A brief description of the CRISPR circuit using SBOL 2.0 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 2.0. Detailed description of properties of the data model is available in the Specification (Data Model 2.0).

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.

Modeling CRISPR repression using `pySBOL`

Creating SBOL Document

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. `pySBOL` organizes the URI collections to enable efficient access

```
1 from sbol import *
2
3 setHomespace("http://sbols.org/CRISPR_Example")
4 toggleSBOLCompliantTypes()
5 version = "1.0.0"
6 doc = Document()
```

The method `setHomespace` 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. The author of any SBOL object should use a URI prefix that either they own or an organization of which they are a member owns. Setting a default namespace is like a signature verifying ownership of objects.

Adding CRISPR-based Repression Template module

Creating TopLevel objects

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. With **FunctionalComponent**, optional fields such as *direction* are used to specify the input, output, both, or neither with regards to the **ModuleDefinition** that contains it. Creation of the generic Cas9 (line 2) **ComponentDefinition** is done by passing its *displayId* "cas9_generic", *version* specified by the `version` string, and *type* to the `ComponentDefinition` constructor. 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

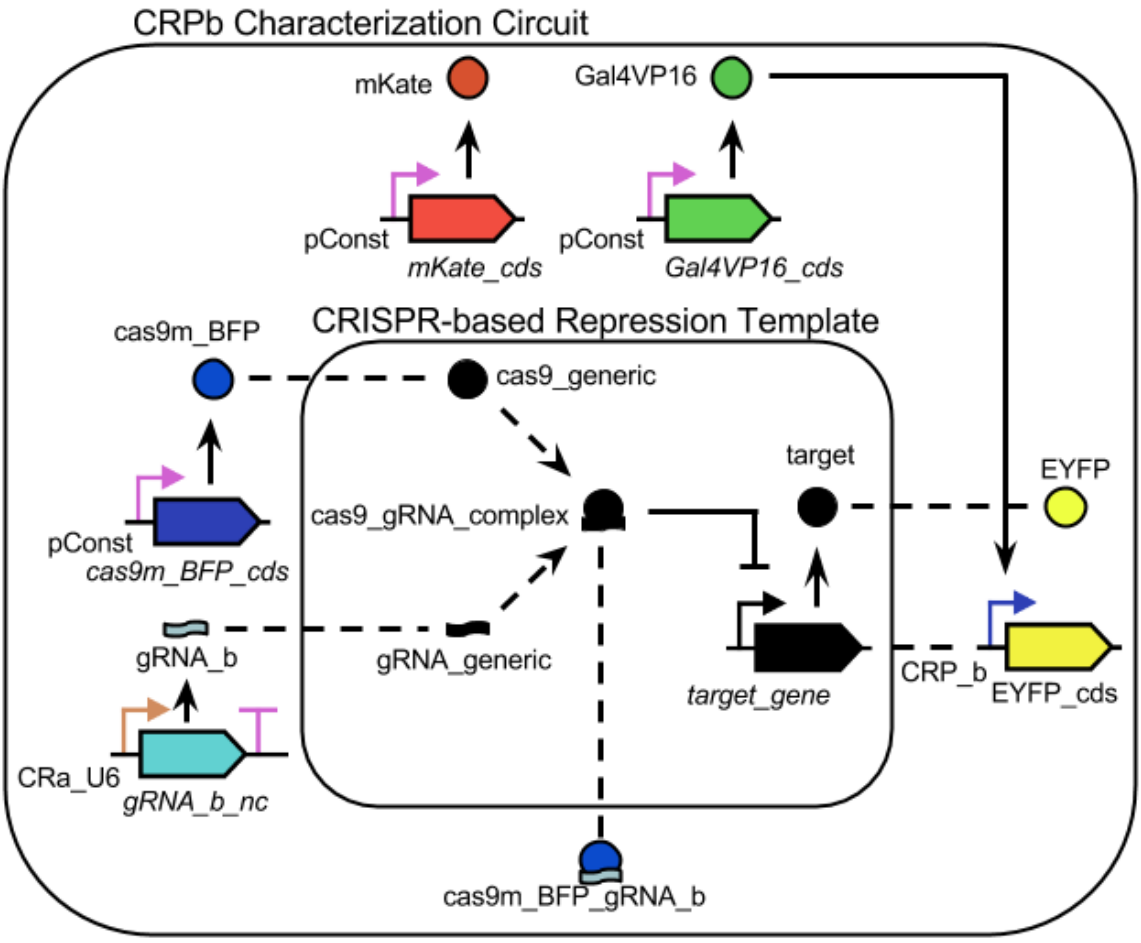

Figure 1: Illustration of a hierarchical CRISPR-based repression module represented in SBOL 2.0 (adapted from Figure 1a in [1]). The CRISPR-based Repression Template **ModuleDefinition** describes a generic CRISPR repression circuit that combines a Cas9 protein with a gRNA to form a complex (represented 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 2.0 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.

