

SOFTWARE COMPLIANCE TESTING FOR THE
SYNTHETIC BIOLOGY OPEN LANGUAGE

by


Meher Samineni

A Senior Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the Degree

Bachelor of Computer Science

School of Computing
The University of Utah
May 2017

Approved:

 /05/05/2017
Chris Myers
Supervisor

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

_____/_____
Ross Whitaker
Director
School of Computing

Copyright © Meher Samineni 2017

All Rights Reserved

ABSTRACT

Data Standards provide a way of expressing data in a uniform manner. In addition to standardizing a format for encoding data, data standards allow for data to be exchanged easily and meaningfully. Standards, commonly, enable applications to easily communicate and pass data to one another; however, this seamless communication between applications is impossible if applications rely on different data standards that encode data differently. This thesis proposes a workflow methodology for best-effort automatic conversion or translation of meta-data from one data standard to another while minimizing the loss of data. The objective of the methodology is to validate the conversion and determine the compatibility between two tools and their underlying data standards. The standard-enabled workflow and methodology created should analyze a given workflow of tools to see if data is lost within the workflow and ensure that the data is still compliant with a standard as the data flows through various tools. To determine how well the methodology works, Synthetic Biology tools are evaluated to see valid connections can be made with other tools while maintaining compliance within the data standard supported by the tool.

CONTENTS

ABSTRACT	
LIST OF FIGURES	ii
ACKNOWLEDGMENTS	iii
CHAPTERS	
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Contributions of this Thesis	2
1.3 Thesis Overview	2
2. SYNTHETIC BIOLOGY OPEN LANGUAGE	3
2.1 Structural Data Classes	6
2.2 Functional Data Classes	7
2.3 Additional Data Classes	7
3. SBOL SOFTWARE SURVEY	9
3.1 Types of Software	10
3.2 SBOL Support Within Applications	13
4. ANALYSIS OF THE SBOL TEST SUITE	15
4.1 Algorithm for the Analysis	16
4.2 Results of the Analysis	19
5. CONCLUSIONS	26
5.1 Summary	26
5.2 Future Work	26
REFERENCES	28

LIST OF FIGURES

2.1 Biological Design Standards Format Evolution. SBOL expands beyond previous formats which only allows expression of sequences to include hierarchical representations of the structure and functional information of a genetic design. SBOL 1 allows for DNA components to be described without requiring sequences to be assigned to each component. SBOL 2 furthers this format by enabling more types of components and their interactions to be described. (figure courtesy of Zundel et al.[26])	4
2.2 Main Classes of SBOL Data Model (figure courtesy of Beal et al.[2]). The dark yellow classes represent the top level structural data classes within the SBOL Data Model with the lighter yellow representing the supporting structural data classes. Dark green classes represent the top level functional data classes with the supporting functional data classes marked in light green. Lastly, the third group of data classes marked in blue represent top level additional classes that do not strictly fall under a specific SBOL data class. .	5
3.1 Availability of SBOL Applications	11
3.2 OS/Platform Requirements for Applications	11
3.3 Structural vs. Functional Levels of Applications	12
3.4 Software Functionality	12
3.5 Applications Supporting SBOL Visual	13
3.6 Applications able to import SBOL	14
3.7 Applications able to export SBOL	14
4.1 A graphical representation of SBOL Examples and their relations based on the data types supported. Diamond nodes are source nodes and have the largest number of SBOL data types represented in this cluster of examples. Nodes colored yellow indicate the examples that only represent the structural SBOL Data Classes. Green nodes indicate the examples that represent the functional SBOL Data Classes.	25

ACKNOWLEDGMENTS

I would like to first and foremost thank my adviser, Chris Myers. I joined Chris' lab as a junior and his mentorship has provided me with valuable support in my time as an undergraduate. I am greatly indebted to Chris' patience in guiding my work. I gained confidence and insight in my work as a researcher and Computer Scientist. I am incredibly thankful and will be forever grateful to all the encouragement and time he has spent with me providing feedback, advice, and mentorship.

I would also like to thank my fellow undergraduate and graduate researchers in the Myers Research Group. Particularly, I would like to thank Zach Zundel, Tramy Nguyen, Leo Watanabe, Zhen Zhang, and Michael Zhang for always being willing to lend a helping hand, providing advice, and always listening during the times I felt incredibly overwhelmed.

Without my parents and all the sacrifices they've made, I would not be able to be the person I am today. I would like to express my immense appreciation for them and for the values they have instilled within me including determination, courage, and the incredible importance of education. Lastly, I would like to express my gratitude to my sister, Sai, for always believing in me even when I did not always share the sentiment. I would not have had the emotional and mental support needed to survive my undergraduate career without her. I would not be the person I am today without her unconditional love and support she has always provided me. I am genuinely grateful for her continued determination in seeing me through every major milestone of my life.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Similar to how a language provides a basis for two people to communicate, data standards provide guidelines for how data can be exchanged meaningfully and in a uniform manner. For example, the Synthetic Biology Open Language (SBOL) Standard is used to represent biological data within a genetic design [2]. Standards are important for the reason that they allow an environment for multiple types of data to be understood. Moreover, standards enable applications to share and translate data across various applications and platforms that do not necessarily all support the same data standard or only support parts of a common data standard. Without data standards applications cannot rely on a seamless communication since various applications can encode data differently. Data standards not only facilitate information to be exchanged freely, but enable workflows to be designed to support data exchanges.

Generally speaking, standards provide crucial support for a software community. Different entities within a community might create different applications, but in order to maintain re-usability and data reproducibility, a standard is used. Furthermore, standards allow for interoperability between groups of applications to connect and share data. However, while standards facilitate exchange of information, they do not guarantee that all applications are compatible to exchange data or if an exchange is achieved, the translated data makes sense.

Compliance is the idea of ensuring that applications are incorporating standards correctly, particularly after an exchange of data has taken place. For applications exchanging data under the purview of a standard, the exchange must be validated using some guidelines of compliance. While it's ideal to assume that an exchange between tools is automatically successful, the data translated might not be legal or valid any longer.

Therefore, data exchanges between applications need to be evaluated under conditions to ensure that compliance with the standard is met.

1.2 Contributions of this Thesis

The SBOL Standard is the main standard used as a case study within this thesis. Specifically, this thesis presents a methodology for compliance testing specifically through analysis of the applications supporting the SBOL standard. Additionally, this thesis produces the updated information regarding current applications supporting the SBOL Standard. This information is gained from the responses of the SBOL software survey. Lastly, this thesis provides the results of analyzing the SBOL Test Suite to determine a testing strategy that determines compatibility between applications and how the SBOL data standard is internally supported within each SBOL application.

The main contributions of this research include

- Tabulated results of the current software tools/applications supporting the SBOL Standard
- An analysis and evaluation of the SBOL Test Suite and an algorithm for compliance testing of SBOL Applications
- The discussion of an algorithm for compliance testing of SBOL Applications.

1.3 Thesis Overview

This thesis is organized within four chapters. Background information on the SBOL standard and data model is provided in Chapter 2. This chapter gives detailed overview of the structural and functional components of the SBOL data model used as a platform to encode biological design information. The results of the SBOL Software survey gathered for various applications supporting the SBOL Standard is detailed in Chapter 3. Chapter 4 details the algorithm created to analyze SBOL Software Applications. Additionally, the results and discussion of applying the testing algorithm to various applications is detailed in Chapter 4. Lastly, Chapter 5 provides a summary of this thesis and the direction of future work.

CHAPTER 2

SYNTHETIC BIOLOGY OPEN LANGUAGE

Given the motivation for the importance of standards and compliance, the rest of this thesis focuses on creating the methodology for compliance testing of standards. However, due to the infinite amount of applications that exist and the standards which they encode, this thesis specifically utilizes the Synthetic Biology Open Language (SBOL) and the applications that support SBOL as the case study to analyze. This chapter introduces the SBOL Standard.

