

**SBOLDESIGNER: A HIERARCHICAL GENETIC DESIGN  
EDITOR**

by  
Michael Zhang

A thesis submitted to the faculty of  
The University of Utah  
in partial fulfillment of the requirements for the degree of

Bachelor's of Science

Computer Science  
The University of Utah  
April 2018

Approved:

\_\_\_\_\_/\_\_\_\_\_  
Chris Myers  
Advisor

\_\_\_\_\_/\_\_\_\_\_  
H. James de St. Germain  
Director of Undergraduate Studies  
School of Computing

\_\_\_\_\_/\_\_\_\_\_  
Ross Whitaker  
Director  
School of Computing

## ABSTRACT

Synthetic biology, as a field of research, applies electrical engineering, systems biology, and bioinformatics to genetic circuit design. Software tools are leveraged to provide rapid iteration through the design space, and data standards are used to encode and characterize complicated genetic circuit designs. Specifically, genetic design automation workflows centered around standards, abstraction, and decoupling are utilized to help experimental biologists accomplish their goals. Unfortunately, the software tools that support this workflow are lacking in some critical features such as combinatorial design and support for an extended range of glyphs. These inadequacies hinder the adoption of data standards, and therefore hurt the reproducibility of experiments and results. Necessary details of experiments are not recorded, and the resulting conclusions are therefore not trusted. SBOLDesigner, a sequence-based computer aided design tool, addresses these issues. This thesis will focus on SBOLDesigner's implementation of combinatorial design and the SBOL Visual 2 standard using the SBOL 2 data model. Using SBOLDesigner, experimental biologists are able to visualize their genetic circuits unambiguously and express the full state of their design robustly. This results in higher productivity when designing genetic circuits, more comprehensive circuit descriptions, and most importantly, enhanced reproducibility in the field of synthetic biology.

# CONTENTS

|                                               |           |
|-----------------------------------------------|-----------|
| <b>ABSTRACT</b> .....                         | <b>ii</b> |
| <b>ACKNOWLEDGEMENTS</b> .....                 | <b>iv</b> |
| <b>CHAPTERS</b>                               |           |
| <b>1. INTRODUCTION</b> .....                  | <b>1</b>  |
| 1.1 Background .....                          | 1         |
| 1.2 Genetic Circuits .....                    | 2         |
| 1.3 The Synthetic Biology Open Language ..... | 5         |
| 1.4 Genetic Design Automation Workflow .....  | 6         |
| 1.5 Thesis Contributions .....                | 9         |
| 1.6 Thesis Overview .....                     | 10        |
| <b>2. SBOLDESIGNER</b> .....                  | <b>11</b> |
| 2.1 Background .....                          | 11        |
| 2.2 libSBOLj .....                            | 13        |
| 2.3 SBOL 2 .....                              | 13        |
| <b>3. SBOL VISUAL</b> .....                   | <b>18</b> |
| 3.1 Background .....                          | 18        |
| 3.2 SBOL Visual 2 .....                       | 20        |
| <b>4. COMBINATORIAL DESIGN</b> .....          | <b>24</b> |
| 4.1 Background .....                          | 24        |
| 4.2 Implementation .....                      | 25        |
| 4.2.1 Data Model .....                        | 25        |
| 4.2.2 User Interface .....                    | 27        |
| 4.2.3 Algorithm .....                         | 28        |
| 4.3 Combinatorial Design Example .....        | 31        |
| <b>5. CONCLUSION</b> .....                    | <b>33</b> |
| 5.1 Summary .....                             | 33        |
| 5.2 Future Work .....                         | 34        |
| 5.2.1 Interactions .....                      | 34        |
| 5.2.2 Computer Aided Manufacturing .....      | 34        |
| 5.2.3 Plugin Support .....                    | 34        |
| 5.2.4 Better Search .....                     | 35        |
| <b>REFERENCES</b> .....                       | <b>36</b> |

## **ACKNOWLEDGEMENTS**

The author would like to thank Professor Myers for being the principal investigator of this research. The author would also like to thank Evren Sirin (Complexible), Michal Galdzicki (Arzeda), Bryan Bartley (U. of Washington), and John Gennari (U. of Washington) for their work on the original version of SBOLDesigner. This work is funded by the National Science Foundation under grants CCF-1218095 and DBI-1356041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

# CHAPTER 1

## INTRODUCTION

### 1.1 Background

Synthetic biology is a relatively new field born out of systems biology, electrical engineering, and bioinformatics [5]. Specifically, synthetic biology enhances these parent fields by taking fundamental engineering principles such as standards, abstraction, and decoupling, and applying them to genetic circuit design. Systems biology, defined as the study of complex biological systems and their pathways of operation and interaction, is used to accelerate the creation of new and novel genetic circuits. Electrical engineering concepts such as circuits and logic gates function as great models for the design process. Finally, synthetic biology borrows from bioinformatics and makes use of massive amounts of data and computational resources.

The direct precursor to synthetic biology was genetic engineering, which focused on modifying organisms' genomes in order to manipulate the characteristics of those organisms. While genetic engineering has existed since the 1960s, it has never really adopted true engineering fundamentals. Therefore, even though synthetic biology is a re-branding of genetic engineering, the focus is to adopt standards, abstraction, and decoupling. This effort can be seen in the adoption of the Synthetic Biology Open Language (SBOL) standard, the use of software tools to rapidly iterate on the design of genetic circuits and their models, simulations, and compositions, and the separation of different tools for different tasks.

SBOL also addresses the problem of reproducibility in synthetic biology. Experiments are inherently complex, and information necessary for reproducing the results found in groundbreaking papers is often incomplete. As a result, much of the research in this field is of reduced value to the larger scientific community. Data models such as SBOL and

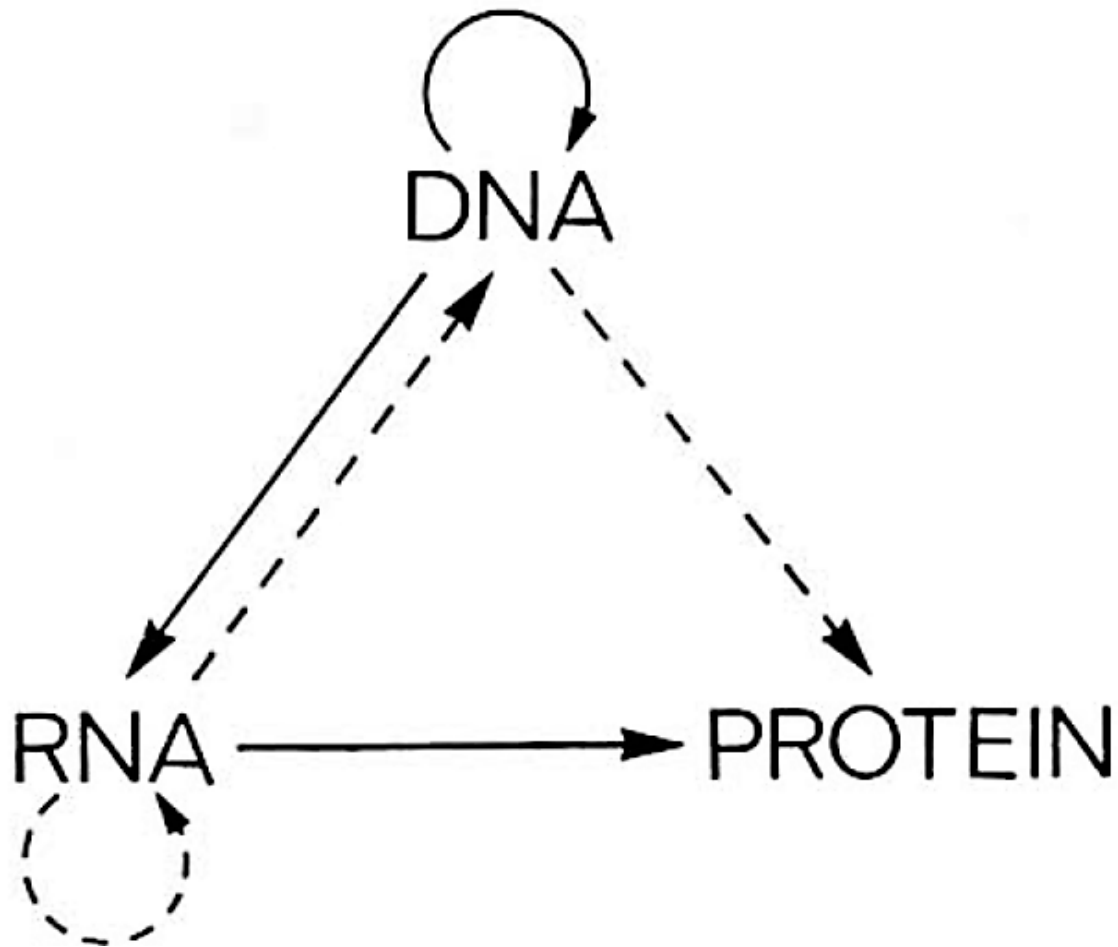
tools that utilize the data model are therefore integral in providing a means to address the issue of reproducibility. Without these blueprints, genetic circuits lack DNA sequence information, proper characterization data, and circuit layout details. To more thoroughly solve the reproducibility problem, tools that utilize SBOL must also support the wide range of ways experimental biologists express their designs.

## 1.2 Genetic Circuits

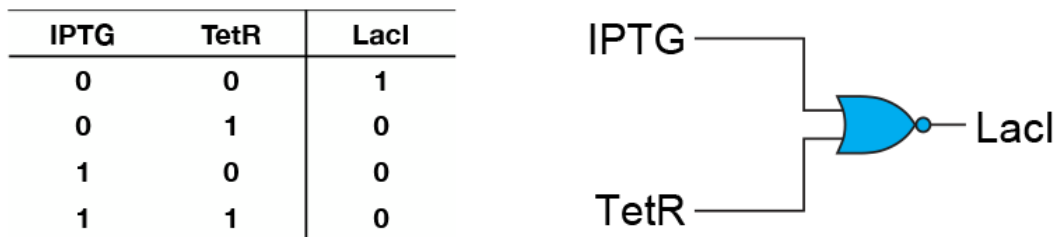
In synthetic biology, genetic circuits are classified as sensor and actuator networks. In the traditional electrical circuit, inputs such as buttons or switches control outputs such as LEDs and motors. This is accomplished through the physical and electrical properties of a variety of components such as resistors, capacitors, inductors, and LEDs. In genetic circuits, much is the same: standardized sequences of DNA control expression of various proteins that results in external factors like cells glowing green [7]. The biggest difference is that the mode of expression is due to the transcription and translation of genes into protein, and not of electrons flowing through wires. This is commonly referred to as the central dogma of molecular biology [3], and is shown in Figure 1.1. Because of these similarities, genetic circuits with similar function to traditional electrical circuits can be built.

For example, one such circuit is the NOR gate. A truth table and electrical circuit schematic is shown in Figure 1.2. IPTG and TetR are the inputs, and LacI is the output. The truth table shows that LacI is only high when both inputs are low. In every other case, LacI is low. The boolean NOR function can also be obtained from a genetic circuit, as shown in Figure 1.3. The circuit schematic describes a backbone of DNA with four glyphs of DNA sequences, defined as parts, drawn on top of it. The yellow bent arrow is called a *promoter*, and facilitates transcription of DNA to RNA. The half circle *ribosome binding site*, arrow *coding sequence*, and T shaped *terminator* are all parts that get transcribed into RNA. From there, a ribosome binds to the *ribosome binding site* and translates the *coding sequence* into a protein before falling off due to the *terminator*. In this case, the *coding sequence* contains the blueprint for the LacI protein. These standard parts are analogous to the electrical components of an electrical circuit.