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). Finally, an optional *version* specified by the `version` string may be specied. If *version* is not specified, it will be set by default to 1.0.0 .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 7 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 16 below has a role of `PROMOTER`, defined as SO term "SO:0000167" (http://identifiers.org/so/SO:0000167). We then create the **ModuleDefinition** template with the displayId "CRISPR_Template" and `version`.

```
1  #Create ComponentDefinition for cas9_generic protein
2  cas9_generic = ComponentDefinition("cas9_generic", BIOPAX_PROTEIN, version)
3  doc.addComponentDefinition(cas9_generic)
4
5  #Create ComponentDefinition for gRNA_generic RNA
6  gRNA_generic = ComponentDefinition("gRNA_generic", BIOPAX_RNA, version)
7  gRNA_generic.roles.set(SO_SGRNA)
8  doc.addComponentDefinition(gRNA_generic)
9
10 #Create ComponentDefinition for cas9_gRNA_complex
11 cas9_gRNA_complex = ComponentDefinition("cas9_gRNA_complex", BIOPAX_COMPLEX,
       version)
12 doc.addComponentDefinition(cas9_gRNA_complex)
13
14 #Create ComponentDefinition for target gene
15 target_gene = ComponentDefinition("target_gene", BIOPAX_DNA, version)
16 target_gene.roles.set(SO_PROMOTER)
17 doc.addComponentDefinition(target_gene)
18
19 #Create ComponentDefinition for target protein
20 target = ComponentDefinition("target", BIOPAX_PROTEIN, version)
21 doc.addComponentDefinition(target)
22
23 #Create ModuleDefinition for CRISPR_Repression_Template
24 CRISPR_Template = ModuleDefinition("CRISPR_Template", version)
25 doc.addModuleDefinition(CRISPR_Template)
```

