Chapter 6

Standardized Golden Gate Assembly Metadata Representation Using SBOL

Gonzalo Vidal, Carolus Vitalis, and Johan Guillén

Abstract

Synthetic biology, also known as engineering biology, is an interdisciplinary field that applies engineering principles to biological systems. One way to engineer biological systems is by modifying their DNA. A common workflow involves creating new DNA parts through synthesis and then using them in combination with other parts through assembly. Assembly standards such as MoClo, Phytobricks, and Loop are based on Golden Gate, and provide a framework for combining parts. The Synthetic Biology Open Language (SBOL) has implemented a best practice for representing build plans to communicate them to other practitioners through whiteboard designs and in a machine-readable format for communication with lab automation tools. Here we present a software tool for creating SBOL representations of build plans to simulate type IIS-mediated assembly reactions and store relevant metadata.

Key words DNA Assembly, Golden Gate, Standard, SBOL, Synthetic Biology

1 Introduction

Synthetic Biology or Engineering Biology is an interdisciplinary field that applies engineering methods to biological systems. In its foundation, it relies on the idea that researchers can "write" arbitrary DNA sequences through DNA synthesis, rather than being constrained to "copy" and "paste" or random mutations from existing DNA sequences found in nature. Although the cost of DNA synthesis is decreasing over time, it is still expensive and remains out of reach for laboratories with limited resources.

DNA assembly techniques allow researchers to use DNA synthesis more efficiently, thereby reducing the cost of building DNA libraries. Two of the main DNA assembly methods are Gibson assembly [1] and type IIS-mediated assembly [2]. Gibson assembly can join two blunt DNA fragments with a ~ 20–40 base pairs homology, using an exonuclease to create complementary sticky ends, a DNA polymerase to close the gaps, and a DNA ligase to seal

Daniel Schindler (ed.), *Golden Gate Cloning: Methods and Protocols*, Methods in Molecular Biology, vol. 2850,
https://doi.org/10.1007/978-1-0716-4220-7_6,

the nicks [1]. Golden Gate is a type IIS-mediated DNA assembly method that employs restriction enzymes to generate DNA fragments with ~3–4 base pair overhangs. Complimentary overhangs anneal and are joined by a DNA ligase that seal the nicks creating a desired DNA fragment or plasmid [2]. Although Gibson assembly is a scarless method, it can produce sequence errors during steps that use a DNA polymerase such as polymerase chain reaction (PCR) fragment amplification or while filling the gaps in DNA. The community primarily use Golden Gate assembly because it is simple, less prone to mutations, and cost-efficient. To allow for the reuse of synthetic DNA fragments, synthetic biologists created a set of DNA parts with a defined biological function and assembly standards to join them. Some basic DNA parts are promoter, ribosome binding site, coding sequence, and terminator. Assembly standards like MoClo [3, 4], Loop [5], Golden Braid [6], and others [7] defined the syntax or the sequence of the sticky ends that flanked a DNA part. Using assembly standards synthetic biologists can share libraries of DNA parts with other laboratories improving collaboration and reproducibility. Combining standard parts from the literature with newly synthesized parts by another group allows them to create combinatorial transcriptional units with fewer resources than by pure DNA synthesis. Having standard parts also benefits the laboratory producing them as they can reuse their DNA parts in future projects. The Golden Gate assembly protocol is part of the workflow of most synthetic biologists and is the way how iGEM has decided to distribute DNA [7]. Furthermore, due to its simple one-pot one-step protocol, it is easy to implement in automated workflows using liquid handling robots [8].

A digital representation of parts and assemblies would help researchers handle larger combinations of parts, design an assembly strategy, and simulate it. This digital representation has to be human-readable to inform researchers the sequences that will be assembled, as well as machine-readable, to provide an interface with lab automation. The Synthetic Biology Open Language (SBOL) can represent these biological designs in a machine-readable format [9, 10]. SBOL visual allows for the visual representation of meta data encoded in SBOL using glyphs to abstract standard objects and relations [11, 12]. SBOL 3 has implemented a best practice for the representation of build plans that can represent Golden Gate assemblies [13].

In the standard representation of parts and assembly for build planning, a terminology that has an SBOL representation where the sequence information and metadata are stored is described. A *Part* is defined as a single contiguous linear DNA construct with a completely specified sequence. There are many types of parts, the simplest being the Part Core, which is a part that has no interfaces nor relation to an assembly, such as the sequence of GFP. Then to

assemble it through Golden Gate following the CIDAR MoClo [3] standard for example, we need to design a *Part Insert* that can be the engineered fluorescent protein *Part Core* plus any 5′ and 3′ flanking sequences, in this case C (AATG) prefix and D (AGGT) suffix, to be placed into a designated insertion site of a *Backbone*. The *Backbone* is a DNA construct into which *Parts* are intended to be inserted at one or more insertion sites, for example, the Plasmid DVA_CD from the CIDAR MoClo kit. When the *Part Insert* is inserted into a *Backbone* insertion site, it creates a *Part in Backbone*, following the example it can be the engineered fluorescent protein in DVA_CD. Now this *Part in Backbone* can be used with other *Parts in Backbone* like a promoter, ribosome binding site (RBS), and terminator, and a *Backbone* in a Golden Gate *Assembly Plan*. When *Parts in Backbones* are digested by BsaI restriction enzyme, they create *Part Extracts*, which are parts plus any 5′ and 3′ flanking sequences that have been extracted from a *Part in Backbone*. On the other hand, the digestion of the *Backbone* will produce an *Open Backbone* and a *Drop-Out Sequence*. The *Part Extracts* and the Open Backbone will align by base complementarity and ligated producing a *Composite Part in Backbone* which is the final result of the assembly, a constitutive transcription unit that expresses the engineered fluorescent protein. All these terms store sequence information and metadata about functions and relations inside the Assembly Plan. For a more detailed description of the terminology and how metadata is stored, please refer to the publication [13].