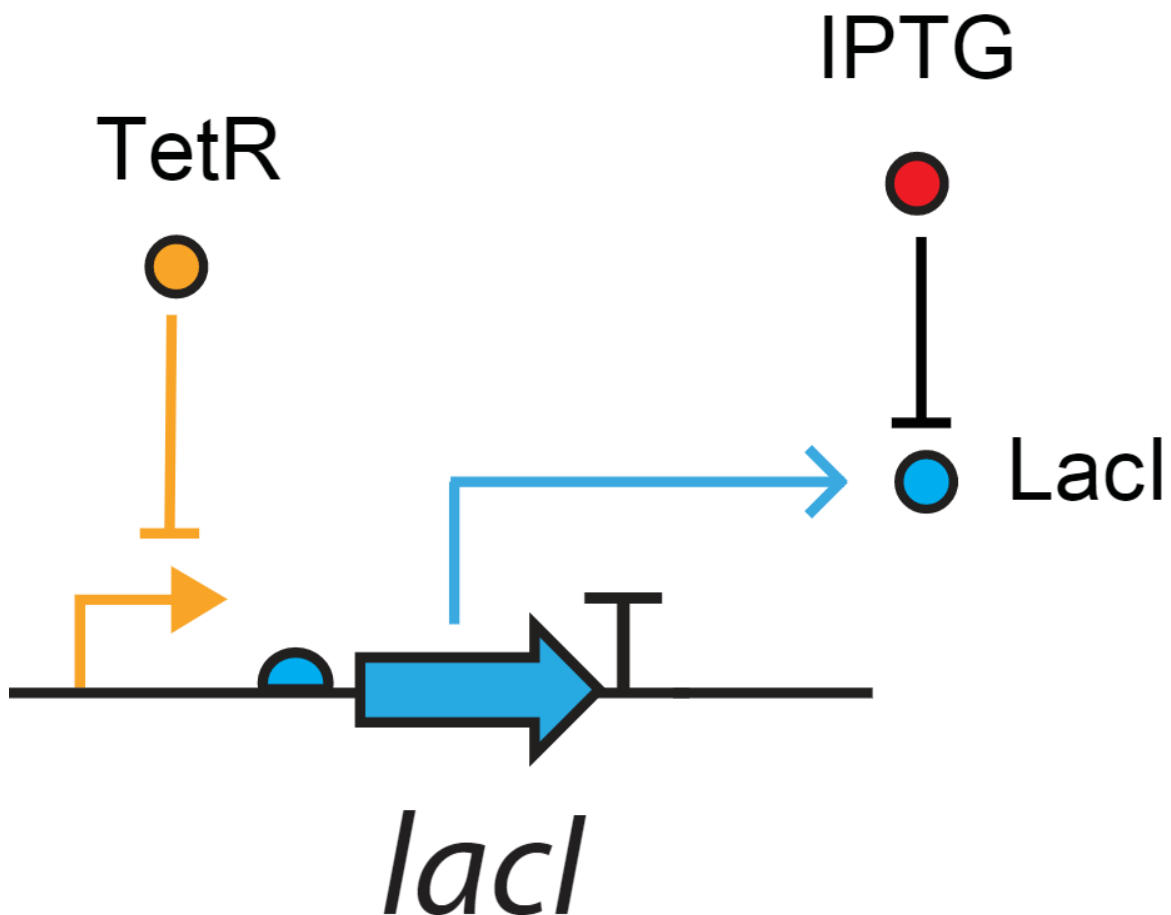
With the foregoing understanding of the these genetic parts' behaviors, the boolean NOR function is realized. TetR is a protein that inhibits the function of the *promoter*, and



**Figure 1.1.** The central dogma of molecular biology. DNA is transcribed into RNA, and RNA is translated into protein. These transformations are shown by the solid arrows, and are called the general transfers. The transformations shown by the dotted arrows are called special transfers, and differ from general transfers in that they don't occur in most cells, but may occur in special circumstances. In genetic engineering and synthetic biology, experimental biologists change the DNA, and the resulting protein is altered. For example, by modifying a genome's DNA to encode for green fluorescent protein, researchers can determine if their genetic circuit is operating correctly by shining ultraviolet light on the cells [7]. Figure from Crick et al. [3]



**Figure 1.2.** The truth table and classical schematic drawing for a NOR gate. The inputs are IPTG and TetR, and the output is LacI. LacI is only produced when TetR is low, and LacI is only expressed when IPTG is not present. Therefore, the only case where LacI is found in the cell is when both TetR and IPTG are low. Figure from Nguyen et al. publication pending.



**Figure 1.3.** The biological circuit schematic of a traditional NOR gate. TetR is repressing the promoter, and IPTG binds with LacI to form a complex that removes LacI from the system. When neither TetR nor IPTG are present, LacI can be produced normally and is expressed by being present in the system. Figure from Nguyen et al. publication pending.



IPTG is a small molecule that binds to LacI and effectively removes it from the system. When TetR is present, LacI doesn't get produced in the first place; when IPTG is present, LacI is neutralized before it can act. The states where LacI is low coincides with the rows in the truth table in Figure 1.2 that show a zero for LacI. When neither TetR nor IPTG are present, LacI can be expressed, resulting in a high state. This high state is the only possible state where LacI is high, and coincides with the row in the truth table in Figure 1.2 that shows a one for LacI.

The NOR gate is a universal gate, meaning all other gates and boolean systems can be built just from NOR. For example, a multiple input AND function could be built [11]. The inputs could be proteins associated with carcinogenic cells, and the output could be green fluorescent protein. When all the inputs are present, all the cancer cells would glow green. Alternatively, the function could be NAND, and cause some vital protein for the cancer cell to stop being produced. Now that we have realized a genetic NOR circuit, all other boolean systems can theoretically be built using genetic circuits.

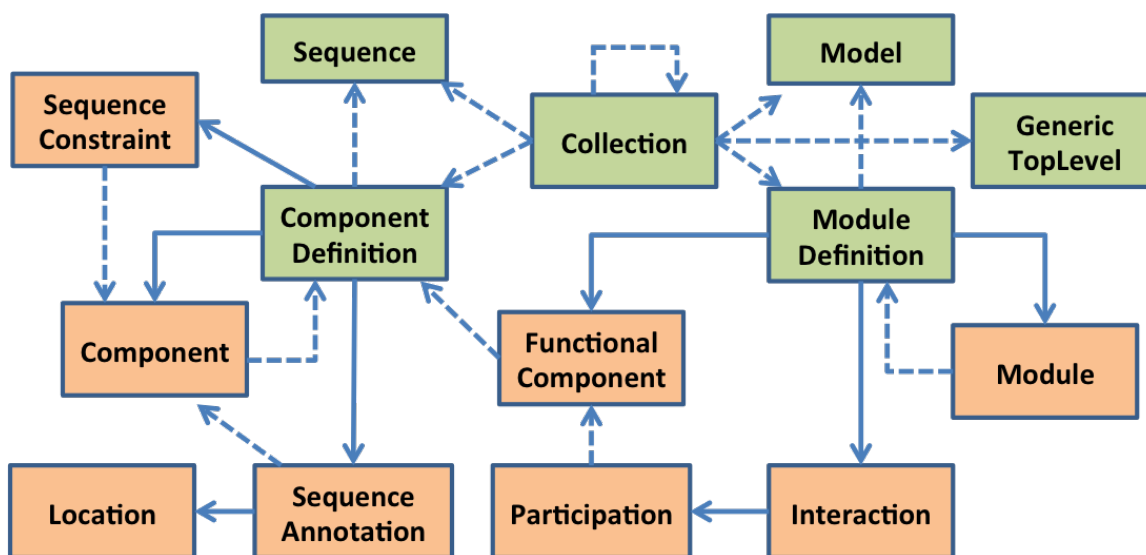
### 1.3 The Synthetic Biology Open Language

SBOL is a standard for describing these genetic circuits [6]. SBOL consists of two parts: the SBOL data model and the SBOL Visual standard [1, 18]. The data model, as shown in Figure 1.4, is a specification of objects and their relationships. Circuits such as the genetic NOR in Figure 1.3 can be encoded in SBOL and passed around as an electronic file. This centralized file format allows software tools to communicate with each other.

The SBOL Visual standard provides a standardized set of schematic glyphs to describe visualizations of genetic circuits. For example, Figure 1.3 is an SBOL Visual compliant depiction of a genetic circuit. Specifically, the genetic parts are drawn in accordance to how their glyphs are defined. These definitions are shown in Figure 1.5.

Both the SBOL data and visual standards are necessary for reproducibility and interoperability in synthetic biology, and further shows how engineering principles have influenced the field. When SBOL is not used to describe research results and genetic circuit layouts, the information published in papers is usually incomplete. Incomplete knowledge of the genetic system results in unreproducible results and a loss in trust for the findings. For this reason, in 2016, *ACS Synthetic Biology* set a precedent by adopting the

SBOL data and visual standards as the official method for depicting and digitally storing genetic constructs [9]. The use of SBOL in publications and the deposition of this data into public repositories tremendously aids reproducibility in this field. However, in order for biologists to generate these designs in SBOL, they need a workflow with tools that have features that enable a straightforward way for generating these constructs [19].










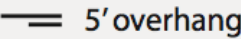

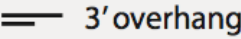

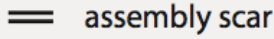



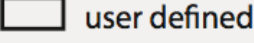



**Figure 1.4.** The SBOL data model. Top level classes are drawn in green, and supporting classes are drawn in yellow. Together, this data specification allows the description of a broad range of genetic circuits. Software tools import and export this format to allow for interoperability and decoupling from tool to tool. Figure from Beal et al. [1].

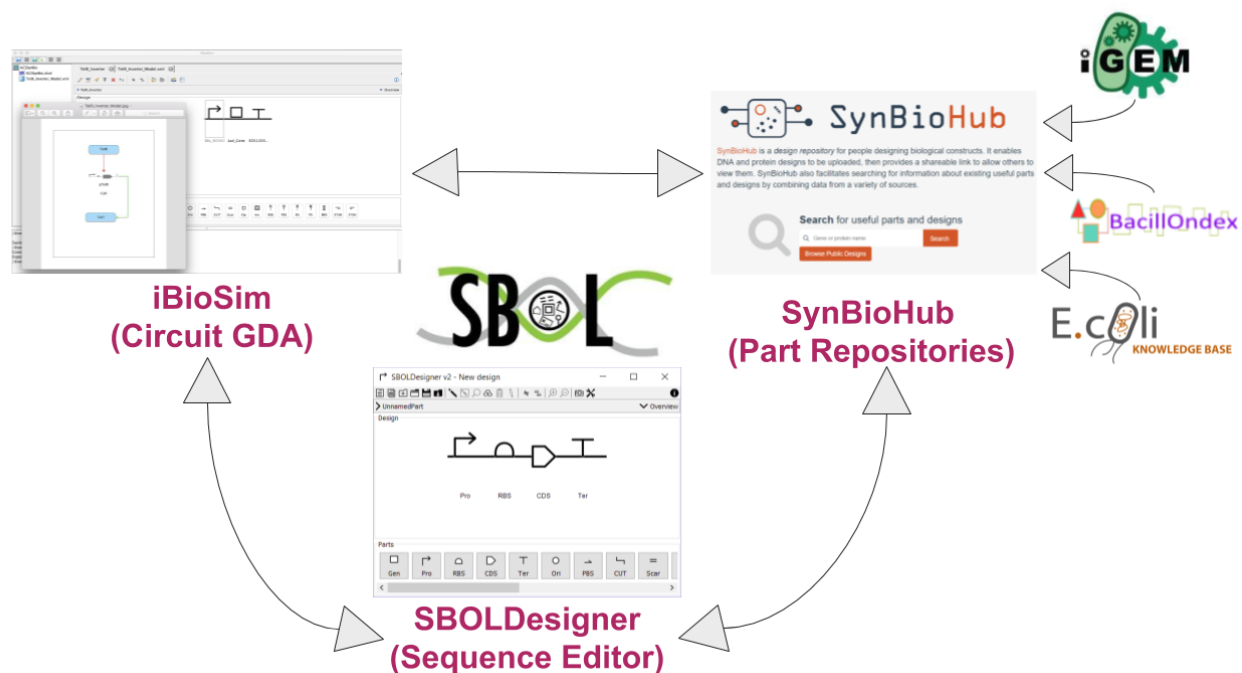
## 1.4 Genetic Design Automation Workflow

Figure 1.6 shows an example genetic design automation workflow that revolves around SBOL. Part repositories, simulation and modeling tools, and sequence level computer aided design tools interact through the SBOL standard. Software tools help experimental biologists abstract their designs and more efficiently prototype their circuits [19]. These tools can also be decoupled by allowing SBOL to be the common language. Because of SBOL, even a tool developed in isolation can contribute to the workflow.