The Synthetic Biology Open Language (SBOL) is developed to specify the information within a biological construct. Biological designs are described in both a structural and functional manner. While other biological design standards support representing information in a unilateral manner, SBOL is able to describe a design in a multi-level fashion. This evolution of Biological Design Standards is shown in Figure 2.1. FASTA represents only the nucleotide sequencing data of a design, GenBank format provides more detail regarding the components within a biological design by annotating the positions of the sequence, but the complete sequence is required. SBOL provides a format describing both structural and functional information of a genetic design. structural description of a design is the information describing the chemical makeup of entities i.e sequencing data [2]. The functional description of a design describes behavior of the design and the interactions between entities [2]. SBOL 1 enables incomplete designs to be expressed in a modular, hierarchical format through composition of DNA components without requiring the sequences for components [1]. This is extended within SBOL 2 which enables more types of components such as non-DNA components, proteins, and small molecules and their interactions to be described [2]. Additionally, various software libraries have been developed to ease the incorporation of the SBOL Standard into applications such as the SBOL java library, *libSBOLj* [25]. In order to achieve the goal of describing biological information on a structural and functional level, a well-defined data model exists.

In addition to the SBOL data standard, there exists the SBOL Visual Standard that enables genetic designs to be visually expressed. SBOL Visual is a graphical notation that uses schematic “glyphs” to specify genetic components and systems [21]. Additionally, SBOL Visual allows different regions of DNA components to be notated using these “glyphs.” For example, Figure 2.1 visually represents the promoter, ribosome binding, and terminator regions using SBOL Visual.

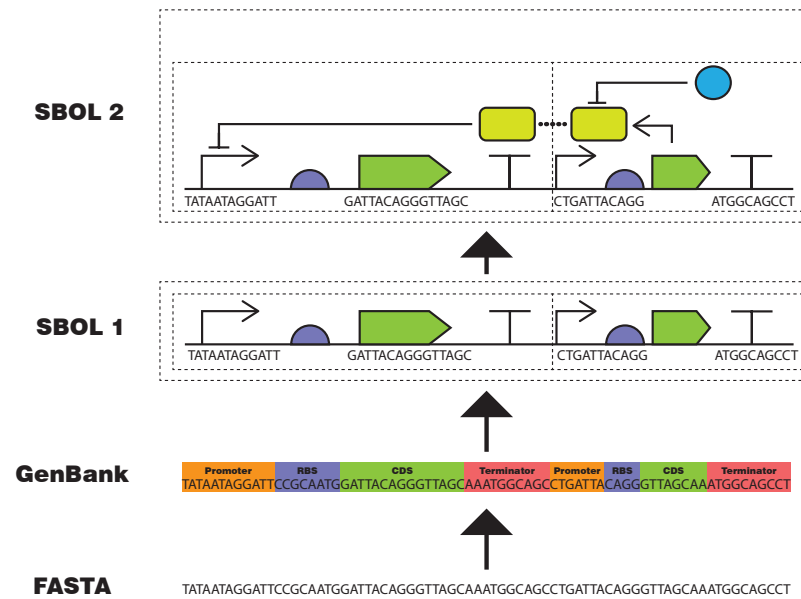


Figure 2.1: Biological Design Standards Format Evolution. SBOL expands beyond previous formats which only allows expression of sequences to include hierarchical representations of the structure and functional information of a genetic design. SBOL 1 allows for DNA components to be described without requiring sequences to be assigned to each component. SBOL 2 furthers this format by enabling more types of components and their interactions to be described. (figure courtesy of Zundel et al.[26])

To give an overview of the SBOL Data Model, all classes stem from the abstract **Top Level** class. As the **Top Level** class is an abstract class, it is not directly referenced, but rather indirectly implemented through six key classes. Those classes that inherit from the Top Level class directly are considered as parent classes that are never nested under any other object. The **Top Level** class is characterized through the following classes: **Sequence**, **ComponentDefinition**, **Model**, **ModuleDefinition**, **Collection**, **GenericTopLevel**. The **ComponentDefinition**, **Sequence**, and **Collection** classes

and their supporting classes represent the structural entities within the SBOL Standard. **ModuleDefinition** and **Model** classes' purpose is to represent functional entities. Figure 2.2 shows the main data type classes representing biological information and their relationships within the SBOL Standard. The dark green colored types are the “top level” functional classes under which all other supporting functional SBOL classes fall that are marked in light green. The dark yellow colored types denote the “top level” structural components of SBOL and the classes marked with light yellow denote the main supporting structural data types represented in the standard. These classes are explained within the next few sections. The associations between classes are indicated using the arrows as specified by UML semantics. Solid arrows indicates ownership of the class which the arrow is pointing towards. Dashed arrows symbolizes one class referring to an object of another class [2].

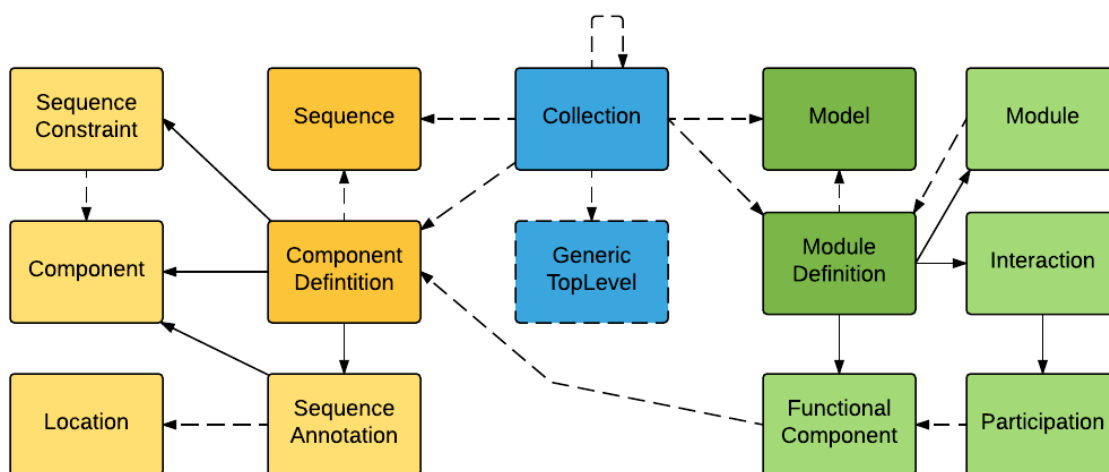


Figure 2.2: Main Classes of SBOL Data Model (figure courtesy of Beal et al.[2]). The dark yellow classes represent the top level structural data classes within the SBOL Data Model with the lighter yellow representing the supporting structural data classes. Dark green classes represent the top level functional data classes with the supporting functional data classes marked in light green. Lastly, the third group of data classes marked in blue represent top level additional classes that do not strictly fall under a specific SBOL data class.

At this point, there is a sufficient overview regarding the representation of the SBOL Data Model to introduce the Top Level classes and their sub-classes essential to understanding how the data model functions as a system. First, Section 2.1 introduces the

structural classes, followed by Section 2.2 that introduces the functional classes. Lastly, Section 2.3 describes a third group of additional data classes that do not necessarily classify completely as structural or functional classes. These last group of classes do not specify biological design data, but act as extension classes to capture data that does not explicitly fall under a SBOL Data class.

2.1 Structural Data Classes

The **ComponentDefinition** class represents the structural entities within a biological design [2]. The components that represent DNA, RNA, protein i.e the structural entities are designed using **ComponentDefinition** objects. While this is the main purpose of this data class, **ComponentDefinition** objects is also used to represent other types of structural entities that exist within a biological design such as small molecules and complexes. A *Sequence* object, which is soon introduced, is used to define the genetic coding within the structural entity. Additionally, there are sub-classes that further assist this class including the **Component**, **SequenceAnnotation**, and **SequenceConstraint** classes which capture more details regarding the entity being represented. These classes will be introduced later in chapter 2.

