SOFTWARE COMPLIANCE TESTING FOR WORKFLOWS USING THE SYNTHETIC BIOLOGY OPEN LANGUAGE

by

Meher Samineni

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

Master of Science

in

Computing

School of Computing

The University of Utah

May 2019

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Meher Samineni**

has been approved by the following supervisory committee members:

Chris Myers ,	Chair(s)	**17 Aug 2018** Date Approved
Jake Beal ,	Member	**17 Aug 2018** Date Approved
Matthew Flatt ,	Member	**17 Aug 2018** Date Approved
Mary Hall ,	Member	**17 Aug 2018** Date Approved

by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Data standards are integral for interoperability between software applications, since they provide guidelines for how data can be meaningfully exchanged and in a uniform manner. While standards provide a bridge for applications to share and translate data, they do not guarantee that applications are compatible to perform a data exchange or that any translated data is legal and valid. As such, data passed from pairing applications must be validated to ensure that the data was not transformed or lost in the process of exchanging information. Ideally we would want an exchange between tools that is automatically successful; however, 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. The proposed research is to develop a compliance methodology that tests compliance of applications against the Synthetic Biology Open Language (SBOL) standard. This research aims to provide a robust test suite, a TestRunner tool implementing the compliance strategy, and a demonstration of the created methodology.

To my sister, Sai

CONTENTS

LIST OF FIGURES

LIST OF TABLES

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 and graduate. I am greatly indebted to Chris for his 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 for all the encouragement and time he has given to me providing feedback, advice, and mentorship.

I would also like to thank my committee members, Jake Beal, Matt Flatt, and Mary Hall. I deeply appreciate the invaluable support each member has provided me, which goes beyond their guidance of this thesis.

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 have 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 the 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

Software communities are integral in various diverse industries. Since users have different needs, software communities cater to their needs by developing unique applications. While users are able to perform tasks more easily due to these developed applications, software communities face some challenges in the management of tools. These challenges include data reproducibility and ensuring data integrity, which are issues that impact the quality of communication between applications. To overcome these challenges, software communities rely on data standards to provide crucial support. By establishing guidelines for data to be meaningfully exchanged and in a uniform manner, standards are integral for interoperability between software applications. While standards enable applications to share and translate data, they do not verify correct implementation within applications. Furthermore, standards do not validate the data exchanged between communicating applications. To certify that an application is functioning correctly as specified by the underlying standard, a compliance methodology is required. Particularly, for applications supporting a data standard, a compliance methodology should ensure that no data loss occurs, data is not harmed, and that all translated data is legal and valid as specified by the standard.

In order to create and test a compliance methodology and identify patterns for non-compliance, a software community with developed applications and a well-defined standard focused in one area is necessary. Synthetic biology is one such particular field in which well-defined standards and developed applications exist. Emerging over the last quarter of a century, synthetic biology grew as a field concerned with the ability to construct biological devices by applying engineering principles [14]. During the development of this field, a synthetic biology

oriented software community created the *Synthetic Biology Open Language* (SBOL) standard that establishes guidelines to represent biological information about genetic devices. In pairing with the standard, developers within this community have created applications with diverse functionalities. Some of these functionalities include genetic design automation, genetic modeling and visualization, and databases hosting genetic information. By creating a compliance methodology to test applications, we can ensure standard compliance and data integrity will be validated by analyzing the developed synthetic biology-related applications for SBOL compliance and verifying data exchanges between applications.

1.1 Contributions of this Thesis

The field of synthetic biology contains well-defined data standards with numerous applications claiming to be standard compliant. A major challenge in the field and in particular the SBOL community is to systematically test applications for standard compliance. One of the key goals of my research is to create a methodology to systematically test and verify applications for compliance. In particular, my research will use software tools specifically supporting the SBOL standard as a case study. The main contributions of this proposed research is briefly summarized in the following paragraphs.

An analysis was performed through a developed SBOL characterization tool on an existing SBOL test suite. The purpose of this analysis is to understand if there is a set of diverse examples representing the entire SBOL data model. A set of metrics are created in order to identify gaps within the SBOL test suite. A more enriched SBOL test suite was created in order to target these gaps and create a more complete test suite.

In addition to a more robust SBOL test suite, a testing methodology including simple and complex round-trip tests is implemented. The purpose of this methodology is to determine compliance of an application to the SBOL standard. This created methodology is instrumented through an **SBOLTestRunner** software tool that performs both types of round-trip testing.

In order to test the created compliance testing methodology, several libraries

and a critical software application were tested. These case studies show proof of concept of the methodology and caught errors within each of the libraries and applications tested. The proposed research has implications within both the SBOL community and within the broader context for software communities maintaining applications encoding a standard. Developers within the SBOL community now have a process to determine if their application is compliant with the SBOL standard. Furthermore, the case studies tested identified critical bugs within each of the applications tested.

While the created methodology utilized the SBOL standard and its supporting applications as case studies, the principles used within this methodology can be applied to various other communities. For example, the concept of a round-trip can be used to test other applications encoding standards other than SBOL. Moreover, this research corroborates that standards should have a notion of compliance testing in order to test applications that support it.

1.2 Thesis Overview

This thesis is organized within five chapters. Background information on the SBOL standard and data model is provided in Chapter 2. This chapter gives a 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 also briefly summarized in this chapter. Chapter 3 describes the algorithm that we developed to analyze the SBOL test suite. It also describes how this algorithm is applied to discover gaps in the test suite. The final result is an enriched test suite that provides better coverage. Chapter 4 describes the round-trip testing methodology that we developed to analyze compliance of applications against the SBOL standard. The chapter also includes the case studies for the application of the SBOLTestRunner tool against various SBOL software tools. 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 that 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 data standard.

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 multilevel 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 components within the sequence, but the complete sequence is required. SBOL provides a format describing both structural and functional information of a genetic design. The 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 [6]. 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 data standard

into applications such as the SBOL java library, *libSBOLj* [12], the SBOL python library, *pySBOL*, the SBOL c++ library, *libSBOL*, and lastly the javascript library, *sboljs*. 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 [11]. Additionally, SBOL Visual allows different regions of DNA components to be depicted using these "glyphs." For example, Figure 2.1 visually represents the promoter, ribosome binding, and terminator regions using SBOL Visual 1, as well as the proteins, small molecules, and their interactions in SBOL Visual 2.

To give an overview of the SBOL data model, all classes stem from the abstract **TopLevel** class. As the **TopLevel** class is an abstract class, it is not directly referenced, but rather indirectly implemented through twelve key classes. Those classes that inherit from the **TopLevel** class directly are considered as parent classes that are never nested under any other object. The **TopLevel** classes include the following derived classes: **Sequence**, **ComponentDefinition**, **Model**, **ModuleDefinition**, **Collection**, **GenericTopLevel**, **CombinationalDerivation**, **Implementation**, **Attachment**, **Activity**, **Agent**, and **Plan**.

The **CombinatorialDerivation**, **ComponentDefinition**, **Sequence**, **Collection**, and **Implementation** classes and their supporting classes represent the structural entities within the SBOL standard. **ModuleDefinition** and **Model TopLevel** classes along with their supporting classes represent functional entities. These classes are explained within the next few sections. The associations between classes are indicated using arrows. Solid arrows indicate ownership of the class towards which the arrow is pointing [2].

The rest of this section describes these classes in a bit more detail. 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 represent auxiliary information that is neither structural nor functional.

2.1 Structural Data Classes

Figure 2.2 depicts the structural classes within the SBOL data model. The **ComponentDefinition** class represents the structural entities within a biological design [2]. The components that represent DNA, RNA, and protein, i.e. the structural entities are described using **ComponentDefinition** objects. While this is the main purpose of this data class, **ComponentDefinition** objects are also used to represent other types of structural entities that exist within a biological design such as small molecules and complexes. A **Sequence** object 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 structure of the entity being represented. 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 denote specific positions of the sequence. This function is achieved through a **SequenceAnnotation** object. To specifically denote 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 denoted 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 denotes the position at a 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 denoting specific positions of a sequence, a **SequenceConstraint** object allows for rules to be specified regarding the relative 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.

In order to represent a physical instance of a synthetic biological construct, the **Implementation** class is used. This class provides the ability to describe the product that was built within a laboratory sample.

The purpose of the **CombinatorialDerivation** class is to describe combinatorial genetic designs without having to specify every possible design variant individually. Its child class is a **VariableComponent** that provides the ability to specify **ComponentDefinition** objects denoting any new **Component** objects.

2.2 Functional Data Classes

Figure 2.3 represents the functional classes and their relationships. 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. There are many entities that connect and interact within a design to produce some function. For this purpose, **Interactions** are utilized in order to provide the information on 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 relationships between the abstract entity and the specific component. It can also be used to indicate when two entities represent the same object.

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

Figure 2.4 represents the additional classes within SBOL. The **Collection** class allows for **TopLevel** objects with a common feature to be grouped together. For example, a set of **ComponentDefinition** objects representing different types of promoters could be placed within a promoter **Collection**.

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 **Attachment** class allows for data files to be associated and to link metadata relating to a SBOL design. For example, experimental data files resulting from a procedure can be represented using an **Attachment**. The **Activity** class is used to track experimental meta-information regarding a genetic design. Submodules of this class include **Association** and **Usage**, which further clarify the roles of entities within an **Activity**. The **Plan** class specifies the steps within a process. For example, this entity could refer to the lab protocols used in an experiment. This class pairs with the **Agent** class. The **Agent** class refers to the entity performing a design process. The **Agent** class could refer to a person, organization, or software tool.

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 that 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.

2.4 SBOL Supporting Applications

We created and dispersed a survey to application developers within the SBOL community with the goal of compiling a list of the current software applications supporting SBOL. The application information gained from the survey is utilized in evaluating the created testing methodology. Twenty-nine responses have been collected as of July, 2018 from the survey and the compiled results are shown in Table 2.1. 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.

2.5 Types of Software

Figure 2.5 and Figure 2.6 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 2.7 shows the breakdown of the applications supporting the SBOL data model. Fourty-one point four percent of applications support SBOL structurally only while thirteen point eight percent of applications specifically are able to support SBOL functional classes. fourty-four point eight percent of developers claim that their application supports both. The testing methodology must take into account that applications that state they can only support one level can only be tested with SBOL data examples supporting that level 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 2.8 references the functionality that various applications provide. An ap-

plication 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 that 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 that state they support visualization of created designs.

2.6 SBOL Support Within Applications

Figure 2.9 represents a summary of results about SBOL applications supporting SBOL Visual. SBOL Visual defines a graphical notation to define genetic components and designs [11]. Of the applications queried, 58.6 percent report they do support SBOL Visual.

Figure 2.10 and Figure 2.11 summarized the results from questions regarding SBOL support integrated within the application. Figure 2.10 shows the results for whether applications are able to read and understand the contents of data within a SBOL example file. Figure 2.11 shows 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 that support SBOL 2.0 which includes multilevel support as well as support within Genbank and FASTA formats.