One critical feature that is not present in the workflow is the ability to create combinatorial designs. Combinatorial design is a method of genetic circuit optimization where many variants of the same template part are synthesized. The performance characteristics

|                                                                                                               |                                                                                                           |
|---------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
|  promoter                    |  origin of replication   |
|  cds                         |  primer binding site     |
|  ribosome entry site         |  blunt restriction site  |
|  terminator                  |  sticky restriction site |
|  operator                    |  5' overhang            |
|  insulator                  |  3' overhang           |
|  ribonuclease site         |  assembly scar        |
|  rna stability element     |  signature            |
|  protease site             |  user defined         |
|  protein stability element |                                                                                                           |

**Figure 1.5.** The SBOL Visual set of defined glyphs. Each glyph represents a type of genetic part that genetic circuits are built from. The visual standard specifies rules and best practices for how these parts should be drawn in software tools, in figures, and on whiteboards. Figure from Quinn et al. [18].



**Figure 1.6.** A workflow consisting of SynBioHub, SBOLDesigner, and iBioSim [13]. Genetic parts from various databases are hosted in the SynBioHub repository [8, 12, 14]. SBOLDesigner and iBioSim can download these parts and use them to construct complete genetic designs. Specifically, iBioSim takes care of modeling and simulation and SBOLDesigner takes care of sequence level design. Figure from Zhang et al. [21].

of each variation are then measured and ranked. This method works well because biology is inherently noisy. Instead of designing a single circuit to achieve an intended result, many variants of the same circuit are built and compared against each other; the best performing variant is then selected. Because each variant is its own unique design, representing this method in SBOL would require many SBOL data files. Each file would encode a single variant, and much of the data would be duplicated.

Another missing piece of the workflow is its limited ability to express genetic circuits visually. An updated version of the SBOL Visual standard, SBOL Visual 2, has been released [18]. This update contains modifications to many of the already established glyphs, and introduces many more. It also contains information on how to visually display both hierarchical and functional relationships between parts.

## 1.5 Thesis Contributions

This thesis will describe SBOLDesigner's upgraded SBOL 2 internal data model, the implementation of combinatorial designs, and the adoption of SBOL Visual 2 within this genetic design automation workflow.

The SBOLDesigner tool was an early adopter of SBOL. The original version, developed at the University of Washington, relied on a heavily modified version of libSBOLj 1.0, a Java library that implemented the original version of SBOL. libSBOLj 2.0 represents a fundamental shift in both the SBOL data model and library APIs. As a result, updating the internal data model of SBOLDesigner to SBOL 2 required reimplementing much of SBOLDesigner's backend. This involved rethinking how SBOL fits into the CAD tool's goals and data representations. This new version of SBOLDesigner has therefore thoroughly adopted SBOL 2's architecture of how biological design should be represented.

Adoption of SBOL Visual 2 required modifying, updating, and creating new glyphs, new user interfaces, and new uses of the SBOL data model. The implications of these benefits will be discussed in the frame of increasing reproducibility in synthetic biology. In particular, visualizations of genetic circuit designs in SBOLDesigner are now much more clear and semantically well defined. Many existing glyphs have been updated to conform to more strict definitions and meanings, and many new glyphs have been added.

Support for combinatorial design within SBOLDesigner required exploring the rela-

tionship between SBOL's representation of the combinatorial design objects and SBOLDesigner's internal data model. New user interfaces also had to be created to give the user an intuitive way to build combinatorially designed genetic circuits. Finally, a combinatorial design enumeration algorithm was devised to explore the design space implied by the combinatorial design. The end result is a cohesive implementation of combinatorial design within SBOLDesigner.

## 1.6 Thesis Overview

The results of these changes are a succinct and descriptive way of specifying combinatorial designs and less ambiguous visualizations of genetic circuits, all in the SBOL 2 data model. Chapter 2 covers SBOLDesigner, its history, and its new SBOL 2 backend. Chapter 3 goes through the improved user interface to adopt SBOL Visual 2 over SBOL Visual 1. Chapter 4 describes combinatorial design as a technique for genetic circuit optimization, and how SBOLDesigner implements combinatorial design through SBOL data model objects, user interface extensions, and the underlying combinatorial enumeration algorithm. Finally, Chapter 5 concludes with a summary of the thesis and an overview of possibilities for future work.

## CHAPTER 2

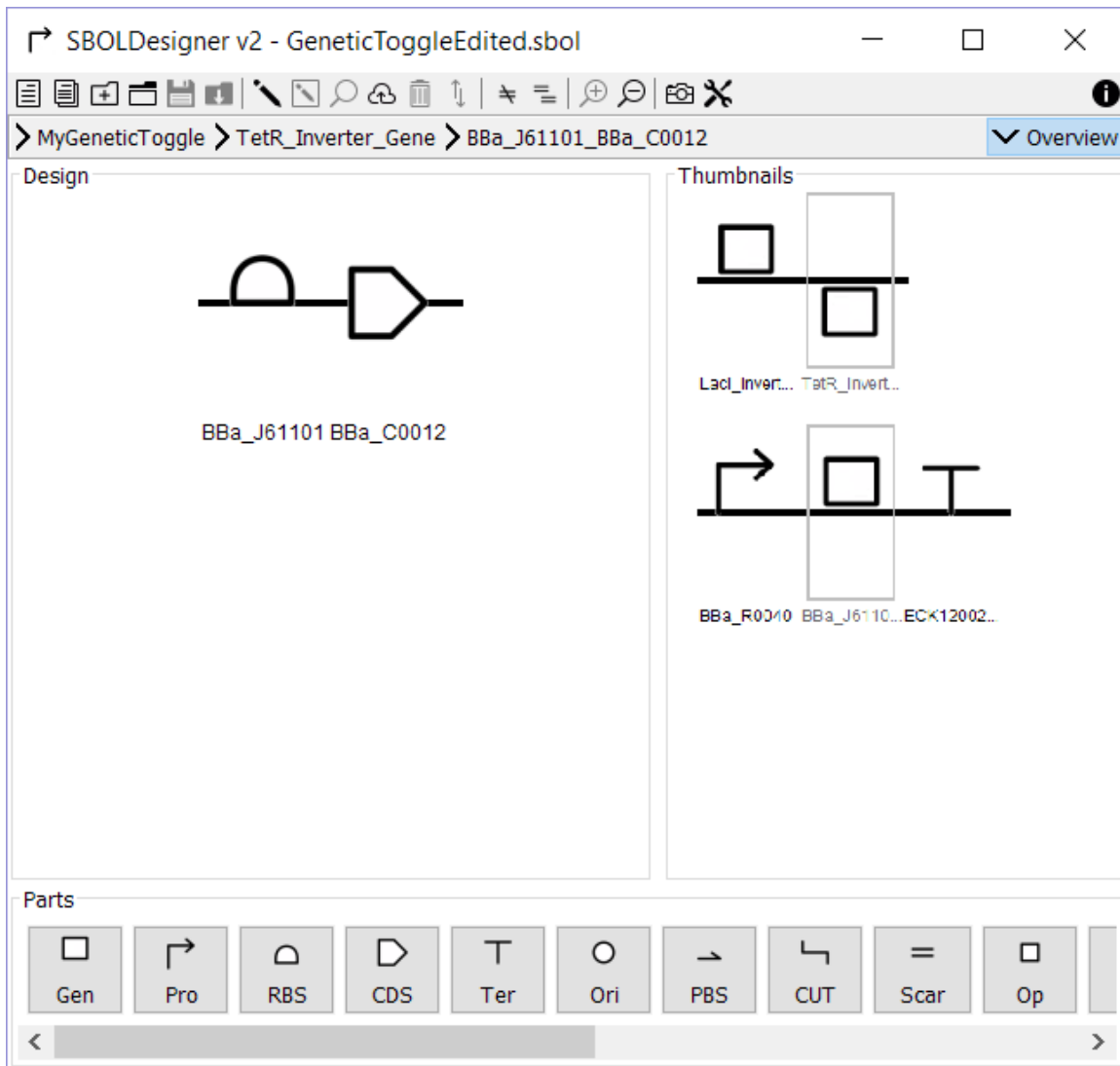
### SBOLDESIGNER

#### 2.1 Background

SBOLDesigner is a key piece of this workflow and the first sequence editor that integrates support of the SBOL 2 data standard with SBOL visual symbols. In particular, SBOLDesigner can obtain DNA sequences and other important metadata from the SynBioHub parts repository [12]. These components can then be composed and edited within SBOLDesigner to create a complete structural design of a genetic circuit. These new composite designs can then be uploaded to the part repository to enable sharing and reuse. In order to add functional information about a genetic design, SBOLDesigner has been integrated into the modeling and simulation *genetic design automation* (GDA) tool, iBioSim [13]. The iBioSim software can be used to construct and analyze functional models using the *Systems Biology Markup Language* (SBML) [10]. These functional models once annotated using genetic designs produced by SBOLDesigner [20] can be converted into an SBOL 2 document including functional information about the product of these genetic circuits and their interactions [15]. Once again, the complete genetic circuit with its functional information can be archived in a part repository for sharing. Throughout this process, researchers can collaborate and pass around files from institution to institution, located anywhere in the world. The SBOL standard provides the means to enable a lossless communication of data between these software tools and repositories.

SBOLDesigner is focused on the sequence and structural levels of genetic circuit design [17,21]. This mainly involves dealing with DNA. The main interface is shown in Figure 2.1. Parts shown in SBOL Visual appear on the bottom row and can be dropped onto the canvas. These parts can be arranged hierarchically, and each part's sequence, type, role, and id can be described. Some additional features are support for importing parts and

sequences from external sources such as other files on disk or SynBioHub, versioning of designs, and support for various file formats such as GenBank, FASTA, and SBOL 1 and 2 [21]. SBOLDesigner also supports rich annotations, such as the provenance standard, through the SBOL annotation framework [1].



**Figure 2.1.** The main canvas of SBOLDesigner is shown. A row of parts on the bottom show what can be placed into the design, and a hierarchically defined genetic circuit is being displayed in the overview. Double clicking on a part in the canvas opens up a part editor that allows the user to change the properties of the part. Parts can also be pulled from various registries and repositories such as SynBioHub, the local filesystem, and other file formats such as GenBank and FASTA. These parts can be combined in many ways, and the resulting design can be uploaded back to SynBioHub. Figure from Zhang et al. [21].



