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 2.0.0 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 `sboljs 2.2.1`

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. `sboljs 2.2.1` organizes the URI collections to enable efficient access. We first create an **SBOLDocument** object by calling its constructor as shown below.

```
let SBOLDocument = require('sboljs');
let filesystem = require('fs');
let createUri = require('./lib/createUri');
let cdTypes = require('./lib/componentDefinitionTypes');
let so = require('./lib/sequenceOntology');
let sbo = require('./lib/systemsBiologyOntology');
let encodings = require('./lib/encodings');

// Create the document
let document = new SBOLDocument();

// Set up namespaces

// Set some constants
let uriPrefix = "http://sbols.org/CRISPR_Example/";
let version = "1.0.0";
```

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.

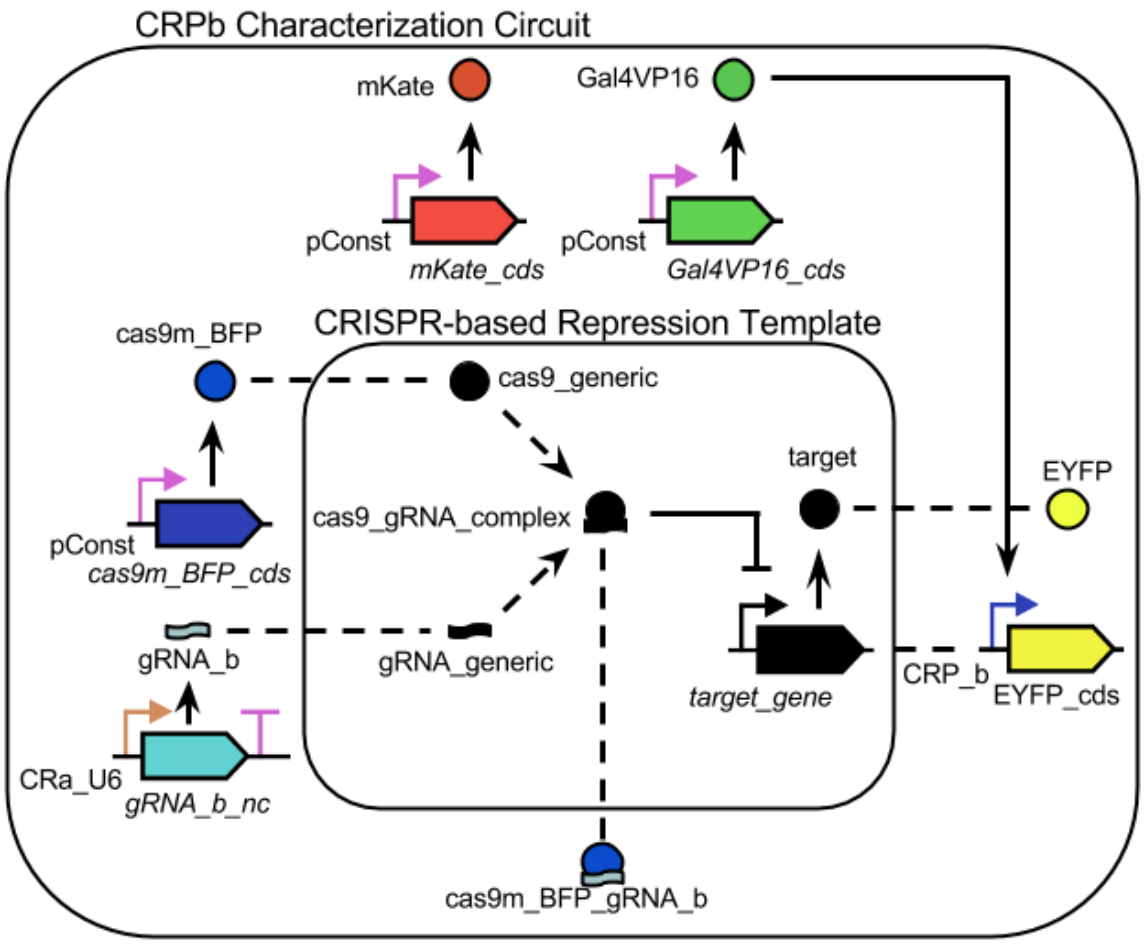

Figure 1: Illustration of a hierarchical CRISPR-based repression module represented in SBOL 2.0 (adapted from Figure 1a in [1]). The CRISPR-based Repression Template **ModuleDefinition** describes a generic CRISPR repression circuit that combines a Cas9 protein with a gRNA to form a complex (represented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