Representing assembly plans in SBOL provides other benefits, such as making your data findable, accessible, interoperable, and reusable (FAIR), as well as the integration within the broader SBOL ecosystem. This ecosystem includes: SBOL Visual for hierarchical design visualization, the SynBioHub repository [14], and design tools such as SynBioSuite [15] and LOICA [16]. Here we present a software tool for the creation of build plan SBOL representations to simulate assembly reactions and store relevant metadata.

2 Materials

2.1 Dependencies

1. Installing SBOL-utilities via PIP will automatically install the required packages. These prerequisites include Python 3.7 or a later version. We recommend the use of an environment manager, such as Anaconda (https://www.anaconda.com/).
2. Additionally, users must install non-Python dependencies, Graphviz (https://graphviz.org/) to render graph-SBOL diagrams and Node.js (https://nodejs.org/en) to use the SBOL-converter.

2.2 Installation

1. The SBOL-utilities Python package is distributed using the Python Package Index (PyPI), which utilizes the Pip Installs Package (PIP) for installation and update management. The latest stable release of SBOL-utilities can be installed using the following commands (*see* **Note 1**):

```
pip3 install SBOL-utilities
```

2. To verify that the installation was successful, users should be able to run the following command with no errors

```
import SBOL-utilities
```

3 Methods

In the following sections, we detail how SBOL-utilities can be used to generate a Golden Gate representation. We exemplify its use by creating an assembly plan for one transcriptional unit (Fig. 1) and a collection with multiple fluorescent proteins (Fig. 2). To learn more about the advanced features, we recommend exploring the source repository https://github.com/SynBioDex/SBOL-utilities with more examples and tutorials, as well as the documentation https://SBOL-utilities.readthedocs.io

3.1 Creating an Assembly Plan for One Transcription Unit

To create an *Assembly Plan* using SBOL, we instantiate the restriction enzyme, the DNA *Parts in Backbone,* and *Backbone* involved in the reaction (*see* **Note 2**). Here we represent the Golde Gate assembly of the following *Parts in Backbone*: a constitutive promoter (J23100), an RBS (B0034), a green fluorescent protein (GFP), and a double terminator (B0015) in the *Backbone* odd receiver 1 (pOdd) from Loop assembly (Fig. 1). This process can be described step by step, as follows:

1. Create a SBOL document

```
doc = SBOL3.Document()
```

2. Set a namespace (*see* **Note 3**)

```
SBOL3.set_namespace('http://SBOLstandard.org')
```

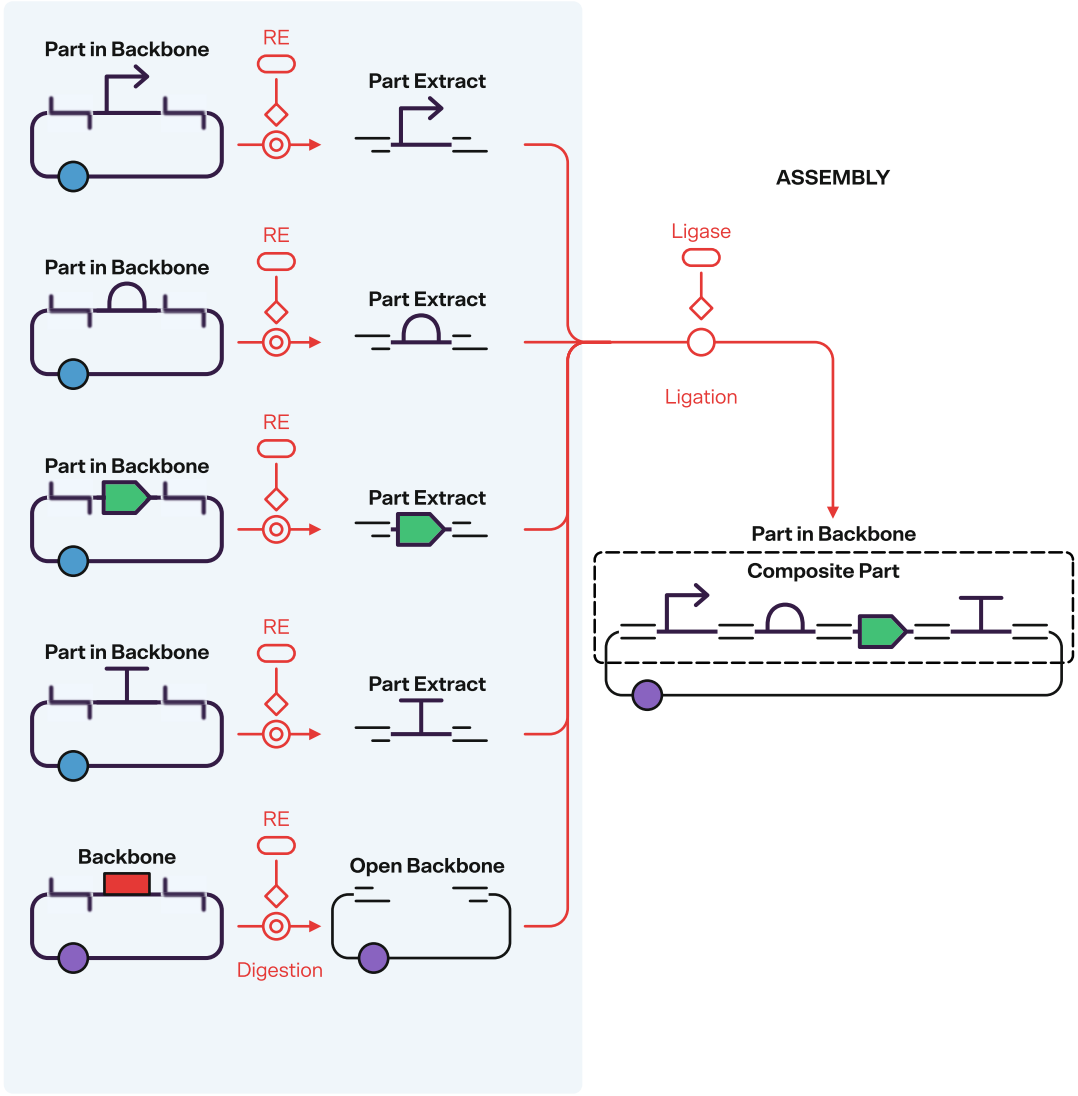

Fig. 1 Simple assembly SBOL visual representation. Using standardized build planning. In this Golden Gate assembly four *Parts in Backbone*: a promoter, an RBS, a CDSs and a Terminator, with an acceptor Backbone are mixed in a single reaction producing one transcription unit. Reagents and reactions are explicit in the diagram representing how the restriction enzyme (RE) regulates the digestion producing Part Extracts as an intermediate state; and how the ligase regulates the ligation producing a *Part in Backbone* with a *Composite Part* as the final product