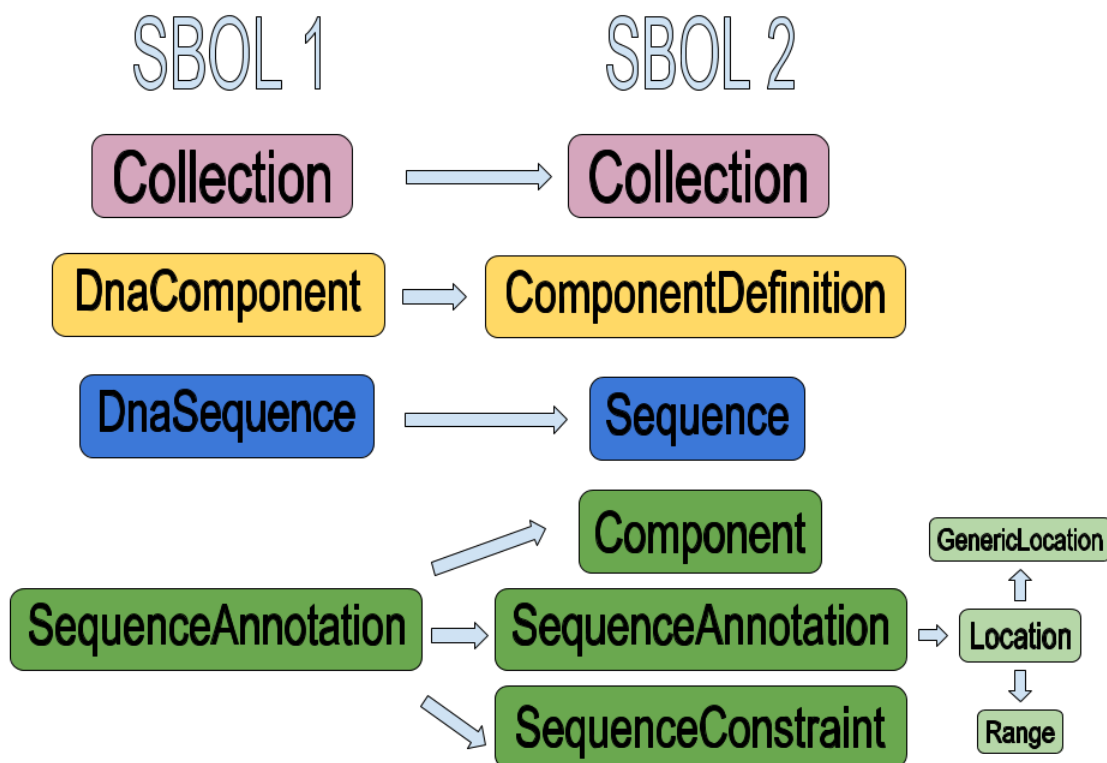
## 2.2 libSBOLj

SBOLDesigner uses the libSBOLj library for its backend data model and SBOL manipulation [22]. libSBOLj is a Java library that implements the SBOL specification. The libSBOLj library has been updated to support combinatorial objects and SBOL Visual 2. Any tool that also uses libSBOLj is able to read and write these constructs. Experimental biologists who were previously limited by what SBOL and SBOLDesigner could represent are now able to fully encode their designs. However, for further adoption of these extended SBOL constructs, more tools and libraries will have to support combinatorial design and SBOL Visual 2. This is a step in the correct direction, but there is a lot more to do before experimental biologists are able to completely abandon their Excel spreadsheets and Word documents.

While the user interface remains similar, the transition from SBOL 1 to SBOL 2 required re-implementing most of the software's back-end to use the data model provided by the libSBOLj 2.0 Java library [22]. Figure 2.2 provides a comparison of the SBOL 1 data model to the structural portion of the SBOL 2 data model. The key difference is that SBOL 2 separates *SequenceAnnotations* from SBOL 1 into *Components*, *SequenceAnnotations*, and *SequenceConstraints*, which requires a fundamental change in the representation of parts in SBOLDesigner. There are also many API changes between libSBOLj 1.0 and libSBOLj 2.0. Part of the design philosophy of libSBOLj 1.0 is the utilization of a factory for creating objects from the data model. Conversely, all libSBOLj 2.0 data model objects inherit from the *Identified* class, and all object creation is handled from a centralized *SBOLDocument* object. Also, the API is more complicated due to the introduction of more specific data model objects and the increase in the number of classes. For example, *SequenceAnnotations* now contain *Locations* which can be specific *Ranges* or more general *GenericLocations*. *SequenceConstraints* allow for more general ordering, and *Components* define explicit instantiations of *ComponentDefinitions*. These changes taken together necessitated a complete re-implementation of SBOLDesigner's back-end.

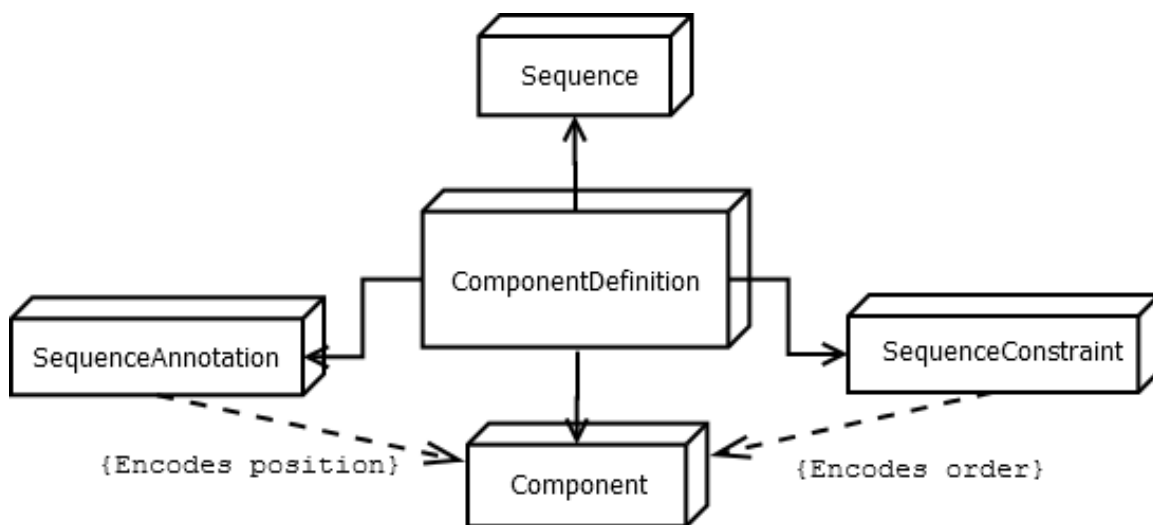
## 2.3 SBOL 2

The SBOL data standard provides a digital format that allows biologists to share genetic designs stored in a principled scheme. SBOL 1 focused solely on the design of *DNA*



**Figure 2.2.** A color coded diagram showing the mapping of classes between SBOL 1 and SBOL 2. While the left-hand side includes the entire SBOL 1 data model, the right-hand side is a simplified view that includes only the structural portion of the SBOL 2 data model. SBOLDesigner currently only supports this structural portion. In particular, DNA *ComponentDefinitions* specify their structure using DNA *Sequences* encoded using IUPAC. These DNA *ComponentDefinitions* can include sub-structure specified using *Components* that are ordered by their *SequenceConstraints* and positioned by the *Locations* within the *SequenceAnnotations*. Finally, these DNA *ComponentDefinitions* can be organized into *Collections*. The key difference between SBOL 1 and SBOL 2 is that *SequenceAnnotations* are now split into *SequenceAnnotations*, *Components*, and *SequenceConstraints*. Also, *DnaComponents* and *DnaSequences* are now more generic *ComponentDefinitions* and *Sequences* enabling them to specify parts of other types, such as RNAs, proteins, small molecules, complexes, etc.

*components* and the annotation of their *sequences* [6]. The latest version, SBOL 2, enables the description of general biological components, along with their *interactions*, and hierarchical composition into *modules*. In particular, SBOL 2 describes designs structurally using *ComponentDefinitions*, *Sequences*, and *SequenceAnnotations* that are composed together into biological designs described functionally using *ModuleDefinitions*, *Interactions*, and *Models* [1]. The SBOLDesigner tool focuses on the structural design of DNA parts. Although data content used by SBOLDesigner is similar to that in SBOL 1, many changes are necessary to support the restructured data model of SBOL 2 and its additional features. Figure 2.3 shows the main classes from the SBOL 2.0 data model that are used by SBOLDesigner.



**Figure 2.3.** A simplified view of the structural portion of the SBOL 2.0 data model that SBOLDesigner uses. *ComponentDefinitions* and their *Components* and *Sequences* are ordered by *SequenceConstraints* and positioned by *SequenceAnnotations*. *Components* can introduce hierarchy by pointing to other *ComponentDefinitions*. A complete genetic circuit design consists of a collection of *ComponentDefinitions* and *Sequences* organized by *SequenceConstraints*, *SequenceAnnotations*, and *Components*.

The main canvas shown in Figure 2.1 represents a *ComponentDefinition* that brings together information on the design's *Sequence*, its *Components*, and their organization. Therefore, every design in SBOLDesigner is inherently hierarchical. All the parts that are added to a specific design are contained transitively within a *root ComponentDefinition*, which is defined as a part that is not included as a sub-part within any other *ComponentDefinition*. Each SBOL file is allowed to include multiple *root ComponentDefinitions*. Therefore, when

an SBOL file is opened in SBOLDesigner, a single root *ComponentDefinition* must be selected for editing. Non-root *ComponentDefinitions* can also be selected for editing. However, it is important to note that while non-root *ComponentDefinitions* appear hierarchically identical to normal root *ComponentDefinitions* from the perspective of the *SBOLDesigner* hierarchy viewer and canvas, they are fundamentally different since there exists another *ComponentDefinition* in the SBOL file that contains a *Component* that references this transitive *ComponentDefinition*. Also, if the original non-root is overwritten, changes are reflected in all designs in the SBOL file that reference this *ComponentDefinition*. If, on the other hand, a new version is created, then the original references continue to use the original design of this *ComponentDefinition*. In this case, the new version of this *ComponentDefinition* would itself become a root *ComponentDefinition* until it is also included in another design.

The fields in the part editor map directly to a *ComponentDefinition* and its *Sequence*. For example, the *role*, *displayId*, *name*, *description*, and *version* are all properties of the *ComponentDefinition*, and the *sequence* text box maps directly to the *elements* property of a separate *Sequence* object. A *ComponentDefinition* and its *Sequence* are individual top-level objects in the data model, and can both exist independently without the other, but to simplify the interface, SBOLDesigner treats them as a single unit.

Below the canvas in SBOLDesigner is a row of genetic elements that can be added to the design. When placed on the canvas, each element represents a *Component*. These *Components* are organized by *SequenceAnnotations* and *SequenceConstraints*, all of which are children of the canvas *ComponentDefinition*. This means that they can only exist within the parent *ComponentDefinition*. *Components* are completely new to SBOL 2, and represent hierarchy in the design. Each *Component* refers to another *ComponentDefinition*, and represents a specific instantiation of that referred *ComponentDefinition*. While *Components* are not top-level object in the data model, the *ComponentDefinitions* they refer to are. Therefore, clicking on the "focus in" button expands a *Component* to expose its *ComponentDefinition*, generating a nested canvas. This new canvas also represents a *ComponentDefinition*, which allows the user to create hierarchically defined designs. However, even though this nested *ComponentDefinition* is a top-level object in the data model, it is not considered a root *ComponentDefinition* since there exists another *ComponentDefinition* that contains a *Component* that references this *ComponentDefinition*. As a result of *Components*, a design can have

any number of layers of *ComponentDefinitions*, as long as there is not a cycle. In other words, *ComponentDefinitions* cannot refer to a *Component* that then directly or indirectly refers to a *ComponentDefinition* that is a parent of the *Component* that refers to the original *ComponentDefinition*.

*SequenceAnnotations* specify the precise *Location* and *orientation* (position and direction) of a *Component* and *SequenceConstraints* encode information on how *Components* are ordered relative to each other. *SequenceConstraints* are only present when there is more than one part on the canvas, since relative ordering only matters when there are multiple parts. However, each *Component* always has a *SequenceAnnotation* that refers to it. Also, it is important to note that *SequenceAnnotations* and *SequenceConstraints* are annotations and constraints on *Components*, not *Sequences*. This distinction means there could be *SequenceAnnotations* and *SequenceConstraints* on *Components* whose *ComponentDefinitions* do not have a defined *Sequence*. By default, *SequenceAnnotations* contain a *Location* of type *GenericLocation* if the *ComponentDefinition* pointed to by its *Component* does not have a defined *Sequence*. However, if the *Component* refers to a *ComponentDefinition* with a defined *Sequence*, then the *SequenceAnnotation* contains a *Location* of type *Range*. This *Range* specifies the *start* and *end* index of exactly where this *Component's* *ComponentDefinition's* *Sequence* belongs in the parent *ComponentDefinition's* *Sequence*. An important assumption used by SBOLDesigner is that parts are abutted as shown in the canvas. This means the *start* position of a *Component* is one base along from the *end* position of the *Component* before it. The first *Component* has a *start* position value of 1. With this assumption, the user does not need to worry about the construction of *Sequences* and their *SequenceAnnotations*, since this is all handled behind the scenes and is automatically generated by SBOLDesigner.