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 1) **ComponentDefinition** is done by passing its *displayId* "cas9_generic". The displayId is appended to the default namespace to create the URI "http://sbols.org/CRISPR_Example/cas9_generic. The next argument is the required field, *type*. 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 `BIOPAX_PROTEIN`(`http://www.biopax.org/release/biopax-level3.owl#Protein`), which is defined as the BioPAX ontology term for 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 by constructing a `ModuleDefinition` with the displayId "CRISPR_Template".

```
// Add top-level Component Definitions
let cas9_generic_cd = document.componentDefinition(createUri(uriPrefix, 'cas9_generic', version));
let grna_generic_cd = document.componentDefinition(createUri(uriPrefix, "gRNA_generic", version));
let cas9_grna_complex_cd = document.componentDefinition(createUri(uriPrefix, "cas9_gRNA_complex", version));
let target_gene_cd = document.componentDefinition(createUri(uriPrefix, "target_gene", version));
let target_cd = document.componentDefinition(createUri(uriPrefix, "target", version));

// Add Display IDs to component definitions
cas9_generic_cd.displayId = "cas9_generic";
grna_generic_cd.displayId = "gRNA_generic";
cas9_grna_complex_cd.displayId = "cas9_gRNA_complex";
target_gene_cd.displayId = "target_gene";
target_cd.displayId = "target";

// Add persistent identities to component definitions
cas9_generic_cd.persistentIdentity = createUri(uriPrefix, "cas9_generic");
grna_generic_cd.persistentIdentity = createUri(uriPrefix, "gRNA_generic");
cas9_grna_complex_cd.persistentIdentity = createUri(uriPrefix, "cas9_gRNA_complex");
target_gene_cd.persistentIdentity = createUri(uriPrefix, "target_gene");
target_cd.persistentIdentity = createUri(uriPrefix, "target");

// Add versions to component definitions
cas9_generic_cd.version = version;
grna_generic_cd.version = version;
cas9_grna_complex_cd.version = version;
target_gene_cd.version = version;
target_cd.version = version;

// Add types to component definitions
cas9_generic_cd.addType(cdTypes.PROTEIN);
grna_generic_cd.addType(cdTypes.RNA);
cas9_grna_complex_cd.addType(cdTypes.COMPLEX);
target_gene_cd.addType(cdTypes.DNA);
target_cd.addType(cdTypes.PROTEIN);

// Add roles to component definitions
grna_generic_cd.addRole(so.SGRNA);
target_gene_cd.addRole(so.PROMOTER);

// Add top-level module definition
let crispr_template_module = document.moduleDefinition(createUri(uriPrefix, "CRISPR_Template", version));

// Add module definition display ID
crispr_template_module.displayId = "CRISPR_Template";

// Add module definition persistent identity
crispr_template_module.persistentIdentity = createUri(uriPrefix, "CRISPR_Template");

// Add module definition version
crispr_template_module.version = version;
```

