ShortBOL tutorial

Abstraction Layer Two

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.

Abstraction layer two

Abstraction layers are layers of templates and processes of building designs that as the abstraction layer increases the design moves away from the underlying SBOL data model (although it is still there!).
This abstraction layer adds the concept of implicit template instance creation. What this means is that you don’t need to explicitly type and understand the underlying SBOL data model much of which doesn’t directly translate to real genetic designs.
This is done by invoking these new inline templates inside of an instance of a template that creates unneeded templates behind the scenes.

Abstraction layers also use all previous abstraction layers to produce this layer, this means that specialised Templates are also still usable and used in this layer, for example a specialised ComponentDefinition is DNA.

If you don’t understand what this means, don’t worry , just know this layer was created to make it easier and quicker for you to describe your designs in ShortBOL.

Downloading and Installing ShortBOL

1. Download or clone the ShortBOL repository:
    a. `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

- `/has_sequence.txt` in the
    - `/examples/examples/examples_abstraction_layer_2` folder is a design for a single promoter and single terminator with associated sequences.
- Compile the `has_sequence.txt` file with `python run.py -s sbolxml /examples/examples/examples_abstraction_layer_2/has_sequence.txt -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 is a Promoter()
lacI 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 can 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_prom 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 anything 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 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 description 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 and Answer files can be found as files supporting this document.

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 properties.

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 type 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 genome 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. A DNA sequence can be defined and added to a part with a single line: hasDNASequence("Sequence Here").
This will link the given sequence with the part it is defined inside (lacI_term in this example).
This is all that is needed to define and link sequences.

```
lacI_term is a Terminator()
(
      hasDNASequence("ttcagccaaaaaac")
)
```

Exercise 3:

Edit the ShortBOL above to also include a new promoter pTetR with its own sequence.

Answer 3:

```
lacI_term is a Terminator()
(
      hasDNASequence("ttcagccaaaaaacttaag")
)
pTetR is a Promoter()
(
      hasDNASequence("tccctatcagtgatagagattgacatccc")
)
```

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 can specify hasComponent(component name). What this means is we would like to create an instance of these sub-components and set them as sub-parts on the overarching component (tetRInverter).

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()

# The composite device for the TetR inverter
tetRInverter is a DNAComponent()
(
  # include the child components
  hasComponent(pTetR)
  hasComponent(lacI_RBS)
  hasComponent(lacI_CDS)
  hasComponent(lacI_term)
)
```

Each hasComponent adds a single component to the `tetRInverter`.

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.

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.

SBOL currently defines four types of constraints. These are `precedes`, `sameOrientationAs`, `differentOrientationAs` and `differentFrom`. These constraints describe rules that must be true between two parts but not any exact details for example positions of the parts or orientation. 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 simply use the `precedes` term. As you can see in the first precedes term in the DNAComponent, this states that the pTetR Promoter must precede the LacI RBS.

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()

# The composite device for the TetR inverter
tetRInverter is a DNAComponent()
(
  # relative positions of child components
  precedes(pTetR, lacI_RBS)
  precedes(lacI_RBS, lacI_CDS)
  precedes(lacI_CDS, lacI_term)
  precedes(lacI_CDS, lacI_term)
)
```

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.

Two new terms are displayed here. Firstly the "`InlineRange`" part. This is a standalone part that simply defines a coordinate like value. The `InlineRange` indicating if the construct is to be inserted inline (on the forward strand) or `ReverseComplementRange` (on the reverse backward strand).

However, this part just specifies a location and not on what part this location is pointing to. This is where the `sequenceAnnotation` is used that is provided with the component and the Range part.
To explain this in example3 the first `sequenceAnnotation(pTetR,pTetR_loc)`, this states that pTetR is located on the forward strand of the tetRInverter from sequence located as position 1 to sequence located at position 55.