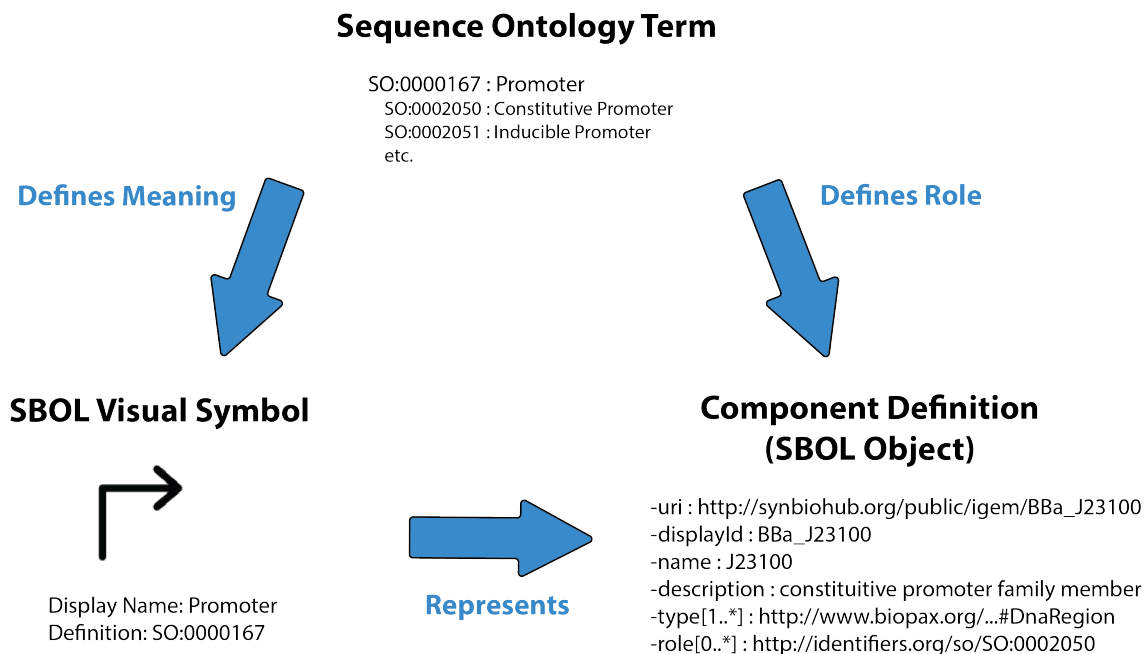
# CHAPTER 3

## SBOL VISUAL

### 3.1 Background

SBOL visual is a specification of standard visual schematic glyphs used to represent various commonly used genetic components [18]. The standards are linked using the *Sequence Ontology* (SO) [4] as shown in Figure 3.1. Namely, a DNA *ComponentDefinition* constructed in the data standard is visualized by looking up the corresponding SBOL visual symbol specified by its *role*. The SBOL visual standard defines how a *ComponentDefinition* with a particular role should be drawn on whiteboards, in computer aided design tools, and in papers. Therefore, the Sequence Ontology term is what connects the role of a *ComponentDefinition* with how it should be drawn. Specifically, the Sequence Ontology term is used to define the meaning of a SBOL Visual glyph symbol and define the role of an SBOL *ComponentDefinition* object.

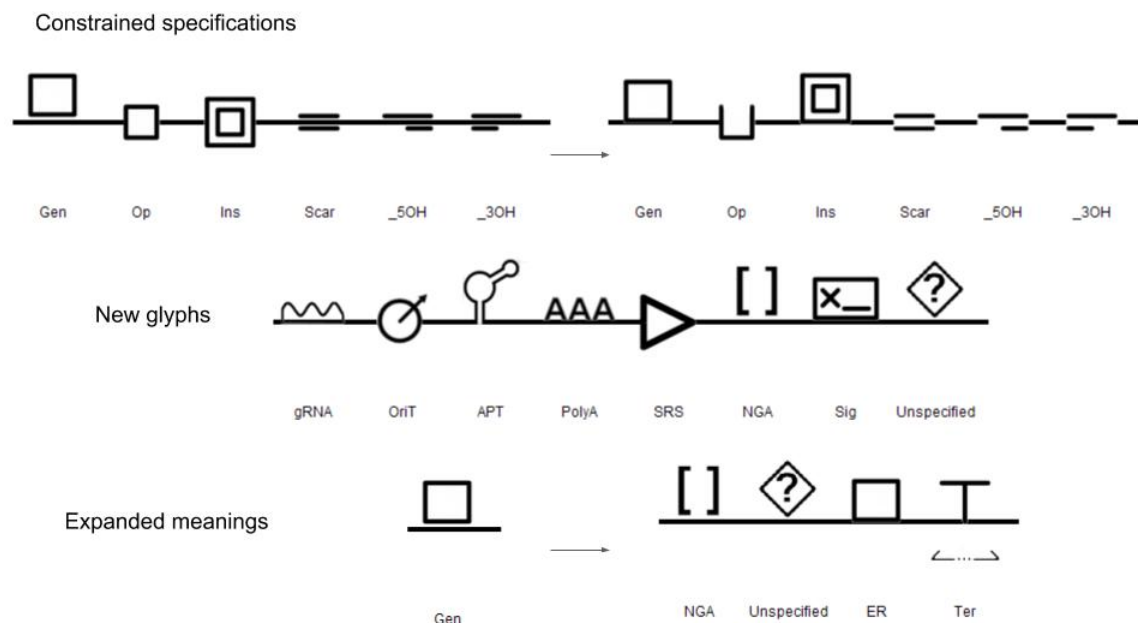
Each glyph therefore has an explicit semantic meaning and representation. When new glyphs are introduced, their specifications clearly indicate the Sequence Ontology term that gives a semantic meaning to the glyph. Therefore, most of the specification is dedicated to defining how the glyph should be drawn. While such detailed specification might seem excessive, it is important to clearly define the commonly used glyphs drawn on whiteboards and in papers. Otherwise, much of the intended meaning of figures of genetic circuits is left to the interpretation of the reader. With SBOL Visual, the intent of the illustrator is made more apparent, and the reader is able to more easily resolve any ambiguity or confusion. As a result, SBOL 2 and SBOL visual taken together facilitate communication between experimental biologists, computational biologists, genetic engineers, and their computational tools.



**Figure 3.1.** *SO:0000167* is the Sequence Ontology term for a promoter. SBOLDesigner is able to associate this well-defined ID to both the SBOL Visual glyph for promoter, as well as a template promoter *ComponentDefinition*. The Sequence Ontology term defines the meaning of the SBOL visual glyph and the role of the *ComponentDefinition* SBOL object. The term is specified by the Sequence Ontology, the visual symbol is defined by the SBOL Visual standard, and the *ComponentDefinition* SBOL object is defined by the SBOL data model standard. Additionally, the Sequence Ontology is organized as a tree, so any descendants of promoter such as constitutive promoter or inducible promoter, are also associated with the correct glyph and *ComponentDefinition*. Also note that many *ComponentDefinition* SBOL objects can share the same Sequence Ontology term role while each SBOL Visual symbol explicitly represents a Sequence Ontology term. Differentiating between different parts that share the same role and glyph is done in SBOLDesigner by writing a text label with the part's displayId below the glyph.

### 3.2 SBOL Visual 2

SBOL Visual 2 is a natural continuation of the original SBOL visual standard. Specifically, it constrains the specification of new glyphs, adds many new glyphs, and gives glyphs more explicit semantic meanings. Figure 3.2 summarizes these changes. These improvements further add to the benefits of having a standardized set of genetic circuit glyphs.



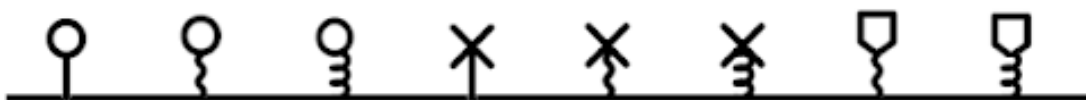
**Figure 3.2.** There are three types of changes introduced in SBOL Visual 2. The top row shows how glyphs now have a more constrained specification. The left row shows how the glyphs used to be shown in the user interface, and the right side shows how they are shown now. For example, there is now a more precise specification as to where the glyph belongs relative to the backbone. The second row of glyphs shows new glyphs that have been added to SBOL Visual. With the extended range of supported and well defined glyphs, more interesting genetic circuits can be represented. Finally, bottom row shows how the generic box glyph has been expanded into more explicit semantic meanings. Specifically, there is now a *no glyph assigned* glyph, *unspecified* glyph, *engineered region* glyph, and *composite plus omitted detail* overlay. As a result of these changes, the scrollable part selection window has been expanded to accommodate the new palette.

All the glyphs now have more precise specifications. For example, some parts sit above the backbone while other sit on the backbone. Some parts cover the backbone and other show the backbone going through the part. SBOL Visual 1 did not provide exact details on how each glyph should be drawn respective to the backbone and relative to other parts.



Now, each glyph specification also includes a bounding box that manages scale and well defined specification style depictions. SBOLDesigner has been updated to follow all these conventions. This results in a more consistent style between SBOLDesigner and other tools and papers. While each illustrator can draw their glyphs creatively, there now exists a minimal set of rules that all instances of each glyph has to follow.

Many new glyphs for various genetic constructs have also been introduced. This allows for a wider range of part encodings. For example, a system of stem and top glyphs have been added where the stem variations can be mixed with the top variations. These glyphs are shown in Figure 3.3. Other new glyphs include the *aptamer*, *non-coding RNA*, *ori-t*, *poly-A site*, *recombination site*, and *signature*. SBOL Visual is a very extensible standard, so even more glyphs are anticipated to be introduced over time.

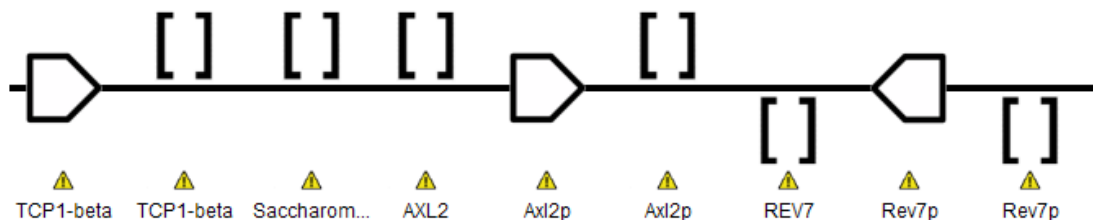


**Figure 3.3.** A systems of glyphs created by mixing variations of stems with variations of tops. From left to right, we have the *base*, *junction*, *amino acid*, *restriction enzyme recognition site*, *ribonuclease site*, *protease site*, *rna stability element*, and *protein stability element*. All these parts can be uniquely represented in SBOLDesigner. Each part is well specified in the SBOL visual standard through its pictorial glyph and Sequence Ontology role.

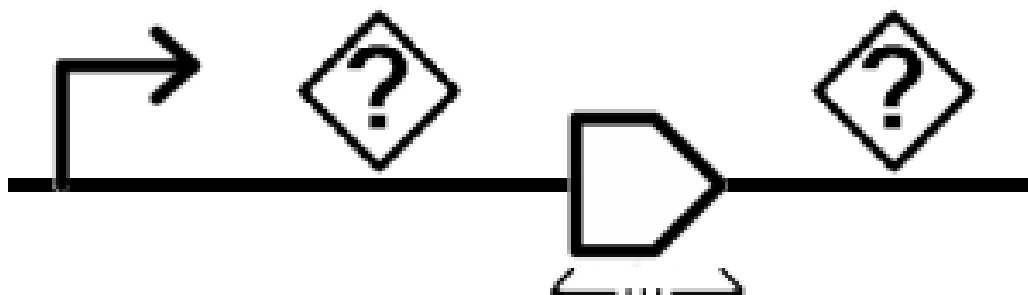
Some additional SBOL Visual 2 changes include adding dotted lines for hierarchically defined composite parts and support for functional relationships between parts. Many new glyphs have also been added to address the semantic shortcomings of the original glyphs. For example, the glyph that used to represent the generic *engineered region* has been replaced by three glyphs that represent *unspecified*, *composite*, and *no glyph assigned* parts. This specific change removes ambiguity in the original SBOL Visual specification by defining different semantics for different usages of the box glyph.

