ShortBOL tutorial

Introduction

Welcome to the ShortBOL Tutorial. ShortBOL is a scripting language, designed to be easy to use, powerful and extensible. ShortBOL is based around structured text to capture your ideas, and doesn't require any prior coding skills. When these scripts are run, they generate SBOL files which can then be used to derive the DNA sequences for your design from its parts, generate diagrams, and can be loaded into any SBOL-compliant computer-aided genome design tools.

This tutorial will get you up to speed in how to rapidly prototype synthetic biology designs with ShortBOL. It works through several steps to introduce the language, and give you practical experience using it to capture your designs. Our running example is a TetR/LacI toggle switch (see Gardner 2000). By the end of the tutorial, you will be able to represent the toggle switch structure and behaviour in ShortBOL and be able to run this script to generate an SBOL file that can then be used in any SBOL-compliant tooling. We then move on to develop a more complex example using CRISPR as described by Crispr transcriptional repression devices and layered circuits in mammalian cells.

Downloading and installing ShortBOL

1. Download or clone the ShortBOL repository:
   `git clone https://github.com/intbio-ncl/shortbol.git`
2. Navigate to your install directory
3. Install dependencies with: `python setup.py install –user`
4. Test the installation using the simple example provided
   - `simple_example.rdfsh` in the `/examples` folder is a design for a single promoter with its associated sequence
   - Compile the `simple_example.rdfsh` file with `python run.py -s sbolxml examples/simple_example.rdfsh -o <output-file>`
   - `<output-file>` is the name of the desired SBOL XML-RDF file

Designing a genetic toggle switch

1) Adding basic parts

We're going to start by building the ShortBOL for the TetR inverter of the TetR/LacI toggle switch. The TetR inverter couples a tetracycline-repressed promoter with the *lacI* coding sequence, so that in the absence of tetracycline, LacI is produced. We are going to describe the design of the TetR inverter using ShortBOL. Create a text document containing the following:

```
pTetR_prom is a Promoter()
lacI_CDS is a CDS()
```

These two lines simply declare a promoter called pTetR, and declare a complement determining an open reading frame, or coding sequence (CDS) called lacI. Comments an be added to the script. Any line starting with a pound '#' character is treated as a comment, and ignored. Blank lines can also be added for formatting and are also ignored.

```
# Declare a promoter named pTetR

pTetR is a Promoter()
lacI_CDS is a CDS()
```

2. Adding properties to SBOL components.

So far we have created a promoter and a CDS and named them. In ShortBOL, we call `pTetR` and `lacI_CDS` instances. An instance is any thing that you have named as part of your description of your design. It may be a piece of DNA, or a large biological module, or a reference to a simulation, or perhaps a publication or co-worker. Instance names are case-sensitive, so `pTetR` and `PTetR` are different instances, as the case of their leading letter differs. You can choose any name you like for an instance. The name is there to refer to it within your script. However, by choosing meaningful names, you will make the script easier to read and understand.

With ShortBOL we can attach properties and values to instances. ShortBOL uses brackets to make lines 'properties of' a containing instance. For example, we can add a human-readable description and comments to pTetR like this:

```
# Declare a promoter named pTetR
pTetR is a Promoter()
(
      # give pTetR a description
      description = "pTet promoter"
)
lacI_CDS is a CDS()
```

Comments *are not* carried through to the final **SBOL** representation, so use them to document your script, for people who will read it in the future (probably you!) and may need some hints. Information needed to understand your design, rather than your script, needs to be added as properties like description, as these *are* available from an **SBOL** design.

You can add any names and values you like to an instance. It is perfectly fine for you to make new ones up as you need them. However, some names mean something special within the SBOL standard. You will frequently use these two SBOL properties for documenting your design:

- **`description`**: associates a human-readable descriptions with things. This can be an extended block of text, that tells us more about an instance.

- **`name`**: a human-readable name, possibly including spaces and special characters. For our *pTetR*, a good choice of name would be "pTetR".

Exercise 1:

Start with the provided skeleton script and modify it so that lacI has the `name` "lacI", `description` "LacI protein coding region. Don't forget your brackets around the property names. The answers can be found at the end of this document.