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_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)

tetRInverter is a DNAComponent()
(
  sequenceAnnotation(pTetR,pTetR_loc)
  sequenceAnnotation(lacI_RBS, lacI_RBS_loc)
  sequenceAnnotation(lacI_CDS, lacI_CDS_loc)
  sequenceAnnotation(lacI_term, lacI_term_loc)
)
```

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:

```
pTetR is a Promoter()
(
       hasDNASequence("tccctatcagtgatagagattgacatccctatcagtgatagagatac")
)

lacI_RBS is a RBS()
(
      hasDNASequence("aaggaggtg")
)

lacI_CDS is a CDS()
(
      hasDNASequence("gtgaaaccagtaacgttatacgatgtcgc")
)

lacI_term is a Terminator()
(
      hasDNASequence("ttcagccaaaaaactta")
)

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)

tetRInverter is a DNAComponent()
(
        sequenceAnnotation(pTetR,pTetR_loc)
        sequenceAnnotation(lacI_RBS, lacI_RBS_loc)
        sequenceAnnotation(lacI_CDS, lacI_CDS_loc)
        sequenceAnnotation(lacI_term, lacI_term_loc)
)
```

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 artefacts of achieving a desired *behaviour*, rather than being an end 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 ModuleDefinition

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 an interaction to say that TetR represses LacI.

Example 4

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

# The TetR inverter
TetR_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  inhibition(TetR, LacI)
)
```

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()

# The TetR inverter
LacI_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  inhibition(LacI,TetR)
)
```

Composing Modules

In the previous section, we have built two ModuleDefinitions, one for TetR inverter and one for the LacI inverter. The next step is to describe how the ModuleDefinitions are linked in the toggle-switch module. To do this, we create a new module that imports the `TetR_inverter`. Then this module is referenced in the `LacI_inverter`. What this actually means is that an instance of `TetR_inverter` is a sub-module of `LacI_inverter`.

```
toggleSwitch is a Module()
(
      description = "LacI/TetR toggle switch"
)

LacI_inverter is a ModuleDefinition()
(
      description = "LacI inverter"
      inhibition(LacI,TetR)
      module = toggleSwitch
)
```

The `LacI_inverter` ModuleDefinition 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 series of terms which Represent the same entity in the overall design.
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 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.

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 the parts together.

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

TetR_inv is a ProteinComponent()
LacI_inv is a ProteinComponent()

# The TetR inverter module
TetR_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  inhibition(TetR,LacI)
)

#The toggle switch module
toggleSwitch is a Module(TetR_inverter)
(
      description = "toggle switch"
      mapsUseLocal(TetR,TetR_inv)
      mapsUseLocal(LacI,LacI_inv)
)

# The LacI inverter module
LacI_inverter is a ModuleDefinition()
(
  description = "LacI inverter"
  inhibition(LacI_inv,TetR_inv)
  module = toggleSwitch
```

Full Toggle Switch Example

The unified example from for the genetic toggle switch example.

```
# tetRInverter sequence definition
pTetR is a Promoter()
(
  hasDNASequence("tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcac")
)
# Declare a RBS named lacI_RBS
lacI_RBS is a RBS()
(
  name = "lacI_RBS"
  description = "RBS for the lacI CDS"
  hasDNASequence("aa")
)
lacI_CDS is a CDS()
(
  name = "lacI"
  description = "LacI protein coding region"
  hasDNASequence("aaggaggtg")
)
# Declare a terminator named lacI_term
lacI_term is a Terminator()
(
  name = "lacI_term"
  description = "Terminator for the lacI CDS"
  hasDNASequence("tccctatcagtgatagagattgacatccctatcagtgatagagatactgagcac")
)
pTetR_loc is a InlineRange(1,55,Inline)
lacI_RBS_loc is a InlineRange(56,68,Inline)
lacI_CDS_loc is a InlineRange(169,1197,Inline)
lacI_term_loc is a InlineRange(1197,1240,Inline)

tetRInverter is a DNAComponent()
(
  # relative positions of child components
  precedes(pTetR, lacI_RBS)
  precedes(lacI_RBS, lacI_CDS)
  precedes(lacI_CDS, lacI_term)
  sequenceAnnotation(pTetR,pTetR_loc)
  sequenceAnnotation(lacI_RBS, lacI_RBS_loc)
  sequenceAnnotation(lacI_CDS, lacI_CDS_loc)
  sequenceAnnotation(lacI_term, lacI_term_loc)
)
```

```
TetR, LacI interaction and maps

TetR is a ProteinComponent()
LacI is a ProteinComponent()

TetR_inv is a ProteinComponent()
LacI_inv is a ProteinComponent()

# The TetR inverter module
TetR_inverter is a ModuleDefinition()
(
  description = "TetR inverter"
  inhibition(TetR,LacI)
)

#The toggle switch module
toggleSwitch is a Module(TetR_inverter)
(
  description = "toggle switch"
  mapsUseLocal(TetR,TetR_inv)
  mapsUseLocal(LacI,LacI_inv)
)

# The LacI inverter module
LacI_inverter is a ModuleDefinition()
(
  description = "LacI inverter"
  inhibition(LacI_inv,TetR_inv)
  module = toggleSwitch
)
```