3. Create a restriction enzyme (Table 1)

```
bsai = ed_restriction_enzyme('BsaI')
```

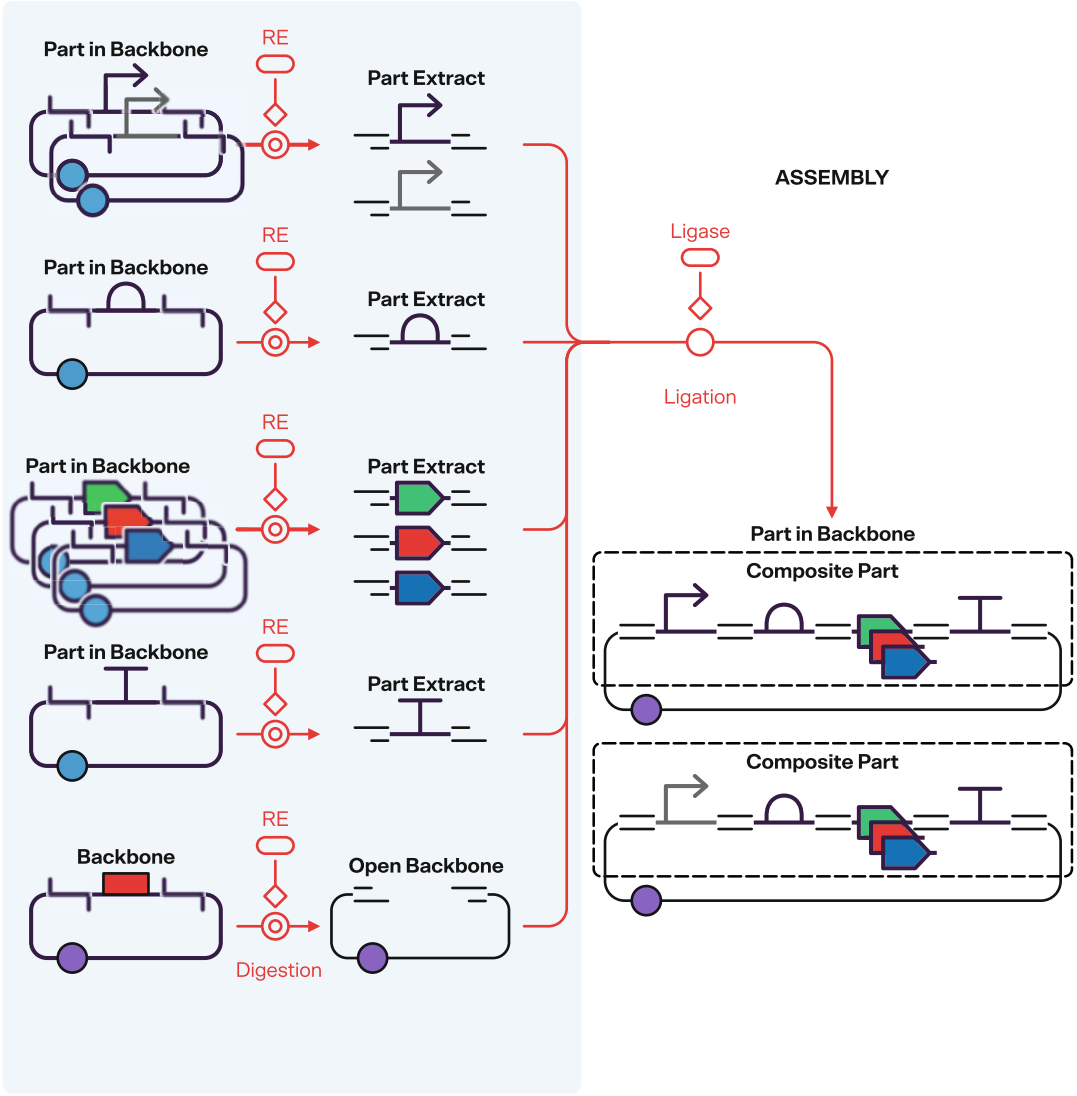

Fig. 2 Combinatorial assembly SBOL visual representation. Using standardized build planning. In this Golden Gate assembly seven *Parts in Backbone*, two promoters, one RBS, three CDSs and one Terminator, with an acceptor Backbone are mixed in six reactions producing a different transcription unit in each one in a combinatorial fashion. Reagents and reactions are explicit in the diagram representing how the restriction enzyme (RE) regulates the digestion producing Part Extracts; and how the ligase regulates the ligation producing *Parts in Backbone* with *Composite Parts*

4. Create an acceptor *Backbone* (Table 2)

```
podd1_dir = os.path.join('podd1.gb')      podd_doc = con-
vert_from_genbank(podd1_dir,
    'https://github.com/Gonza10V')
    podd_af = [top_level for top_level in podd_doc
```

Table 1
ed_restriction_enzyme

Argument	Description	Format/ values	Default
name	Name of the SBOL ExternallyDefined, used by PyDNA. Case sensitive, follow standard restriction enzyme nomenclature, i.e., "BsaI"	String	n/a
kwargs	Keyword arguments of any other ExternallyDefined attribute	n/a	n/a

Table 2
Backbone from SBOL

Argument	Description	Format/ Values	Default
identity	The identity of the Component. The identity of Sequence is also identity with the suffix '_seq'	String	n/a
SBOL_comp	The SBOL Component containing the DNA sequence to use	String	n/a
dropout_location	List of 2 integers that indicates the start and the end of the dropout sequence including overhangs. Note that the index of the first location is 1, as is typical practice in biology, rather than 0, as is typical practice in computer science	List [Integer]	n/a
fusion_site_length	Integer of the length of the fusion sites (eg. BsaI fusion site length is 4, SapI fusion site length is 3)	Integer	n/a
linear	Boolean that indicates if the backbone is linear, by default it is set to Flase which means that it has a circular topology	Boolean	False
kwargs	Keyword arguments of any other Component attribute	n/a	n/a

```
if type(top_level)==SBOL3.Component][0]
podd_backbone, podd_backbone_seq =
backbone_from_SBOL('pOdd_bb', podd_af,
[680,1770], 4, False, name='pOdd_bb')
doc.add([podd_backbone,podd_backbone_seq])
```

5. Obtain *Parts in Backbone* sequences from GeneBank files (*see* **Note 4**)

```
j23100_dir = os.path.join('ab_j23100.gb')      b0034_dir =
os.path.join('bc_b0034.gb')
    gfp_dir = os.path.join('ce_gfp.gb')
    b0015_dir = os.path.join('ef_b0015.gb')
```

6. Convert sequences to SBOL (*see* **Note 3**)

```
j23100_doc = convert_from_genbank
(j23100_dir,      'https://github.com/Gonza10V')
    j23100_ab = [top_level for top_level in j23100_doc
    if type(top_level)==SBOL3.Component][0]
    b0034_doc = convert_from_genbank(b0034_dir,
    'https://github.com/Gonza10V')
    b0034_bc = [top_level for top_level in b0034_doc
    if type(top_level)==SBOL3.Component][0]
    gfp_doc = convert_from_genbank(gfp_dir,
    'https://github.com/Gonza10V')
    gfp_ce = [top_level for top_level in gfp_doc
    if type(top_level)==SBOL3.Component][0]
    b0015_doc = convert_from_genbank(b0015_dir,
    'https://github.com/Gonza10V')
    b0015_ef = [top_level for top_level in b0015_doc
    if type(top_level)==SBOL3.Component][0]
```

7. Create standard *Parts in Backbone* (Table 3)

```
j23100_ab_in_bb, j23100_ab_in_bb_seq =      part_in_back-
bone_from_SBOL('j23100_ab_in_bb',
    j23100_ab, [479,513], [SBOL3.SO_PROMOTER], 4,
    False, name='j23100_ab_in_bb')
    doc.add([j23100_ab_in_bb, j23100_ab_in_bb_seq])
    b0034_bc_in_bb, b0034_bc_in_bb_seq =
    part_in_backbone_from_SBOL('b0034_bc_in_bb',
    b0034_bc, [479,499], [SBOL3.SO_RBS], 4, False,
    name='b0034_bc_in_bb')
    doc.add([b0034_bc_in_bb, b0034_bc_in_bb_seq])
    gfp_ce_in_bb, gfp_ce_in_bb_seq =
    part_in_backbone_from_SBOL('gfp_ce_in_bb',
    gfp_ce, [479,1195], [SBOL3.SO_CDS], 4, False,
    name='gfp_ce_in_bb')
    doc.add([gfp_ce_in_bb, gfp_ce_in_bb_seq])
    b0015_ef_in_bb, b0015_ef_in_bb_seq =
    part_in_backbone_from_SBOL('b0015_ef_in_bb',
    b0015_ef, [518,646], [SBOL3.SO_TERMINATOR], 4,
    False, name='b0015_ef_in_bb')
    doc.add([b0015_ef_in_bb, b0015_ef_in_bb_seq])
```