Answer 1:

```
# Declare a promoter named pTetR
pTetR is a Promoter()
(
      # give pTetR a description
      description = "pTet promoter"
)
lacI_CDS is a CDS()
(
      # Properties of lacI CDS
      name = "lacI"
      description = "LacI protein coding region"
)
```

3. Working with types.

In the previous example, we created instances to represent *pTetR* and *lacI* in the *TetR* inverter device, and gave them `names`, `descriptions` and `displayIds`. When we put it all together, that example looks like this:

```
# Declare a promoter named pTetR
pTetR is a Promoter()
(
      # give pTetR a description
      description = "pTetR promoter"
)
lacI_CDS is a CDS()
(
      # Properties of lacI CDS
      name = "lacI"
      description = "LacI protein coding region"
)
```

Let's look at this example again. It declares two instances, a Promoter called pTetR and a CDS called lacI. The Promoter and CDS are types. They say what sort of thing pTetR and LacI are.

In ShortBOL, whenever you declare an instance, you construct it with a type. The name of the type is linked to the name of the instance with '`is a`' to denote that the instance is a type of something. A type can be distinguished from an instance since it will have a "()" suffix which indicates the tyoe constructor. A constructor can be used to initialise values of properties in an instance when it is created. More about this later in the section on creating sequences.

SBOL provides a pallet of types that can be used in your designs for all the common types of genetic parts. Here are some of the ones you may use most frequently:

`Promoter`: A genomic region where transcription is initiated.
`CDS`: A complement determining sequence; a genomic region that encodes a protein.
`Terminator`: A genomic region that terminates transcription.
`RBS`: A ribosome binding region, where the ribosome will bind to a transcript.
`Operator`: A region where proteins bind to regulate transcription.

You can add any number of these genetic parts to your design. Just give them each a unique name within your script.

Exercise 2:

The *TetR* inverter is made of four parts. A promoter, RBS, CDS and terminator. Edit the design above to include additional instances for an RBS instance called lacI_RBS and a Terminator instance called lac_term.

Answer 2:

```
# Declare a promoter named pTetR
pTetR is a Promoter()
(
      # give pTetR a description
      description = "pTet promoter"
)

# Declare a CDS named pTetR
lacI_CDS is a CDS()
(
      # Properties of lacI CDS
      name = "lacI"
      description = "LacI protein coding region"
)
# Declare a RBS named lacI_RBS
lacI_RBS is a RBS()
(
      name = "lacI_RBS"
      description = "RBS for the lacI CDS"
)
# Declare a terminator named lacI_term
lacI_term is a Terminator()
(
      name = "lacI_term"
      description = "Terminator for the lacI CDS"
`
```

4. Adding sequences

Ultimately, when you build a genetic design, you need the corresponding DNA sequence. Each individual genetic part in your design will have its own sequence, and the sequence of the whole design is composed from these. ShortBOL has a type called **`DnaSequence`** that lets you specify a DNA sequence, and a property **`sequence`** that lets you associate this with an instance representing a genetic part. e.g.

```
lacITSeq is a DnaSequence ("ttcagccaaaaaacttaagaccgccggtct
tgtccactaccttgcagtaatgcggtggacaggatcggcggttttcttttctcttctcaa")
```

Here we have constructed a `DnaSequence` named `lacITSeq`, and rather than setting a property, the DNA sequence string is passed into the `DnaSequence` constructor. ShortBOL instances are often created by giving the type constructor some values to work with. The constructor will use these to set up properties for you.

Now that we know how to make a sequence, we need to attach it to the corresponding part. This is done in the same way that we set the name, description and displayId for the parts earlier. SBOL defines a property called sequence that links from a genetic part back to the sequence it has. This time, rather than quoting the value, we use the naked value. This tells ShortBOL that we are linking to another instance, rather than capturing some text. Instances are always linked by the name that their ShortBOL instance was declared with, rather than by the value of their name, or any other data property.

```
lacITSeq is a DnaSequence ("ttcagccaaaaaacttaagaccgccggtct
tgtccactaccttgcagtaatgcggtggacaggatcggcggttttcttttctcttctcaa")

lacIT is a Terminator()
(
  sequence = lacITSeq
)
```

