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 libSBOL 2.0.0

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. libSBOL 2.0.0 organizes the URI collections to enable efficient access. We first create an SBOLDocument object by calling its constructor as shown below.

```c++
setHomespace("http://sbols.org/CRISPR_Example");
toggleSBOLCompliantTypes();
Document &doc = *new Document();
std::string 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.

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". 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
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.
version specified by the version string may be specified. 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”.

```cpp
1  // Create ComponentDefinition for cas9_generic protein
2  ComponentDefinition& cas9_generic = *new ComponentDefinition("cas9_generic",
3     BIOPAX_PROTEIN, version);
4  doc.add<ComponentDefinition>(cas9_generic);

5  // Create ComponentDefinition for gRNA_generic RNA
6  ComponentDefinition& gRNA_generic = *new ComponentDefinition("gRNA_generic",
7     BIOPAX_RNA, version);
8  gRNA_generic.roles.set("http://identifiers.org/so/SO:0001998");
9  doc.add<ComponentDefinition>(gRNA_generic);

10 // Create ComponentDefinition for cas9_gRNA_complex
11 ComponentDefinition& cas9_gRNA_complex = *new ComponentDefinition("cas9_gRNA_complex",
12     BIOPAX_COMPLEX, version);
13 doc.add<ComponentDefinition>(cas9_gRNA_complex);

14 // Create ComponentDefinition for target gene
15 ComponentDefinition& target_gene = *new ComponentDefinition("target_gene",
16     BIOPAX_DNA, version);
17 target_gene.roles.set(SO::PROMOTER);
18 doc.add<ComponentDefinition>(target_gene);

19 // Create ComponentDefinition for target protein
20 ComponentDefinition& target = *new ComponentDefinition("target", BIOPAX_PROTEIN,
21     version);
22 doc.add<ComponentDefinition>(target);

23 // Create ModuleDefinition for CRISPR_Repression_Template
24 ModuleDefinition &CRISPR_Template = *new ModuleDefinition("CRISPR_Template",
25     version);
26 doc.add<ModuleDefinition>(CRISPR_Template);
```

By default, libSBOL 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

```cpp
// Create Interaction for cas9_complex_formation
Interaction cas9_complex_formation = *new Interaction(cas9_generic, gRNA_generic);
```

```cpp
// Create Interaction for cas9_gRNA_complex and target
Interaction cas9_gRNA_target = *new Interaction(cas9_gRNA_complex, target);
```

```cpp
// Create Interaction for target_gene and target
Interaction target_gene_target = *new Interaction(target_gene, target);
```
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 to 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.

```java
Interaction &Cas9Complex_Formation = CRISPR_Template.interactions.create("cas9_complex_formation");
Cas9Complex_Formation.types.set(SBO_NONCOVALENT_BINDING);

FunctionalComponent &cas9_generic_fc = CRISPR_Template.functionalComponents.create("cas9_generic");
cas9_generic_fc.definition.set(cas9_generic.persistentIdentity.get());
cas9_generic_fc.access.set(SBOL_ACCESS_PUBLIC);
cas9_generic_fc.direction.set(SBOL_DIRECTION_IN_OUT);
cas9_generic_fc.version.set(version);

Participation &cas9_generic_participation = Cas9Complex_Formation.participations.create("cas9_generic");
cas9_generic_participation.roles.set(SBO_REACTANT);
cas9_generic_participation.participant.set(cas9_generic_fc.identity.get());

FunctionalComponent &gRNA_generic_fc = CRISPR_Template.functionalComponents.create("gRNA_generic");
gRNA_generic_fc.definition.set(gRNA_generic.persistentIdentity.get());
gRNA_generic_fc.access.set(SBOL_ACCESS_PUBLIC);
gRNA_generic_fc.direction.set(SBOL_DIRECTION_IN_OUT);
gRNA_generic_fc.version.set(version);

Participation &gRNA_generic_participation = Cas9Complex_Formation.participations.create("gRNA_generic");
gRNA_generic_participation.roles.set(SBO_REACTANT);
gRNA_generic_participation.participant.set(gRNA_generic_fc.identity.get());