The **Component** class is a child of the **ComponentDefinition** class. Its purpose is to define sub-entities and their structural uses. For example, a gene is represented using a **ComponentDefinition**. However, the substructures within a gene include a promoter, terminator, and a coding region which are represented by **Components**.

Within a **Sequence** object belonging to a **ComponentDefinition**, it is ideal to notate specific positions of the sequence. This function is achieved through a **SequenceAnnotation** object. To specifically notate the position, a **Location** object is used. The **Location** class is an abstract that allows a region of a coding sequence within a **SequenceAnnotation** object to be notated either through a **Range**, **Cut**, or **GenericLocation** object. A **Range** object denotes the sequencing data between a given start and end position of the data. Alternatively, a **Cut** object notates position at an specified index within a sequence. Lastly, **GenericLocation** allows position access within a **Sequence** object containing different genetic encodings or to annotate objects that lack sequence data. In addition to notating specific positions of a sequence, a **SequenceConstraint** object allows for rules to be specified regarding the location and orientation of substructures.

The **Sequence** class represents the genetic code within a **ComponentDefinition** object. The main purposes of this class include representing the genetic coding of the

constituents of a biological entity and identifying the meaning behind the genetic encoding. For example, the nucleotide bases of a DNA molecule. Using the same example, the elements of a nucleotide bases must be *t*, *c*, *a*, *g*.

2.2 Functional Data Classes

The **ModuleDefinition** class allows for grouping of the structural and functional entities in a biological design [2]. The main purpose of this class is to track the function and molecular interactions between entities within a biological design. A **ModuleDefinition** references a set of **FunctionalComponent** objects, the **Interactions** between entities, and **Modules** of a biological design.

As discussed earlier, the entities within a design are represented as **ComponentDefinition** objects. In order to instantiate the created object, a **FunctionalComponent** object is defined. A **FunctionalComponent** object is the use case of a created entity. Within a biological construct, rarely are entities the only participants in creating biological processes. Rather, there are many entities that connect and interact within a design to produce some function. For this purpose, **Interactions** are designed in order to provide the context for how **FunctionalComponents** behave together such as representing the biological processes of transcription and translation. Within a **Interaction**, a set of **Participation** objects are typically created to denote the entities participating within an **Interaction**.

ModuleDefinition objects can contain abstract entities representing various components. These components do not necessarily reference a specific part with genetic information, but act as placeholders for more specific entities to replace the abstract entities. This functionality is achieved through a **MapsTo** object. A **MapsTo** object defines the the relationships between the abstract entity and the specific component.

The **Model** class allows for an external computational model to be referenced and for meta-data of the contents of a model to be tracked. This class allows for an abstraction so that there isn't duplication of designs.

2.3 Additional Data Classes

The **Collection** class allows for **TopLevel** objects with a common feature to be grouped together. For example, a set of **ComponentDefinition** objects representing the biological entity of a promoter are placed within a **Collection** to be accessed later.

Annotation objects are created to attach information to any SBOL object. This attached information does not change the meaning of the SBOL object, but adds extra description to the referenced part. For example, a **ComponentDefinition** object might contain an annotation with the location of the imported source data [2].

The last class to briefly mention within the SBOL Data Model are the **GenericTopLevel** objects. These objects act as a catch all mechanism to retain information regarding a biological construct which cannot be internally well-defined by an existing SBOL class. The entities that are created using a **GenericToplevel** object contain annotations with information that can be used to exchange non-SBOL related data.

CHAPTER 3

SBOL SOFTWARE SURVEY

This thesis is concerned with creating a testing methodology specifically for software applications that support the SBOL Standard. In order to test applications, a survey was created and dispersed to developers of applications within the SBOL community with the goal of creating a compiled list of the software applications that currently support SBOL. This chapter discusses the responses from the survey in Section 3.1. The application information gained from the survey is utilized in evaluating the created testing methodology which will be discussed in Chapter 4. Twenty-nine applications responses were collected from the survey and the compiled results are shown in Table 3.1. The questions within the survey focused on gaining a comprehensive understanding of an application's capabilities and extent of SBOL support the application provides. To meet this objective, there are three main types of questions asked. The first was a general overview of information regarding the application. The second type of questions included the functionality and usage of the applications. The last type of questions related to the capacity in which the SBOL Standard was supported. Section 3.1 discusses the results of the first two types of questions whereas Section 3.2 discusses the current SBOL support within applications.

Name	Function					SBOL			URL
	R	S	V	G	M	1	2	v	
BOOST [19]		•				•	•		boost.jgi.doe.gov
Cello [18]				•	•		•		cellocad.org
DeviceEditor[8]		•	•	•	•	•		•	j5.jbei.org
DNAPlotLib[9]			•			•		•	dnaplotlib.org
Eugene [5]		•		•		•		•	http://www.eugenecad.org
Finch	•	•	•	•			•	•	synbiotools.org
GenoCAD	•	•	•		•			•	www.genocad.com
GeneGenie		•				•			gene-genie.org
Graphviz			•					•	www.graphviz.org
ICE[10]	•		•			•	•	•	public-registry.jbei.org
iBioSim[17]		•	•	•	•	•	•	•	www.async.ece.utah.edu/ibiosim
j5[11]		•							j5.jbei.org
MoSeC[16]		•		•		•			ico2s.org/software/mosec.html
Pigeon[4]			•					•	pigeoncad.org
Pinecone	•	•						•	serotiny.bio
Pool Designer[23]		•				•	•		github.com/CIDARLAB/poolDesigner
Proto BioCompiler[3]			•	•	•	•		•	synbiotools.bbn.com
SBOLDesigner [24]		•	•			•	•	•	www.async.ece.utah.edu/SBOLDesigner
SBOLme[12]				•				•	www.cbrc.kaust.edu.sa/sbolme
ShortBol[20]		•		•				•	shortbol.ico2s.org/sandbox.html
SynBioHub[13]	•		•			•	•	•	synbiohub.org
Tellurium[22]		•		•	•			•	tellurium.analogmachine.org
TeselaGen		•	•			•		•	www.teselagen.com
TinkerCell[7]			•	•	•	•		•	www.tinkercell.com
VisBOL[14]			•					•	visbol.org
VirtualParts[15]				•				•	www.virtualparts.org

Table 3.1: A partial list of software supporting SBOL. An up-to-date list is maintained on <http://sbolstandard.org>. The function column indicates if the tool is a (R)epository, (S)equence design tool, (G)enetic circuit design tool, (M)odeling and simulation tool, or a (V)isualization tool. The SBOL column indicates if it supports SBOL(1), (2), or (v)isual. (figure courtesy of Myers et al [6])

3.1 Types of Software

Figures 3.1 and Figure 3.2 are a few examples of the questions asked regarding the platforms and licenses applications supported. Most applications are hosted under an Open-Source license. There is not any preference for any particular OS, but there are slightly higher statistics for web-based applications.

One of the key points of this survey is the breakdown between how applications supported both structural and functional aspects of SBOL. Figure 3.3 shows the breakdown of the applications supporting the SBOL Data Model. 41.4 percent of applications support SBOL structurally only while only 13.8 percent of applications specifically are able to support SBOL functionally. 44.8 percent of applications claim to support both. The testing methodology must take into account that applications that state they can only support only one level can only be tested with SBOL data examples supporting that level

What is the availability of this software tool?