Given 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.

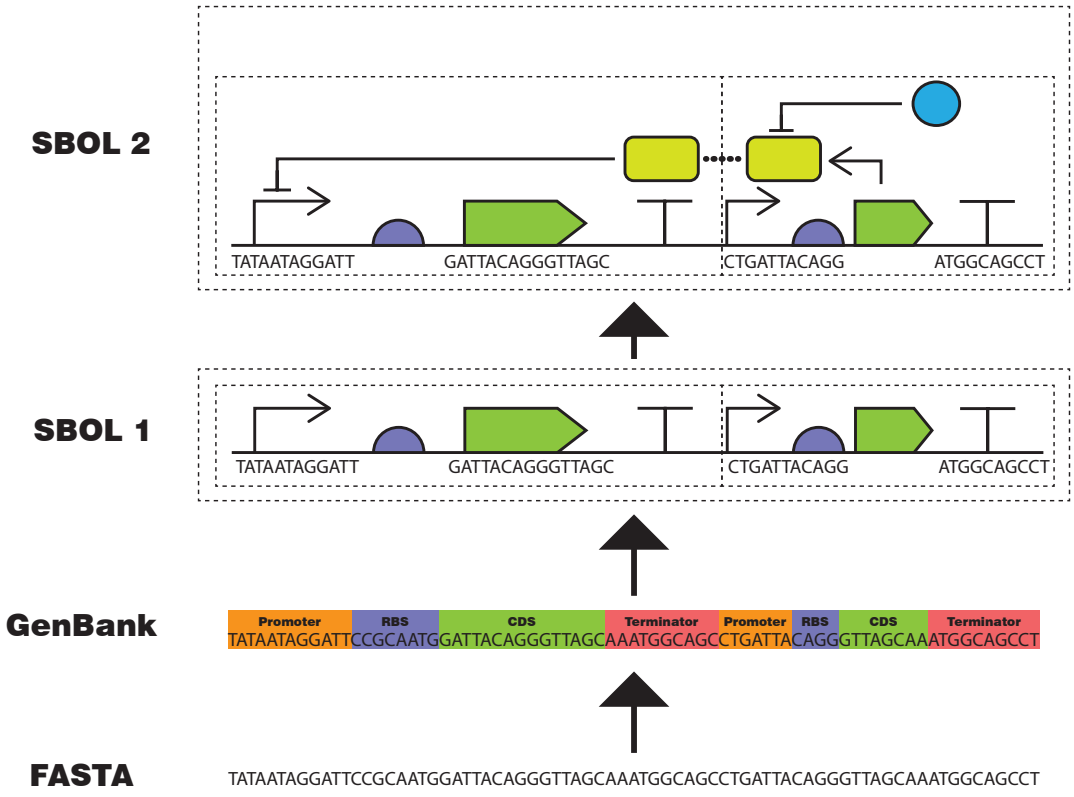

Figure 2.1. Biological design standards format evolution. SBOL expands beyond previous formats that only allow expression of sequences to include hierarchical representations of the structure and functional information of a genetic design. SBOL 1 allows for DNA components to be described without requiring sequences to be assigned to each component. SBOL 2 further extends this format by enabling more types of components and their interactions to be described (figure courtesy of Zundel et al [13]).

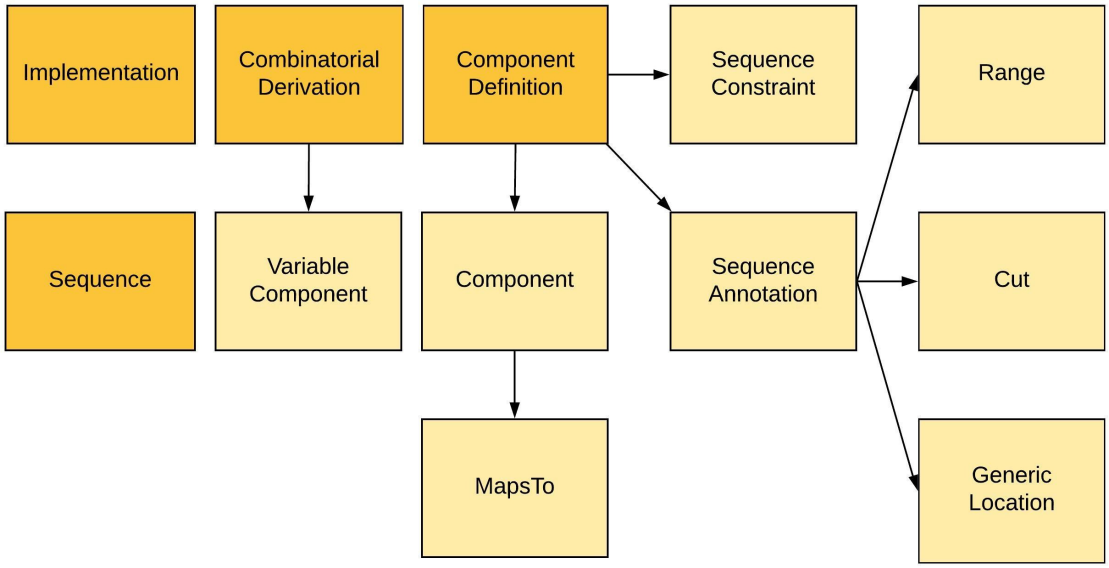

Figure 2.2. Structural classes of the SBOL data model. The dark yellow classes represent the **TopLevel** structural data classes within the SBOL data model with the lighter yellow representing the supporting structural data classes.

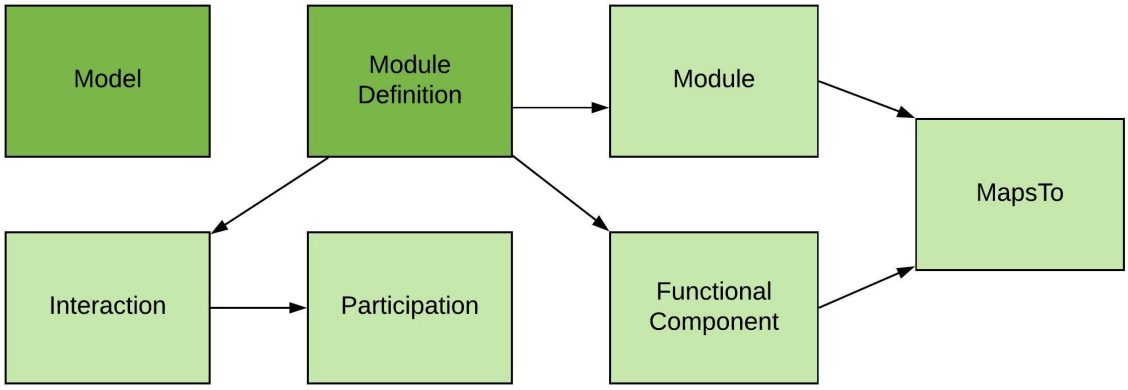

Figure 2.3. Functional classes of the SBOL data model. Dark green classes represent the top level functional data classes with the supporting functional data classes marked in light green.

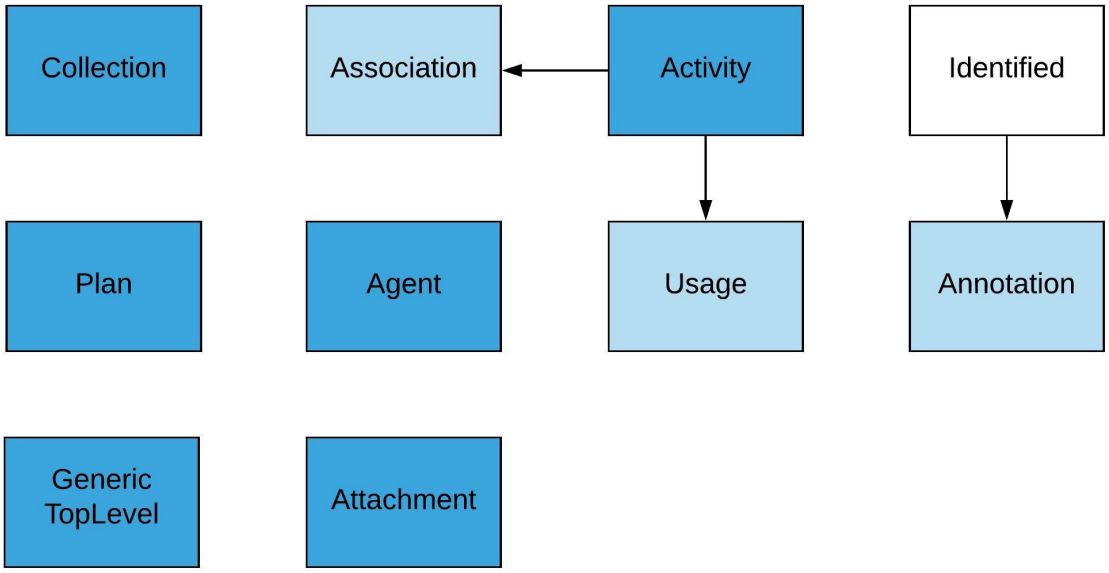

Figure 2.4. Additional classes of the SBOL data model are marked in dark blue. These classes represent top level additional classes that do not strictly represent structural or functional information. The lighter shade of blue represents the supporting classes. All of the classes within the SBOL data model inherit from the abstract **Identified** class; therefore, any class can have a child annotation.

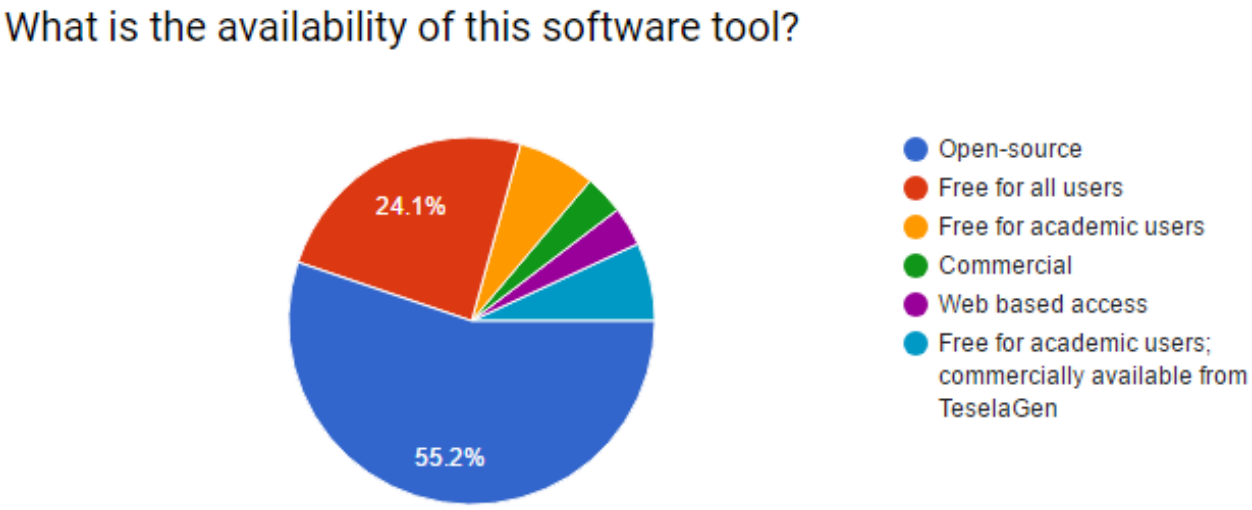

Figure 2.5. Results of the availability of SBOL applications and licensing and user requirements to acquire the applications.