Abstraction Layer One to Two

Below are some examples of ShortBOL that produce the same output (are functionally the same) but typed differently.
This is useful for users who are coming from using layer one but want the reduced typing and simpler format of layer two.

Creating Sequences

Abstraction Layer One

```
lacI_term_seq is a DNASequence("ttcagccaaaaaac ")

lacI_term is a Terminator()
(
  sequence = lacI_term_seq
)
```

Abstraction Layer Two

```
lacI_term is a Terminator()
(
  hasDNASequence("ttcagccaaaaaac")
)
```

Defining Sequence Annotations

Abstraction Layer One

```
pTetR is a Promoter()
pTetR_c is a Component(pTetR)
pTetR_loc is a InlineRange(1,55)
pTetR_sa is a SequenceAnnotation(pTetR_loc)
(
  component = pTetR_c
)

tetRInverter is a DNAComponent()
(
  component = pTetR_c
  sequenceAnnotation = pTetR_sa
)
```

Abstraction Layer Two

```
pTetR is a Promoter()
pTetR_loc is a InlineRange(1,55,Inline)
tetRInverter is a DNAComponent()
(
  sequenceAnnotation(pTetR,pTetR_loc)
)
```

Specifying Interactions

Abstraction Layer One

```
TetR is a ProteinComponent()
LacI is a ProteinComponent()

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

LacI_lacinv_part is a Participation(LacI_fc, inhibitor)
TetR_lacinv_part is a Participation(TetR_fc, inhibited)

LacITetR_int is a Interaction(inhibition)
(
  participation = TetR_lacinv_part
  participation = LacI_lacinv_part
)

LacI_inverter is a ModuleDefinition()
(
  functionalComponent = TetR_fc
  functionalComponent = LacI_fc
  interaction = LacITetR_int
)
```

Abstraction Layer Two

```
TetR is a ProteinComponent()
LacI is a ProteinComponent()

LacI_inverter is a ModuleDefinition()
(
  inhibition(LacI,TetR)
)
```

Adding Sequence Constraints

Abstraction Layer One

```
pTetR is a Promoter()
lacI_RBS is a RBS()

pTetR_c is a Component(pTetR)
lacI_RBS_c is a Component(lacI_RBS)

pair1 is a Precedes(pTetR_c, lacI_RBS_c)

tetRInverter is a DNAComponent()
(
  component = pTetR_c
  component = lacI_RBS_c
  sequenceConstraint = pair1
)
```


Abstraction Layer Two

```
pTetR is a Promoter()
lacI_RBS is a RBS()

tetRInverter is a DNAComponent()
(
  precedes(pTetR, lacI_RBS)
)
```

Mapping mapsTo

Abstraction Layer One

```
TetR is a ProteinComponent()

TetR_fc is a FunctionalComponent(TetR,inout)
TetR_lacinv_fc is a FunctionalComponent(TetR,inout)

TetR_map is a MapsUseLocal(TetR_lacinv_fc,TetR_fc)
toggleSwitch is a Module(TetR_inverter)
(
  mapsTo = TetR_map
)
```


Abstraction Layer Two

```
TetR is a ProteinComponent()
TetR_inv is a ProteinComponent()

toggleSwitch is a Module(TetR_inverter)
(
  description = "toggle switch"
  mapsUseLocal(TetR,TetR_inv)
)
```

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.