FunctionalComponent &cas9_gRNA_complex_fc = CRISPR_Template.functionalComponents.create("cas9_gRNA_complex");
cas9_gRNA_complex_fc.definition.set(cas9_gRNA_complex.persistentIdentity.get());
cas9_gRNA_complex_fc.access.set(SBOL_ACCESS_PUBLIC);
cas9_gRNA_complex_fc.direction.set(SBOL_DIRECTION_IN_OUT);
cas9_gRNA_complex_fc.version.set(version);

Participation &cas9_gRNA_complex_participation = Cas9Complex_Formation.participations.create("cas9_gRNA_complex");
cas9_gRNA_complex_participation.roles.set(SBO_PRODUCT);
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.
Creating CRPb Characterization Circuit

So far, we have completed the repression template. In order to construct the CRPb Characterization Circuit, we must realize the template with specific components. We first create Sequence objects for those provided in [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.

Unfortunately, as usual, not all sequences are provided in the paper.

---

1 Unfortunately, as usual, not all sequences are provided in the paper.
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 `c_ds`, 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 libSBOL as predefined constants. See the API documentation or the constants.h header file for predefined constants associated with an SBOL property.

```java
// Create ComponentDefinition for CRP_b promoter
ComponentDefinition& CRP_b = *new ComponentDefinition("CRP_b", BIOPAX_DNA, version);
CRP_b.roles.set(SO_PROMOTER);
CRP_b.sequences.add(CRP_b_seq.persistentIdentity.get());
doc.add<ComponentDefinition>(CRP_b);

// Create ComponentDefinition for EYFP coding sequence
ComponentDefinition& EYFP_cds = *new ComponentDefinition("EYFP_cds", BIOPAX_DNA, version);
EYFP_cds.roles.set(SO_CDS);
doc.add<ComponentDefinition>(EYFP_cds);

// Create ComponentDefinition for EYFP gene
ComponentDefinition& EYFP_gene = *new ComponentDefinition("EYFP_gene", BIOPAX_DNA, version);
EYFP_gene.roles.set(SO_PROMOTER);
doc.add<ComponentDefinition>(EYFP_gene);

Component &CRP_b_c = EYFP_gene.components.create("CRP_b");
CRP_b_c.definition.set(CRP_b.persistentIdentity.get());
CRP_b_c.access.set(SBOL_ACCESS_PUBLIC);
CRP_b_c.version.set(version);

Component &EYFP_cds_c = EYFP_gene.components.create("EYFP_cds");
EYFP_cds_c.definition.set(EYFP_cds.persistentIdentity.get());
EYFP_cds_c.access.set(SBOL_ACCESS_PUBLIC);
EYFP_cds_c.version.set(version);

SequenceConstraint &EYFP_gene_constraint = EYFP_gene.sequenceConstraints.create("EYFP_gene_constraint");
EYFP_gene_constraint.subject.set(CRP_b_c.identity.get());
EYFP_gene_constraint.object.set(EYFP_cds_c.identity.get());
EYFP_gene_constraint.restriction.set(SBOL_RESTRICTION_PRECEDES);
```

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 libSBOL 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:

```java
// Create ModuleDefinition for CRISPR Repression
ModuleDefinition &CRPb_circuit = *new ModuleDefinition("CRPb_characterization_Circuit", version);
doc.add<ModuleDefinition>(CRPb_circuit);
```

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.
Table 1: **ComponentDefinition** objects