By default, pySBOL operates in SBOL-compliant mode. In SBOL-compliant mode, each constructor creates an SBOL-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 2.0) 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 `cas9_generic_participation` **Participation** object refers to the `cas9_generic_fc` **FunctionalComponent** object. Note that even if we do not specify such object, 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" will be automatically created and it connects the reference from the `cas9_generic_participation` **Participation** to the `cas9_generic` **ComponentDefinition**. We also assign a `REACTANT` role to it.

```
1  Cas9Complex_Formation = CRISPR_Template.interactions.create("
       cas9_complex_formation")
2  Cas9Complex_Formation.types.set(SBO_NONCOVALENT_BINDING)
3
4  cas9_generic_fc = CRISPR_Template.functionalComponents.create("cas9_generic")
5  cas9_generic_fc.definition.set(cas9_generic.persistentIdentity.get())
6  cas9_generic_fc.access.set(SBOL_ACCESS_PUBLIC)
7  cas9_generic_fc.direction.set(SBOL_DIRECTION_IN_OUT)
8  cas9_generic_fc.version.set(version)
9
10 cas9_generic_participation = Cas9Complex_Formation.participations.create("
       cas9_generic")
11 cas9_generic_participation.roles.set(SBO_REACTANT)
12 cas9_generic_participation.participant.set(cas9_generic_fc.identity.get())
13
14 gRNA_generic_fc = CRISPR_Template.functionalComponents.create("gRNA_generic")
15 gRNA_generic_fc.definition.set(gRNA_generic.persistentIdentity.get())
16 gRNA_generic_fc.access.set(SBOL_ACCESS_PUBLIC)
17 gRNA_generic_fc.direction.set(SBOL_DIRECTION_IN_OUT)
18 gRNA_generic_fc.version.set(version)
19
20 gRNA_generic_participation = Cas9Complex_Formation.participations.create("
       gRNA_generic")
21 gRNA_generic_participation.roles.set(SBO_REACTANT)
22 gRNA_generic_participation.participant.set(gRNA_generic_fc.identity.get())
23
24 cas9_gRNA_complex_fc = CRISPR_Template.functionalComponents.create("
       cas9_gRNA_complex")
25 cas9_gRNA_complex_fc.definition.set(cas9_gRNA_complex.persistentIdentity.get())
26 cas9_gRNA_complex_fc.access.set(SBOL_ACCESS_PUBLIC)
27 cas9_gRNA_complex_fc.direction.set(SBOL_DIRECTION_IN_OUT)
28 cas9_gRNA_complex_fc.version.set(version)
29
30 cas9_gRNA_complex_participation = Cas9Complex_Formation.participations.create("
       cas9_gRNA_complex")
31 cas9_gRNA_complex_participation.roles.set(SBO_PRODUCT)
32 cas9_gRNA_complex_participation.participant.set(cas9_gRNA_complex_fc.identity.get
       ())
```

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
EYFP_production = CRISPR_Template.interactions.create("target_production")
EYFP_production.types.set(SBO_GENETIC_PRODUCTION)

target_gene_fc = CRISPR_Template.functionalComponents.create("target_gene")
target_gene_fc.definition.set(target_gene.persistentIdentity.get())
target_gene_fc.access.set(SBOL_ACCESS_PUBLIC)
target_gene_fc.direction.set(SBOL_DIRECTION_IN_OUT)
target_gene_fc.version.set(version)

target_gene_participation = EYFP_production.participations.create("target_gene")
target_gene_participation.roles.set(SBO_PROMOTER)
target_gene_participation.participant.set(target_gene_fc.identity.get())

target_fc = CRISPR_Template.functionalComponents.create("target")
target_fc.definition.set(target.persistentIdentity.get())
target_fc.access.set(SBOL_ACCESS_PUBLIC)
target_fc.direction.set(SBOL_DIRECTION_IN_OUT)
target_fc.version.set(version)

target_participation = EYFP_production.participations.create("target")
target_participation.roles.set(SBO_PRODUCT)
target_participation.participant.set(target_fc.identity.get())

#Inhibition of target by cas9m_BFP_gRNA
target_generic_gene_inhibition = CRISPR_Template.interactions.create("
    target_gene_inhibition")
target_generic_gene_inhibition.types.set(SBO_INHIBITION)

cas9_gRNA_complex_participation1 = target_generic_gene_inhibition.participations.
    create("cas9_gRNA_complex")
cas9_gRNA_complex_participation1.roles.set(SBO_INHIBITOR)
cas9_gRNA_complex_participation1.participant.set(cas9_gRNA_complex_fc.identity.get
    ())