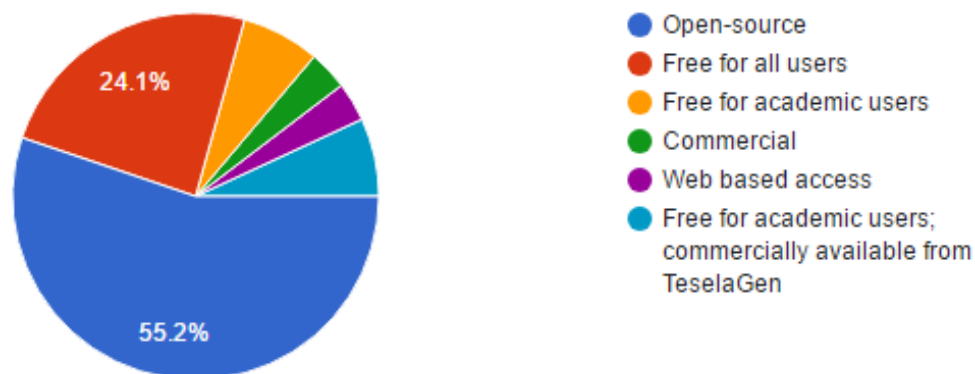


Figure 3.1: Availability of SBOL Applications

OS/platform that this software tool supports (check all that apply)

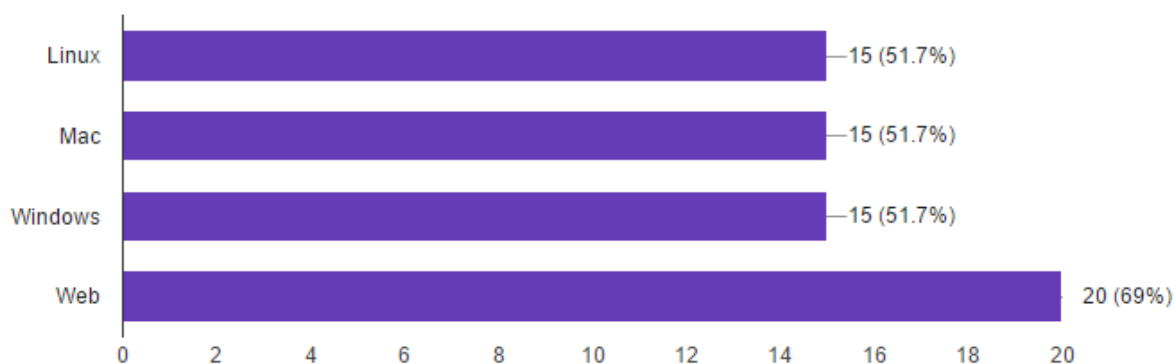


Figure 3.2: OS/Platform Requirements for Applications

of data. Another point is the applications that fall under 'both' levels still could only partially support parts of the classes within the data model.

Figure 3.4 references the functionality in which various applications provide. An application can have multiple capabilities, so there is overlap among the categories. Predictably, however, the largest category is that applications allow for designing sequence and biological/genetic constructs. Fifteen of the Twenty-nine applications state they could support Biological/Genetic Circuit design and fourteen of twenty-nine state functionality

for Sequence designing capability. Other categories include twelve of twenty-nine applications state they support Visualization of created designs.

What level does this software work at?

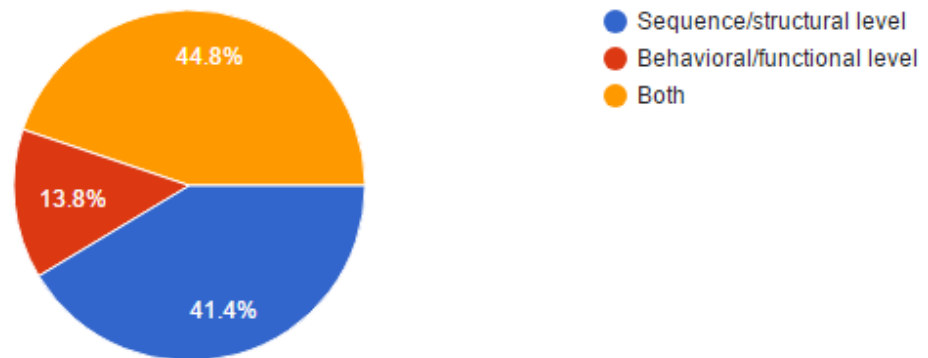


Figure 3.3: Structural vs. Functional Levels of Applications

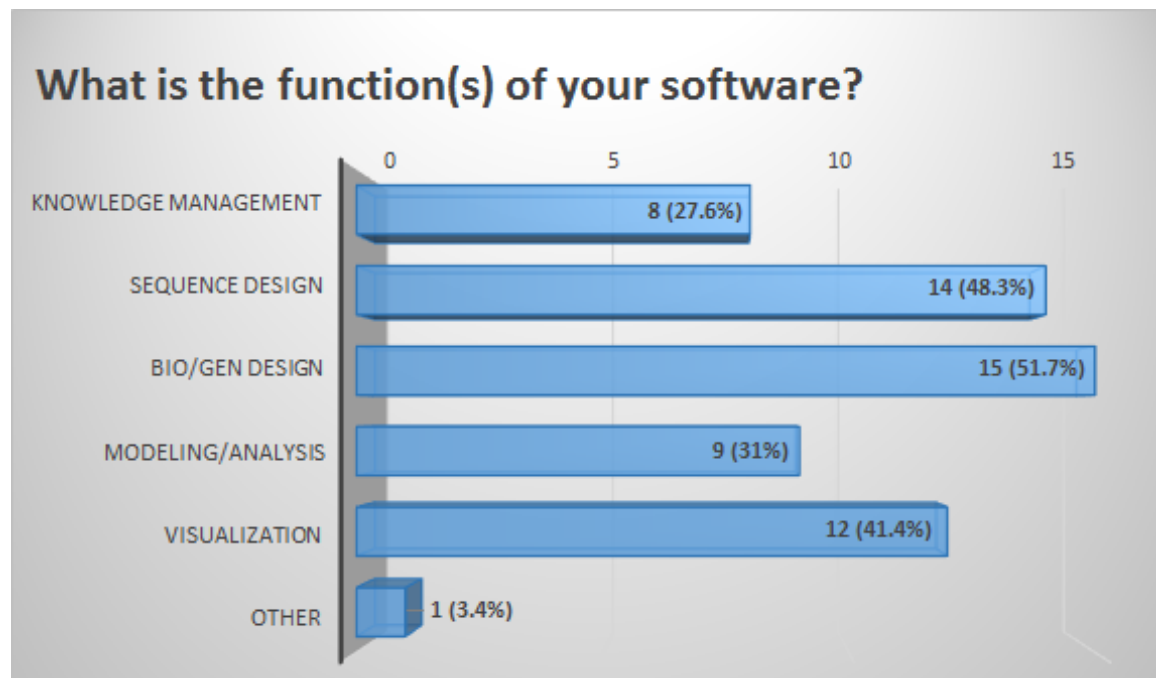


Figure 3.4: Software Functionality

3.2 SBOL Support Within Applications

Figure 3.5 represents SBOL applications supporting SBOL Visual. SBOL Visual defines a graphical notation to define genetic components and designs [21]. Of the applications queried, 58.6 percent report they do support SBOL Visual.

Does this software tool support SBOL Visual?

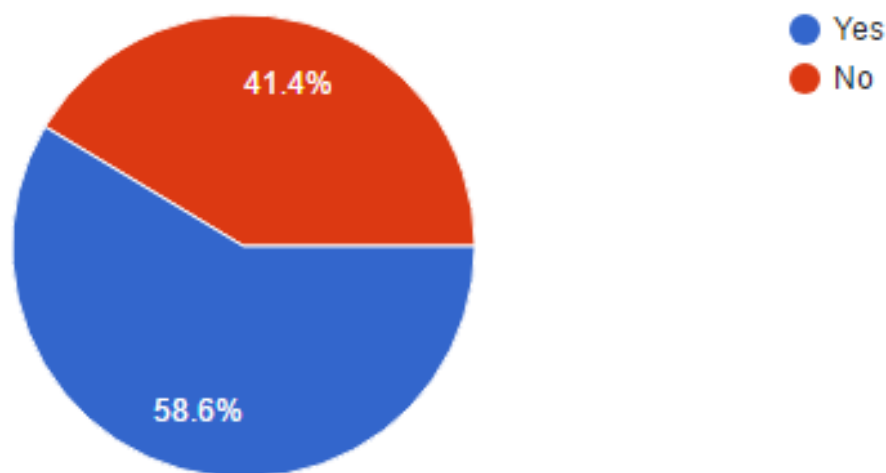


Figure 3.5: Applications Supporting SBOL Visual

Figures 3.6 and 3.7 are questions asked regarding SBOL support integrated within the application. Figure 3.6 are the results for whether applications are able to read and understand the contents of a data within an SBOL example file. Figure 3.7 are the results for whether applications are able to export a SBOL file containing valid SBOL data of the design built within the application. Applications largely support SBOL 1.0 which only supports structural classes within the data model. However, there were nine of twenty-nine applications which support SBOL 2.0 which includes multi-level support as well support within Genbank and FASTA formats.