Table 2.1. 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 (K)nowledge Management, (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 (table courtesy of Myers et al. [10]).

	Function					SBOL				
							Import		Export	
Name	K	S	V	G	M	V	1	2	1	2
BOOST		•					•	•	•	•
Cello				•	•	•				•
DeviceEditor		•	•	•	•	•		•		
DNAPlotLib			•			•	•	•	•	•
D-VASim					•					
Eugene		•		•		•	•		•	
Finch	•	•	•	•		•				•
GeneGenie		•							•	
GenoCAD	•	•		•		•		•		•
Graphviz			•			•				
iBioSim			•	•	•	•	•	•	•	•
ICE	•	•	•			•		•		•
j5		•			•	•	•		•	
MoSeC		•		•					•	
Parts&Pools				•	•					
Pigeon			•			•				
Pinecone	•	•				•		•		•
Pool Designer		•					•	•		
Proto BioCompiler			•	•	•	•			•	
SBOL-GB Converter							•		•	
SBOL Validator	•						•	•	•	•
SBOLDesigner	•	•	•			•	•	•	•	•
SBOLme				•						•
ShortBol		•		•						•
SynBioHub	•		•			•	•	•		•
Tellurium		•		•	•			•	•	
TinkerCell			•	•	•	•	•		•	
VisBOL			•			•		•		
VirtualParts	•			•	•	•	•		•	•

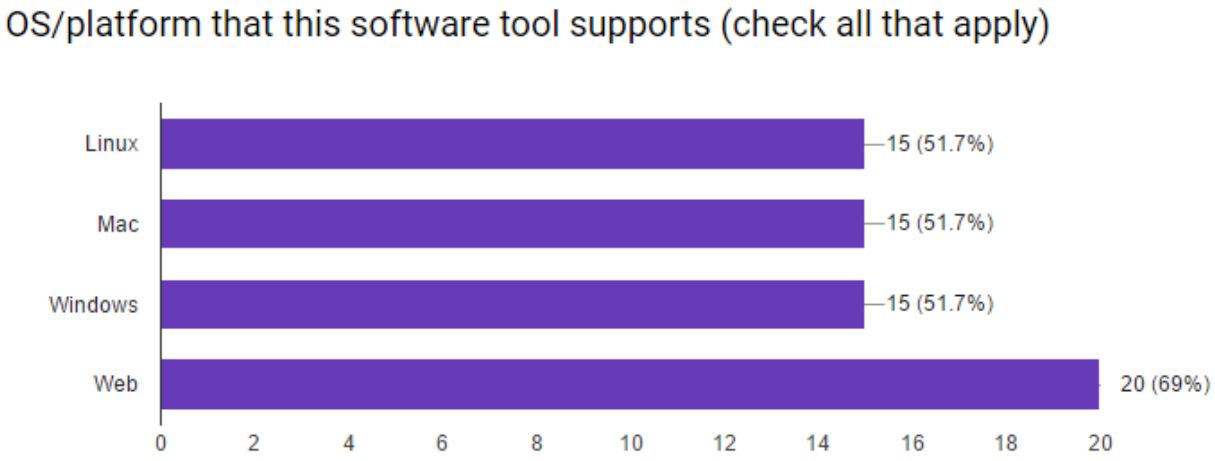

Figure 2.6. The results of the survey question regarding the OS/Platform requirements for SBOL applications. The results of the breakdown of the platform usage are shown.

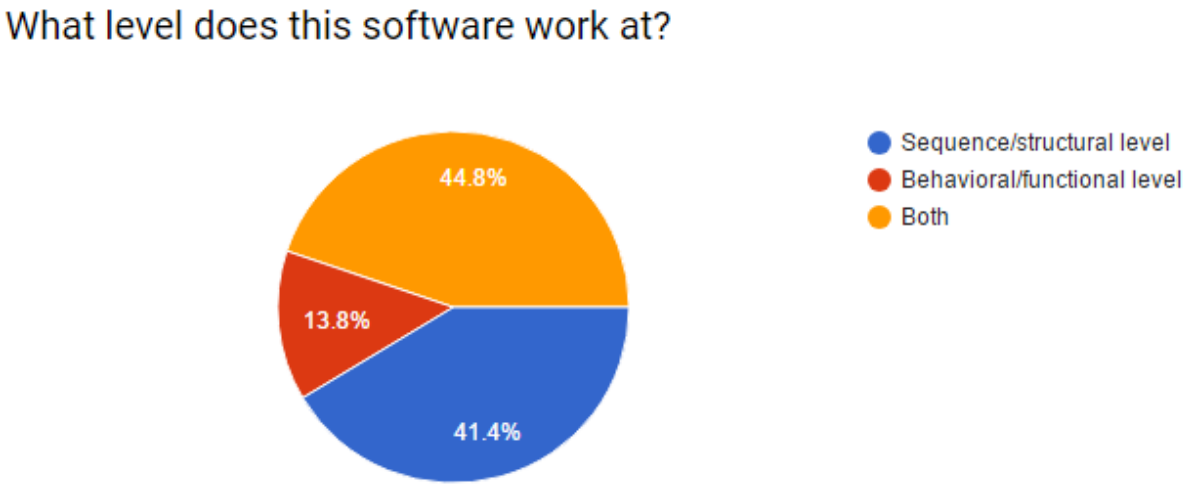

Figure 2.7. Results of SBOL applications supporting structural or functional information.

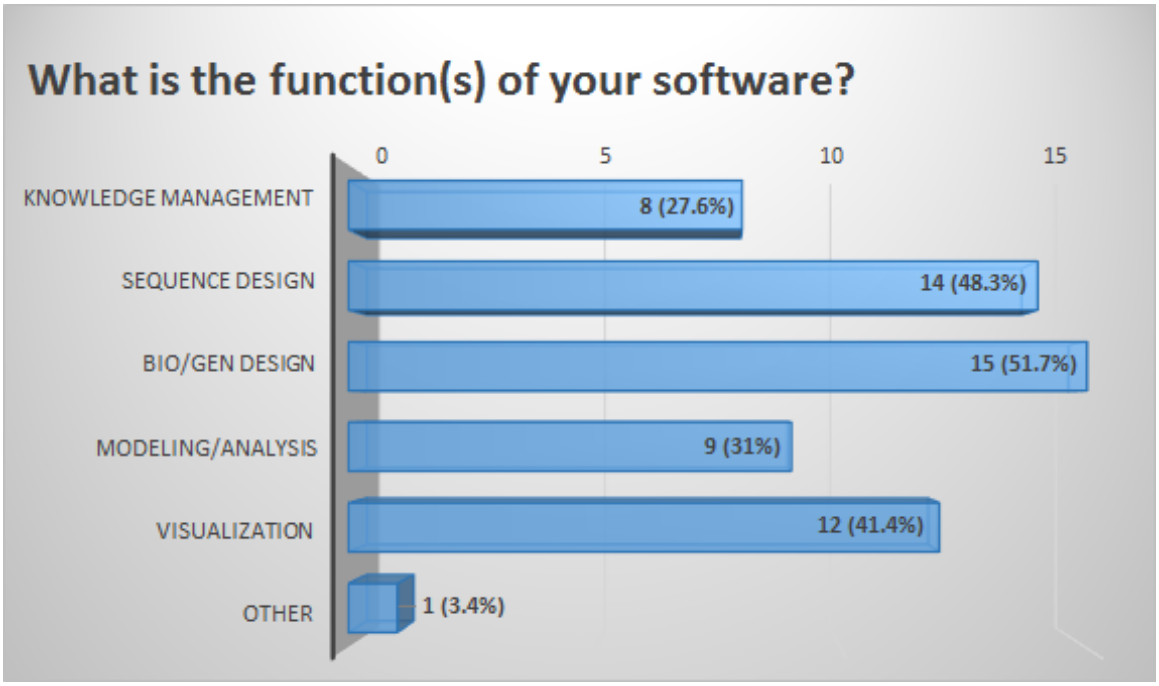

Figure 2.8. The breakdown of the various capabilities that SBOL applications have.

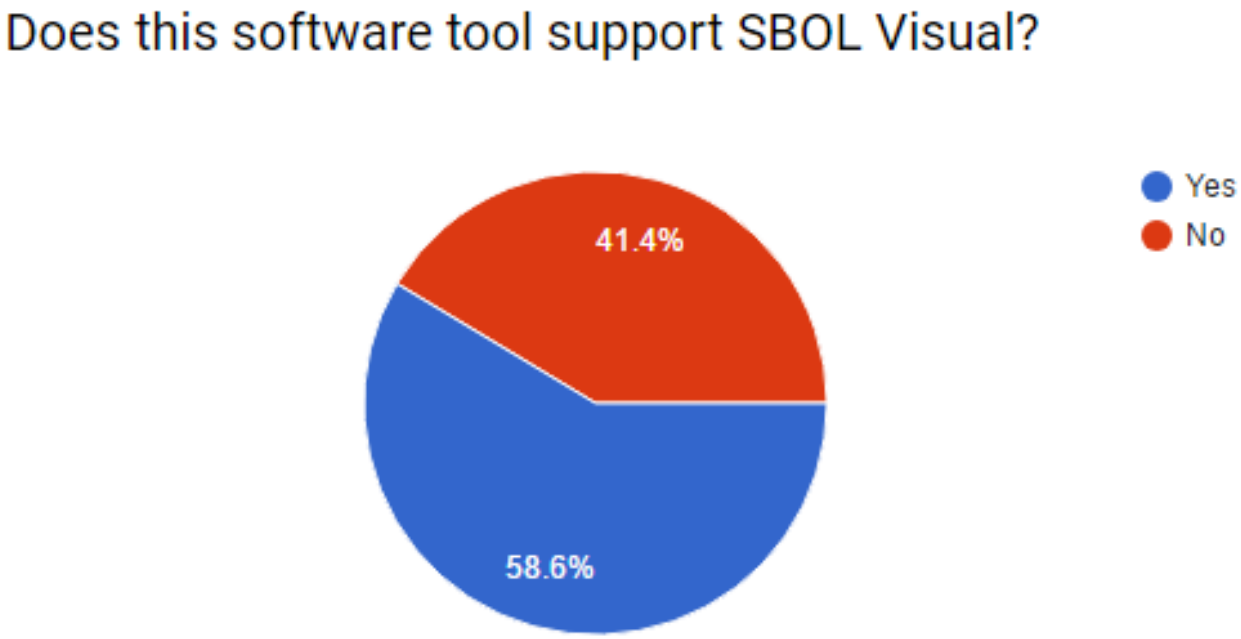

Figure 2.9. Survey results determining the level of SBOL Visual in SBOL applications.