target_gene_participation2 = target_generic_gene_inhibition.participations.create(
    "target_gene")
target_gene_participation2.roles.set(SBO_PROMOTER)
target_gene_participation2.participant.set(target_gene_fc.identity.get())
```

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 53, 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
CRa_U6_seq_elements = ("GGTTTACCGAGCTCTTATTGGTTTTCAAACTTCATTGACTGTGCC"
                       "AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT"
                       "TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT"
                       "TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA"
                       "AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT"
                       "AAAATGGACTATCATATGCTTACCGTAACTTGAAATATAGAACCG"
                       "ATCCTCCCATTGGTATATATTATAGAACCGATCCTCCCATTGGCT"
                       "TGTGGAAAGGACGAAACACCGTACCTCATCAGGAACATGTGTTTA"
                       "AGAGCTATGCTGGAAACAGCAGAAATAGCAAGTTTAAATAAGGCT"
                       "AGTCCGTTATCAACTTGAAAAAGTGGCACCGAGTCGGTGCTTTTT"
                       "TTGGTGCGTTTTTATGCTTGTAGTATTGTATAATGTTTTT")
CRa_U6_seq = Sequence("CRa_U6_seq", CRa_U6_seq_elements, SBOL_ENCODING_IUPAC, version)
doc.addSequence(CRa_U6_seq)

#Create Sequence for gRNA_b coding sequence
gRNA_b_elements = ("AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT"
                   "TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT"
                   "TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA"
                   "AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT"
                   "AAAATGGACTATCATATGCTTACCGTAACTTGAAAGTATTTCGAT"
                   "TTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACCGTACCT"
                   "CATCAGGAACATGTGTTTAAGAGCTATGCTGGAAACAGCAGAAAT"
                   "AGCAAGTTTAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGG"
                   "CACCGAGTCGGTGCTTTTTTT")
gRNA_b_seq = Sequence("gRNA_b_seq", gRNA_b_elements, SBOL_ENCODING_IUPAC, version)
doc.addSequence(gRNA_b_seq)

#Create Sequence for mKate
mKate_seq_elements = ("TCTAAGGGCGAAGAGCTGATTAAGGAGAACATGCACATGAAGCTG"
                      "TACATGGAGGGCACCGTGAACAACCACCACTTCAAGTGCACATCC"
                      "GAGGGCGAAGGCAAGCCCTACGAGGGCACCCAGACCATGAGAATC"
                      "AAGGTGGTCGAGGGCGGCCCTCTCCCCTTCGCCTTCGACATCCTG"
                      "GCTACCAGCTTCATGTACGGCAGCAAAACCTTCATCAACCACACC"
                      "CAGGGCATCCCCGACTTCTTTAAGCAGTCCTTCCCTGAGGTAAGT"
                      "GGTCCTACCTCATCAGGAACATGTGTTTTAGAGCTAGAAATAGCA"
                      "AGTTAAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGGCACC"
                      "GAGTCGGTGCTACTAACTCTCGAGTCTTCTTTTTTTTTTTCACAG"
                      "GGCTTCACATGGGAGAGAGTCACCACATACGAAGACGGGGGCGTG"
                      "CTGACCGCTACCCAGGACACCAGCCTCCAGGACGGCTGCCTCATC"
                      "TACAACGTCAAGATCAGAGGGGTGAACTTCCCATCCAACGGCCCT"
                      "GTGATGCAGAAGAAAACACTCGGCTGGGAGGCCTCCACCGAGATG"
                      "CTGTACCCCGCTGACGGCGGCCTGGAAGGCAGAAGCGACATGGCC"
                      "CTGAAGCTCGTGGGCGGGGGCCACCTGATCTGCAACTTGAAGACC"
                      "ACATACAGATCCAAGAAACCCGCTAAGAACCTCAAGATGCCCGGC"
                      "GTCTACTATGTGGACAGAAGACTGGAAAGAATCAAGGAGGCCGAC"
                      "AAAGAGACCTACGTCGAGCAGCACGAGGTGGCTGTGGCCAGATAC"
                      "TGCG")
mKate_seq = Sequence("mKate_seq", mKate_seq_elements, SBOL_ENCODING_IUPAC, version)
doc.addSequence(mKate_seq)

#Create Sequence for CRP_b promoter
CRP_b_seq_elements = ("GCTCCGAATTTCTCGACAGATCTCATGTGATTACGCCAAGCTACG"
                      "GGCGGAGTACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAG"
                      "TACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAGTTCTGTC"
                      "CTCCGAGCGGAGACTCTAGATACCTCATCAGGAACATGTTGGAAT"
                      "TCTAGGCGTGTACGGTGGGAGGCCTATATAAGCAGAGCTCGTTTA"
                      "GTGAACCGTCAGATCGCCTCGAGTACCTCATCAGGAACATGTTGG"
                      "ATCCAATTCGACC")
CRP_b_seq = Sequence("CRP_b_seq", CRP_b_seq_elements, SBOL_ENCODING_IUPAC, version)
doc.addSequence(CRP_b_seq)
```