With an understanding of the software applications that currently support SBOL and the information of how they support SBOL, the next chapter discusses the testing algorithm created to test SBOL applications and analyze a set of SBOL Examples used as the input to test SBOL applications.

Which standards can this software tool import?

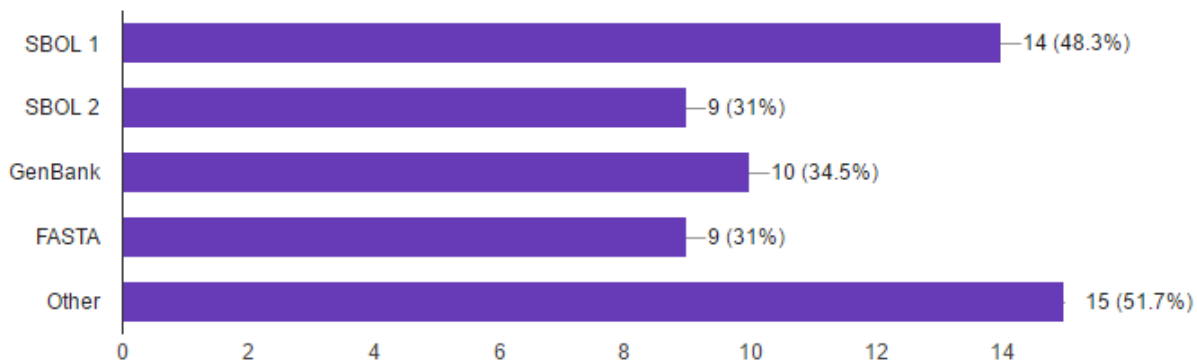


Figure 3.6: Applications able to import SBOL

Which standards can this software tool export?

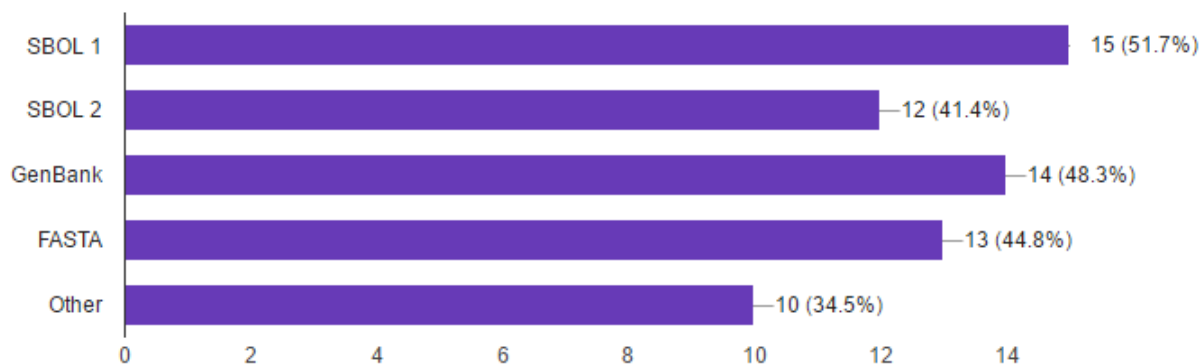


Figure 3.7: Applications able to export SBOL

CHAPTER 4

ANALYSIS OF THE SBOL TEST SUITE

The survey results provide a group of software applications to test the compatibility between applications through their data exchanges. Furthermore, the survey provides information regarding how each application supports SBOL. Since these results are self-reported, the claims of what an application can support must be verified. The three questions relating to SBOL support within the survey include if an application can support SBOL Visual, import SBOL, and export SBOL. Different testing methods are required to verify each level of SBOL support. First, to verify an application supports SBOL Visual, there is simply no other way to verify an application's correct usage other than through manual inspection. Additionally, to verify the claims that an application can export legal SBOL data, the SBOL Validator tool can be used. The SBOL Validator is a software tool that checks the validity of SBOL files to verify that the SBOL Standard is correctly implemented [26]. Checking for the first two types of SBOL support is fairly easy. However, verifying the claims of whether an application imports SBOL data correctly requires more in-depth testing. Import operations are slightly more complex to ensure that the meaning of data imported is not changed upon an import operation. Round-trip testing is used to test import operations. A round-trip test consists of importing SBOL data into an application and then exporting the imported data and performing a comparison operation on the imported data and exported data. If no differences exist between the input and output data, then no data has been transformed or lost and the application is able to correctly import SBOL data.

A series of SBOL files representing various biological designs were previously created to test the *libSBOLj* library. These examples are used as the input data to the applications learned from the survey to implement any of the various testing strategies of SBOL support. First, these examples are analyzed to determine areas of the SBOL Data Model represented and how robustly SBOL was represented. This analysis begins the start of the testing methodology to robustly test and verify the claims from the survey responses of

the SBOL software applications. Section 4.1 of this chapter details the algorithm created to analyze the SBOL Suite of Examples. The results of the analysis are discussed within Section 4.2.

4.1 Algorithm for the Analysis

In order to analyze applications supporting the SBOL Standard, the algorithm established creates a logical understanding of the various biological designs in relation to the SBOL Data Model. These biological designs are created for the purpose of testing SBOL's java library *libSBOLj*. While these designs are robust in the diversity of both structural and functional classes integrated from the SBOL Data Model and the pairings of classes, there is not an obvious way to identify how an application supports SBOL when given one of these biological designs as input. Therefore, this following section explains the algorithm created to organize the created biological designs.

Algorithm 4.1: Pseudocode to Count Data Types in an SBOL Example

Input: SBOLDocument doc, Set *Files*, *SetTypes*

Output: Map $\langle S, C \rangle$ m

```

1 foreach f in Files do
2   doc = read(f)
3   foreach t in Types do
4     if doc.contains(t) then
5       c = count(doc, t)
6       map t to c

```

The goal of this algorithm is to understand the current data given within the series of SBOL files representing various biological designs. The files are placed into a set and then each file is read into an **SBOLDocument** individually using *libSBOLj*. As explained within Algorithm 4.1, the types introduced within the SBOL Data Model are given as set. Then each file identifies the types contained and a count is associated with each type. The purpose of this is to understand the extent of the SBOL Data Model support within the existing biological designs.

The second goal of the algorithm is to create relationships between biological designs that contain the same type of data. A cluster is defined to have a set of SBOL data types and a set of the files with the data that contains those specific data types.

Algorithm 4.2 shows the pseudocode of the function to create the clusters. The set of files with the associated data counts is given as input. The function works such that

Algorithm 4.2: Pseudocode to Create Data Type Clusters

Input: Map $\langle F, \text{Map} \langle T, C \rangle \rangle$ m, Set Files
Output: Map $\langle \text{Set}F, \text{Set } clus \rangle$ clusters

```

1 while Files is not empty do
2   f = choose f in Files
3   remove f from Files
4   Set givenTypes = m.get(f)
5   create Set cluster
6   create Set types
7   add f to cluster
8   foreach t in givenTypes do
9     if t(c) != 0 then
10      types.add(t)
11  foreach f in Files do
12    Set check = m(F)
13    flag = true
14    foreach t in givenTypes do
15      if given(t) != 0 and check(t) == 0 then
16        flag = false
17      if given(t) == 0 and check(t) != 0 then
18        flag = false
19    if flag then
20      cluster.add(F)

```

an SBOL file is chosen at random and removed from the remaining list of files. A cluster is created with the chosen file as the only member included. Then using the information previously gained from the algorithm 4.1, the types existing in the file with counts greater than zero are placed into a set of data types. In order to determine what other files are members of this cluster, every other file is checked such that the count data is the same for each file as the chosen file, then the current file is placed into the cluster. Once all the files are checked and then a set of clusters with files that contain common data types is created.