By default, sboljs 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`). 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 `cas9_complex_formation` in `CRISPR_Template`, with the displayId "cas9_complex_formation" and a non-covalent binding type (line 1 to 2). 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 represents a species participating in a biochemical reaction. The components which participate in an interaction must be assigned using the `participate` method.

```
let cas9_generic_fc = document.functionalComponent(createUri(uriPrefix, "
    CRISPR_Template/cas9_generic", version));
let grna_generic_fc = document.functionalComponent(createUri(uriPrefix, "
    CRISPR_Template/gRNA_generic", version));
let cas9_grna_complex_fc = document.functionalComponent(createUri(uriPrefix, "
    CRISPR_Template/cas9_gRNA_complex", version));
let target_gene_fc = document.functionalComponent(createUri(uriPrefix, "
    CRISPR_Template/target_gene", version));
let target_fc = document.functionalComponent(createUri(uriPrefix, "CRISPR_Template
    /target", version));

// Add display IDs to functional components
cas9_generic_fc.displayId = "cas9_generic";
grna_generic_fc.displayId = "gRNA_generic";
cas9_grna_complex_fc.displayId = "cas9_gRNA_complex";
target_gene_fc.displayId = "target_gene";
target_fc.displayId = "target";

// Add persistent identities to function components
cas9_generic_fc.persistentIdentity = createUri(uriPrefix, "CRISPR_Template/
    cas9_generic");
grna_generic_fc.persistentIdentity = createUri(uriPrefix, "CRISPR_Template/
    gRNA_generic");
cas9_grna_complex_fc.persistentIdentity = createUri(uriPrefix, "CRISPR_Template/
    cas9_gRNA_complex");
target_gene_fc.persistentIdentity = createUri(uriPrefix, "CRISPR_Template/
    target_gene");
target_fc.persistentIdentity = createUri(uriPrefix, "CRISPR_Template/target");

// Add versions to functional components
cas9_generic_fc.version = version;
grna_generic_fc.version = version;
cas9_grna_complex_fc.version = version;
target_gene_fc.version = version;
target_fc.version = version;

// Point functional components to their component definitions
cas9_generic_fc.definition = cas9_generic_cd.persistentIdentity;
grna_generic_fc.definition = grna_generic_cd.persistentIdentity;
cas9_grna_complex_fc.definition = cas9_grna_complex_cd.persistentIdentity;
target_gene_fc.definition = target_gene_cd.persistentIdentity;
target_fc.definition = target_cd.persistentIdentity;

// Add functional components to module definition for CRISPR template
crispr_template_module.addFunctionalComponent(cas9_generic_fc);
crispr_template_module.addFunctionalComponent(grna_generic_fc);
crispr_template_module.addFunctionalComponent(cas9_grna_complex_fc);
crispr_template_module.addFunctionalComponent(target_gene_fc);
crispr_template_module.addFunctionalComponent(target_fc);
```

```
// Create complex formation reaction
let cas9complex_formation_interaction = document.interaction(createUri(uriPrefix,
    "CRISPR_Template/cas9_complex_formation", version));

// Add complex formation reaction display ID
cas9complex_formation_interaction.displayId = "cas9_complex_formation";

// Add complex formation reaction persistent identity
cas9complex_formation_interaction.persistentIdentity = createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation");

// Add complex formation version
cas9complex_formation_interaction.version = version;

// Add type to complex formation interaction
cas9complex_formation_interaction.addType(sbo.NON_COVALENT_BINDING);

// Create complex formation interaction participations
let cas9_generic_participation = document.participation(createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation/cas9_generic", version));
let grna_generic_participation = document.participation(createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation/gRNA_generic", version));
let cas9_grna_complex_participation = document.participation(createUri(uriPrefix,
    "CRISPR_Template/cas9_complex_formation/cas9_gRNA_complex", version));

// Add complex formation interaction participations display IDs
cas9_generic_participation.displayId = "cas9_generic";
grna_generic_participation.displayId = "gRNA_generic";
cas9_grna_complex_participation.displayId = "cas9_gRNA_complex";

// Add complex formation interaction participations persistent identity
cas9_generic_participation.persistentIdentity = createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation/cas9_generic");
grna_generic_participation.persistentIdentity = createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation/gRNA_generic");
cas9_grna_complex_participation.persistentIdentity = createUri(uriPrefix, "
    CRISPR_Template/cas9_complex_formation/cas9_gRNA_complex");

// Add complex formation interaction participations version
cas9_generic_participation.version = version;
grna_generic_participation.version = version;
cas9_grna_complex_participation.version = version;

// Point complex formation interaction participants to their functional components
cas9_generic_participation.participant = cas9_generic_fc;
grna_generic_participation.participant = grna_generic_fc;
cas9_grna_complex_participation.participant = cas9_grna_complex_fc;

// Add participant roles for complex formation
cas9_generic_participation.addRole(sbo.REACTANT);
grna_generic_participation.addRole(sbo.REACTANT);
cas9_grna_complex_participation.addRole(sbo.PRODUCT);

// Add participants to complex formation interaction
cas9complex_formation_interaction.addParticipation(cas9_generic_participation);
cas9complex_formation_interaction.addParticipation(grna_generic_participation);
cas9complex_formation_interaction.addParticipation(cas9_grna_complex_participation
    );

// Create target production interaction
let eyfp_production_interaction = document.interaction(createUri(uriPrefix, "
    CRISPR_Template/target_production", version));

// Add target production interation display ID
eyfp_production_interaction.displayId = "target_production";
```