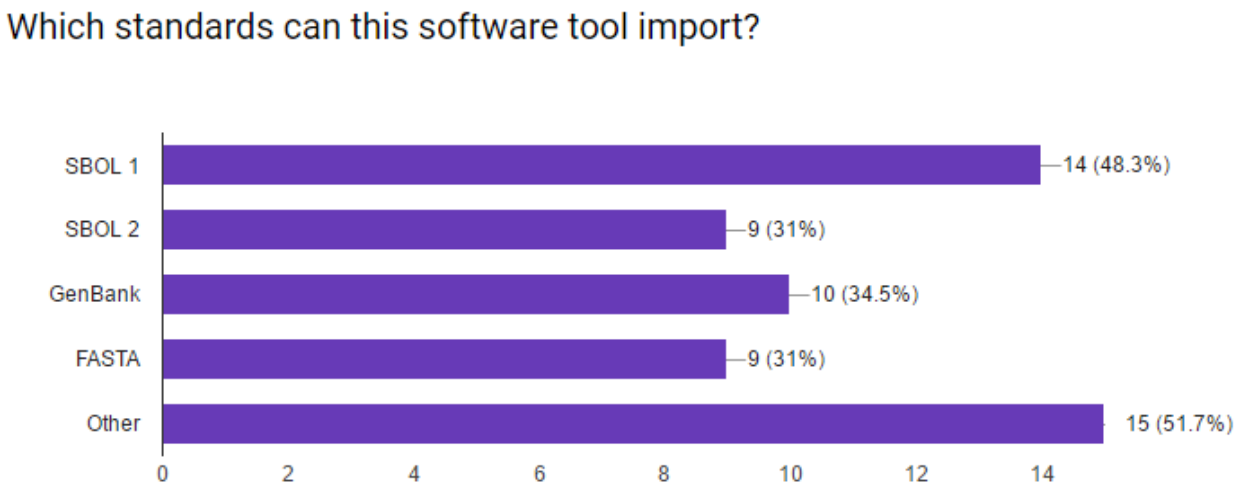

Figure 2.10. Survey results determining the number of applications able to import SBOL.

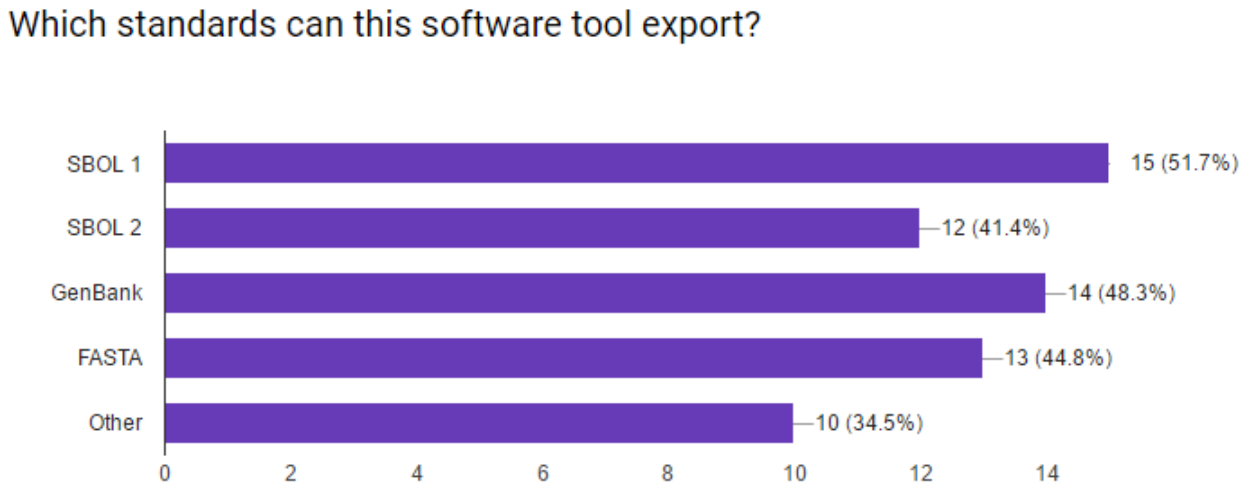

Figure 2.11. Survey results determining the number of applications able to export SBOL.

CHAPTER 3

ENRICHMENT 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. In order to verify the self-reported data, a robust test suite with examples representing the SBOL standard is required. During the beginning work of this thesis, the initial SBOL test suite contained different sets of tests created for various purposes. This test suite contained a series of existing SBOL files used for conversion testing of SBOL 1 to SBOL 2 [1, 6]. Examples also representing the SBOL 2 specification were created [2]. Additionally, some examples belonging to the test suite were utilized to test the SBOL Java library [12]. Alternatively, the initial test contained a series of invalid files that were used to test the SBOL validation rules, and applications supporting SBOL should throw errors for these files. The focus of this thesis is to analyze the initial example set to understand its coverage of the SBOL data model and to augment the test suite with new examples for a full coverage of the data model. 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 3.1 of this chapter details the algorithm created to analyze the SBOL Suite of Examples. The results of the analysis are discussed within Section 3.2.

3.1 Algorithm for the Analysis of the SBOL Test Suite

In order to test applications supporting the SBOL standard, the SBOL 2 examples within the test suite are analyzed to create 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* and have been reorganized to determine SBOL compliance, example completeness, and adherence to best practices. 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 based on the data types represented within each biological design.

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 1, the types introduced within the SBOL data model are given as a 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.

Algorithm 1: Pseudocode to determine what SBOL data types exist within an SBOL example.

```
Input: SBOLDocument doc, Set files, Set types
Output: Map ⟨ f, Map ⟨ t, c ⟩⟩ m
foreach f ∈ files do
    doc = read(f)
    foreach t ∈ types do
        c = doc.count(t)
        if m[f][t] ≠ 0 then
            m[f][t] := c
        end
    end
end
```

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 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 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 Algorithm 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, a set of clusters with files that contain common data types is created.

Algorithm 2: Pseudocode to create clusters with a set of SBOL data types common to a set of SBOL examples.

```
Input: Map ⟨ f, Map ⟨ t, c ⟩⟩ m, Set files
Output: Set clusters
foreach f ∈ files do
    cTypes = ∅
    Map ⟨ t, c ⟩ givenTypes = m[f]
    foreach t ∈ givenTypes do
        if m[f][t] ≠ 0 then
            cTypes = cTypes ∪ t
        end
    end
    clusters[cTypes] = clusters[cTypes] ∪ f
end
```

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

Determining relations between the created clusters provides an organized un-

Algorithm 3: Pseudocode to create parent-child relationships between clusters representing common SBOL data types across SBOL examples.

```
Input: Set clusters
Output: Graph g
for p ∈ clusters do
    for c ∈ clusters do
        if p == c then
        end
        continue
        if c ⊊ p then
            continue
        end
        flag = true;
        for otherCluster ∈ clusters do
            if otherCluster == parent or otherCluster == child then
                continue
            end
            if otherCluster ⊆ parent and child ⊆ otherCluster then
                flag = false
            end
        end
        if flag then
            Edge e from parent to child
        end
    end
end
```

Algorithm 4: This algorithm determines if any data types found within a parent cluster are not found within at least one of the child clusters.

```
Input: Set clusters
Output: String state
for p ∈ clusters do
    childDataTypes = ∅
    if size(p) ≠ 0 then
        if isTopLevel(p) then
            state = "leaf node : complete set"
        end
        else
            for c ∈ p do
                childDataTypes = childDataTypes ∪ c
            end
            if size(childDataTypes) == 0 then
                state = "children have no data types: complete set"
            end
            else if getDataTypes(p) == childDataTypes then
                state = "children exist in parent: complete set"
            end
            else
                state = "children do not exist in parent : incomplete Set"
            end
        end
    end
end
```

derstanding of how the SBOL standard is represented through the series of SBOL examples. The created graph of clusters determines what tests do exist, not necessarily what tests do not exist in the current test suite. Further analysis determined the gaps in the test suite. In particular, the ”completeness” of the cluster graph was determined. Each cluster was checked against its children to verify whether they form complete tree subset relations. As shown in Algorithm 4, this method iterates through the set of clusters and for each cluster retrieves data types its child clusters. If the union of all the data types of the children equals the data types within the parent cluster, then a perfect subset relation exists. For each cluster, if the union of all the child types do not equal the parent's types, then the graph is incomplete. In this case, a new test case can be added for the missing data types to create a perfect subset. In certain cases, the union of the child types cannot equal the parent because within the SBOL data model, only the top level types are allowed to exist alone. All other data types must exist along with their top level parent type. For example, a parent cluster with the types **ComponentDefinition** and **Component** that has one child cluster containing a **ComponentDefinition** is not a perfect subset. To complete this subset, there would need to be a cluster containing just the **Component** data type. However, this case is not possible because a **Component** cannot exist without a **ComponentDefinition**. Within the SBOL data model, all child data types must exist with their top level classes. Therefore, when checking for the ”completeness” of the graph, these parent-child data type relationships are taken into account before creating any new test cases. Additionally, there are also sibling relationships that exist. In extending the previous example, a cluster that contains the types: **ComponentDefinition**, **Component**, and **SequenceConstraint** cannot have a child cluster with **ComponentDefinition** and **SequenceConstraint**. A **SequenceConstraint** references a **Component**, therefore a **Component** is required to exist. In adhering to top level parent-child and sibling relationships, the graph structure will formulate such that the leaf nodes will be single top level classes as shown in Algorithm 5. Then the parents of those leaf nodes will be a combination of the top level classes with their child classes and in some cases, their required sibling classes. This structure can follow to create larger and more

complex cases with various classes.