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 2-4). 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

Table 1: **ComponentDefinition** objects

component definition	type	role	sequence	sequence constraint
pConst	BIOPAX_DNA	SO_PROMOTER	n/a	n/a
cas9m_BFP_cds	BIOPAX_DNA	SO_CDS	n/a	n/a
cas9m_BFP_gene	BIOPAX_DNA	SO_PROMOTER	n/a	cas9m_BFP_gene_constraint
cas9m_BFP	BIOPAX_PROTEIN	n/a	n/a	n/a
CRa_U6	BIOPAX_DNA	SO_PROMOTER	CRa_U6_seq	n/a
gRNA_b_nc	BIOPAX_DNA	SO_CDS	gRNA_b_seq	n/a
gRNA_b_terminator	BIOPAX_DNA	SO_TERMINATOR	n/a	n/a
gRNA_b_gene	BIOPAX_DNA	SO_PROMOTER	n/a	gRNA_b_gene_constraint1 gRNA_b_gene_constraint2
gRNA_b	BIOPAX_RNA	SGRNA	n/a	n/a
cas9m_BFP_gRNA_b	BIOPAX_COMPLEX	n/a	n/a	n/a
mKate_cds	BIOPAX_DNA	SO_CDS	mKate_seq	n/a
mKate_gene	BIOPAX_DNA	SO_PROMOTER	n/a	mKate_gene_constraint
mKate	BIOPAX_PROTEIN	n/a	n/a	n/a
Gal4VP16_cds	BIOPAX_DNA	SO_CDS	n/a	n/a
Gal4VP16_gene	BIOPAX_DNA	SO_PROMOTER	n/a	GAL4VP16_gene_constraint
Gal4VP16	BIOPAX_PROTEIN	n/a	n/a	n/a
CRP_b	BIOPAX_DNA	SO_PROMOTER	CRP_b_seq	n/a
EYFP_cds	BIOPAX_DNA	SO_CDS	n/a	n/a
EYFP_gene	BIOPAX_DNA	SO_PROMOTER	n/a	EYFP_gene_constraint
EYFP	BIOPAX_PROTEIN	n/a	n/a	n/a

Table 2: **SequenceConstraint** objects

displayId	restriction type	subject	object
cas9m_BFP_gene_constraint	SBOL_RESTRICTION_PRECEDES	pConst	cas9m_BFP_cds
gRNA_b_gene_constraint1	SBOL_RESTRICTION_PRECEDES	CRa_U6	gRNA_b_nc
gRNA_b_gene_constraint2	SBOL_RESTRICTION_PRECEDES	gRNA_b_nc	gRNA_b_terminator
mKate_gene_constraint	SBOL_RESTRICTION_PRECEDES	pConst	mKate_cds
GAL4VP16_gene_constraint	SBOL_RESTRICTION_PRECEDES	pConst	Gal4VP16_cds
EYFP_gene_constraint	SBOL_RESTRICTION_PRECEDES	CRP_b	EYFP_cds