Examples of the new distinction between *unspecified*, *no glyph assigned*, and *composite* parts are also shown in Figure 3.4 and Figure 3.5. Figure 3.4 shows how the no glyph assigned glyph can be used to explicitly say a circuit contains parts that have no glyph assigned. Due to the GenBank format, many GenBank files that are converted to SBOL

end up having parts with no glyph assigned. In general, there have always been loosely specified genetic circuits with no glyph assigned parts. Prior to SBOL Visual 2, all of these parts were simply overloaded as engineered regions. Not only was this not informative for the users of SBOLDesigner, it also caused confusion where there were both engineered regions, composites, and actual parts with no glyph assigned.



**Figure 3.4.** Many imported GenBank files contain annotations that specify parts with no assigned glyph. To more explicitly differentiate between engineered regions, hierarchically defined parts, and unspecified parts, these no glyph assigned glyphs are used. Prior to this change, each part would have looked like a box regardless of whether it is actually an engineered region or a hierarchically defined composite. The yellow warning overlay further specifies that these parts do not have sequences associated with them.



BBa\_K154... BBa\_K184... BBa\_M45178BBa\_J1071...

**Figure 3.5.** A simple genetic device is shown. The second and fourth parts don't have specified roles, and are therefore marked with the unspecified glyph. There is actually no role information in the SBOL file that encodes these parts. Also, the coding sequence is defined hierarchically, as shown with the composite overlay beneath the glyph. This overlay is expandable by selecting the coding sequence and clicking on the focus in button. Before, all three of these parts would have been shown as a plain box in SBOLDesigner.

Figure 3.5 shows another example genetic circuit that is more explicitly rendered using the composite and unspecified glyphs. Before, all three variations of these glyphs would have shown up as plain boxes. Now, the unspecified parts are marked as such, and the composite coding sequence has a composite overlay underneath the glyph. This overlay is expandable by selecting the coding sequence and clicking on the focus in button. It is also important to note that an unspecified part is truly unspecified, and occurs whenever there is no role information in the actual SBOL file. As defined within the SBOL data model specification, this kind of unspecified *ComponentDefinition* is against best practices.

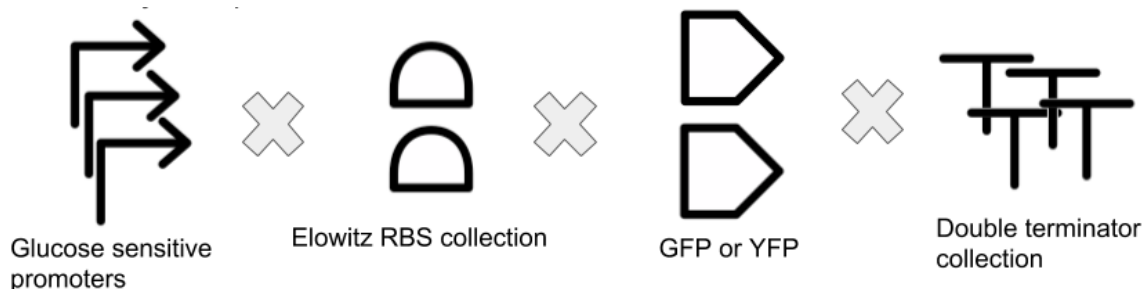
Supporting the semantics of each glyph required lots modification of the main canvas in SBOLDesigner. For example, composite parts, or parts composed hierarchically using other parts, are represented with a dotted line underneath the glyph that implies substructure. This dotted line is how SBOLDesigner represents the *omitted detail* glyph. By clicking on the composite part, the user can zoom into the part and view the substructure. This is effectively viewing the detail that is omitted in the omitted detail glyph. Also, parts without sequence information have a yellow warning icon below the glyph. Examples of these overlays are shown in Figure 3.4 and Figure 3.5 as well. These overlays help the user visually reason about the genetic circuit's parts outside of what role each part has. The overlays can also be extended in the future to support further such notifications such as functional detail and sequence length.

## CHAPTER 4

### COMBINATORIAL DESIGN

#### 4.1 Background

Combinatorial designs are a natural way to express experiments on genetic circuits where variants of many parts are simultaneously created and then tested. Specifically, each part can have many variants, and each variant of each part can show up in an instance of the generated genetic circuit. This results in exponential growth in terms of the number of possible generated instances. Figure 4.1 shows this. As a new part is added, all of the previous parts, their variants, and all the generated instances have to choose which of the new variants to use in the design.



**Figure 4.1.** An example combinatorial design is shown. The promoters are chosen from a set of glucose sensitive promoters, the ribosome binding site comes from the Elowitz collection, the coding sequence can either be green fluorescent protein or yellow fluorescent protein, and the terminator can be one of a bunch of double terminators. To represent this in SBOL, a template part is created for each part, and variants are connected to each template part. SBOLDesigner supports the creation and design space exploration of combinatorial designs.

It can therefore be very inefficient to represent the design space by simply serializing all the different possibilities. To this end, SBOL supports combinatorial design by allowing each part to be tagged with variants, what possibilities can occur, and what strategies to

use while creating the generated instances. These new additions to the SBOL data model were introduced in SBOL 2.2.0 [2].

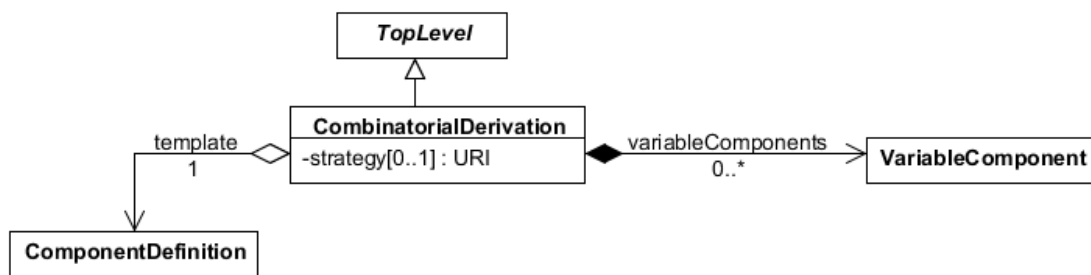
An emphasis has been put on making the new features conceptually easy to understand. This also gives more tools in the SBOL workflow an opportunity for adopting combinatorial design. As combinatorial design becomes an increasingly useful design strategy, support for these objects in tools such as SBOLDesigner and libraries such as libSBOLj will become increasingly valuable as well.

## 4.2 Implementation

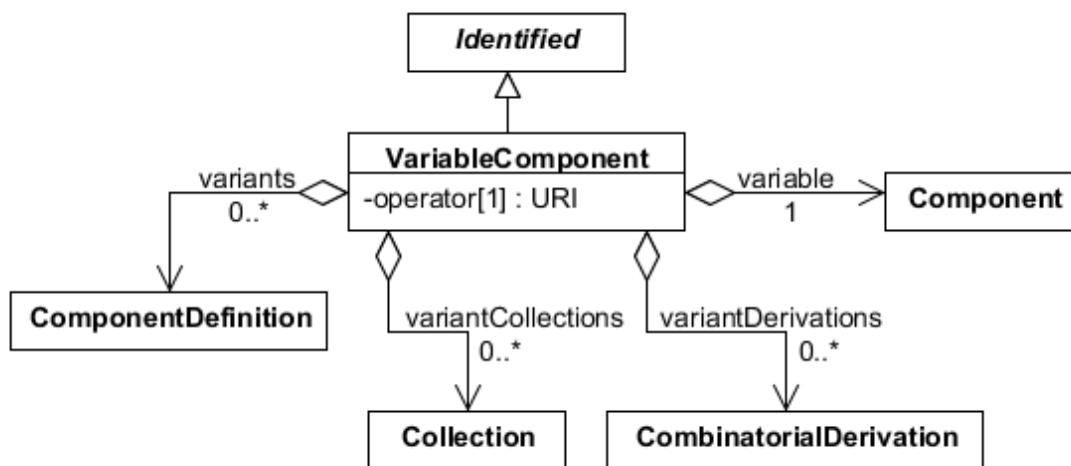
### 4.2.1 Data Model

A UML diagram for the combinatorial design object that has been added to the SBOL standard is shown in Figure 4.2 and Figure 4.3. Instead of having a separate design for each variant, a single combinatorial derivation is created. These combinatorial derivations represent a single combinatorial design. The template part is a *ComponentDefinition* that contains hierarchically defined child parts, and each child part can be associated with a variable component. The variable components are linked to that specific child part through the variable field. The variable field is a *Component* and refers to another *ComponentDefinition*. This allows the child part to specify many variant parts without unnecessarily duplicating other design information.

The combinatorial derivation also has a strategy property. This can either be *enumerate* or *sample*. *Enumerate* specifies that all possible combinations of the variants should be derived, and *sample* specifies that only a selected subset of all possible combinations should be derived. Variable components also have an operator property that specifies exactly how to group the collected variants that the variable component refers to. The operator can either be *one, zero or one, one or more, or zero or more*. Variants are able to be specified and collected using a variety of ways. Specifically, variants can be specified as a set of individual *ComponentDefinitions*, a set of *Collections* containing *ComponentDefinitions*, and hierarchically nested *CombinatorialDerivations*. Any grouping of the collected variants allowed by the operator can replace the *ComponentDefinition* referred to by the *Component* variable field in a generated instance of the *CombinatorialDerivation* that owns this variable component.



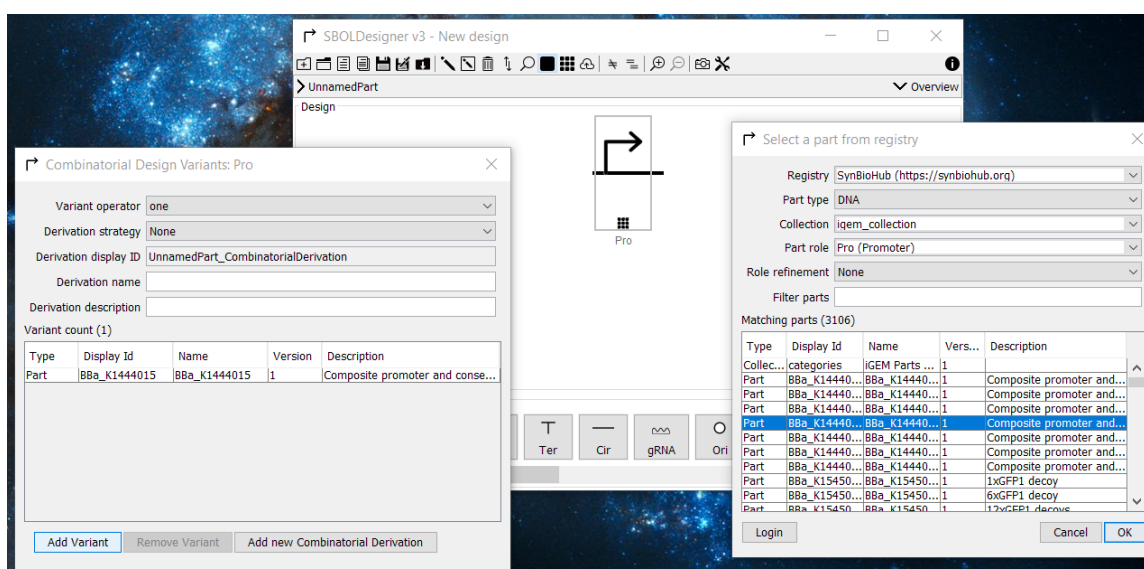
**Figure 4.2.** A UML diagram of the combinatorial derivation object. Each combinatorial derivation points to a specific part through its template reference and specifies the variable components that that part has through its variableComponents reference. As with all SBOL data model top level objects, it inherits properties such as a displayId, description, and version, and can exist independently of other SBOL data model objects. The variable components are not top level object, and therefore are owned by and live within the top level combinatorial derivation. Each combinatorial derivation also has a strategy field of type uniform resource identifier that specifies how the combinatorial derivation should be expanded.