Exercise 3: Edit the shortbol above to also include a new promoter pTetR with its own sequence

Answer 3:

```
lacITSeq is a DnaSequence ("ttcagccaaaaaacttaagaccgccggtct
tgtccactaccttgcagtaatgcggtggacaggatcggcggttttcttttctcttctcaa")

pTetRSeq is a DnaSequence
("tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcac")
)

lacIT is a Terminator()
(
  sequence = lacITSeq
)
pTetR is a promoter()
(
   sequence = pTetRSeq
)
```

5. Composition

A core principle of synthetic biology design is that larger designs are built up from smaller, well-validated components. This paradigm is exemplified by BioBricks, an assembly standard and parts registry of genomic parts. The SBOL data standard provides a lot of tooling for describing how a design is composed.

In this tutorial, we are going to look at several strategies for using ShortBOL to compose a larger design from smaller ones, by building up the *TetR* inverter from its component parts using the approach specified in the SBOL specification document.

In the tutorial exercise 2 above, we made instances for the four parts of the *TetR* inverter device. However, we stopped short of assembling them into a composite device. The SBOL type for a composite DNA device is a type of `ComponentDefinition` called a `DnaComponent`. `Components` are used to compose objects into a structural hierarchy of a `DnaComponent`

To place the genetic parts we've made within a larger `DnaComponent`, we create a Component from each `DnaComponent` to be composed and then send each of these `components` the `component` property of `DnaComponent` as shown below in example 1:

Example 1

```
# The genetic parts of the TetR inverter
pTetR      is a Promoter()
lacI_RBS   is a RBS()
lacI_CDS   is a CDS()
lacI_term  is a Terminator()

pTetR_c is a Component(pTetR)
lacI_RBS_c is a Component(lacI_RBS)
lacI_CDS_c is a Component(lacI_CDS)
lacI_term_c is a Component(lacI_term)

# The composite device for the TetR inverter
tetRInverter is a DnaComponent()
(
  # include the child components
  component = pTetR_c
  component = lacI_RBS_c
  component = lacI_CDS_c
  component = lacI_term_c
)
```

Because we are adding four sub-components, we set the component property four times. When you assign to a property multiple times, you add new values rather than over-writing previous ones.

We have built a *pTetR* inverter device that contains its four genetic parts as sub-components. However, we haven't specified anything about how these parts are to be assembled. There are two complementary ways to specify this. Firstly, we can attach constraints on their relative positions. Secondly, we can say exactly where the sub-components are located within the composite component.

6. Composition using constraints and locations

Constraints

In this section we are going to explore constraints. Sequence constraints are declared using the `sequenceConstraint` property. The values of this property are `sequenceConstraint` instances. In this version of ShortBOL (v1.0) we are true to the SBOL data model and so there is a bit of setting up to do.

SBOL currently defines three types of constraints. These are `precedes`, `sameOrientationAs` and `differentOrientationAs`. These last two tell you if the two components share the same orientation or have different orientations, but not what the orientation of either component is.

The constraint we need in this design is `precedes`. This says that one component comes before the other in the design. In this way, we can place the genetic parts, left-to-right. In order to do this we need to create a precedes relationship for pairs of `Component` sand then include them in a `Component` to form the correct ordering as shown below:

Example 2

```
# The genetic parts of the TetR inverter
pTetR     is a Promoter()
lacI_RBS  is a RBS()
lacI_CDS  is a CDS()
lacI_term  is a Terminator()

pTetR_c is a Component(pTetR)
lacI_RBS_c is a Component(lacI_RBS)
lacI_CDS_c is a Component(lacI_CDS)
lacI_term_c is a Component(lacI_term)

pair1 is a Precedes(pTetR_c, lacI_RBS_c)
pair2 is a Precedes(lacI_RBS_c, lacI_CDS_c)
pair3 is a Precedes(lacI_CDS_c, lacI_term_c)

# The composite device for the TetR inverter
tetRInverter is a DnaComponent()
(
  # include the child components
  component = pTetR_c
  component = lacI_RBS_c
  component = lacI_CDS_c
  component = lacI_term_c
  # relative positions of child components
  sequenceConstraint = pair1
  sequenceConstraint = pair2
  sequenceConstraint = pair3
)
```

Locations and ranges

In the previous section we saw how ShortBOL can describe the relative positions of children within a parent design. Here we will see how it can give them exact positions. The SBOL property used to position sub-components is called `sequenceAnnotation`. See example 3 below.