Table 3: **Interaction** objects

interaction	type	participant	role
mKate_production	SBO_GENETIC_PRODUCTION	mKate_gene mKate	SBO_PROMOTER SBO_PRODUCT
Gal4VP16_production	SBO_GENETIC_PRODUCTION	Gal4VP16_gene Gal4VP16	SBO_PROMOTER SBO_PRODUCT
cas9m_BFP_production	SBO_GENETIC_PRODUCTION	cas9m_BFP_gene cas9m_BFP	SBO_PROMOTER SBO_PRODUCT
gRNA_b_production	SBO_GENETIC_PRODUCTION	gRNA_b_gene gRNA_b	SBO_PROMOTER SBO_PRODUCT
EYFP_Activation	SBO_STIMULATION	EYFP_gene Gal4VP16	SBO_PROMOTER SBO_STIMULATOR
mKate_deg	SBO_DEGRADATION	mKate	SBO_REACTANT
Gal4VP16_deg	SBO_DEGRADATION	Gal4VP16	SBO_REACTANT
cas9m_BFP_deg	SBO_DEGRADATION	cas9m_BFP	SBO_REACTANT
gRNA_b_BFP_deg	SBO_DEGRADATION	gRNA_b_BFP	SBO_REACTANT
EYFP_deg	SBO_DEGRADATION	EYFP	SBO_REACTANT
cas9m_BFP_gRNA_b_deg	SBO_DEGRADATION	cas9m_BFP_gRNA_b	SBO_REACTANT

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 27-30). 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**.

```
1  # Create ComponentDefinition for CRP_b promoter
2  CRP_b = ComponentDefinition("CRP_b", BIOPAX_DNA, version)
3  CRP_b.roles.set(SO_PROMOTER)
4  CRP_b.sequences.add(CRP_b_seq.persistentIdentity.get())
5  doc.addComponentDefinition(CRP_b)
6
7  # Create ComponentDefintiion for EYFP coding sequence
8  EYFP_cds = ComponentDefinition("EYFP_cds", BIOPAX_DNA, version)
9  EYFP_cds.roles.set(SO_CDS)
10 doc.addComponentDefinition(EYFP_cds)
11
12 # Create ComponentDefinition for EYFP gene
13 EYFP_gene =  ComponentDefinition("EYFP_gene", BIOPAX_DNA, version)
14 EYFP_gene.roles.set(SO_PROMOTER)
15 doc.addComponentDefinition(EYFP_gene)
16
17 CRP_b_c = EYFP_gene.components.create("CRP_b")
18 CRP_b_c.definition.set(CRP_b.persistentIdentity.get())
19 CRP_b_c.access.set(SBOL_ACCESS_PUBLIC)
20 CRP_b_c.version.set(version)
21
22 EYFP_cds_c = EYFP_gene.components.create("EYFP_cds")
23 EYFP_cds_c.definition.set(EYFP_cds.persistentIdentity.get())
24 EYFP_cds_c.access.set(SBOL_ACCESS_PUBLIC)
25 EYFP_cds_c.version.set(version)
26
27 EYFP_gene_constraint = EYFP_gene.sequenceConstraints.create("EYFP_gene_constraint"
       )
28 EYFP_gene_constraint.subject.set(CRP_b_c.identity.get())
29 EYFP_gene_constraint.object.set(EYFP_cds_c.identity.get())
30 EYFP_gene_constraint.restriction.set(SBOL_RESTRICTION_PRECEDES)
31
32 EYFP =  ComponentDefinition("EYFP", BIOPAX_PROTEIN, version)
33 doc.addComponentDefinition(EYFP)
```

Other **ComponentDefinition** objects can be created using the same set of method calls. As an exercise, the reader is encouraged to specify them according to Table 1 and 2. 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 2.0) document.