Creating CRPb Characterization Circuit

So far, we have completed the repression template. In order to construct the the CRPb Characterization Circuit, we must realize the template with specific components. 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 `Sequence` constructor, 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.

```
// Define the sequence strings
let cra_u6_seq_elements = "GGTTTACCGAGCTCTTATTGGTTTTCAAACTTCATTGACTGTGCC" +
                          "AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT" +
                          "TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT" +
                          "TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA" +
                          "AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT" +
                          "AAAATGGACTATCATATGCTTACCGTAACTTGAAATATAGAACCG" +
                          "ATCCTCCCATTGGTATATATTATAGAACCGATCCTCCCATTGGCT" +
                          "TGTGGAAAGGACGAAACACCGTACCTCATCAGGAACATGTGTTTA" +
                          "AGAGCTATGCTGGAAACAGCAGAAATAGCAAGTTTAAATAAGGCT" +
                          "AGTCCGTTATCAACTTGAAAAAGTGGCACCGAGTCGGTGCTTTTT" +
                          "TTGGTGCGTTTTTATGCTTGTAGTATTGTATAATGTTTTT";

let grna_b_elements = "AAGGTCGGGCAGGAAGAGGGCCTATTTCCCATGATTCCTTCATAT" +
                      "TTGCATATACGATACAAGGCTGTTAGAGAGATAATTAGAATTAAT" +
                      "TTGACTGTAAACACAAAGATATTAGTACAAAATACGTGACGTAGA" +
                      "AAGTAATAATTTCTTGGGTAGTTTGCAGTTTTAAAATTATGTTTT" +
                      "AAAATGGACTATCATATGCTTACCGTAACTTGAAAGTATTTCGAT" +
                      "TTCTTGGCTTTATATATCTTGTGGAAAGGACGAAACACCGTACCT" +
                      "CATCAGGAACATGTGTTTAAGAGCTATGCTGGAAACAGCAGAAAT" +
                      "AGCAAGTTTAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGG" +
                      "CACCGAGTCGGTGCTTTTTTT";

let mkate_seq_elements = "TCTAAGGGCGAAGAGCTGATTAAGGAGAACATGCACATGAAGCTG" +
                         "TACATGGAGGGCACCGTGAACAACCACCACTTCAAGTGCACATCC" +
                         "GAGGGCGAAGGCAAGCCCTACGAGGGCACCCAGACCATGAGAATC" +
                         "AAGGTGGTCGAGGGCGGCCCTCTCCCCTTCGCCTTCGACATCCTG" +
                         "GCTACCAGCTTCATGTACGGCAGCAAAACCTTCATCAACCACACC" +
                         "CAGGGCATCCCCGACTTCTTTAAGCAGTCCTTCCCTGAGGTAAGT" +
                         "GGTCCTACCTCATCAGGAACATGTGTTTTAGAGCTAGAAATAGCA" +
                         "AGTTAAAATAAGGCTAGTCCGTTATCAACTTGAAAAAGTGGCACC" +
                         "GAGTCGGTGCTACTAACTCTCGAGTCTTCTTTTTTTTTTTCACAG" +
                         "GGCTTCACATGGGAGAGAGTCACCACATACGAAGACGGGGGCGTG" +
                         "CTGACCGCTACCCAGGACACCAGCCTCCAGGACGGCTGCCTCATC" +
                         "TACAACGTCAAGATCAGAGGGGTGAACTTCCCATCCAACGGCCCT" +
                         "GTGATGCAGAAGAAAACACTCGGCTGGGAGGCCTCCACCGAGATG" +
                         "CTGTACCCCGCTGACGGCGGCCTGGAAGGCAGAAGCGACATGGCC" +
                         "CTGAAGCTCGTGGGCGGGGGCCACCTGATCTGCAACTTGAAGACC" +
                         "ACATACAGATCCAAGAAACCCGCTAAGAACCTCAAGATGCCCGGC" +
                         "GTCTACTATGTGGACAGAAGACTGGAAAGAATCAAGGAGGCCGAC" +
                         "AAAGAGACCTACGTCGAGCAGCACGAGGTGGCTGTGGCCAGATAC" +
                         "TGCG";


let crp_b_seq_elements = "GCTCCGAATTTCTCGACAGATCTCATGTGATTACGCCAAGCTACG" +
                         "GGCGGAGTACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAG" +
                         "TACTGTCCTCCGAGCGGAGTACTGTCCTCCGAGCGGAGTTCTGTC" +
                         "CTCCGAGCGGAGACTCTAGATACCTCATCAGGAACATGTTGGAAT" +
                         "TCTAGGCGTGTACGGTGGGAGGCCTATATAAGCAGAGCTCGTTTA" +
                         "GTGAACCGTCAGATCGCCTCGAGTACCTCATCAGGAACATGTTGG" +
                         "ATCCAATTCGACC";

// Create the sequences
let cra_u6_seq_sequence = document.sequence(createUri(uriPrefix, "CRa_U6_seq", version));
let grna_b_seq_sequence = document.sequence(createUri(uriPrefix, "gRNA_b_seq", version))
let mkate_seq_sequence = document.sequence(createUri(uriPrefix, "mKate_seq", version));
let crp_b_seq_sequence = document.sequence(createUri(uriPrefix, "CRP_b_seq", version));

// Set the sequence display IDs
cra_u6_seq_sequence.displayId = "CRa_U6_seq";
grna_b_seq_sequence.displayId = "gRNA_b_seq";
mkate_seq_sequence.displayId = "mKate_seq";
crp_b_seq_sequence.displayId = "CRP_b_seq";

// Set the sequence persistentIdentities
cra_u6_seq_sequence.persistentIdentity = createUri(uriPrefix, "CRa_U6_seq");
grna_b_seq_sequence.persistentIdentity = createUri(uriPrefix, "gRNA_b_seq");
mkate_seq_sequence.persistentIdentity = createUri(uriPrefix, "mKate_seq");
crp_b_seq_sequence.persistentIdentity = createUri(uriPrefix, "CRP_b_seq");

// Set the sequence display IDs
cra_u6_seq_sequence.version = version;
grna_b_seq_sequence.version = version;
mkate_seq_sequence.version = version;
crp_b_seq_sequence.version = version;
```