Let's unpack that a bit. Firstly, we have the parts for the inverter defined as `DNAComponents` and then their corresponding `Components` also defined too. We then define some InlineRange objects that define the sequence range that a genetic part exists in nucleotides on the composite sequence. InLine means that the sequences are on the top strand. Similarly a `ReverseComplementRange` could also be used here to indicate that the sequences lie on the bottom strand. The `InlineRange` objects are then each used to create corresponding `SequenceAnnotation` objects. We then define the tetInverter composite `DnaComponent` as in the previous examples but also call the `components` and the `sequenceAnnotation` methods on `DNAComponent` with the corresponding `Component` and SequenceAnnotation objects.

The value of `sequenceAnnotation` is actually a complex object. This expects four values; the component to locate, the start and end coordinates, and a flag indicating if the construct is to be inserted inline (on the forward strand), or `reverseComplement` (on the reverse backward strand).

Example 3.

```
# The genetic parts of the TetR inverter
pTetR      is a Promoter()
lacI_RBS   is a RBS()
lacI_CDS   is a CDS()
lacI_term  is a Terminator()

pTetR_c is a Component(pTetR)
lacI_RBS_c is a Component(lacI_RBS)
lacI_CDS_c is a Component(lacI_CDS)
lacI_term_c is a Component(lacI_term)

pTetR_loc is a InlineRange (1, 55)
lacI_RBS_loc is a InlineRange (56, 68)
lacI_CDS_loc is a InlineRange (169, 1197)
lacI_term_loc is a InlineRange (1197, 1240)


pTetR_sa is a SequenceAnnotation (pTetR_loc)
lacI_RBS_sa is a SequenceAnnotation (lacI_RBS_loc)
lacI_CDS_sa is a SequenceAnnotation (lacI_CDS_loc)
lacI_term_sa is a SequenceAnnotation (lacI_term_loc)


tetRInverter is a DnaComponent()
(
  # include the child components
  component = pTetR_c
  component = lacI_RBS_c
  component = lacI_CDS_c
  component = lacI_term_c

  # absolute positions of child components
  sequenceAnnotation = pTetR_sa
  sequenceAnnotation = lacI_RBS_sa
  sequenceAnnotation = lacI_CDS_sa
  sequenceAnnotation = lacI_term_sa
)
```

Exercise 4: In the previous *pTetR* inverter positions example, we specified the positions of the four parts. However, we haven't specified their sequence. Add a second terminator lacI_term2 and add the sequences to the example above so that the final SBOL design is able to generate the DNA sequence for the pTetR-lacI_RBS-lacI_CDS-lacI_term-lacI_term2 device.

Answer 4:

```
#The sequences of the TetR inverter parts
pTetR_seq is a DnaSequence
("tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcac")
lacI_CDS_seq is a DnaSequence
("gtgaaaccagtaacgttatacgatgtcgcagagtatgccggtgtctcttatcagaccgtttcccgcgtg
gtgaaccaggccccaatacgcaaaccgcctctccccgcgcgttggccgattcattaatgcagctggcacga
caggtttcccgactggaaagcgggcag")
lacI_RBS_seq is a DnaSequence ("aaggaggtg")
lacI_term_seq is a DnaSequence
("ttcagccaaaaaacttaagaccgccggtcttgtccactaccttgcagtaatgcggtggacaggatcggc
ggttttcttttctcttctcaa")
lacI2_term_seq is a DnaSequence
("ccggcttatcggtcagtttcacctgatttacgtaaaaacccgcttcggcgggtttttgcttttggaggg
gcagaaagatgaatgactgtccacgacgctatacccaaaagaaa")


# The genetic parts of the TetR inverter
pTetR  is a Promoter()
(
     sequence = lacI_term_seq
)
lacI_RBS is a RBS()
(
     sequence = pTetR_seq
)
lacI_CDS is a CDS()
(
     sequence = lacI_CDS_seq
)
lacI_term is a Terminator()
(
     sequence = lacI_term_seq
)
lacI2_term is a Terminator()
(
     sequence = lacI2_term_seq
)
```