**Figure 4.3.** A UML diagram of the variable component object. Each variable component points to a specific part and specifies the variant parts that could replace the component. These variants could be specified as plain references to Component Definitions, references to collections of Component Definitions, and other nested Combinatorial Derivations. The operator of the variable component is a uniform resource identifier that specifies how the collected variants can be grouped together. The variable reference points to a Component that refers to a ComponentDefinition that can be replaced by any one of the possible groupings.

## 4.2.2 User Interface

The interface for the combinatorial design editor as implemented within SBOLDesigner is shown in Figure 4.4. When a part is combinatorially defined, there is an overlay underneath the part that makes it clear to the user that that part has variants. Viewing the variants is done through the variant editor. The variant editor shows all the attached variants, fields that can be set regarding this part's variants, and ways to add new variants or add new combinatorial derivations. There is also a way to add a new combinatorial derivation. This means a single genetic circuit design can have multiple combinatorial derivations that can each be independently expanded. The variant editor is connected to the template promoter on the canvas through a variable component reference. This variable component is referenced to by a parent combinatorial derivation that references the promoter's parent as the template part. The variable field of the variable component specifies the part that will be replaced by a grouping of the collected variants.



**Figure 4.4.** SBOLDesigner's combinatorial design interface is shown. The variant editor (leftmost window) enables the user to quickly add and remove variants of the promoter from a variety of sources. It also allows the variable component's operator and combinatorial derivation's strategy, displayId, name, and description to be set. Variants can then be added, which pulls up an import dialog (rightmost window) that allows for parts to be imported from other files on disk, various SynBioHub instances, and a collection of built in parts. Since the *promoter* shown on the main canvas (center window) has combinatorial design variants, SBOLDesigner shows a grid overlay underneath the *promoter* glyph.

Design space exploration of the combinatorial design is also supported within SBOLD-designer. Depending on whether the enumerate or sample field was selected, an SBOL file will be generated that contains an enumeration of all possible designs or a sampling of a single generated design. Figure 4.5 shows an opened SBOL file with many generated instances. The algorithm that implemented enumeration of combinatorial designs is described below. The result of SBOLDesigner's combinatorial design implementation is that a small and simple genetic circuit with template parts associated with variants can result in an enumerated library of hundreds or thousands of generated instances. The complexity of managing the internal SBOL data model scaffolding, enumeration algorithm, and combinatorial design creation is abstracted away and hidden from the user.

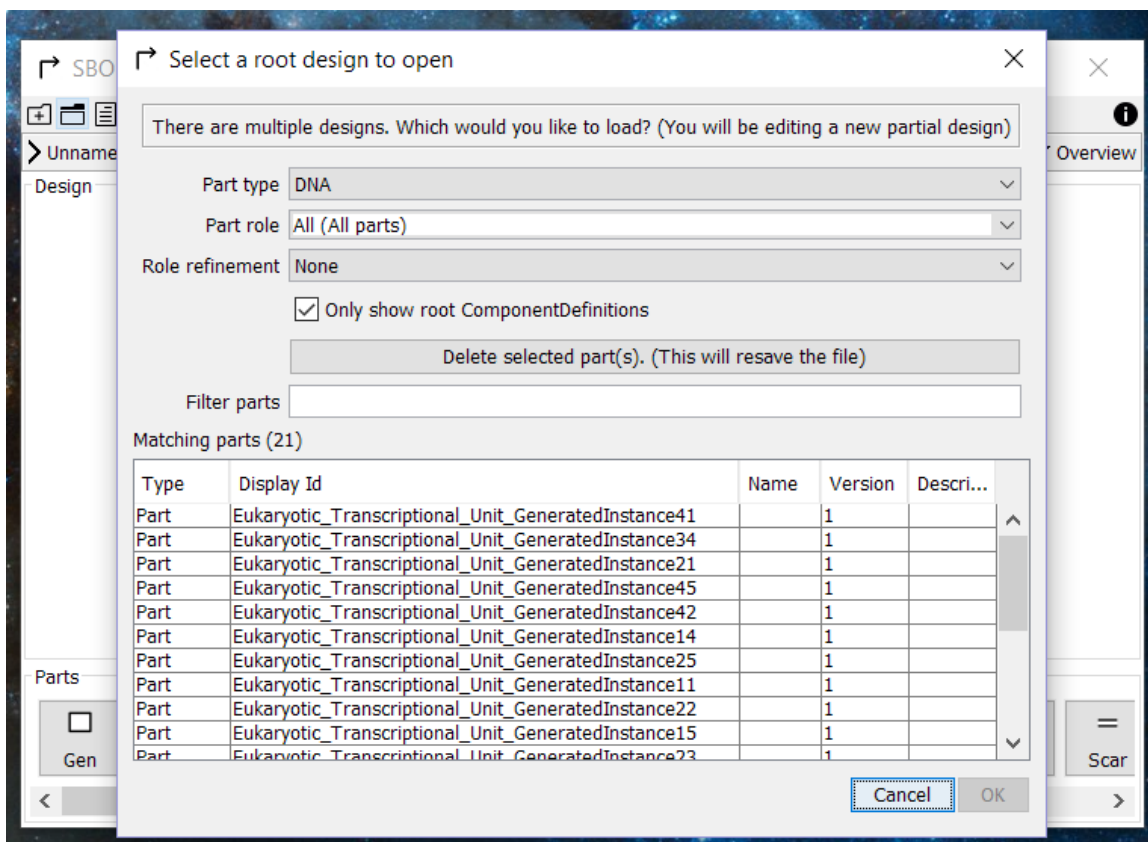
### 4.2.3 Algorithm

The combinatorial design enumeration algorithm is shown in Algorithm 1. The main driving function is *enumerate*, which takes in a combinatorial derivation and returns a set of all of the generated instances. On a high level, this function goes through each of the variable components in the derivation one at a time, and takes all the variant possibilities from this variable component and mixes it in with the current partially explored design space. This can be thought of as multiplying all the current possibilities by all the new possible variants within this variable component.

Specifically, a set of component definitions call parents is created. This represents all the possible generated instances. As we go through each variable component referenced by the combinatorial derivation, a new empty set of component definitions is created called newParents. Then, for each parent in the original parents set, and for each group of children specified by the variable component, a newParent is created by copying the original parent and adding this group of children. Finally, the newParent is added to newParents and parents is replaced with newParents for the variable component. In the end, parents will contain all the possible generated instances from this enumeration.

Two important details are how the variants are collected together from each variable component, and how the collected variants are grouped together to form the possible sets of children. The collectVariants function takes in a variable component and returns all of the referenced component definitions as a set. This includes the singly specified





**Figure 4.5.** A eukaryotic transcriptional unit genetic circuit is defined combinatorially and enumerated. In this specific case, the enumeration resulted in 21 generated instances. It is much easier to define and reason about the base genetic circuit as a combinatorial design than as a collection of all the possible generated instances. How many generated instances from enumeration exist is purely a function of the combinatorial derivation and variable components. Because of the exponential nature of specifying more variants and more template parts, enumerated combinatorial designs can grow to many more parts.

component definition variants, aggregated component definition collections, and nested combinatorial derivations. It is important to note that when variants are specified using a nested combinatorial derivation, this combinatorial derivation must first be recursively enumerated over in order to get its possible generated instances. The group function simply takes all the variants that were previously collected as well as the operator this variable component specified. Depending on whether the operator specifies to group the variants in groups of ONE, ZEROORONE, ONEORMORE, or ZEROORMORE, the algorithm returns a set of sets of these groupings. Finally, the addChildren function is also important since it has to preserve the sequence constraints, ordering, and other rules specified by the combinatorial design.

---

**Algorithm 1** Enumerate Combinatorial Design
 

---

```

1: procedure ENUMERATE(DERIVATION)
2:   parents ← {originalTemplate}
3:   for vc : derivation.variableComponents do
4:     newParents ← {}
5:     for parent : parents do
6:       for children : group(collectVariants(vc), vc.operator) do
7:         newParent ← copy(parent)
8:         addChildren(newParent, children)
9:         newParents.add(newParent)
10:    parents ← newParents
11:   return parents
12:
13: procedure COLLECTVARIANTS(VARIABLECOMPONENT)
14:   variants ← {}
15:   variants.addAll(variableComponent.variants)
16:   variants.addAll(variableComponent.variantCollections.variants)
17:   variants.addAll(enumerate(variableComponent.variantDerivations))
18:   return variants
19:
20: procedure GROUP(VARIANTS, OPERATOR)
21:   if operator is ONE then return {a}, {b}, {c}
22:   else if operator is ZEROORONE then return {}, {a}, {b}, {c}
23:   else if operator is ONEORMORE then return {a}, {ab}, {abc}, ...
24:   else if operator is ZEROORMORE then return {}, {a}, {ab}, {abc}, ...

```

---

### 4.3 Combinatorial Design Example

Figure 4.6 shows an example combinatorial design for a green fluorescent protein reporter circuit. As shown in the main canvas, the circuit is comprised of a *promoter*, *ribosome binding site*, *coding sequence*, and *terminator*. The *promoter* and *coding sequence* are template parts that can be swapped out for any of their variants. SBOLDesigner's main canvas indicates this through the three by three cube overlay underneath the respective glyphs. The *coding sequence's* variants are shown in the variant editor. There are three variants, and each variant is a version of the green fluorescent protein. The *ribosome binding site* is an Elowitz *ribosome binding site*, and the *terminator* is a composite double terminator as seen from the composite overlay.

Different variations of this circuit can be generated depending on what *promoter* and what *coding sequence* are chosen. However, they are all green fluorescent protein reporter circuits since each *coding sequence* encodes a different variation of the green fluorescent protein. To represent all the variations of this circuit, the user would normally need to create each generated instance independently. With combinatorial design, the user only needs to create one genetic circuit using templates and variants, and the combinatorial enumeration algorithm can handle the rest. After enumeration, each combination of *promoter* would be mixed with each combination of *coding sequence*. Specifically, this kind of design space exploration occurs since the *enumeration* strategy is specified on the *combinatorial derivation* object and both of the *variable components* that reference the *promoter* and *coding sequence* specify the *one* operator.

The screenshot shows the SBOLDesigner v3 interface. The main canvas displays a genetic circuit diagram with the following components: a promoter (Pro), a ribosome binding site (RBS), a coding sequence (CDS), and a terminator (Ter). The CDS is highlighted with a combinatorial design variant editor. The variant editor shows a table of three variants for the CDS, each with a different display ID and name, all using the same GFP part. The variant editor also includes fields for variant operator, derivation strategy, display ID, name, and description, along with an 'Add Variant' button.

| Type | Display Id  | Name        | Version | Description                    |
|------|-------------|-------------|---------|--------------------------------|
| Part | BBa_E0040   | GFP         | 1       | green fluorescent protein d... |
| Part | BBa_M36577  | BBa_M36577  | 1       | GFP                            |
| Part | BBa_I766210 | BBa_I766210 | 1       | GFP                            |

**Figure 4.6.** A combinatorial design of a green fluorescent protein reporter circuit is shown in SBOLDesigner’s main canvas. The *promoter* and *coding sequence* are described using combinatorial variants. The variants of the template green fluorescent protein *coding sequence* are shown in the variant editor. After enumeration, each combination of *promoter* and *coding sequence* will be generated using the same *ribosome binding site* and hierarchically defined *terminator*.

## CHAPTER 5

### CONCLUSION

#### 5.1 Summary

The ability to express these new kinds of constructs means a wider variety of designs are able to be captured. The use of SBOLDesigner to create well-formed SBOL documents also helps the adoption of data standards in synthetic biology. Before, valuable information about how a genetic circuit was specified was omitted due to not having a straightforward method of encoding all the details other than Excel spreadsheets and Word documents. This use of unstructured and non-machine interpretable data made transcription of knowledge a mess and bookkeeping a chore. Now, with automated tooling and biologist friendly user interfaces, even some of the most daunting and most advanced synthetic biology techniques can be captured with the click of a button. Ease of use spurs adoption, especially if the intended users traditionally are not subscribers to computer aided design workflows. However, the rewards for adopting these new synthetic biology workflows are great.