<table>
<thead>
<tr>
<th>component definition</th>
<th>type</th>
<th>role</th>
<th>sequence</th>
<th>sequence constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>pConst</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cas9m_BFP_cds</td>
<td>BIOPAX_DNA</td>
<td>SO_CDS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cas9m_BFP_gene</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>cas9m_BFP_gene_constraint</td>
</tr>
<tr>
<td>cas9m_BFP</td>
<td>BIOPAX_PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>CRa_U6</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>CRa_U6_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b_nc</td>
<td>BIOPAX_DNA</td>
<td>SO_CDS</td>
<td>gRNA_b_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b_terminator</td>
<td>BIOPAX_DNA</td>
<td>SO_TERMINATOR</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>gRNA_b_gene</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>gRNA_b_gene_constraint1</td>
</tr>
<tr>
<td>gRNA_b</td>
<td>BIOPAX_RNA</td>
<td>SGRNA</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>mKate_cds</td>
<td>BIOPAX_DNA</td>
<td>SO_CDS</td>
<td>mKate_seq</td>
<td>n/a</td>
</tr>
<tr>
<td>mKate_gene</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>mKate_gene_constraint</td>
</tr>
<tr>
<td>mKate</td>
<td>BIOPAX_PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Gal4VP16_cds</td>
<td>BIOPAX_DNA</td>
<td>SO_CDS</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Gal4VP16_gene</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>GAL4VP16_gene_constraint</td>
</tr>
<tr>
<td>EYFP</td>
<td>BIOPAX_PROTEIN</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>cas9m_BFP_gRNA_b</td>
<td>BIOPAX_COMPLEX</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Gal4VP16</td>
<td>BIOPAX_DNA</td>
<td>SO_PROMOTER</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>EYFP_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>pConst</td>
<td>cas9m_BFP_cds</td>
<td></td>
</tr>
<tr>
<td>gRNA_b_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>CRa_U6</td>
<td>gRNA_b_nc</td>
<td></td>
</tr>
<tr>
<td>mKate_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>gRNA_b_nc</td>
<td>gRNA_b_terminator</td>
<td></td>
</tr>
<tr>
<td>GAL4VP16_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>pConst</td>
<td>mKate_cds</td>
<td></td>
</tr>
<tr>
<td>EYFP_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>CRP_b</td>
<td>EYFP_cds</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: **SequenceConstraint** objects

<table>
<thead>
<tr>
<th>displayId</th>
<th>restriction type</th>
<th>subject</th>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>cas9m_BFP_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>pConst</td>
<td>cas9m_BFP_cds</td>
</tr>
<tr>
<td>gRNA_b_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>CRa_U6</td>
<td>gRNA_b_nc</td>
</tr>
<tr>
<td>mKate_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>gRNA_b_nc</td>
<td>gRNA_b_terminator</td>
</tr>
<tr>
<td>GAL4VP16_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>pConst</td>
<td>mKate_cds</td>
</tr>
<tr>
<td>EYFP_gene_constraint</td>
<td>SBOL_RESTRICTION_PRECEDES</td>
<td>CRP_b</td>
<td>EYFP_cds</td>
</tr>
</tbody>
</table>

Now, the CRISPR-based Repression Template can be connected to the CRPb Characterization Circuit using Modules. Modules are used to instantiate 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.
### Table 3: Interaction objects

<table>
<thead>
<tr>
<th>interaction</th>
<th>type</th>
<th>participant</th>
<th>role</th>
</tr>
</thead>
<tbody>
<tr>
<td>mKate_production</td>
<td>SBO_GENETIC_PRODUCTION</td>
<td>mKate</td>
<td>SBO_PROMOTER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>mKate</td>
<td>SBO_PRODUCT</td>
</tr>
<tr>
<td>Gal4VP16_production</td>
<td>SBO_GENETIC_PRODUCTION</td>
<td>Gal4VP16</td>
<td>SBO_PROMOTER</td>
</tr>
<tr>
<td>cas9m_BFP_production</td>
<td>SBO_GENETIC_PRODUCTION</td>
<td>cas9m_BFP</td>
<td>SBO_PROMOTER</td>
</tr>
<tr>
<td>gRNA_b_production</td>
<td>SBO_GENETIC_PRODUCTION</td>
<td>gRNA_b</td>
<td>SBO_PROMOTER</td>
</tr>
<tr>
<td>EYFP_Activation</td>
<td>SBO_STIMULATION</td>
<td>EYFP</td>
<td>SBO_PROMOTER</td>
</tr>
<tr>
<td>mKate_deg</td>
<td>SBO_DEGRADATION</td>
<td>mKate</td>
<td>SBO_REACTANT</td>
</tr>
<tr>
<td>Gal4VP16_deg</td>
<td>SBO_DEGRADATION</td>
<td>Gal4VP16</td>
<td>SBO_REACTANT</td>
</tr>
<tr>
<td>cas9m_BFP_deg</td>
<td>SBO_DEGRADATION</td>
<td>cas9m_BFP</td>
<td>SBO_REACTANT</td>
</tr>
<tr>
<td>gRNA_b_deg</td>
<td>SBO_DEGRADATION</td>
<td>gRNA_b</td>
<td>SBO_REACTANT</td>
</tr>
<tr>
<td>EYFP_deg</td>
<td>SBO_DEGRADATION</td>
<td>EYFP</td>
<td>SBO_REACTANT</td>
</tr>
<tr>
<td>cas9m_BFP_gRNA_b_deg</td>
<td>SBO_DEGRADATION</td>
<td>cas9m_BFP_gRNA_b</td>
<td>SBO_REACTANT</td>
</tr>
</tbody>
</table>

```python
Module &CRISPR_Template_Module = CRPb_circuit.modules.create("CRISPR_Template");
CRISPR_Template_Module.definition.set(CRISPR_Template.identity.get());
MapsTo &cas9m_BFP_map = CRISPR_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());
MapsTo &gRNA_b_map = CRISPR_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());
MapsTo &cas9m_BFP_gRNA_map = CRISPR_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_fc.identity.get());
cas9m_BFP_gRNA_map.remote.set(cas9_gRNA_complex_fc.identity.get());
MapsTo &EYFP_map = CRISPR_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());
MapsTo &EYFP_gene_map = CRISPR_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());
doc.write("CRISPR_example.xml");
```