Answer 4 contd.:

```
#Build components for each of the DNAComponents
pTetR_c is a Component(pTetR)
lacI_RBS_c is a Component(lacI_RBS)
lacI_CDS_c is a Component(lacI_CDS)
lacI_term_c is a Component(lacI_term)
lacI2_term_c is a Component(lacI2_term)

#Specify the range for each part
pTetR_loc is a InlineRange(1,55)
lacI_RBS_loc is a InlineRange(56,68)
lacI_CDS_loc is a InlineRange(169,1197)
lacI_term_loc is a InlineRange(1197,1240)
lacI2_term_loc is a InlineRange(1241,1280)

pTetR_sa is a SequenceAnnotation (pTetR_loc)
(
     component = pTetR_c
)
lacI_RBS_sa is a SequenceAnnotation (lacI_RBS_loc)
(
     component = lacI_RBS_c
)
lacI_CDS_sa is a SequenceAnnotation (lacI_CDS_loc)
(
     component = lacI_CDS_c
)
lacI_term_sa is a SequenceAnnotation (lacI_term_loc)
(
     component = lacI_term_c
)
lacI2_term_sa is a SequenceAnnotation (lacI2_term_loc)
(
     component = lacI2_term_c
)

tetRInverter is a DnaComponent()
(
  # include the child components
  component = pTetR_c
  component = lacI_RBS_c
  component = lacI_CDS_c
  component = lacI_term_c
  component = lacI2_term_c

  # absolute positions of child components
  sequenceAnnotation = pTetR_sa
  sequenceAnnotation = lacI_RBS_sa
  sequenceAnnotation = lacI_CDS_sa
  sequenceAnnotation = lacI_term_sa
  sequenceAnnotation = lacI2_term_sa
```

Modules

Up until now we have been building descriptions of the physical design, by describing the stuff that makes it up. Usually, the physical parts are artifacts of achieving a desired *behaviour*, rather than being an ends in their own right. The SBOL data standard provides a rich, compositional model for describing the intended behaviour of a design, in parallel to the desired structure. This is captured by the ModuleDefinition type. The ModuleDefinition data model groups together the participating physical parts, their interactions, links to numerical models if they exist, and sub-modules. Here we will cover the physical parts and their interactions.

A Module

In the functional design of the TetR inverter, the TetR protein represses the expression of the LacI protein. To capture this functionality, we first need to create a module, and add components for TetR and LacI to it. Then we add an interaction to say that TetR represses LacI. Here we are true to the SBOL data model and so we need to create `FunctionalComponents` which in turn become `Participants` in an `Interaction` as described below in Example 4.

Example 4

```
# Example 4
# The TetR and LacI proteins
TetR is a ProteinComponent()
LacI is a ProteinComponent()

TetR_fc is a FunctionalComponent(TetR,none)
LacI_fc is a FunctionalComponent(LacI,none)

#Make a participation for the two proteins
TetR_part is a Participation(TetR_fc, inhibitor)
LacI_part is a Participation(LacI_fc, inhibited)

#Make an Interaction for the participants
TetRLacI_int is a Interaction(genetic_production)
(
     participation = TetR_part
     participation = LacI_part
)

# The TetR inverter module
TetR_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  functionalComponent = TetR_fc
  functionalComponent = LacI_fc
  interaction = TetRLacI_int
)
```

Exercise 5: The LacI inverter is very similar, but in this module LacI represses TetR. Write a script to include this interaction.