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 2-5). Then, we create two **ComponentDefinition** objects, one for the EYFP *coding sequence* (CDS) and another for the EYFP gene (lines 8-15). We use a **SequenceConstraint** object (lines 27-30) 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**. The **restriction** property uses flags defined in the formal specification, which are provided in sboljs as predefined constants. See the API documentation or the constants.h header file for predefined constants associated with an SBOL property.

```
1  // Create ComponentDefinition for CRP_b promoter
2  let crp_b_cd = document.componentDefinition(createUri(uriPrefix, "CRP_b", version)
       );
3
4  // Create ComponentDefintiion for EYFP coding sequence
5  let eyfp_cd = document.componentDefinition(createUri(uriPrefix, "EYFP", version));
6
7  // Create ComponentDefinition for EYFP gene
8  let eyfp_gene_cd = document.componentDefinition(createUri(uriPrefix, "EYFP_gene",
       version));
9
10 let crp_b_cd = document.componentDefinition(createUri(uriPrefix, "CRP_b", version)
       );
11
12 let eyfp_cds_cd = document.componentDefinition(createUri(uriPrefix, "EYFP_cds",
       version));
13
14 let eyfp_gene_constraint_sequence_constraint = document.sequenceConstraint(
       createUri(uriPrefix, "EYFP_gene/EYFP_gene_constraint", version));
```

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. Entries "type" and "roles" column in the table are constants corresponding to a SequenceOntology term. URIs for these terms are described in Table 3 of the Specification (Data Model 2.0) document.