Table 3
Part in backbone from SBOL

Argument	Description	Format/Values	Default
identity	The identity of the Component. If it is a String it builds a new SBOL Component, and if it is None it adds on top of the input. The identity of Sequence is also identity with the suffix '_seq'	String—None	n/a
SBOL_comp	The SBOL3 Component that will be used to create the *part in backbone* Component and Sequence	SBOL3. Component	n/a
part_location	List of 2 integers that indicates the start and the end of the unitary part. Note that the index of the first location is 1, as is typical practice in biology, rather than 0, as is typical practice in computer science	List[Integer]	n/a
part_roles	List of strings that indicates the roles to add on the part	List[String]	n/a
fusion_site_length	Integer of the length of the fusion sites (eg. BsaI fusion site length is 4, SapI fusion site length is 3)	Integer	n/a
linear	Boolean that indicates if the backbone is linear, by default it is set to False which means that it has a circular topology	Boolean	False
kwargs	Keyword arguments of any other Component attribute	n/a	n/a

Table 4
Assembly plan composite in backbone single enzyme

Parameter	Description	Format/Values	Default
name	Name of the assembly plan Component	String	n/a
parts_in_backbone	Parts in backbone to be assembled	List[SBOL3. Component]	n/a
acceptor_backbone	Backbone in which parts are inserted into the assembly	SBOL3.Component	n/a
restriction_enzyme	Restriction enzyme with the correct name from Bio.Restriction as Externally Defined	String—SBOL3. ExternallyDefined	n/a
document	SBOL Document where the assembly plan will be created	SBOL3.Document	n/a

8. Create *Assembly Plan* (Table 4)

```
simple_assembly_plan =      Assembly_plan_composite_in_-
backbone_single_enzyme(
    name='simple_green_transcriptional_unit',
```

```
parts_in_backbone=[j23100_ab_in_bb,
b0034_bc_in_bb, gfp_ce_in_bb, b0015_ef_in_bb],
acceptor_backbone=podd_backbone,
restriction_enzyme=bsai,
document=doc)
simple_assembly_plan.run()
```

9. Inspect assembly products, its length should be 1 as the only product is a constitutive transcription unit expressing GFP, the Composite Part in Backbone J23100-B0034-GFP-B0015 in pOdd (Fig. 1).

```
len(simple_assembly_plan.products)
```

10. Inspect objects in assembly documents

```
for obj in      simple_assembly_plan.document.objects:
    print(obj.identity)
```

11. Inspect the sequence of an object. Select an object identity from the printed list and use it as check_obj. The user can swap object identities in check_obj to inspect the sequence of other objects. In this example, we selected the promoter J23100 Part Extract.

```
check_obj = 'http://SBOLstandard.org/testfiles/j23100_ab_
in_bb_part_extract'      for obj in simple_assembly_plan.doc-
ument.objects:
    if obj.identity == check_obj:
    print(obj.sequences[0].lookup().elements)
```

The output of this code is the string containing the sequence of the promoter J23100 and corresponds to the expected 43 bp long DNA sequence: "GGAGttgacggctagctcagtcctaggtacagtgctagcTACT".

This process can be iterated for all *Part Extract*, *Backbone,* and *Composite* to inspect its arrangement (Fig. 3).

3.2 Creating an Assembly Plan for a Combinatorial Design

Combinatorial designs allow researchers to simultaneously explore many conditions or variables systematically and efficiently. These approaches are used in synthetic biology to systematically generate diverse genetic constructs or pathways. This facilitates the screening of multiple variants to identify those with desired properties, such as enhanced productivity or improved function.