While the clusters are able to group the SBOL data files, the third and final goal of the algorithm is to create relations between the clusters. In order to do this function 4.3 iterates through the clusters and chooses two different clusters at random. One cluster is marked arbitrarily as the parent and the other cluster as the child. To see if a direct relation exists between the two clusters, each cluster within the remaining clusters is checked to ensure that the third cluster is not a subset of the parent cluster and the child is not a subset of the third cluster. The subset relation in this function is defined through

Algorithm 4.3: Graph Creation pseudocode representing SBOL Examples

Input: Map $\langle \text{SetF}, \text{Set } \textit{clus} \rangle$ clusters
Output: Graph g

```

1 for Parent p in clusters do
2   for Child c in clusters do
3     if  $p == c$  then
4       continue
5     if child not subset of parent then
6       continue
7      $\textit{flag} = \text{true}$ ; for otherCluster in clusters do
8       if  $\textit{otherCluster} == \textit{parent}$  or  $\textit{otherCluster} == \textit{child}$  then
9         continue
10      if otherCluster subset of parent and child subset of otherCluster then
11         $\textit{flag} = \text{false}$ 
12      if  $\textit{flag}$  then
13        Edge  $e$  from parent to child

```

the common data types. If every data type belonging to a cluster also is contained within another cluster, then the first cluster is considered a subset of the second cluster.

4.2 Results of the Analysis

This section aims to discuss the key details of the graph from analyzing SBOL examples to test SBOL applications. Figure 4.1 is the graphical representation of analyzing SBOL examples based on their data types and their relations. The diamond shaped source nodes represent a superset of the data types common to a group of examples. No overlaps exist between the source nodes because each source node represents one unique superset with one particular set of data types that are represented. The remaining portion of the graph consists of child nodes that follow parent-children relationships based on the groups of data types found within examples. Each child node stems from one or more of the source nodes and each child node consists of examples that contain a subset of the SBOL data types contained by its immediate parent. An example of how to interpret nodes in the graph includes the parent node *6* which is a source node whose set of data types consists of **{Participation, Interaction, ModuleMapsTo, SequenceAnnotation, Sequence, FunctionalComponent, Range, ModuleDefinition, Model, Component, Module, ComponentDefinition, Location}**. There is one biological design example that represents the data types belonging to node *6*. Furthermore, there exists no such node containing a superset of the data types belonging to node *6*. Every child node descending from *6* contains a subset of data types belonging to this node. For example, Node *11* is a child node of *6* and its data types consist of **{String_Annotation, URI_Annotation, SequenceAnnotation, GenericTopLevel, Collection, Sequence, Component, Range, Annotation, ComponentDefinition, Location}**.

As mentioned before, SBOL is organized into structural and functional data types. Each node is marked in either the color green or yellow to denote whether the set of examples belonging to that node represent either structural or functional data types. Yellow colored nodes represent structural data types and green colored nodes represent functional data types. One main observation derived from the graph is there exists subtrees of nodes that point to examples representing only structural or functional data type sets. For example, Node *19* is a source node which only contains structural data types. The details from this node is seen more clearly within Table 4.2 which shows the data types represented by each node. The data types within Node *19* include Top Level classes such as **ComponentDefinition, Sequence**, and supporting classes including **SequenceAnnotation, Location**, and **SequenceConstraint, GenericLocation, Component**, and

Range. Any nodes whose ancestors include this source node only represents structural data types. For example, child nodes *13*, *22*, *8*, *12*, *24*, and *7* contain only structural data. This is however different for source nodes marked yellow. These source nodes consist of examples that either contain functional or structural data. For example, Node *6* is marked green, but its two child nodes are marked different colors. Child node *11* is marked yellow, so there is only structural data that is being represented in this subset of examples. However, Node *5* contains examples that represent functional data. Identifying the nodes as structural vs. functional data is an important feature of the graph to identify what type of data examples represent. This information is used as a testing strategy for testing applications claiming to support either structural, functional, or both types of data.

The last key benefit the graph provides are the pathways that can be followed from any one node to narrow down the possibilities of a unique set of data types represented by a group of examples. An example pathway is Node *19* which contains 12 examples that represent the same group of data types. From this node, Node *13* contains a subset of the data types within Node *19*. If this chain of nodes continues, Node *22* will contain a subset of the data types contained within Node *13*. These pathways are significant because they provide precision to narrow down the possibilities when testing applications to determine exactly what areas of the SBOL Data Model the application can support. For example, if an application fails to read the data from any of the examples that fall under source node *6*, then the next step to choose to data belonging to from a child node such as node *5*. Since the child node represents a subset of the data types, then the testing method include attempting to read in the data from an example within this node and determining if the application properly read the data. If the application once again fails to read the data belonging to the next node within a pathways, then this process can be continued until a node is reached where an application can successfully read the data belonging to an example from that node.

Table 4.3 provides a brief overview of the results examined from Figure 4.1. The results are gained from analyzing eighty-eight SBOL examples. Furthermore, excluding abstract classes within the SBOL Data Model, there are nineteen data classes. Given this information, Node *6* contains the maximum number of data types that exist in a set of examples which is fifteen types. One of the main insights the graph provides indicate that there does not exist at least one example with every data type of the SBOL Data

Model being represented. While there does not exist an all-inclusive example, every single data type is at least represented once within an example. This statistic is determined by performing a union operation of the data types within all the source nodes within the graph and determining that one instance of every data type exists in at least one example. Of the nineteen existing data types, all nineteen types are represented in an example. Nineteen of these data types are represented at least once within an example.

Another detail the graph provides from analyzing the robustness of the examples is the disproportionate variety of examples representing certain data types. The largest group of examples with a set of common data types is represented by Node 28 which contains twenty-eight examples. This statistic is significant because of the complete set of examples given, about 31 percent of the examples are specifically testing only types such as **Sequence**, **ComponentDefinition**, and **Collection**, which are the data types belonging to Node 28. Node 19, which is also a source node, represents the second largest group with twelve examples and there are eight data types common to the examples. The specific set of data types can be found in Table 4.2. The graph allows for analysis that the examples are not broadly testing the scope of the SBOL Data Standard proportionally. Results determine that most examples contain **Sequence** and **ComponentDefinition** data types, but not many examples contain instances of the *Cut* or *MapsTo* data types. Therefore, the examples do not proportionally cover the data types being represented.

One last key insight is the imbalance in the types of data existing within the examples. The analysis shows that 69 percent of the examples represent only structural data classes. The nodes with the most number of examples, as mentioned above, contain only structural data classes in each example. Sixty-one of the eighty-eight examples are marked as supporting only structural data types internally. In contrast, only 31 percent of examples which is composed of twenty-seven examples contained functional data.

The purpose of analyzing the examples and creating a graphical representation of the data types being represented is to provide some insight into how to logically test applications that support SBOL. There now exists a method to be able to test applications and validate the self-reported information taken from the survey responses. One possible strategy is given an application, an example from each the source nodes can be imported into an application to determine the classes it can support, If the application fails to properly import the data within any of these examples, then the examples belonging to the next level child nodes can be imported and checked. This process can be continued

to see what parts of the SBOL data model the application truly does support. This type of testing provides some confidence that the application successfully supports those data types of the SBOL Data Model. Another testing strategy to verify an application claims to supporting both structural and functional data, the data within the examples belonging to the clusters identifying only as structural or functional can be imported into an application. Lastly, round-trip tests can be used to determine if an application is able to exchange data accurately. An example of a round-trip test includes importing a file containing SBOL data into an application, then exporting the data that was just imported into a file. If the data within the output file matches the data within the imported data file, then no data was lost or transformed which is considered a successful round-trip test.