Algorithm 5: Algorithm to determine if a cluster is a leaf node. The cluster must contain one data type that must be a **TopLevel** type.

```
Input: cluster c
Output: boolean b
b = false
if size(dataTypes(c)) == 1 then
    for dataType in c do
        if dataType == Collection then
            b = true
        end
        else if dataType == ModuleDefinition then
            b = true
        end
        else if dataType == ComponentDefinition then
            b = true
        end
        else if dataType == Sequence then
            b = true
        end
        else if dataType == Model then
            b = true
        end
        else if dataType == GenericTopLevel then
            b = true
        end
        else if dataType == Attachment then
            b = true
        end
        else if dataType == Implementation then
            b = true
        end
        else if dataType == CombinatorialDerivation then
            b = true
        end
        else if dataType == Activity then
            b = true
        end
    end
end
```

The second main extended analysis performed determined the total number

of valid combinations of data types. A complete test suite would ideally have at least one example for each valid combination. Utilizing the *combos* function within the **itertools** Python library, each valid combination of data types was identified. Each top level and its children paired with their siblings were recorded in a list. If the paired top level for the children did not exist within the combination, then it is not considered valid and was not recorded. For example, for the top level, **Sequence**, it has a child class **Annotation**. The outputted combinations include **(Sequence)**, **(Sequence, Annotation)**, and **(Annotation)**. All of the combinations are valid except for the last because in **Annotation**, the data type is not a top level and its paired top level, i.e the **Sequence** class, does not exist. Therefore, it is excluded. Table 3.1 provides the number of unique valid combinations for each top level paired with their children and required siblings. If these combinations were once again paired across with each other, then the total number of combinations is 105,840. It is not practical to create tests for all of these cases, but it is reasonable to create examples for each of the individual top level combinations. The number of combinations for each top level combination is shown in Table 3.1.

The total number of valid combinations identified points to the minimum number of valid examples that should exist within a test suite. The data types within each cluster provide one possible combination. To understand how many combinations had a paired example within the test suite, all of the data type sets from each cluster were concatenated into a set. This set was matched against the total number of possible combinations to determine the coverage amount and also which combinations did not have a paired example.

3.2 Results of the Analysis of the SBOL Test Suite

There is now a methodology to which the SBOL test suite can be systematically tested to determine the robustness of the examples. The purpose of analyzing the examples and creating a graphical representation of the test suite is to provide some insight into how to logically test applications that support SBOL. The created graph provides 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 of 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's 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, the examples within the SBOL test suite acts as the input for the round-trip tests, which are used to determine if an application is able to exchange data accurately. Additionally, the process determines whether the created graph and the examples create a robust test suite or if there are gaps that are to be filled. Lastly, the methodology measures the test suite against the total valid number of test cases that can exist.

The SBOL test suite was enriched through utilizing the methodology and identifying areas of areas of improvement. To show the evolution of the SBOL test suite, various statistics and the created graph will be discussed from analyzing the test suite at three different time points. The first time point is previous to the addition of classes to the SBOL standard. The second time point is immediately after examples were added to the test suite representing the new SBOL classes. The last is the current test suite with examples added as part of the work of this thesis. While the created graphs at each time point can be discussed, only the graph for the first time point is discussed in detail due to space consideration. There are a variety of statistics tabulated from analyzing the SBOL test suite at each of the three time points. The information provided is organized in three different sections. The first is overall data statistics of the test suite including the number of examples and clusters that make up the test set. The second section breaks down the data type diversity of the SBOL examples. This includes the number of structural, functional, and auxiliary examples that exist as well as their overlaps. The last section provides statistics for cluster coverage of the data types.

Original SBOL Test Suite

A graphical representation of the resulting POSET for the example test suite is shown in Figure 3.1. The nodes with no incoming arrows are the source nodes within the graph. Source nodes represent a superset of the data types common to a group of examples. Each source node denotes a set of examples with a unique superset of the data types. In other words, no other node exists that includes the same data types and more. The remaining portion of the graph consists of paths made up of nodes that follow parent-child relationships. Pathways exist as a child node that stems from one or more parent nodes and each child node consists of examples that contain a subset of the SBOL data types contained by its immediate parent. These pathways are significant because they provide a test strategy to narrow down which data types are being correctly supported. There are three numbers within each cluster. The top number is an arbitrary number used to reference the cluster node. The second number represents the number of common data types found within a group of examples. The last number within the parenthesis denotes the number of examples. Red colored nodes represent clusters that include structural, functional, and auxiliary data types. The yellow colored nodes represent clusters that include structural and possibly auxiliary data types (**Sequences**, **ComponentDefinitions**, etc.) while the green colored nodes represent clusters that are functional and possibly auxiliary data types (**Models**, **ModuleDefinitions**, etc.). Blue colored nodes represent only auxiliary data types (**Collection**, **GenericTopLevel**, etc). Lastly, white colored nodes represent nodes with data types. For example, source Node *11* contains one example with eighteen data types represented in the example and there exists no example representing these data types and more. Node *22* stems from *10* with three examples representing eleven types and are also a subset of the types found in the previous node. Node *17* contains one example with five data types that are also within node *22* and so on. The pathway is followed depth-wise with each child level providing a more constrained subset of the data types existing within the parent node. Table 3.2 provides the details for each node including the number of example files that belong to that node as well as the type of examples found within the node.

Furthermore, Table 3.3 provides the exact data types found within each node.

In the SBOL data model, there are twenty-seven classes excluding abstract classes. The SBOL data model has expanded to include new classes. The SBOL data model that existed during the first time point had eighteen non-abstract data classes. Given this, the maximum number of data types that exists in a set of examples is eighteen types as represented by Node *10*. Furthermore, there does not exist any example with every data type represented. While there does not exist an all-inclusive example, every single data type is at least represented once within an example. One last key insight is the imbalance in the types of data existing within the examples. Thirty-four percent of the examples represent only structural data classes, while only five percent of the examples include functional data type.

The inspection of the example test suite whose examples are organized in partial order set relationships provides a way to test SBOL applications. However, the example test suite is incomplete. A full summary of the analysis statistics is provided in Table 3.4. There are ninety-five examples and twenty-seven clusters. There are nine data types that are not currently being represented in any one example. The test suite contains an imbalance in the structural and functional examples. Thirty-four percent of the test suite contains examples representing structural data types that are representative of SBOL 1 examples. In contrast, the test suite only contains five percent of examples representing functional data types. The example test suite needs to expand to include more SBOL 2 examples that include functional data types. This imbalance across structural, functional, and auxiliary types shows in cluster formation.

In analyzing the current suite of examples, we observed that different combinations of classes, especially key classes such as **ComponentDefinition** and **ModuleDefinition**, do exist. However, many combinations of data classes are still missing. Furthermore, some classes are being tested very thoroughly whereas other classes only are tested by a few examples. The number of examples is not evenly distributed across types.

3.2.1 SBOL Test Suite with Provenance Examples for SBOL 2.2

Along with the expansion of the SBOL data model, there were new tests contributed by various developers within the SBOL community. These new tests were representative of the new classes. Once again, performing the test suite analysis, the statistics regarding the expanded test suite were gathered. There were 114 tests total that did increase the test count as well as increased the number of clusters to thirty-six. However, the same issues that were seen with the original test suite remained prevalent. The same imbalance between structural versus functional tests did not improve. Twenty-nine percent of total examples represented structural data only. In comparison, about one-eighth the amount of structural tests were functional. Additionally, there is an improvement in the number of clusters representing various combinations. While all combinations cannot be covered in a realistic manner, at least more of the first two levels of the graph that are the top level leaf nodes and the top levels paired with their children should exist. These statistics are shown in Table 3.5.

3.2.2 Current State of the SBOL Test Suite

From identifying the problems that existed in the previous states of the test suite, a main goal of this thesis was to enrich the *SBOLTestSuite*. Once again, performing the test suite analysis, the statistics regarding the expanded test suite were gathered and are shown in Table 3.6. There are now 200 tests total and 113 clusters total. Every single data type is represented in at least one example. New tests were created with the goal of creating a balanced test suite representative of every category of data type. In the previous test suites, there were far more structural examples than functional. Therefore, more examples now exist with only functional data types. The ratio of structural to functional examples is half rather than an eighth as before. There is also an increase of examples including auxiliary data types. This is particularly important since SBOL 2.2 expansion included more types that were neither structural or functional. With the added clusters, there is an improvement in the combinations covered. There are still many uncovered combinations, but this is a difference from the number of clusters that

represented valid combinations in the original test suite. Increasing clusters covers more of the first two levels of the graph that have examples representing top level and their direct children combinations.

3.3 Discussion

In creating a methodology for analyzing SBOL software applications and their support of the SBOL standard, an algorithm is created to analyze the SBOL test suite. The inspection of the examples results in a graph that can be used to test SBOL applications and verify the self-reported data from the survey. A goal of this thesis is to enrich the SBOL test suite and provide a process to prove its robustness. The test suite was expanded to fill in gaps in the diversity of SBOL data types represented. In extension, the developed *SBOLTestRunner* software tool will utilize the enriched test suite to perform compliance testing of applications supporting SBOL. The SBOL test suite is available at `https://github.com/SynBioDex/SBOLTestSuite`. The algorithm for analyzing the test suite is located at `https://github.com/mehersam/SBOLTestCharacterization`.

Table 3.1. This table represents the valid combinations for each top level paired with its children and siblings.

Valid Combinations	
ModuleDefinition	32
ComponentDefinition	80
Sequence	2
Model	2
Collection	2
GenericTopLevel	2
CombinatorialDerivation	4
Implementation	2
Attachment	2
Activity	8
Plan	2
Agent	2
Total Combos	105,840

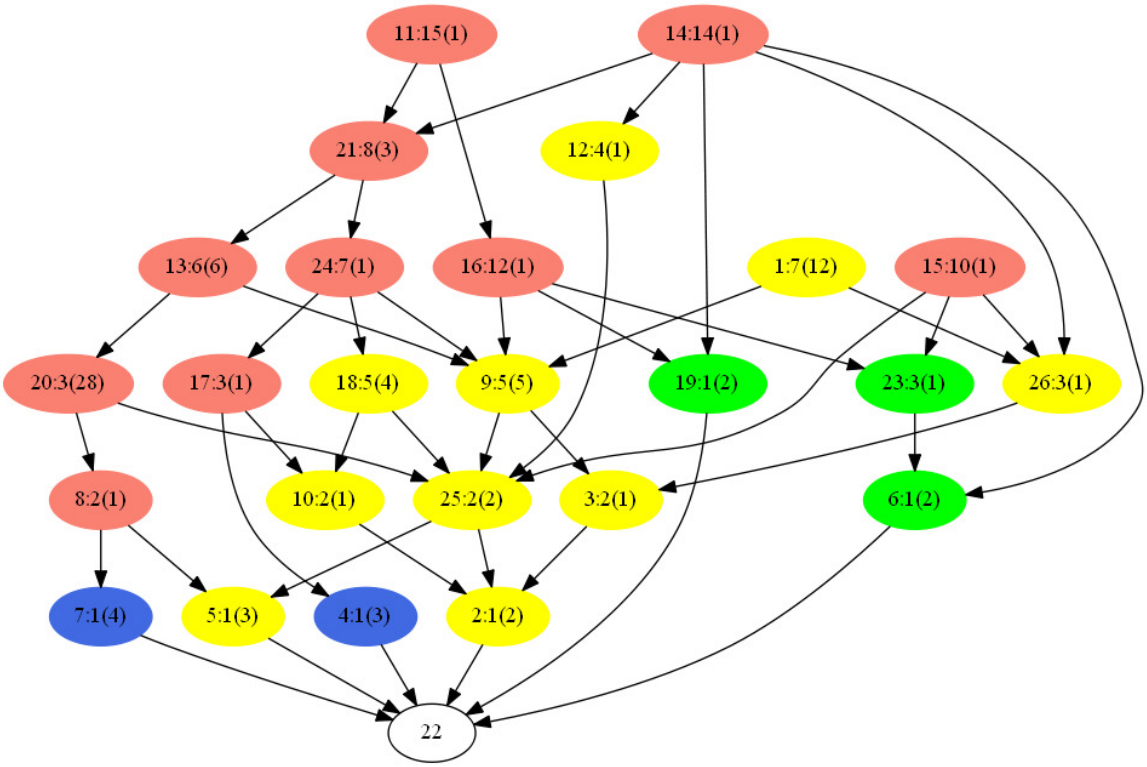

Figure 3.1. This is a graphical representation of the original SBOL test suite and their relations based on the data types supported. The nodes are clusters of examples with the same SBOL data types that are numbered arbitrarily for easy referencing. The second number references the number of data types common to a group of examples. The numbers within the parentheses are the number of examples in the cluster. Diamond nodes are source nodes with unique supersets of data types. The yellow colored nodes represent clusters that include structural and possibly auxiliary data types, while the green colored nodes represent clusters that are functional, and possibly auxiliary data types. The red colored nodes represent clusters that include structural, functional, and possibly auxiliary data types. Blue colored nodes represent only auxiliary data types. All of the examples analyzed are available at `https://github.com/SynBioDex/SBOLTestSuite/tree/master/valid/SBOL2.0`

Table 3.2. 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	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 3.3. Each entry represents the data types found within a set of examples within that node cluster. **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*

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 3.4. The results observed from analyzing the original SBOL Biological Design Examples.