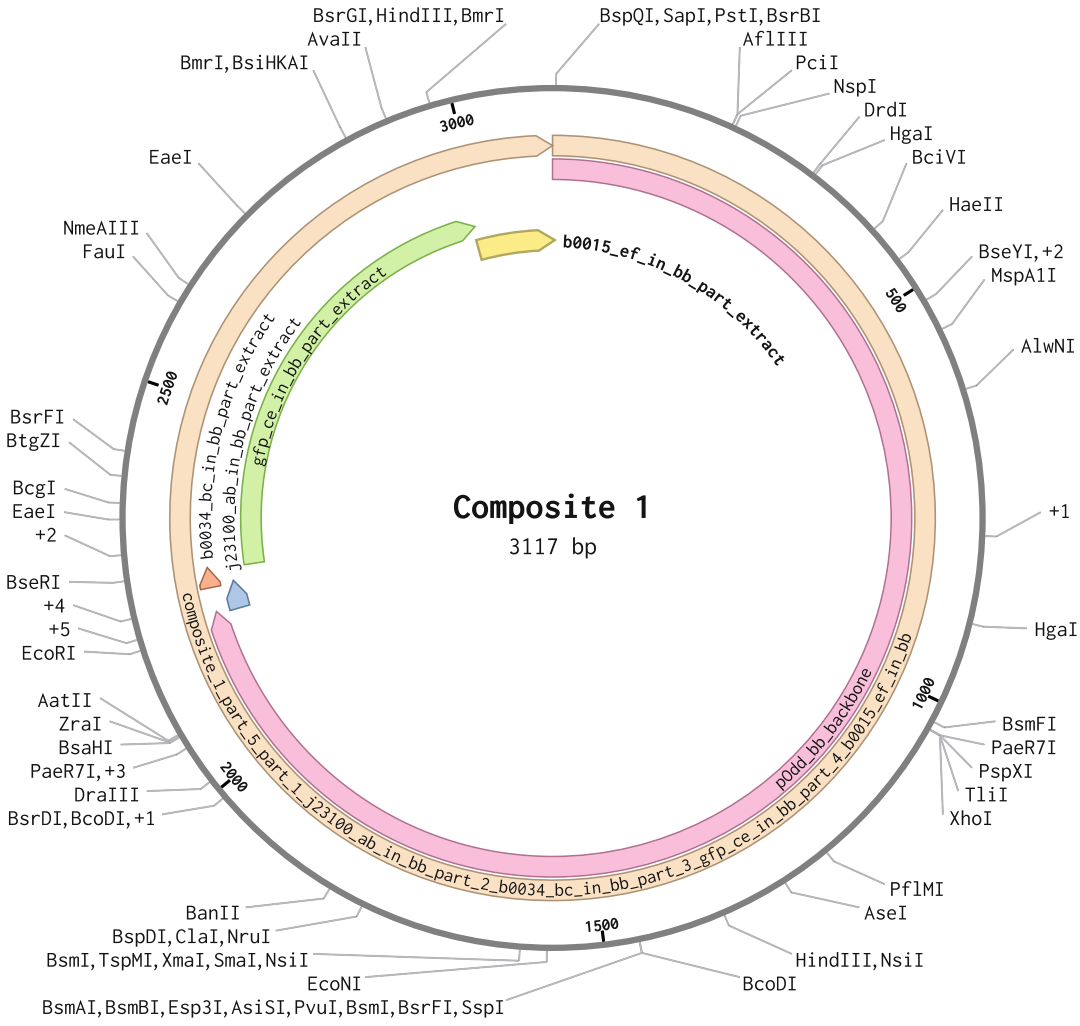

Fig. 3 Composite sequence diagram. DNA sequence of a *Composite* created using *Assembly plan composite in backbone single enzyme* was obtained from step 11 and uploaded to Benchling sequence manager. DNA sequences from *Composite*, *Part Extracts* and *Backbone* sequences were aligned and added as features using their identity as name

To create an *Assembly Plan* using SBOL, we instantiate the restriction enzyme, the DNA *Parts in Backbone*, and *Backbone* involved in the reaction (*see* **Note 2**). Here we represent the Golde Gate assembly of the following *Parts in Backbone*: two constitutive promoter variants (J23100 and J23101), an RBS (B0034), a green fluorescent protein (GFP), red fluorescent protein (RFP), cyan fluorescent protein (CFP), and a double terminator (B0015) in the *Backbone* odd receiver 1 (pOdd) from Loop assembly (Fig. 2). This process can be described step by step, as follows:

1. Create a SBOL document

```
doc = SBOL3.Document()
```

2. Set a namespace (*see* **Note 3**)

```
SBOL3.set_namespace('http://SBOLstandard.org')
```

3. Create a restriction enzyme (Table 1)

```
bsai = ed_restriction_enzyme('BsaI')
```

4. Create an acceptor *Backbone* (Table 2)

```
podd1_dir = os.path.join('podd1.gb')      podd_doc = con-
vert_from_genbank(podd1_dir,  'https://github.com/
Gonza10V')
    podd_af = [top_level for top_level in podd_doc
    if type(top_level)==SBOL3.Component][0]
    podd_backbone, podd_backbone_seq =
    backbone_from_SBOL('pOdd_bb', podd_af,
    [680,1770], 4, False, name='pOdd_bb')
    doc.add([podd_backbone,podd_backbone_seq])
```

5. Obtain *Parts in Backbone* sequences from GeneBank files (*see* **Note 4**)

```
j23100_dir = os.path.join('ab_j23100.gb')      j23101_dir =
os.path.join('ab_j23101.gb')
    b0034_dir = os.path.join('bc_b0034.gb')
    gfp_dir = os.path.join('ce_gfp.gb')
    rfp_dir = os.path.join('ce_mrfp1.gb')
    cfp_dir = os.path.join('ce_ecfp.gb')
    b0015_dir = os.path.join('ef_b0015.gb')
```

