas9_generic",
3          version
4          );
5  ComponentDefinition cas9_generic2 = doc.getComponentDefinition(
6          "cas9_generic",
7          null
8          );
9  if (cas9_generic1.equals(cas9_generic2)) {
10     System.out.println("Two Cas9 generic protein objects are equal.");
11 }
12 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.

```
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," +
        "Journal: Nature Methods, vol. 11, no. 7, pp. 723  7 2 6 , 2014."
        );
```

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.

```
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`.

```
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.

```
CRP_b.clearSequences();
CRP_b.addSequence("CRP_b_seq");
CRP_b.addSequence(
  URI.create("http://partsregistry.org/seq/partseq_154")
  );
```

Creating Annotations

In order to allow representation of data that can not 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".

```
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.

```
String myersLabURI = "http://www.async.ece.utah.edu";
String myersLabPrefix = "myersLab";
GenericTopLevel datasheet=doc.createGenericTopLevel(
      "datasheet",
      "1.1",
      new QName(myersLabURI, "datasheet", myersLabPrefix));
datasheet.setName("Datasheet for Custom Parameters");
datasheet.createAnnotation(
        new QName(myersLabURI, "characterizationData", myersLabPrefix),
        URI.create(myersLabURI + "/measurement/Part:BBa_J23119:Experience"));
datasheet.createAnnotation(
        new QName(myersLabURI, "transcriptionRate", myersLabPrefix),
        0.75);
pConst.createAnnotation(
        new QName(myersLabURI, "datasheet", myersLabPrefix),
        datasheet.getIdentity());
```

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.

```
ComponentDefinition pConst_alt = (ComponentDefinition) doc.createCopy(pConst, "
    pConst_alt");
Sequence pConst_alt_seq = doc.createSequence(
        "pConst_alt_seq",
        version,
  "ttgacggctagctcagtcctaggtacagtgctagc",
  Sequence.IUPAC_DNA);
pConst_alt.addSequence(pConst_alt_seq);
```

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-chackable 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 modesl 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.

```
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
9  // TODO: uncomment to compare with a "golden" version of the RepressionModel.xml
10   SBOLDocument doc2 = SBOLReader.read(<path of RepressionModel.xml>);
11   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 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.

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

Reading of a file or an input stream in the form of 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.

```
1  public static SBOLDocument writeThenRead(SBOLDocument doc)
2                  throws SBOLValidationException, IOException,
                       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

[1] S. Kiani, J. Beal, M. Ebrahimkhani, J. Huh, R. Hall, Z. Xie, Y. Li, and R. Weiss, "Crispr transcriptional repression devices and layered circuits in mammalian cells," *Nature Methods*, vol. 11, no. 7, pp. 723–726, 2014.

[2] M. Courtot, N. Juty, C. Knüpfer, D. Waltemath, A. Zhukova, A. Dräger, M. Dumontier, A. Finney, M. Golebiewski, J. Hastings, S. Hoops, S. Keating, D. Kell, S. Kerrien, J. Lawson, A. Lister, J. Lu, R. Machne, P. Mendes, M. Pocock, N. Rodriguez, A. Villeger, D. Wilkinson, S. Wimalaratne, C. Laibe, M. Hucka, and N. Le Novère, "Controlled vocabularies and semantics in systems biology," *Molecular Systems Biology*, vol. 7, 2011.

[3] M. Hucka, A. Finney, H. Sauro, H. Bolouri, J. Doyle, H. Kitano, A. Arkin, B. Bornstein, D. Bray, A. Cornish-Bowden, *et al.*, "The systems biology markup language (sbml): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[4] N. Roehner, Z. Zhang, T. Nguyen, and C. Myers, "Generating systems biology markup language models from the synthetic biology open language," *ACS Synthetic Biology*, vol. 4, no. 8, pp. 873–879, 2015.