Graph Statistics	
SBOL Examples	95
SBOL Clusters	27
Covered Data Classes	18
Missing Data Classes	9
Example Statistics	
Structural Examples	32 (34%)
Functional Examples	5 (5%)
Auxiliary Examples	7 (8%)
S & F Examples	2 (2%)
S & A Examples	40 (42%)
F & A Examples	0
All Types Examples	2 (2%)
No Types Examples	7 (7%)
Cluster Statistics	
Structural Clusters	11
Functional Clusters	3
Auxiliary Clusters	2
S & F Clusters	2
S & A Clusters	6
F & A Clusters	0
All Types Clusters	10
No Types Clusters	1

Table 3.5. The results observed from analyzing SBOL Biological Design Examples with the addition of SBOL 2.2 examples.

Graph Statistics	
SBOL Examples	114
SBOL Clusters	36
Covered Data Classes	27
Missing Data Classes	0
Example Statistics	
Structural Examples	33 (29%)
Functional Examples	5 (4.4%)
Auxiliary Examples	8 (7%)
S & F Examples	2 (2.8%)
S & A Examples	54 (47%)
F & A Examples	0
All Types Examples	5 (4.3%)
No Types Examples	7 (6.1%)
Cluster Statistics	
Structural Clusters	12
Functional Clusters	3
Auxiliary Clusters	3
S & F Clusters	2
S & A Clusters	11
F & A Clusters	0
All Types Clusters	4
No Types Clusters	1

Table 3.6. The results observed from analyzing SBOL Biological Design Examples with the addition of tests added from the work of this thesis.

Graph Statistics	
SBOL Examples	200
SBOL Clusters	113
Covered Data Classes	27
Missing Data Classes	0
Example Statistics	
Structural Examples	62 (31%)
Functional Examples	31 (15%)
Auxiliary Examples	29 (14.5%)
S & F Examples	3 (1.5%)
S & A Examples	59 (30%)
F & A Examples	0
All Types Examples	9 (4.5%)
No Types Examples	7 (3.5%)
Cluster Statistics	
Structural Clusters	36
Functional Clusters	27
Auxiliary Clusters	25
S & F Clusters	3
S & A Clusters	13
F & A Clusters	0
All Types Clusters	8
No Types Clusters	1

CHAPTER 4

COMPLIANCE TESTING FOR THE SBOL DATA STANDARD

Standard compliance provides many benefits to applications supporting a standard. These benefits as previously mentioned include data reproducibility, interoperability across platforms, and data exchanges between applications. While frameworks for compliance testing are not common or largely in use, some do exist in order to verify that an application is able to perform as specified by a standard [4]. This section explores some of these approaches.

Compliance testing generally consists of checking a system's behavior in relation to defined specifications [5]. A compliance testing approach exists to test software tools supporting a standard for modeling software systems [4]. The approach is to analyze the outputs produced and maintained by a software application supporting the standard. The components within a test case consists of a software model and an expected result that follows the specifications of the standard. Each software tool then imports the input model, verifies the model, and produces its own result. The produced result is then compared against the expected result [3]. This approach was experimented with a case study for testing software tools supporting the UML standard. A test case consists of a UML model and the expected test result, which verifies whether UML well-formedness rules are violated. The software tool then imports the UML model provided in the test case and outputs a verification of the correctness of the model. This verification result is compared with the expected result on a pass/fail scale. In building a test suite of these types of test cases, there are two different sets. One set of test cases contain valid models that should be accepted by a software tool. The other set includes invalid models. A tool is fully compliant if it accepts all the valid models

and rejects all the invalid models [3].

Another compliance testing approach to test applications is to model an underlying specification as a finite state machine (FSM). This type of verification is utilized in applications supporting the Trusted Computing standard guidelines. *Trusted Computing* (TC) is a technology used to ensure computer security and involves establishing a trustworthy environment. *Trusted Computing Group* (TCG) defined specifications for TC and a standard platform called *Trusted Platform Module* (TPM). In pairing with these resources, researchers established a compliance validation approach to test TC applications [5]. To expand on TPM, the module represents a microprocessor chip attached to a motherboard. From a programmatic view, the chip has three parts: the functional units that provide the functions for cryptographic operations, and the nonvolatile and volatile memory that act as storage for keys. TPM is used by executing commands in an order-dependent manner that produce codes as the output. The proposed testing approach has two parts; the first is the commands reliability tests and the second is the functional execution tests [5]. The first type of tests consist of pairing commands with their different possible values. These tests are utilized in the second type of tests, which consists of categorizing TPM commands with a function and building a finite state machine [5]. When testing a TC application, the application should verify its state against the built state machine if a test is run with a series of commands that represent different functions.

Different compliance testing schemes in both hardware and software have been discussed. Within the synthetic biology software community, there has not been a formalized compliance testing approach to test software compliance with the SBOL data standard. However, a conformance testing system exists for applications supporting the *Systems Biology Markup Language* (SBML) standard. The SBML standard provides a format for modeling and describing biological processes. In particular, SBML allows description of biochemical networks, cell signaling, and metabolism models within a biological system [7]. The SBML test suite is a conformance testing system in which SBML supporting applications are able to verify the extent and correct usage of SBML support [8]. The test suite consists of three differ-

ent test collections that includes semantic tests, stochastic tests, and syntactic tests. Each different test in the collections contains a test biological model, the expected SBML output file, a visualization of the expected simulation, and the statistical results in the form of a *csv* file. Applications supporting the SBML standard can represent the given model and compare it to the expected output given. Semantic tests represent deterministic simulation behavior of various biological models. Stochastic tests cases represent the expected result for the stochastic simulation behavior of a model. Lastly, the syntactic test suite represents whether SBML data was parsed correctly within the supporting application.

4.1 Compliance Testing Approach for the SBOL Data Standard

The analysis of the examples and the graph structure allows us to test the applications compiled from the results of the SBOL survey for standard compliance and verify self-reported data. In particular, the graph allows us to understand what parts of the SBOL standard an application does in fact support. The process of checking includes selecting one test case from each source node and using it to test the application. If an example fails, then one test case would be drawn from all child clusters. This process would repeat until the sink node is reached or no further failures are discovered. By analyzing the point at which examples succeed, we can accurately determine the data classes that the software supports.

While the graph provides a way to test compliance of the SBOL data model, the methodology needs to be extended to evaluate data exchanges across applications. There are three different elements for testing. The first is to test applications' support of SBOL Visual, an application's ability to import SBOL data, and an applications' ability to export SBOL data. SBOL Visual must be manually inspected to ensure the correct usage of symbols. Additionally, to verify an application can export SBOL data, the *SBOL Validator* tool can check the validity of the SBOL files produced [13]. Verifying that SBOL data is correctly imported requires a round-trip test. A round-trip test as shown in Figure 4.1 consists of importing SBOL data into an application and then exporting the imported data. A comparison is performed of the imported and exported data to ensure that no semantically

important data has been transformed or lost. If the comparison produces no semantic differences, then the application can correctly import SBOL data. While this test strategy could be excruciatingly tedious if the comparison is performed manually, the ideal scenario is to instrument the software through an interface that enables the tool to import and export SBOL data programmatically, then perform the comparison. This simple round-trip test works perfectly well if an application does not internally modify the data in any way. However, applications can modify data either by removing, adding, or transforming data internally. In some cases, these data modifications are not always harming the data, but for these cases, the simple round-trip does not actually work. Therefore, we have expanded our methodology to include a slightly more complex test as shown in Figure 4.2. In the instance that an application modifies the data internally, an emulator is built which follows the same operations that the application performs on the data and outputs its own SBOL file. The SBOL file that is output from the application is compared against the emulator's output file to check that no significant differences exist. The result of this type of compliance test is that the emulator effectively characterizes how the application modifies the data, so these modifications can be analyzed to determine if they are significant.

The goal of creating and testing a compliance methodology is to analyze a piece of software to programmatically determine what ways an application is modifying the data through the use of an emulator. The methodology should identify an algorithm and a set of rules on how to abstractly test compliance of applications supporting standards in general as well as identify patterns that cause non-compliance. An **SBOLTestRunner** tool is the means of testing each SBOL application for compliance using the methodology. Once the created methodology is published to the community, developers can utilize the **SBOLTestRunner** to ensure compliance for an application. In order to use the **SBOLTestRunner**, developers must provide an emulator function that can be written using any SBOL library. This function is executed in parallel with the application with the same input. The results that are output from the emulator and the application are checked for semantic differences. The **SBOLTestRunner** has three main features. The software

has the ability to filter examples that are used to test the application. For example, functional examples are not used for an application which does not support functional information and states that it cannot support functional information. Furthermore, the **SBOLTestRunner** has the ability to configure what factors are considered important during comparisons such as identifying fields that can be ignored when performing a comparison. As mentioned before, there is a way to accept an emulator provided by the developer as input. Each of the applications tested requires some form of an interface either through an API or a command line program in order to programmatically interact with the application to import and export SBOL data.

Additionally, the methodology ideally is used to validate workflows of SBOL applications to show how data is preserved. The methodology is applied to each of the applications within a workflow to demonstrate that the methodology can be utilized to show how the data is changed as it passes through each application. This demonstration also tests each application to determine the extent of SBOL support based on which examples from the developed test suite each application supports. The output of a successful evaluation of an application is determining the SBOL support, determining what modifications happen in passing data to the application, characterizing the modifications to build an emulator, and answering key questions. The key questions that ideally can be answered include how an application handles receiving data representing the areas of SBOL it does not support and is data lost in the process of importing and exporting. Successfully analyzing the applications in the workflow demonstrates the usage of the methodology to test for software compliance against an underlying standard.