We are now ready to create all remaining **FunctionalComponent** objects. We first create a **ModuleDefinition** object for the CRPb Characterization Circuit as shown below. 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.

```
#Create ModuleDefintion for CRISPR Repression
CRPb_circuit =  ModuleDefinition("CRPb_characterization_Circuit", version)
doc.addModuleDefinition(CRPb_circuit)

EYFP_fc = CRPb_circuit.functionalComponents.create("EYFP")
EYFP_fc.definition.set(EYFP.identity.get())
EYFP_fc.access.set(SBOL_ACCESS_PRIVATE)
EYFP_fc.direction.set(SBOL_DIRECTION_NONE)
EYFP_fc.version.set(version)
```

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.

Now, the CRISPR-based Repression Template **Module** is instantiated and connected to the CRPb Characterization Circuit using **MapsTo** objects. **Modules** are used to instantiates a submodule in the parent **ModuleDefinition**. **MapsTo** is then created to provide an identity relationship between two **ComponentInstance** objects, the first contained by the lower level definition of the **ComponentInstance** or **Module** that owns the MapsTo, and the second contained by the higher level definition that contains the **ComponentInstance** or **Module** that owns the **MapsTo**. The remote property of a **MapsTo** refers to the first lower level **ComponentInstance**, while the local property refers to the second higher level **ComponentInstance**. The code below shows these mappings.

```
Template_Module = CRPb_circuit.modules.create("CRISPR_Template")
Template_Module.definition.set(CRISPR_Template.identity.get())

cas9m_BFP_map = Template_Module.mapsTos.create("cas9m_BFP_map")
cas9m_BFP_map.refinement.set(SBOL_REFINEMENT_USE_LOCAL)
cas9m_BFP_map.local.set(cas9m_BFP_fc.identity.get())
cas9m_BFP_map.remote.set(cas9_generic_fc.identity.get())

gRNA_b_map = Template_Module.mapsTos.create("gRNA_b_map")
gRNA_b_map.refinement.set(SBOL_REFINEMENT_USE_LOCAL)
gRNA_b_map.local.set(gRNA_b_fc.identity.get())
gRNA_b_map.remote.set(gRNA_generic_fc.identity.get())

cas9m_BFP_gRNA_map = Template_Module.mapsTos.create("cas9m_BFP_gRNA_map")
cas9m_BFP_gRNA_map.refinement.set(SBOL_REFINEMENT_USE_LOCAL)
cas9m_BFP_gRNA_map.local.set(cas9m_BFP_gRNA_b_fc.identity.get())
cas9m_BFP_gRNA_map.remote.set(cas9_gRNA_complex_fc.identity.get())

EYFP_map = Template_Module.mapsTos.create("EYFP_map")
EYFP_map.refinement.set(SBOL_REFINEMENT_USE_LOCAL)
EYFP_map.local.set(EYFP_fc.identity.get())
EYFP_map.remote.set(target_fc.identity.get())

EYFP_gene_map = Template_Module.mapsTos.create("EYFP_gene_map")
EYFP_gene_map.refinement.set(SBOL_REFINEMENT_USE_LOCAL)
EYFP_gene_map.local.set(EYFP_gene_fc.identity.get())
EYFP_gene_map.remote.set(target_gene_fc.identity.get())
```

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.

```
doc.write("CRISPR_example.xml")
```

Other Features of pySBOL

So far, we have demonstrated how one can build the CRISPR-based repression module [1] using pySBOL. 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 retrieve top level objects from a document by calling a templated "get" method using the class of the target object as the template argument. For example, if we want to get the cas9_generic protein **ComponentDefinition** object, we can use the get<ComponentDefinition> method shown below (lines 1-4) by providing the display ID of the object. By default this retrieves the latest version of an object. Alternatively, one may pass a full URI as an argument to the getter, which may be necessary when retrieving previous versions of an object.

```
1 cas9_generic2 = doc.getComponentDefinition("cas9_generic")
```

Manipulating Optional Fields

Objects may include optional fields. These are indicated in the UML specification as properties having 0 or more possible values. For example, the role property of a ComponentDefinition is optional while the molecular type field is required. Optional properties can only be set after the object is created. The following code creates a DNA component which is designated as a promoter:

```
1 TargetPromoter = ComponentDefinition("TargetPromoter", BIOPAX_DNA, "1.0.0")
2 TargetPromoter.roles.set(SO_PROMOTER)
```