We are now ready to create the CRPb Characterization Circuit which realizes the template design. We first create a **ModuleDefinition** object as shown below:

```
1  // Create ModuleDefinition for CRISPR Repression
2  let crpb_circuit = document.moduleDefinition(createUri(uriPrefix, "
       CRPb_characterization_circuit", 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 can be connected to the CRPb Characterization Circuit using **Modules**. **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**.

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.

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

```
1 // Serialize the result
2 let xml = document.serializeXML();
3
4 filesystem.writeFile('RepressionModel-js.rdf', xml);
```

Other Features of `libSBOL`

So far, we have demonstrated how one can build the CRISPR-based repression module [1] using `libSBOL`. 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.

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 PRODUCT
cas9m_BFP_production	SBO_GENETIC_PRODUCTION	cas9m_BFP_gene cas9m_BFP	SBO_PROMOTER 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_deg	SBO_DEGRADATION	gRNA_b	SBO_REACTANT
EYFP_deg	SBO_DEGRADATION	EYFP	SBO_REACTANT
cas9m_BFP_gRNA_b_deg	SBO_DEGRADATION	cas9m_BFP_gRNA_b	SBO_REACTANT

```
ComponentDefinition &cas9_generic1 = doc.get<ComponentDefinition>("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:

```
ComponentDefinition& TargetPromoter = *new ComponentDefinition("TargetPromoter",
    BIOPAX_DNA, "1.0.0");
TargetPromoter.roles.set(SO_PROMOTER)
```

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

```
cout << TargetPromoter.roles.get() << endl;
// 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:

```
TargetPromoter.roles.add(SO "0000568");
```

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

```
// Iterate through a property to get multiple values
for (auto i_role = reaction_participant.roles.begin(); i_role !=
    reaction_participant.roles.end(); i_role++)
{
    string role = *i_role;
    cout << role << endl;
}
```

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 number of values contained by a property can be obtained by calling the size method.

```
1 TargetPromoter.roles.size();
```

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.

```
1 ComponentDefinition& EYFPGene = *new ComponentDefinition("EYFPGene", BIOPAX_DNA);
2 Sequence& seq = *new Sequence("EYFPSequence", "atgnnntaa", SBOL_ENCODING_IUPAC);
3 EYFPGene.sequences.set(seq.identity.get());
```

Creating Extension Classes

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. These data are exchanged unmodified between software tools that adopt SBOL employing `libSBOL` or its sister libraries such as `libSBOLj`. `LibSBOL` employs custom extension classes in order to embed data. Extension classes are defined like any other C++ class, as long as the user adheres to some simple patterns. The extension class approach differs slightly from the custom annotation mechanism used by `libSBOLj`, but the end result is the same. The following snippet illustrates an extension class for biological parts compatible with the iGEM parts registry (`http://parts.igem.org`).