4.2 SBOLTestRunner

The **SBOLTestRunner** tool allows for testing SBOL applications for compliance. Utilizing the testing methodology and the complete SBOL test suite, the tool tests applications' compliance of the SBOL standard.

As specified in the methodology, there are two different types of round-trip tests— a simple and complex test. Within both testing processes, an SBOL data file

is fed to the application and the data that is output is compared to the input data to ensure that no changes have been made. This idea is followed exactly within a simple round-trip, the application is fed an SBOL data file and data is simply output and absolutely no internal data changes are made in processing the file. However, within a complex round-trip test, the application does make changes to the data file passed, and an emulator is necessary when testing the application. The emulator specifies the steps of what data changes and modifications are made to the data when given to the application. The **SBOLTestRunner** software tool follows the methodology and two types of applications are tested. The tool is used to test applications that perform data changes on an SBOL data file and require an emulator as well as applications that simply pass the data file from input to output. There are two main inputs required by the software tool regardless of the type of application. The first input is the application program command and the second input is location path for recording retrieved files. The third input is determined by the type of application tested. For applications that require an emulator, users can use "-e" as a command line argument followed by the command to run the emulator program. Once the arguments are provided, the tool performs round-trip testing by running the application as a system *exec* command and utilizing the input of the arguments provided by the user. The tool then passes SBOL data files as input to the application, and outputs the retrieved and emulated files and result of comparison. If the round-trip produces no differences, then the application successfully processed the data file.

Within the process of testing an application, multiple failures can occur. The **SBOLTestRunner** attempts to carefully document any failures that occur and are provided to the user. Moreover, the **SBOLTestRunner** tries to identify the point at which the application failure occurred since failures can occur outside executing the round-trip. The failure classifications include exceptions that occurred when running the application and emulator program command, test file validation errors, round-trip execution errors, and round-trip comparison errors. By clarifying the point of failure, the *SBOLTestRunner* provides if the application is non-compliant with the SBOL standard by failing the actual round-trip test, or rather

the application tested is failing for some other reason either by the commands provided or maybe the SBOL data file passed as input is bad.

Some applications only support a certain subset of data specified within the SBOL standard. If the application provides information on exactly which data types they support, then these applications should be tested with the appropriate SBOL data files. The **SBOLTestRunner** classifies data test files into different categories. Users can specify to provide *SBOL2*, *SBOL1*, *Genbank*, or *FASTA* data files only. Additionally, files can be classified as *structural*, *functional*, *auxiliary*, or a mixture between those categories.

4.3 Case Studies

Using the **SBOLTestRunner** software tool that implements the created methodology, various SBOL applications have been tested. An emulator was created for each application that made data modifications or additions when processing data. The emulators recorded each step of how the data was internally changed. Additionally, the software tool determined if the application successfully performed a round-trip test. If not, errors were identified and documented. A brief summary of the process and errors identified for applications performing under a round-trip test is provided.

4.3.1 SBOL Library Applications

This section summarizes the results of round-trip testing the individual libraries that code the SBOL data standard. The libraries do not require emulators as they should not change data processed in anyway. Therefore, a simple round-trip is used to test each of the libraries. Each data file passed is simply read using the library and written to an output file. The two data files are compared for any data modifications.

In using a simple round-trip test to determine if the **libSBOLj** application modified any data, no errors were found. There are 200 SBOL 2 tests that include both structural, functional, auxiliary, and a combination of all types of tests. **libSBOLj** passed the round-trip test for each of these data files without reporting any data

modifications or validation errors. The input data file compared exactly to the output data file. Furthermore, this also proves that **libSBOLj** does support all data types represented within the SBOL data model.

In testing the **pySBOL** application, both round-trip and validation errors were reported. Thirty-two data files failed to compare successfully in the round-trip testing process. From comparing the input file and the file output from the application, multiple data objects differed. These specific files were then hand-compared individually to determine the specific point of difference in the round-trip test. These error results are tabulated in Table 4.1. There were also seventeen data files that were throwing validation errors and one input file that could not be read and a *SBOLReader* occurred. A total of fifty files are failing in some manner.

The **sboljs** application is the SBOL Javascript library. Similar to the SBOL Python and Java libraries, the **SBOLTestRunner** was utilized to determine data modification and validation errors. The following error results found are tabulated in Table 4.2. There are nine files that are failing round-trip testing and no validation errors were reported. From comparing the specific file output by the application and the corresponding input files, three bugs were identified.

4.3.2 SynBioHub Application

SynBioHub [9] is a software application that is able to store and publish synthetic biology designs using a Web interface. Additionally, **SynBioHub** functionality is available to use through the SBOL java library, **libSBOLj**. An emulator was created that records which steps are taken in processing the data. Table 4.3 shows these steps. Table 4.4 briefly summarizes the errors found through round-trip testing of the **SynBioHub** application. In addition to compliance testing of the application, timing statistics are provided of the round-trip testing process. Statistics were gathered for the overall emulation and retrieval of SBOL data files from SynBioHub. Additionally, the time taken to perform each individual emulation step is almost insignificant. Changing the URI prefix is the emulation step with the longest average time of 1.25 seconds. The examples sizes used to the test the application ranged from 1 kilobyte to 3.5 megabytes. On average, the example size

is 106 kilobytes. Therefore, most of the files test were not very large. A summary of the timing information for smaller sized files is provided in Table 4.5. The timing information for larger sized SBOL data files is provided in Table 4.6.

4.4 Discussion

A methodology is created to test the compliance of SBOL software applications and their support of the SBOL standard. The **SBOLTestRunner** software tool automates round-trip testing of applications to test for compliance of the SBOL standard. This software is available `https://github.com/mehersam/SBOLTestRunner`. Additionally, the emulators for the applications with which the **SBOLTestRunner** was utilized to test are available at `https://github.com/mehersam/SBOLEmulators`.

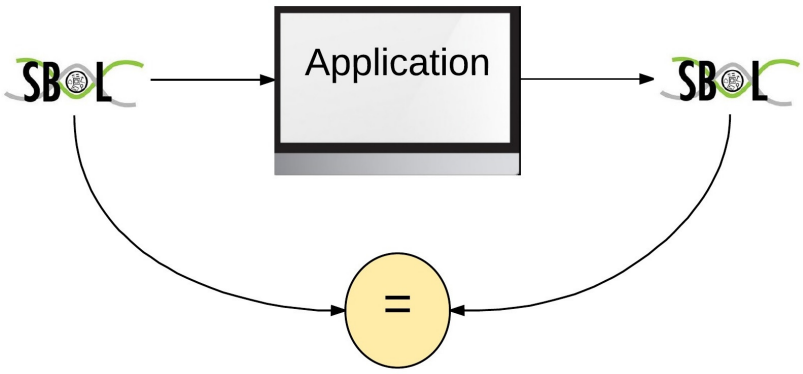

Figure 4.1. Simple round-trip test used to verify compliance of a SBOL application by ensuring that an SBOL file imported by an application and the corresponding exported SBOL file on output contains no semantically different data.

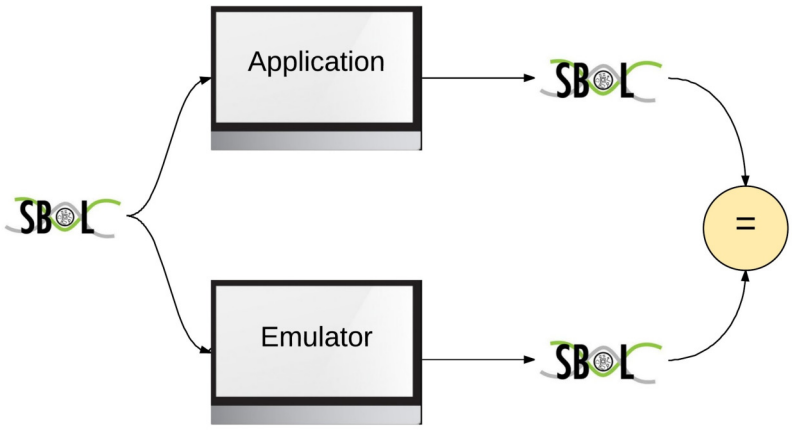

Figure 4.2. A slightly more complex round-trip test used to verify compliance of a SBOL application that modifies imported data internally on export. A SBOL data file is imported into the application and exported. In parallel, an emulator mimics the same operations to the same imported SBOL data file as the tested application and exports its own version of a SBOL data file. The different SBOL data files are then compared to verify no significant differences and ensure that no data is lost or harmed. The emulator effectively characterizes how the application modifies the data.

Table 4.1. The errors identified in pySBOL through round-trip testing using the **SBOLTestRunner**.

Bug Number	Description	Resolved
pySBOL#77	**Activity** *displayId* field is modified	Open
pySBOL#84	**Activity** *hasPlan* field is dropped	Open
pySBOL#78	**ComponentDefinition Nested Annotations** are dropped	Open
pySBOL#79	**Attachment** *source* field URI is changed to the full URI	Open
pySBOL#83	**Model** *source* field URI is changed to the full URI	Open
pySBOL#80	**Attachment Annotations** are dropped i.e time created	Open
pySBOL#81	Version tag was inserted into the **VariableComponent** object	Open
pySBOL#82	**GenericTopLevel Annotations** were dropped	Open
pySBOL#82	Version tag was inserted into the **GenericTopLevel** object	Open

Table 4.2. The errors identified in sboljs through round-trip testing using the **SBOLTestRunner**.

Bug Number	Description	Resolved
sboljs#39	**GenericTopLevel** object is dropped	Open
sboljs#40	**ComponentDefinition** and **Sequence** objects are dropped	Open
sboljs#41	**Attachment** object URI is changed to ”undefined”	Open

Table 4.3. The data modifications the **SynBioHub** emulator makes in processing SBOL data.

Emulator Changes	
	Add **Collection** to **SBOLDocument**.
	Create **Annotation** for *creator* field of created **Collection**
	Include **TopLevel** objects as members of the created **Collection**
	Create **Annotations** for*ownedBy*, *topLevel*, and *type* fields for each **TopLevel** added
Accepted Changes	
	Created *timestamp* on the created **Collection**
	Modified *source location* in **Model** classes

Table 4.4. The errors identified in **SynBioHub** through round-trip testing using the **SBOLTestRunner**.