6. Convert sequences to SBOL (*see* **Note 3**)

```
j23100_doc = convert_from_genbank
(j23100_dir,      'https://github.com/Gonza10V')
    j23100_ab = [top_level for top_level in j23100_doc
    if type(top_level)==SBOL3.Component][0]
    j23101_doc = convert_from_genbank(j23101_dir,
```

```
'https://github.com/Gonza10V')
j23101_ab = [top_level for top_level in j23101_doc
if type(top_level)==SBOL3.Component][0]
b0034_doc = convert_from_genbank(b0034_dir,
'https://github.com/Gonza10V')
b0034_bc = [top_level for top_level in b0034_doc
if type(top_level)==SBOL3.Component][0]
gfp_doc = convert_from_genbank(gfp_dir,
'https://github.com/Gonza10V')
gfp_ce = [top_level for top_level in gfp_doc
if type(top_level)==SBOL3.Component][0]
rfp_doc = convert_from_genbank(rfp_dir,
'https://github.com/Gonza10V')
rfp_ce = [top_level for top_level in rfp_doc
if type(top_level)==SBOL3.Component][0]
cfp_doc = convert_from_genbank(cfp_dir,
'https://github.com/Gonza10V')
cfp_ce = [top_level for top_level in cfp_doc
if type(top_level)==SBOL3.Component][0]
b0015_doc = convert_from_genbank(b0015_dir,
'https://github.com/Gonza10V')
b0015_ef = [top_level for top_level in b0015_doc
if type(top_level)==SBOL3.Component][0]
```

7. Create standard *Parts in Backbone* (Table 3)

```
j23100_ab_in_bb, j23100_ab_in_bb_seq = part_in_back-
bone_from_SBOL('j23100_ab_in_bb',
    j23100_ab, [479,513], [SBOL3.SO_PROMOTER], 4,
    False, name='j23100_ab_in_bb')
    doc.add([j23100_ab_in_bb, j23100_ab_in_bb_seq])
    j23101_ab_in_bb, j23101_ab_in_bb_seq =
    part_in_backbone_from_SBOL('j23101_ab_in_bb',
    j23101_ab, [479,513], [SBOL3.SO_PROMOTER], 4,
    False, name='j23101_ab_in_bb')
    doc.add([j23101_ab_in_bb, j23101_ab_in_bb_seq])
    b0034_bc_in_bb, b0034_bc_in_bb_seq =
    part_in_backbone_from_SBOL('b0034_bc_in_bb',
    b0034_bc, [479,499], [SBOL3.SO_RBS], 4, False,
    name='b0034_bc_in_bb')
    doc.add([b0034_bc_in_bb, b0034_bc_in_bb_seq])
    gfp_ce_in_bb, gfp_ce_in_bb_seq =
    part_in_backbone_from_SBOL('gfp_ce_in_bb',
    gfp_ce, [479,1195], [SBOL3.SO_CDS], 4, False,
    name='gfp_ce_in_bb')
    doc.add([gfp_ce_in_bb, gfp_ce_in_bb_seq])
```

```
rfp_ce_in_bb, rfp_ce_in_bb_seq =
part_in_backbone_from_SBOL('rfp_ce_in_bb',
rfp_ce, [479,1156], [SBOL3.SO_CDS], 4, False,
name='rfp_ce_in_bb')
doc.add([rfp_ce_in_bb, rfp_ce_in_bb_seq])
cfp_ce_in_bb, cfp_ce_in_bb_seq =
part_in_backbone_from_SBOL('cfp_ce_in_bb',
cfp_ce, [479,1198], [SBOL3.SO_CDS], 4, False,
name='cfp_ce_in_bb')
doc.add([cfp_ce_in_bb, cfp_ce_in_bb_seq])
b0015_ef_in_bb, b0015_ef_in_bb_seq =
part_in_backbone_from_SBOL('b0015_ef_in_bb',
b0015_ef, [518,646], [SBOL3.SO_TERMINATOR], 4,
False, name='b0015_ef_in_bb')
doc.add([b0015_ef_in_bb, b0015_ef_in_bb_seq])
```

8. Create *Assembly Plan* (Table 4)

```
combinatorial_assembly_plan =
Assembly_plan_composite_in_backbone_single_enzyme
(     name='combinatorial_rgb_transcriptional_units',
      parts_in_backbone=[j23100_ab_in_bb,
      j23101_ab_in_bb, b0034_bc_in_bb, gfp_ce_in_bb,
      rfp_ce_in_bb, cfp_ce_in_bb, b0015_ef_in_bb],
      acceptor_backbone=podd_backbone,
      restriction_enzyme=bsai,
      document=doc)
      combinatorial_assembly_plan.run()
```

9. Inspect assembly products; its length should be 6 as the products are constitutive transcription units expressing GFP, RFP, and CFP the *Composite Parts in Backbone* J23100-B0034-GFP-B0015 in pOdd, J23100-B0034-RFP-B0015 in pOdd, J23100-B0034-CFP-B0015 in pOdd, J23101-B0034-GFP-B0015 in pOdd, J23101-B0034-RFP-B0015 in pOdd, and J23101-B0034-CFP-B0015 in pOdd (Fig. 2).

```
len(combinatorial_assembly_plan.products)
```

10. Inspect objects in *Assembly Plan* documents

```
for obj in      combinatorial_assembly_plan.document.
objects:
      print(obj.identity)
```

11. Inspect the sequence of an object. Select an object identity from the printed list and use it as check_obj. The user can swap object identities in check_obj to inspect the sequence of other objects. In this example, we selected the promoter J23100 Part Extract.

```
check_obj = 'http://SBOLstandard.org/testfiles/j23100_ab_
in_bb_part_extract'      for obj in simple_assembly_plan.doc-
ument.objects:
    if obj.identity == check_obj:
    print(obj.sequences[0].lookup().elements)
```

The output of this code is the string containing the sequence of the promoter J23100 and corresponds to the expected 43 bp long DNA sequence: "GGAGttgacggctagctcagtcctaggtacagtgctagcTACT".