In addition, properties have a get method. To view the value of a property:

```
1 print(TargetPromoter.roles.get())
2 # This returns the string "http://identifiers.org/so/SO:0000167" which is the
    Sequence Ontology term for a promoter.
```

Note also that some properties may contain more than one value. In the specification diagrams, an asterisk symbol next to a property indicates that the property may hold an arbitrary number of values. For example, a ComponentDefinition may be assigned multiple roles. To append a new value to the values already assigned:

```
1 TargetPromoter.roles.add(SO + "0000568")
```

To get multiple values back from a property, it is necessary to iterate over the property:

```
1 # Iterate through a property to get multiple values
2 for i in range(len(reaction_participant.roles.get())):
3     role = reaction_participant.roles[i]
4     print(role)
```

An important thing to remember is that the set method will always overwrite the first value of a property, while the add method will always append a new value. To remove a value, one may use the remove method. Currently the remove method requires a numerical index, though this will likely change in the future.

```
1 TargetPromoter.roles.remove(0)
```

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 2.0) for more details).

Creating and Editing References

Some SBOL objects point to other objects by way of references. For example, ComponentDefinitions point to their corresponding Sequences. Properties of this type should be set with the URI of the related object.

```
EYFPGene = ComponentDefinition("EYFPGene", BIOPAX_DNA)
seq = Sequence("EYFPSequence", "atgnnntaa", SBOL_ENCODING_IUPAC)
EYFPGene.sequences.set(seq.identity.get())
```

Creating and Editing Child Objects

Some SBOL objects can be composed into hierarchical parent-child relationships. In the specification diagrams, these relationships are indicated by black diamond arrows. For example ComponentDefinitions are parents of SequenceAnnotations.

If operating in SBOL-compliant mode, you will almost always want to use the create method rather than constructors in order to create a child object. The create method constructs and adds the SequenceAnnotation in a single function call. The create method ALWAYS takes one argument-the displayId of the new object. Some values may be initialized with default values. Refer to documentation of specific constructors to learn which parameters are assigned default values. After object creation, these fields and optional fields may be changed.

```
point_mutation = TargetPromoter.sequenceAnnotations.create("point_mutation")
```

In SBOL-compliant mode, directly adding a child to a parent object is prohibited, in order to maintain URI persistence between them. In 'open-world mode' the library makes no assumptions about how URIs are formed and leaves URI generation entirely up to the user. In this case child objects can be directly created using constructors and added to the parent. Use `toggleSBOLCompliance()` if you prefer to generate your own URIs and operate in open-world mode. In future developments, constructors may be opened up for use in SBOL-compliant mode as well.

```
point_mutation = SequenceAnnotation("point_mutation")
TargetPromoter.sequenceAnnotations.add(point_mutation)
```

Serialization

The library supports reading and writing data encoded in RDF/XML format. All file I/O operations are performed on the Document object. The read and write methods are used for reading and writing files in SBOL format.

```
doc = Document()
doc.read("CRISPR_example.xml")
doc.write("CRISPR_example_new.xml")
```

The complete repression model described in this tutorial is provided in the libSBOL source code in the 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.

Validation

The library also supports validation of RDF/XML file to ensure that it conforms with SBOL specification. Validation is performed on a Document object over online validator. To run it, simply run validate() on a Document object. The returned string will contain the results of validation.

```
>>> result = doc.validate()
>>> print(result)
Validation successful, no errors.
```

Validation is also run when a SBOL file is created through `write()` function. The output of validation is returned as a string when Document.write() function is executed. Keep in mind that the file will be generated regardless of whether it passes the validation step or not.

```
>>> result = doc.write("CRISPR_example.xml")
>>> print(result)
Validation successful, no errors.
```

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.