Additionally, support for SBOL 2, combinatorial design, and SBOL Visual 2 helps address the problem of non-reproducible science in synthetic biology. With the adoption of the SBOL workflow using SBOLDesigner and its new features, data published in papers will be well organized, and biologists will be able to easily search through and find the information they need to rapidly iterate and validate their new and novel designs.

Using SBOLDesigner, a critical missing part of the synthetic biology workflow centered around SBOL has been addressed. Experimental biologists are now able to more completely represent their design methodology in these tools, and visualizations of genetic circuits are more clear. These additions to SBOLDesigner's ability to express genetic circuits allows for a easier path towards SBOL adoption. Increased SBOL adoption in turn drives

experimental biologists towards the end goal of enhancing reproducibility in synthetic biology. The benefits of reproducibility include higher quality papers, more trustworthy results and conclusions, and faster accumulation of synthetic biology's body of knowledge.

## 5.2 Future Work

### 5.2.1 Interactions

The SBOL community is currently limited in its ability to express interactions. The SBOL data model supports parts interacting with other parts through *ModuleDefinitions*, *FunctionalComponents*, and *MapsTos*, but there is currently no tool that specializes in encoding and creating this information in an easy and user friendly manner. Figure 1.3 shows an example genetic circuit with parts interacting with other small molecules.

To support interactions within SBOLDesigner, the main canvas would need to be able to understand the functional side of the SBOL data model, draw interaction arcs, draw small molecules, and present the user with a series of dialogs that controls the data model. This presents a major user interface challenge. However, SBOLDesigner's main canvas has already been extended with overlays, so initial proofs of concepts have been completed.

### 5.2.2 Computer Aided Manufacturing

Computer aided manufacturing (CAM) tools in synthetic biology are used to prepare DNA sequences for fabrication and manufacturing. In short, CAM tools are used to prepare for the building of the systems designed using CAD tools. The Build OptimizatiOn Software Tools (BOOST) service provides an API for common CAM sequence operations such as reverse translation, codon juggling, and synthesis constraint verification [16]. It would be beneficial to integrate SBOLDesigner with BOOST so a user could simply click on a "prepare for synthesis" button and run CAM operations on their completed genetic circuit design. BOOST would return with a manufacturing optimized SBOL file which SBOLDesigner could present to the user.

### 5.2.3 Plugin Support

SBOLDesigner is very useful both standalone and embedded in a larger tool. For example, SBOLDesigner is currently embedded within iBioSim, resulting in a robust platform for genetic circuit design, modeling, and simulation. iBioSim provides the simulation and

modeling capabilities, and SBOLDesigner provides a simple and intuitive way to construct the structural portions of genetic circuits. Embedding SBOLDesigner as a plugin within other suites of tools such as Geneious or Benchling would be similarly useful.

#### **5.2.4 Better Search**

One of the biggest pain points in genetic circuit design is finding the right parts. Most part repositories are not well curated and/or consist of lackluster quality user submitted parts. Data mining and data infrastructure techniques can be used to extract or filter out parts to provide a better part selection process. While SBOLDesigner currently has great integrations with SynBioHub, it is only really useful when the user already knows exactly what part they are looking for. Also, the search query results in a somewhat randomly ordered list of parts. To improve this, parts in SynBioHub can be merged to reduce the number of useless or low quality parts, resulting cleaned and normalized parts could be ranked to give a better order of usefulness based on popularity, and the backend datastore could be improved so that queries evaluate quicker.

## REFERENCES

- [1] J. BEAL, R. COX, R. GRUNBERG, J. MCLAUGHLIN, T. NGUYEN, B. BARTLEY, M. BISSELL, K. CHOI, K. CLANCY, C. MACKLIN, C. MADSEN, G. MISIRLI, E. OBERORTNER, M. POCOCK, N. ROEHNER, M. SAMINENI, M. ZHANG, Z. ZHANG, Z. ZUNDEL, J. GENNARI, C. MYERS, H. SAURO, AND A. WIPAT, *Synthetic Biology Open Language (SBOL) Version 2.1.0*, *Journal of Int. Bioinfo.*, (2016).
- [2] R. S. COX, C. MADSEN, J. A. MCLAUGHLIN, T. NGUYEN, N. ROEHNER, B. BARTLEY, J. BEAL, M. BISSELL, K. CHOI, K. CLANCY, ET AL., *Synthetic biology open language (sbol) version 2.2. 0*, *Journal of Integrative Bioinformatics*, (2018).
- [3] F. CRICK, *Central Dogma of Molecular Biology*, *Nature*, 227 (1970), pp. 561–563.
- [4] K. EILBECK, S. LEWIS, C. MUNGALL, M. YANDELL, L. STEIN, R. DURBIN, AND M. ASHBURNER, *The Sequence Ontology: A Tool for the Unification of Genome Annotations*, *Genome Biology*, 6 (2005), p. 1.
- [5] D. ENDY, *Foundations for Engineering Biology*, *Nature*, 438 (2005), pp. 449–453.
- [6] M. GALDZICKI, K. P. CLANCY, E. OBERORTNER, M. POCOCK, J. Y. QUINN, C. A. RODRIGUEZ, N. ROEHNER, M. L. WILSON, L. ADAM, J. C. ANDERSON, B. A. BARTLEY, J. BEAL, D. CHANDRAN, J. CHEN, D. DENSMORE, D. ENDY, R. GRUENBERG, J. HALLINAN, N. J. HILLSON, J. D. JOHNSON, A. KUCHINSKY, M. LUX, G. MISIRLI, J. PECCOUD, H. A. PLAHAR, E. SIRIN, G.-B. STAN, A. VILLALOBOS, A. WIPAT, J. H. GENNARI, C. J. MYERS, AND H. M. SAURO, *The Synthetic Biology Open Language (SBOL) Provides a Community Standard for Communicating Designs in Synthetic Biology*, *Nature Biotechnology*, 32 (2014), pp. 545–550.
- [7] T. S. GARDNER, C. R. CANTOR, AND J. J. COLLINS, *Construction of a Genetic Toggle Switch in Escherichia Coli*, *Nature*, 403 (2000), pp. 339–342.
- [8] T. S. HAM, Z. DMYTRIV, H. PLAHAR, J. CHEN, N. J. HILLSON, AND J. D. KEASLING, *Design, Implementation and Practice of JBEI-ICE: an Open Source Biological Part Registry Platform and Tools*, *Nucleic Acids Res.*, 40 (doi: 10.1093/nar/gks531, 2012).
- [9] N. HILLSON, H. PLAHAR, J. BEAL, AND R. PRITHVIRAJ, *Improving synthetic biology communication: Recommended practices for visual depiction and digital submission of genetic designs*, *ACS synthetic biology*, 5 (2016), pp. 449–451.
- [10] M. HUCKA, A. FINNEY, H. M. SAURO, H. BOLOURI, J. C. DOYLE, H. KITANO, , THE REST OF THE SBML FORUM:, A. P. ARKIN, B. J. BORNSTEIN, D. BRAY, A. CORNISH-BOWDEN, A. A. CUELLAR, S. DRONOV, E. D. GILLES, M. GINKEL, V. GOR, I. I. GORYANIN, W. J. HEDLEY, T. C. HODGMAN, J.-H. HOFMEYR, P. J. HUNTER, N. S. JUTY, J. L. KASBERGER, A. KREMLING, U. KUMMER, N. LE NOVÈRE, L. M. LOEW, D. LUCIO, P. MENDES, E. MINCH, E. D. MJOLSNESS, Y. NAKAYAMA, M. R. NELSON, P. F. NIELSEN,



- T. SAKURADA, J. C. SCHAFF, B. E. SHAPIRO, T. S. SHIMIZU, H. D. SPENCE, J. STELLING, K. TAKAHASHI, M. TOMITA, J. WAGNER, AND J. WANG, *The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models*, *Bioinformatics*, 19 (2003), pp. 524–531.
- [11] Y. LIU, Y. ZENG, L. LIU, C. ZHUANG, X. FU, W. HUANG, AND Z. CAI, *Synthesizing and Gate Genetic Circuits Based on CRISPR-Cas9 for Identification of Bladder Cancer Cells*, *Nature Communications*, 5 (2014), p. 5393.
- [12] C. MADSEN, J. A. McLAUGHLIN, G. MISIRLI, M. POCOCK, K. FLANAGAN, J. HALLINAN, AND A. WIPAT, *The SBOL Stack: A Platform for Storing, Publishing, and Sharing Synthetic Biology Designs*, *ACS Synthetic Biology*, (2016).
- [13] C. MADSEN, C. MYERS, T. PATTERSON, N. ROEHNER, J. STEVENS, AND C. WINSTEAD, *Design and Test of Genetic Circuits Using iBioSim*, *IEEE Design and Test of Computers*, 29 (2012), pp. 32–39.
- [14] G. MISIRLI, A. WIPAT, J. MULLEN, K. JAMES, M. POCOCK, W. SMITH, N. ALLENBY, AND J. S. HALLINAN, *Bacillondex: An Integrated Data Resource for Systems and Synthetic Biology*, *Journal of Integrative Bioinformatics (JIB)*, 10 (2013), pp. 103–116.
- [15] T. NGUYEN, N. ROEHNER, Z. ZUNDEL, AND C. J. MYERS, *A converter from the systems biology markup language to the synthetic biology open language*, *ACS synthetic biology*, 5 (2016), pp. 479–486.
- [16] E. OBERORTNER, J.-F. CHENG, N. J. HILLSON, AND S. DEUTSCH, *Streamlining the design-to-build transition with build-optimization software tools*, *ACS synthetic biology*, 6 (2016), pp. 485–496.
- [17] C. OLSEN, K. QAADRI, H. SHEARMAN, AND H. MILLER, *Synthetic Biology Open Language Designer*, 2014 International Workshop on Bio-Design Automation, (2014), pp. 60–61.
- [18] J. QUINN, R. COX, A. ADLER, J. BEAL, S. BHATIA, Y. CAI, J. CHEN, K. CLANCY, M. GALDZICKI, N. HILLSON, N. NOVRE, A. MAHESHWARI, J. ALASTAIR, C. MYERS, P. UMESH, M. POCOCK, C. RODRIGUEZ, L. SOLDATOVA, G. STAN, N. SWAINSTON, A. WIPAT, AND H. SAURO, *SBOL Visual: A Graphical Language for Genetic Designs*, *PLOS Biology*, (2015).
- [19] N. ROEHNER, J. BEAL, K. CLANCY, B. BARTLEY, G. MISIRLI, R. GRUNBERG, E. OBERORTNER, M. POCOCK, M. BISSELL, C. MADSEN, T. NGUYEN, M. ZHANG, Z. ZHANG, Z. ZUNDEL, D. DENSMORE, J. GENNARI, A. WIPAT, H. SAURO, AND C. MYERS, *Sharing Structure and Function in Biological Design with SBOL 2.0*, *ACS Synthetic Biology*, 5 (2016), pp. 498–506.
- [20] N. ROEHNER AND C. MYERS, *A methodology to annotate systems biology markup language models with the synthetic biology open language*, *ACS synthetic biology*, 3 (2013), pp. 57–66.
- [21] M. ZHANG, J. A. McLAUGHLIN, A. WIPAT, AND C. J. MYERS, *SBOLDesigner 2: An Intuitive Tool for Structural Genetic Design*, *ACS Synthetic Biology*, (2017).

- [22] Z. ZHANG, T. NGUYEN, N. ROEHNER, G. MISIRLI, M. POCOCK, E. OBERORTNER, M. SAMINENI, Z. ZUNDEL, J. BEAL, K. CLANCY, A. WIPAT, AND C. MYERS, *libsbolj 2.0: A Java Library to Support SBOL 2.0*, IEEE Life Sciences Letters, 1 (2015), pp. 34–37.