Node	Example Count	Data Count	Source	Structural	Functional
1	4	6		•	
3	1	15	•		•
4	1	10	•		
5	1	13			•
6	1	15	•		•
7	3	1		•	
8	2	1		•	
9	4	1		•	
10	1	2		•	
11	3	9		•	
12	5	6		•	
13	1	3		•	
14	2	1			•
15	1	3			•
16	1	3		•	
17	1	8	•	•	
18	1	5		•	
19	12	8	•	•	
20	3	1			
21	28	3		•	
22	1	2		•	
23	6	7		•	
24	2	2		•	
25	1	2		•	
26	2	1			•

Table 4.1: Each entry represents information within each cluster. The *Example Count* column indicates the number of examples within that cluster. The *Data Count* column represents the number of unique data types found within that set of examples. The *Source* column indicates whether that cluster is a root node in the graph. The *Structural* and *Functional* columns indicate the examples within that cluster represent structural or functional data.

Node	Data Types
1	A, SA, S, R, CD, L
3	A,C,I,SA,S,R,MD,SC,MDL,GTL,C,Comp,Mod,CD,L
4	P, MD, SC, I, MPS, S, FC, Comp, Mod, CD
5	P, I, MPS, SA, S, FC, R, MD, MDL, Comp, Mod, CD, L
6	P,A,MPS,SA,S,FC,R,MD,MDL,GTL,C,Comp,Mod,CD,L
7	S
8	CD
9	C
10	A, CD
11	A, SA, GTL, C, S, Comp, R, CD, L
12	SA, S, Comp, R, CD, L
13	SC, Comp, CD
14	MD
15	MD, I, FC
16	A, GTL, CD
17	A, SA, GTL, S, Comp, R, CD, L
18	Cut, SA, S, CD, L
19	SC, SA, GL, S, Comp, R, CD, L
20	GTL
21	C, S, CD
22	Comp, CD
23	SA, C, S, Comp, R, CD, L
24	S, CD
25	C, S
26	MDL

Table 4.2: Each entry represents the data types found within a set of examples within that node clusters. **CD** denotes *ComponentDefinition*, **S** denotes *Sequence*, **C** denotes *Collection*, **SA** denotes *SequenceAnnotation*, **SC** denotes *SequenceConstraint*, **L** denotes *Location*, **R** denotes *Range*, **Cut** denotes *Cut*, **Comp** denotes *Component*, **MD** denotes *ModuleDefinition*, **FC** denotes *FunctionalComponent*, **Mod** denotes *Module*, **I** denotes *Interaction*, **P** denotes *Participation*, **MDL** denotes *Model*, **MPS** denotes *MapsTo*, **GTL** denotes *GenericTopLevel*, **A** denotes *Annotation*

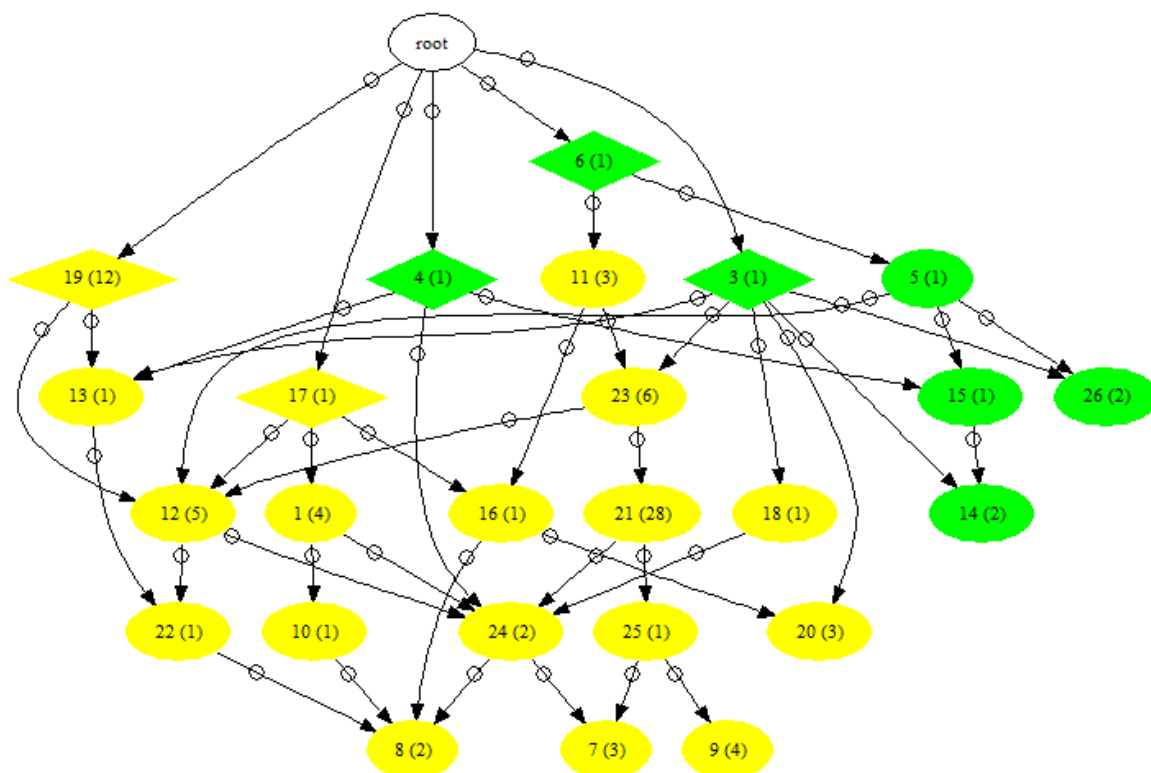


Figure 4.1: A graphical representation of SBOL Examples and their relations based on the data types supported. Diamond nodes are source nodes and have the largest number of SBOL data types represented in this cluster of examples. Nodes colored yellow indicate the examples that only represent the structural SBOL Data Classes. Green nodes indicate the examples that represent the functional SBOL Data Classes.

Graph Statistics	
SBOL Examples	88
SBOL Data Classes	19
Maximum Data Classes	15 (79%)
Data classes Tested	19 (100%)
Structural Examples	61 (69%)
Functional Examples	27 (31%)

Table 4.3: This table represents the results observed from analyzing SBOL Biological Design Examples

CHAPTER 5

CONCLUSIONS

5.1 Summary

This thesis presents a methodology for analyzing software applications and their support of the SBOL Standard. In order to create the methodology, a list of the current SBOL software applications and the data regarding each application's functionality, purpose, and capacity of SBOL support was compiled.

In addition to this compiled data, an algorithm was created to analyze the list of existing biological designs that were created to test the *libSBOLj* java library. By inspecting each biological design to determine the classes of the SBOL Data Model represented internally, clusters were created such that each cluster tracks the same set of SBOL data types for a specific group of biological design examples. These clusters were then paired to create parent-child relationships and segregated into two separate sets based on if the data types within the cluster represent structural or functional data classes.

The algorithm to analyze the SBOL examples produced a testing graph that can be used to test an SBOL Application and determine its level of SBOL Data Model support. Applications claiming to support both structural and functional SBOL data classes can be more accurately verified by performing import and export operations using the examples belonging to the nodes within the graph marked as structural or functional only. The testing methodology created is able to test applications and validate the self-reported information taken from the survey responses.

5.2 Future Work

This thesis began the work to validate the compatibility between SBOL applications through data exchanges. However, to be able to perform the data exchanges, the SBOL Suite of examples need to be completed to have multiple examples representing the full range of the SBOL Data Model. One aspect in representing all the SBOL classes within

various contexts is if the union of all the data types of a node's children is taken, then every single data type belonging to the parent node should be represented within at least one example in a child node. The examples also need to have more distribution of examples supporting a set of data types. Certain nodes contain twenty-eight examples whereas other nodes only contain one example representing a particular data type set. The solution to this goal is to simply include more examples with a wider representation of data type pairings.