```
1  #include "sbol.h"
2
3  using namespace sbol;
4  using namespace std;
5
6  // These constants determine the appearance of nodes in the output file
7  #define EXTENSION_NS "http://igem.org#"    // Must end in a hash or forward-slash
8  #define EXTENSION_CLASS "iGEMCDef"         // Name of the class in XML output
9  #define EXTENSION_PREFIX "igem"            // Namespace prefix in XML output
10
11 class iGEMComponentDefinition : public ComponentDefinition  // Derive an extension
       class
12 {
13
14 public:
15     TextProperty partStatus;
16     TextProperty notes;
17     TextProperty source;
18     URIProperty results;
19
20     // Define the constructor. Put required fields in the argument list. Each
           required field must have a default value specified, even if only an empty
           string.
21     iGEMComponentDefinition(std::string uri = "", std::string partStatus = "Under
           construction") :
22
23         // Call public base class constructor
24         ComponentDefinition(uri),
25
26         // Initialize member properties. The second argument must ALWAYS be 'this
               '.
27         partStatus(EXTENSION_NS "partStatus", this, partStatus),  // The field is
               initialized to the value "Under construction"
28         notes(EXTENSION_NS "notes", this),                        // The optional
               field is not initialized with any value
29         source(PURL_URI "source", this),                        // Dublin Core
               namespace is already defined as part of the SBOL Core
30         results(EXTENSION_NS "results", this)
31     {
32         // Register the extension class.
33         register_extension_class < iGEMComponentDefinition >(EXTENSION_NS,
               EXTENSION_PREFIX, EXTENSION_CLASS);
34     };
35
36     // Destructor
37     ~iGEMComponentDefinition() {};
38 };
```

An extension class requires an extension namespace, a class name, and a required namespace prefix (which is just a shorthand symbol for the namespace in the output file). In the example above, the extension class iGEMCDef will be defined in the namespace http://igem.org, or simply 'igem'. Note that a properly formed namespace MUST end with '/' or '#'.

In line 9 the extension class is derived from the core SBOL class ComponentDefinition. This means that new properties defined in the extension class will be serialized as annotations under the ComponentDefinition class. In addition, entirely new TopLevel extension classes can be defined, but this is covered in the next section.

Lines 13-16 define the extension properties. Each object in SBOL 2.0 can be annotated by having any number of extension properties of type `TextProperty`, `URIProperty`, `IntProperty`, or `ReferencedObject` objects that store data in the form of name/value property pairs. In addition, extension classes can be assembled into composite

data structures using `OwnedObject` properties (not shown).

Line 19 defines the constructor signature. Like all SBOL classes, the first argument to an extension class should be a URI that identifies the new object. Also, as a best practice consistent with the rest of the core SBOL constructors, the remaining arguments should be required fields. All fields MUST have a default value specified such that a default constructor (see http://en.cppreference.com/w/cpp/language/default_constructor) is defined.

Lines 25-28 initialize the member properties. These also follow a simple pattern. The first argument is the URI of the property, which consists of the extension namespace followed by the property name as it appears in the serialized XML file. The second argument MUST always be **`this`**. In the optional third argument, an initial value for the property is specified. In this example, only the required argument partStatus is assigned an initial value. All the other properties are left blank by default.

Finally, the extension class is registered in the data model. The class interface can now be used like any other SBOL core class, and can be written to and parsed from an SBOL file. The following code demonstrates this.

```
1  int main()
2  {
3      Document& doc = *new Document();
4      setHomespace("http://sys-bio.org");
5
6      iGEMComponentDefinition& cd = *new iGEMComponentDefinition("My_component", "
           Available");
7      cd.notes.set("This component works in E. coli");
8      cd.source.set("This component was isolated from B. subtilis");
9      cd.results.set("http://synbiohub.org/igem/results/Works");
10     doc.add < iGEMComponentDefinition > (cd);
11     doc.write("igem_example.xml");
12 }
```

The extension class appears in the serialized RDF/XML as a ComponentDefinition with custom annotations embedded:

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <rdf:RDF xmlns:dcterms="http://purl.org/dc/terms/"
3     xmlns:igem="http://igem.org#"
4     xmlns:prov="http://www.w3.org/ns/prov#"
5     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6     xmlns:sbol="http://sbols.org/v2#">
7    <sbol:ComponentDefinition rdf:about="http://sys-bio.org/ComponentDefinition/
         My_component/1.0.0">
8      <igem:notes>This component works in E. coli</igem:notes>
9      <igem:partStatus>Available</igem:partStatus>
10     <igem:results rdf:resource="http://synbiohub.org/igem/results/Works"/>
11     <dcterms:source>This component was isolated from B. subtilis</dcterms:source>
12     <sbol:displayId>My_component</sbol:displayId>
13     <sbol:persistentIdentity rdf:resource="http://sys-bio.org/ComponentDefinition/
           My_component"/>
14     <sbol:type rdf:resource="http://www.biopax.org/release/biopax-level3.owl#
           DnaRegion"/>
15     <sbol:version>1.0.0</sbol:version>
16   </sbol:ComponentDefinition>
17 </rdf:RDF>
```

Sharing and Distributing Extensions

Simple extensions, such as those described above can be distributed as simple header files. Simply by including the header file with any application that links libSBOL, users can read and write extension classes and use basic accessor

methods. In more advanced cases, extensions may include .cpp implementation files. In these cases, it is possible to distribute extensions as binary files. An example command line for how to compile an extension called libdummy on Mac OSX is below:

```
$ g++ -dynamiclib -std=c++11 -I/usr/local/include/sbol -I/usr/local/include/
    raptor2 -L/usr/local/lib -lraptor2 -lsbol dummy_extension.cpp -o libdummy.
    dylib
```