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.

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.

```cpp
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:

```cpp
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:

```cpp
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:

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

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

```cpp
// 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.

```cpp
TargetPromoter.roles.remove(0);
```
The number of values contained by a property can be obtained by calling the size method.

```java
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.

```java
ComponentDefinition& EYFPGene = *new ComponentDefinition("EYFPGene", BIOPAX_DNA);
Sequence& seq = *new Sequence("EYFPSequence", "atgnntaa", SBOL_ENCODING_IUPAC);
EYFPGene.sequences.set(seq.identity.get());
```

### Creating Extension Classes

In order to allow representation of data that cannot currently be represented by the SBOL data model or data that are outside the scope of SBOL, SBOL offers developers the ability to embed custom data. 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).
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.

```c++
int main()
{
    Document& doc = *new Document();
    setHomespace("http://sys-bio.org");

    iGEMComponentDefinition& cd = *new iGEMComponentDefinition("My_component", "Available");
    cd.notes.set("This component works in E. coli");
    cd.source.set("This component was isolated from B. subtilis");
    cd.results.set("http://synbiohub.org/igem/results/Works");
doc.add < iGEMComponentDefinition > (cd);
doc.write("igem_example.xml");
}
```

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

```xml
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:dcterms="http://purl.org/dc/terms/"
    xmlns:igem="http://igem.org#"
    xmlns:prov="http://www.w3.org/ns/prov#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:sbol="http://sbols.org/v2#">
    <sbol:ComponentDefinition rdf:about="http://sys-bio.org/ComponentDefinition/My_component/1.0.0">
        <igem:notes>This component works in E. coli</igem:notes>
        <igem:partStatus>Available</igem:partStatus>
        <igem:results rdf:resource="http://synbiohub.org/igem/results/Works"/>
        <dcterms:source>This component was isolated from B. subtilis</dcterms:source>
        <sbol:displayId>My_component</sbol:displayId>
        <sbol:persistentIdentity rdf:resource="http://sys-bio.org/ComponentDefinition/My_component"/>
        <sbol:version>1.0.0</sbol:version>
    </sbol:ComponentDefinition>
</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:

```bash
$ 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.

```cpp
1  doc.read("igem_example.xml");
2  ComponentDefinition& cd = doc.componentDefinitions["My_component"];
3  cout << cd.getPropertyValue(EXTENSION_NS "partStatus") << endl;
4  cout << cd.getPropertyValue(EXTENSION_NS "notes") << endl;
5  cout << cd.getPropertyValue(PURL_URI "source") << endl;
6  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 SequenceAn-
notations.

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 persis-
tence 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.getComponentDefinition('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