Bug Number	**Description**	**Resolved**
libSBOLj#493, SBH#479	**GenericTopLevel** namespace is missing/not being recognized	Closed
libSBOLj#492	*createGenericTopLevel* function does not allow SBOL1 namespace	Closed
libSBOLj#491	*equals* function for **Annotations** and **GenericTopLevels** should not compare QName prefix	Closed
SBH#477	Downloading test file with a **GenericTopLevel** object causes an exception	Closed
SBH#474	**Nested Annotations** are returned as **GenericTopLevel**	Closed
SBH#473	Empty set of collection choices on remote submit creating spurious isMemberOf annotations	Closed
SBH#486	Fetching collections with Unicode characters creates invalid URIs	Closed
SBH #481	Sorting collection for labhost_All.xml causes a hang	Closed
SBH #478	Uploading files with non-compliant	Closed
SBH #476	two objects with same displayId but different URI prefixex cannot be uploaded to the same Collection	Closed

Table 4.5. A summary of various timing statistics of the round-trip testing process against **SynBioHub** application.

SynBioHub Round-Trip Testing Timing Statistics	
# of Examples	**194**
Average Size of Files (in KB)	**47.96**
Average Submission Time (in sec)	**0.73**
Average File Retrieval Time (in sec)	**1.38**
Average Emulation Time (in sec)	**0.07**
Average Removal Time (in sec)	**0.0004**
Average URI Prefix Change Time (in sec)	**0.07**
Average Add Collection Time (in sec)	**0.0004**
Average Add Annotations Time (in sec)	**0.002**
libSBOLj work performed (%)	**4.5**

Table 4.6. A summary of various timing statistics of the round-trip testing process against **SynBioHub** application using large-scale examples.

SynBioHub Round-Trip Testing Timing Statistics of Large-Scale Examples	
# of Examples	**6**
Average Size of Files (in MB)	**2**
Average Submission Time (in sec)	**54.87**
Average File Retrieval Time (in sec)	**6.7**
Average Emulation Time (in sec)	**39.49**
Average Removal Time (in sec)	**0.007**
Average URI Prefix Change Time (in sec)	**39.462**
Average Add Collection Time (in sec)	**0.0004**
Average Add Annotations Time (in sec)	**0.02**
libSBOLj work performed (%)	**58.47**

CHAPTER 5

CONCLUSIONS

This thesis presents a methodology for analyzing compliance of software applications for the SBOL standard. A list of current SBOL software applications and data regarding each application's functionality, purpose, and degree of SBOL support was compiled. Since this data is self-reported by the application's developer, each software requires verification of the claims made. In order to do this, a main goal of this thesis is to create a robust, diverse test suite that is representative of the entire SBOL data standard. The purpose of this test suite is to test each application and understand how well it supports the SBOL standard as well as if it is compliant with the standard. The work to create such a test suite began through analyzing the existing biological designs created to test *libSBOLj*, the SBOL Java library and are representative of the SBOL 2 standard. 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 separate sets based on whether or not the data types within the cluster represent structural, functional, or auxiliary data classes. The inspection of the examples results in a graph that can be used to test SBOL applications and verify the self-reported data from the survey. However, after analyzing the original examples, the results showed gaps within the SBOL test suite. These gaps included that a majority of the test suite contained mostly examples representing structural data only. Additionally, the graph created was considered "incomplete" since the data types found in the parent did not also exist in at least one child node. This is an issue as applications cannot be correctly verified if the test suite does not contain a full set of examples. Therefore, this

thesis provides a metric system to analyze the SBOL test suite that determines the breakdown of structural, functional, and auxiliary tests available. Additionally, one metric includes whether the graph output from characterizing the examples into clusters is complete and each parent-child cluster is complete. The last main metric is a set of combinations determining the set of valid combinations that are available given the SBOL data model. These combinations are the total number of minimum tests that would ideally exist in the test suite. While this is not realistic, as the test suite grows, this thesis provides a way to determine how many combinations are represented in the test suite. In creating these metrics, this thesis provides the enriched test suite that covers majors gaps such as the imbalance across data types as well as making incomplete parent-child relationships complete.

The second major goal of this test suite is to create a methodology to test the compliance of a SBOL application against the SBOL standard. The enriched test suite is utilized as the testing input for the **SBOLTestRunner** software tool to perform simple and complex round-trip tests. This compliance methodology and the paired software tool that executes the methodology provides a way to test a SBOL application. This thesis shows the case studies for which the applications were successfully tested and determined compliance failure points. This tool is available along with the enriched SBOL test suite to developers to test their own applications. Developers must provide an emulator recording any data changes made to an input example for the test runner to utilize in performing the round-trip test. Furthermore, the self-reported data regarding SBOL application compliance retrieved from the SBOL survey can now be accurately verified.

For applications encoding a standard, it is important to verify that the application is using the standard correctly. Standards need a compliance methodology in order to ensure application data reproducibility and integrity. The research of this thesis focuses on the impact of developing a process to determine compliance guidelines and testing applications against those guidelines. In creating a compliance methodology specifically for the SBOL standard and testing various key software, there were numerous bugs identified. A major impact of this thesis showed that applications that were used within the the SBOL community still did

not completely maintain data integrity. While it is unlikely to extinguish all errors within a software application, it is necessary for software communities to rely on a process to ensure whether an application is reliable and compliant. However, the created software methodology to test for compliance is not applicable to just the SBOL standard, but should be maintained for many existing standards. The impact of this thesis verifies that standards need some qualification for applications to determine if they are compliant and correctly using an underlying standard.

5.1 Future Work

While this thesis provides the basis for compliance testing of SBOL applications for the SBOL standard, this research can be furthered. Further contributions include expanding the test suite, testing more applications using the **SBOLTestRunner**, and creating an interface for the software tool created.

5.1.1 SBOL Test Suite Strategies and Expansion

This thesis provides metrics and a cluster formation algorithm to organize SBOL data types. The metrics provided include imbalances in the test suite. Future developments can be to add tests to decrease the gaps identified in the test suite. However, it is not practical to hand create all of the examples to meet every single possible valid test. It would be ideal if property-based automated testing existed to auto-generate tests based on a set of parameters given. This particularly would be ideal to generate tests for new validation rules that are added to the SBOL standard. Additionally, the test suite metrics are only based on existence of data types within an example. However, individual data types have fields that the test suite characterization does not analyze whether they exist or not.

5.1.2 Round-Trip Testing Methodology Case Study Expansion

The test suite currently provides case studies testing a few SBOL applications to show proof of concept for the created methodology. However, emulators should be created for each of the applications reported in the SBOL survey. There are many different types of applications with various functionalities such as sequence design, genetic circuit design, knowledge management, modeling, and visual-

ization tools. All of these tools support different parts of the SBOL data model including functional types. It would be ideal to test these applications using the created cluster graph. Additionally, the case studies provided utilize applications that follow simple round-trip tests, but it would be interesting to see applications that require complex round-trip testing as they process and potentially change SBOL data internally. Lastly, automating testing using the round-trip methodology focuses on testing compliance of the SBOL data model. However, possible future development could be used to determine automated compliance testing for applications supporting SBOL Visual.

5.1.3 SBOLTestRunner Software Tool Development

While the **SBOLTestRunner** software tool currently has functionality that automates round-trip testing, it does not currently have an interface. It is currently command-line based and compliance reporting is basic. It would ideal to expand it in the future to have both an interface that users can use to provide emulators and better specify what capacity their application supports SBOL. In pairing with an interface, it would be a future development to output a more visual graph of the tests the application supports and fails to support.

5.1.4 Integrating Compliance into the SBOL Standard

In order to ensure that this work does not stop at the conclusion of this thesis, it would be ideal to integrate the compliance methodology into a workflow to automatically test SBOL applications. It would be ideal if a process could be created such that application developers within the community can ”check-in” an emulator software into a central GitHub repository. Each of the applications within this created repository can fit into an automated system such that each application within the repository is tested using the **SBOLTestRunner** and the examples within the **SBOLTestSuite**. Furthermore, using continuous integration software, it would good to periodically re-test each application or if a change is detected in any application software, then the automated system re-tests that particular application to ensure that it is still in compliance with the SBOL standard.

REFERENCES

[1] Adames, N., et al. GenoLIB: a database of biological parts derived from a library. *IEEE Life Sciences Letters 1*, 4 (2016), 34–37.

[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] Bunyakiati, P., and Finkelstein, A. The compliance testing of software tools with respect to the uml standards specification - the argouml case study. *2009 ICSE Workshop on Automation of Software Test* (2009).

[4] Bunyakiati, P., Finkelstein, A., and Rosenblum, D. The certification of software tools with respect to software standards. *2007 IEEE International Conference on Information Reuse and Integration* (2007).

[5] Cui, Q., and Shi, W. An approach for compliance validation of trusted computing applications. *Workshop on Knowledge Discovery and Data Mining* (2008).

[6] Galdzicki, M., Clancy, K., Oberotner, E., M. Pocock, J. Q., Rodriguez, C., Roehner, N., Wilson, M., Adam, L., Anderson, J., Bartley, B., Beal, J., Chandran, D., Chen, J., Densmore, D., Endy, D., Grunberh, R., Hallinan, J., Hillson, N., Johnson, J., Kunchinsky, A., Lux, M., Misirli, G., Peccoud, J., and Plahar, H. The synthetic biology open language (SBOL) provides a community standard for communicating designs in synthetic biology. *Nature Biotechnology*, 32 (2014), 545–550.

[7] Hucka, M., Bergmann, F. T., Hoops, S., Keating, S. M., Sahle, S., Schaff, J. C., Smith, L. P., and Wilkinson, D. J. The systems biology markup language (sbml): Language specification for level 3 version 1 core. *Journal of Integrative Bioinformatics 12*, 2 (2015), 266.

[8] Keating, S., Evans, T., Smith, L., and et al. SBML test suite, 2016.

[9] McLaughlin, J., Myers, C., Zundel, Z., Mirsirli, G., Zhang, M., Ofiteru, I., Goni-Moreno, A., and Wipat, A. Synbiohub: A standards-enabled design repository for synthetic biology. *ACS Synthetic Biology 7*, 2 (2018), 682–688.

[10] Myers, C., Beal, J., Gorochowski, T., Kuwahara, H., Madsen, C., McLaughlin, J., Mirsirli, G., Nguyen, T., Oberortner, E., Samineni, M., Wipat, A., Zhang, M., and Zundel, Z. A standard-enabled workflow for synthetic biology. *Biochemical Society Transactions 45* (2017), 793–803.

[11] Quinn, J., Cox, R., Adler, A., Beal, J., S. Bhatia, Y. C., Chen, J., Clancy, K., Galdzicki, M., Hillson, N., Novere, N., Maheshwari, A., Mclaughlin, J., and Sauro, H. SBOL visual: A graphical language for genetic designs. *PLoS Biology 13*, 12 (12 2015), e1002310.

[12] Zhang, Z., Nguyen, T., Roehner, N., Misirli, G., Pocock, M., Oberotner, 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.

[13] Zundel, Z., Samineni, M., Zhen, Z., and Myers, C. A validator and converter for the Synthetic Biology Open Language. *ACS Synthetic Biology 6*, 7 (2016), 1161–1168.