Accessing Annotations Without an Extension

Extensions provide a means for users to quickly extend the SBOL Core data model and API. However, even if you don't have a particular extension installed, a user can still access extension data and custom annotations embedded in a file. The downside is that the user won't gain the full advantage of interfacing with the new class through an object-oriented API. Instead, the user has to access annotations using special methods in the base class. In the following example code snippet, the user interfaces with the iGEMComponentDefinition extension class by using its ComponentDefinition base class.

```
doc.read("igem_example.xml");
ComponentDefinition& cd = doc.componentDefinitions["My_component"];
cout << cd.getPropertyValue(EXTENSION_NS "partStatus") << endl;
cout << cd.getPropertyValue(EXTENSION_NS "notes") << endl;
cout << cd.getPropertyValue(PURL_URI "source") << endl;
cout << cd.getPropertyValue(EXTENSION_NS "results") << endl;
```

Creating a TopLevel Extension Class

Custom data can also be embedded at the top level of an SBOL document. The purpose of a top level extension class is to contain a set of annotations that are independent of any other class of SBOL object. To define a new top level class, simply derive from the TopLevel class. When deriving new classes from TopLevel class, a constant URI that defines the extension class must be specified. This URI specifies the namespace and name of the XML node for the serialized data. In this example, the Datasheet class contains two properties, a TextProperty that contains transcription data and a ReferencedObject which contains a reference to the iGEMComponentDefinition that the Datasheet describes.

```
class Datasheet : public TopLevel
{
public:
    TextProperty transcription_rate;
    ReferencedObject part;

    // Define the constructor. Put required fields in the argument list. Each
        required field must have a default value specified, even if only an empty
        string.
    Datasheet(std::string uri = "") :

        // Call the TopLevel constructor. Note that the first argument to the
            TopLevel constructor is a constant URI that defines the class. The
            second argument defines URIs for new object instances
        TopLevel(EXTENSION_NS "Datashee", uri),
        transcription_rate(EXTENSION_NS "transcription_rate", this),
        part(EXTENSION_NS "part", EXTENSION_NS "iGEMCDef", this)
    {
        // Register the extension class.
        register_extension_class < Datasheet >(EXTENSION_NS, EXTENSION_PREFIX, "
            Datasheet");
    };

    // Destructor
    ~Datasheet() {};
};

int main()
{
    Document& doc = *new Document();
    setHomespace("http://sys-bio.org");
    Datasheet& data = *new Datasheet("test");
    data.transcription_rate.set("0.75");
    data.part.set("http://sys-bio.org/ComponentDefinition/My_component/1.0.0");
    doc.add < Datasheet > (data);
    doc.write("datasheet.xml");
}
```

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.

```
SequenceAnnotation& point_mutation = TargetPromoter.annotations.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.

```
SequenceAnnotation& point_mutation = *new SequenceAnnotation("point_mutation");
TargetPromoter.annotations.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.

```
Document& doc = *new Document();
doc.read("CRISPR_example.xml");
doc.write("CRISPR_example.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.

Copying Objects

The library can make copies of **TopLevel** objects using the templated `copy` methods. This method takes a number of optional arguments. If no arguments are specified, a copy is made and the version is incremented. An object can be copied from one Document to another by passing a pointer to the target Document as the first argument. In addition, the object can be copied to a new namespace, which is specified as the optional second argument. Finally, a custom version tag can be specified in the third argument. The following code snippet demonstrates how the copy method may be used to copy a ComponentDefinition to a new namespace and Document.

```
ComponentDefinition& venus = old_doc.get<ComponentDefinition>('venus_yfp');
ComponentDefinition& venus_copy = venus.copy<ComponentDefinition>(&new_doc, "http
    ://igem.org");
new_doc.write("copy_example.xml");
```

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.

```
cout << doc.validate() << endl;
```

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.

```
std:string response = doc.write(std::string("CRISPR_example.xml"));
cout << response << endl;
```

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.