Additionally, there need to much more representation of functional classes within various examples. There is an overwhelming amount of examples that only represent structural classes currently. Applications that claimed to support functional classes cannot be properly tested since there is not nearly enough examples with which to test. Once this test suite is complete, then SBOL applications can be more accurately tested to determine their compliance. The testing methodology will then be extended to validate the compatibility between applications through their data exchanges.

REFERENCES

- [1] BARTLEY, B., BEAL, J., CLANCY, K., MISIRLI, G., ROEHNER, N., OBERORTNER, E., POCOCK, M., BISSELL, M., MADSEN, C., NGUYEN, T., ZHANG, Z., GENNARI, J. H., MYERS, C., WIPAT, A., AND SAURO, H. The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology. *Nature Biotechnology*, 32 (2014), 545–550.
- [2] BEAL, J., COX, R. S., GRNBERG, R., MCLAUGHLIN, J., NGUYEN, T., BARTLEY, B., BISSELL, M., CHOI, K., CLANCY, K., MACKLIN, C., MADSEN, C., MISIRLI, G., OBERORTNER, E., POCOCK, M., ROEHNER, N., SAMINENI, M., ZHANG, M., ZHANG, Z., ZUNDEL, Z., GENNARI, J., MYERS, C., SAURO, H., AND WIPAT, A. Synthetic biology open language (SBOL) version 2.1.0. *J. Integrative Bioinformatics* 13, 3 (2016).
- [3] BEAL, J., LU, T., AND WEISS, R. Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks. *PLoS one* 6, 8 (2011), e22490.
- [4] BHATIA, S., AND DENSMORE, D. Pigeon: a design visualizer for synthetic biology. *ACS Synth. Biol.* 2, 6 (2013), 348–350.
- [5] BILITCHENKO, L., LIU, A., CHEUNG, S., WEEDING, E., XIA, B., LEGUIA, M., ANDERSON, J. C., AND DENSMORE, D. Eugene—a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PLoS ONE* 6, 4 (doi: 10.1371/journal.pone.0018882, 2011), e18882.
- [6] C. MYERS, J. BEAL, T. G. H. K. C. M. J. M. G. M. T. N. E. O. M. S. A. W. M. Z. Z. Z. A standard-enabled workflow for synthetic biology.
- [7] CHANDRAN, D., AND SAURO, H. M. Hierarchical modeling for synthetic biology. *ACS synthetic biology* 1, 8 (2012), 353–364.
- [8] CHEN, J., DENSMORE, D., HAM, T. S., KEASLING, J. D., AND HILLSON, N. J. Deviceeditor visual biological cad canvas. *Journal of Biological Engineering* 6, 1 (2012), 1.
- [9] DER, B. S., GLASSEY, E., BARTLEY, B. A., ENGHUUS, C., GOODMAN, D. B., GORDON, D. B., VOIGT, C. A., AND GOROCHOWSKI, T. E. Dnaplotlib: Programmable visualization of genetic designs and associated data. *ACS Synthetic Biology* 0, 0 (0), null.
- [10] HAM, T. S., DMYTRIV, Z., PLAHAR, H., CHEN, J., HILLSON, N. J., AND KEASLING, J. D. 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).

- [11] HILLSON, N. J., ROSENGARTEN, R. D., AND KEASLING, J. D. j5 DNA assembly design automation software. *ACS Synth. Biol.* 1 (2012), 14–21.
- [12] KUWAHARA, H., CUI, X., UMAROV, R., GRÜNBERG, R., MYERS, C. J., AND GAO, X. SBOLme: a repository of SBOL parts for metabolic engineering. *ACS Synth Biol* (Jan. 2017).
- [13] MADSEN, C., McLAUGHLIN, J., MISIRLI, G., POCOCK, M., FLANAGAN, K., HALLINAN, J., AND WIPAT, A. The sbol stack: A platform for storing, publishing, and sharing synthetic biology designs. *ACS synthetic biology* 5, 6 (2016), 487–497.
- [14] McLAUGHLIN, J., POCOCK, M., MISIRLI, G., MADSEN, J., AND WIPAT, A. Visbol: Web-based tools for synthetic biology design visualization. *ACS Synthetic Biology* 5, 8 (2016), 874–876.
- [15] MISIRLI, G., HALLINAN, J., AND WIPAT, A. Composable modular models for synthetic biology. *ACM J. Emerg. Technol. Comput. Syst.* (2014).
- [16] MISIRLI, G., HALLINAN, J., YU, T., LAWSON, J., WIMALARATNE, S., COOLING, M., AND WIPAT, A. Model annotation for synthetic biology: automating model to nucleotide sequence conversion. *Bioinformatics* 27, 7 (2011), 973–979.
- [17] MYERS, C. J., BARKER, N., JONES, K., KUWAHARA, H., MADSEN, C., AND NGUYEN, N.-P. D. ibiosim: a tool for the analysis and design of genetic circuits. *Bioinformatics* 25, 21 (2009), 2848.
- [18] NIELSEN, A., DER, B., SHIN, J., VAIDYANATHAN, P., PARALANOV, V., STRYCHALSKI, E., ROSS, D., DENSMORE, D., AND VOIGT, C. Genetic circuit design automation. *Science* 352, 6281 (2016), aac7341.
- [19] OBERORTNER, E., CHENG, J.-F., HILLSON, N. J., AND DEUTSCH, S. Streamlining the design-to-build transition with build-optimization software tools. *ACS Synthetic Biology* 0, 0 (0), null. PMID: 28004921.
- [20] POCOCK, M., TAYLOR, C., McLAUGHLIN, J., MISIRLI, G., AND WIPAT, A. An environment for augmented biodesign using integrated data resources. In *8th International Workshop on Bio-Design Automation* (2016).
- [21] QUINN, J., COX, R., ADLER, A., BEAL, J., BHATIA, S., CAI, Y., CHEN, J., CLANCY, K., GALDZICKI, M., HILLSON, N., LE NOVÈRE, N., MAHESHWARI, A., McLAUGHLIN, J., MYERS, C., P, U., POCOCK, M., RODRIGUEZ, C., SOLDATOVA, L., STAN, G., SWAINSTON, N., WIPAT, A., AND SAURO, H. Sbol visual: A graphical language for genetic designs. *PLoS Biol* 13, 12 (12 2015), e1002310.
- [22] SAURO, H. M., CHOI, K., MEDLEY, J. K., CANNISTRA, C., KONIG, M., SMITH, L., AND STOCKING, K. Tellurium: A python based modeling and reproducibility platform for systems biology. *bioRxiv* (2016), 054601.
- [23] WOODRUFF, L. B., GOROCHOWSKI, T. E., ROEHNER, N., MIKKELSEN, T. S., DENSMORE, D., GORDON, D. B., NICOL, R., AND VOIGT, C. A. Registry in a tube: multiplexed pools of retrievable parts for genetic design space exploration. *Nucleic Acids Research* (2016), gkw1226.

- [24] ZHANG, M., MCLAUGHLIN, J., WIPAT, A., AND MYERS, C. Sboldesigner 2: An intuitive tool for structural genetic design. (*forthcoming*) (2017).
- [25] ZHANG, Z., NGUYEN, T., ROEHNER, N., MISIRLI, G., POCOCK, M., OBERORTNER, E., SAMINENI, M., ZUNDEL, Z., BEAL, J., CLANCY, K., WIPAT, A., AND MYERS, C. libsbolj 2.0: a java library to support sbol 2.0. *IEEE Life Sciences Letters* 1, 4 (2016), 34–37.
- [26] ZUNDEL, Z., SAMINENI, M., ZHANG, Z., AND MYERS, C. A validator and converter for the synthetic biology open language. *ACS Synthetic Biology* (2016).