This process can be iterated for all *Part Extract*, *Backbone* and *Composite* to inspect its arrangement (Fig. 3).

4 Notes

1. The version of SBOL-utilities used in this example is 1.0a17 in this branch (https://github.com/Gonza10V/SBOL-utilities/tree/gonzalo-internship3).
2. The DNA parts can be a linear or circular vectors.
3. The namespace can be any valid internationalized resource identifier (IRI).
4. The paths used in the example assume that the file is in the same folder as the Python file. Be mindful of the file location and format to use it here. When running the Jupyter Notebook if you get the error "SSLCertVerificationError," please add the following two lines of code at the top of the script:
   import ssl ssl._create_default_https_context = ssl._create_-unverified_context If you don't have the package ssl, you have to install it by running the following command in the terminal: pip install ssl.
5. The script uses the Python library Biopyhton, if you don't have it in your environment, please install it running the following command in the terminal: pip install Bio.

Acknowledgements

The authors would like to acknowledge Jacob Beal, for all the guidance and feedback provided during the creation of this work

References

1. Gibson DG, Young L, Chuang RY, Venter JC, Hutchison CA, Smith HO (2009) Enzymatic assembly of DNA molecules up to several hundred kilobases. Nat Methods 6(5):343–345. https://doi.org/10.1038/nmeth.1318
2. Engler C, Kandzia R, Marillonnet S (2008) A one pot, one step, precision cloning method with high throughput capability. PLoS One 3(11). https://doi.org/10.1371/journal.pone.0003647
3. Iverson SV, Haddock TL, Beal J, Densmore DM (2016) CIDAR MoClo: Improved MoClo assembly standard and new E. coli part library enable rapid combinatorial design for synthetic and traditional biology. ACS Synth Biol 5(1):99. https://doi.org/10.1021/acssynbio.5b00124
4. Weber E, Engler C, Gruetzner R, Werner S, Marillonnet S (2011) A modular cloning system for standardized assembly of multigene constructs. PLoS One 6(2):e16765. https://doi.org/10.1371/journal.pone.0016765
5. Pollak B et al (2019) Universal loop assembly: Open, efficient and cross-kingdom DNA fabrication. Synth Biol 5(1):1–13. https://doi.org/10.1093/synbio/ysaa001
6. Sarrion-Perdigones A et al (2011) GoldenBraid: An iterative cloning system for standardized assembly of reusable genetic modules. PLoS One 6(7):e21622. https://doi.org/10.1371/journal.pone.0021622
7. Bird JE, Marles-Wright J, Giachino A (2022) A user's guide to Golden Gate cloning methods and standards. ACS Synth Biol 11(11):3551–3563. https://doi.org/10.1021/acssynbio.2c00355
8. Bryant JA, Kellinger M, Longmire C, Miller R, Wright RC (2023) AssemblyTron: flexible automation of DNA assembly with Opentrons OT-2 lab robots. Synth Biol 8(1). https://doi.org/10.1093/synbio/ysac032
9. McLaughlin JA et al (2020) The Synthetic Biology Open Language (SBOL) Version 3: simplified data exchange for bioengineering. Front Bioeng Biotechnol 8(September):1–15. https://doi.org/10.3389/fbioe.2020.01009
10. Buecherl L et al (2023) Synthetic biology open language (SBOL) *version 3.1.0.* J Integr Bioinform 20(1):20220058. https://doi.org/10.1515/jib-2022-0058
11. Quinn JY et al (2015) SBOL visual: a graphical language for genetic designs. PLoS Biol 13(12):e1002310. https://doi.org/10.1371/journal.pbio.1002310
12. Baig H et al (2021) Synthetic biology open language visual (SBOL Visual) version 2.3. J Integr Bioinform 18(3). https://doi.org/10.1515/jib-2020-0045
13. Beal J et al (2023) Standardized representation of parts and assembly for build planning. ACS Synth Biol 12:3655. https://doi.org/10.1021/ACSSYNBIO.3C00418
14. McLaughlin JA et al (2018) SynBioHub: a standards-enabled design repository for synthetic biology. ACS Synth Biol 7(2):682–688. https://doi.org/10.1021/ACSSYNBIO.7B00403/ASSET/IMAGES/LARGE/SB-2017-004037_0006.JPEG
15. Sents Z, Stoughton TE, Buecherl L, Thomas PJ, Fontanarrosa P, Myers CJ (2023) SynBioSuite: a tool for improving the workflow for genetic design and modeling. ACS Synth Biol 12(3):892–897. https://doi.org/10.1021/ACSSYNBIO.2C00597/ASSET/IMAGES/LARGE/SB2C00597_0002.JPEG
16. Vidal G, Vitalis C, Rudge TJ (2021) LOICA: integrating models with data for genetic network design automation. ACS Synth Biol 2022. https://doi.org/10.1021/ACSSYNBIO.1C00603/ASSET/IMAGES/LARGE/SB1C00603_0003.JPEG