Answer 5:

```
# Answer 5
# The TetR and LacI proteins
TetR is a ProteinComponent()
LacI is a ProteinComponent()

TetR_fc is a FunctionalComponent(TetR,none)
LacI_fc is a FunctionalComponent(LacI,none)

#Make a participation for the two proteins
TetR_part is a Participation(TetR_fc, inhibitor)
LacI_part is a Participation(LacI_fc, inhibited)

#Make an Interaction for the participants
LacITetR_int is a Interaction(inhibition)
(
      participation = TetR_part
      participation = LacI_part
)

# The TetR inverter module
LacI_inverter is a ModuleDefinition()
(
  description = "LacI inverter"
  functionalComponent = TetR_fc
  functionalComponent = LacI_fc
  interaction = LacITetR_int
)
```

Composing Modules

In the previous section, we have built two modules, one for TetR inverter and one for the LacI inverter. The next step is to combine these into a toggle-switch module. To do this, we create a new ModuleDefinition that imports the two inverters. The basics for this are shown below.

```
toggleSwitch is a ModuleDefinition()
(
      description = "LacI/TetR toggle switch"
      module =  TetR_Inverter
      module = LacI_Inverter
)
```

This composite module contains all of the behaviour of both the TetR and LacI inverter modules. However, at the moment both of the inverters are 'black box', with completely independent behaviour. What we want to do is glue them together, so that they are using the same pool of TetR and LacI molecules. This will cause them to repress one-another, flip-flopping between repressing TetR levels and LacI levels.

To achieve this, we need to wire components in the sub-modules. This is done using the mapsTo property. We create placeholder components in the super-module, and then use mapsTo to wire components in the sub-modules to that component. By wiring TetR from both inverters to the same component in the super-module, we identify them with a shared molecule pool. This couples the behaviour of the two inverters, so that one now affects the levels of molecules used by the other. The other change in this example is that because the mapsTo property is defined on the Module, we have to create the Module instance the long way, with an explicit definition, rather than relying upon ShortBOL to generate one for us given a reference.

The final design that includes both Tet inverter and Lac inverter modules glued together to form the final toggleswitch design is shown below in example 5. Note that here we have added the '`inout`' parameter to the constructor of the `FunctionalComponent` class.

We have also included the Class `MapsUseLocal` which establishes that the TetR protein is the same protein in both the TetR inverter and the LacI inverter and that the LacI protein also is the same protein in both the TetR inverter and the LacI inverter, essentially linking their `functionalcomponents` to each other.

```
# Example 5
# The TetR and LacI proteins
TetR is a ProteinComponent()
LacI is a ProteinComponent()

TetR_fc is a FunctionalComponent(TetR,inout)
LacI_fc is a FunctionalComponent(LacI,inout)

#Make a participation for the two proteins
TetR_part is a Participation(TetR_fc, inhibitor)
LacI_part is a Participation(LacI_fc, inhibited)

#Make an Interaction for the participants
TetRLacI_int is a Interaction(inhibition)
(
      participation = TetR_part
      participation = LacI_part
)

# The TetR inverter module
TetR_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  functionalComponent = TetR_fc
  functionalComponent = LacI_fc
  interaction = TetRLacI_int
)

#The toggle switch module

TetR_map is a MapsUseLocal(TetR_lacinv_fc,TetR_fc)
LacI_map is a MapsUseLocal(LacI_lacinv_fc,LacI_fc)
toggleSwitch is a Module(TetR_inverter)
(
      description = "toggle switch"
      mapsTo = TetR_map
      mapsTo = LacI_map
)

#Make new FunctionalComponents
TetR_lacinv_fc is a FunctionalComponent(TetR,inout)
LacI_lacinv_fc is a FunctionalComponent(LacI,inout)

#Make a participation for the two proteins in the Lacinverter module
LacI_lacinv_part is a Participation(LacI_lacinv_fc, inhibitor)
TetR_lacinv_part is a Participation(TetR_lacinv_fc, inhibited)

#Make an Interaction for the participants in the Lacinverter module
LacITetR_int is a Interaction(inhibition)
(
      participation = TetR_lacinv_part
      participation = LacI_lacinv_part
)
```

Example 5 contd.

```
# The LacI inverter module
LacI_inverter is a ModuleDefinition()
(
  description = "LacI inverter"
  functionalComponent = TetR_lacinv_fc
  functionalComponent = LacI_lacinv_fc
  interaction = LacITetR_int
  module = toggleSwitch
)
```

Acknowledgments

This document draws heavily on a original tutorial by Matt Pocock (see http://shortbol.ico2s.org/tutorial.html#/) relating to an earlier Scala implementation version of ShortBOL with a slightly different syntax.