TECHNOLOGY MAPPING OF GENETIC CIRCUIT DESIGNS

by

Nicholas Roehner

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

Doctor of Philosophy

Department of Bioengineering

The University of Utah

December 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Nicholas Roehner**

has been approved by the following supervisory committee members:

Chris J. Myers, Chair	**July 31, 2014**	Date Approved
Orly Alter, Member	**July 30, 2014**	Date Approved
Frederick R. Adler, Member	**August 4, 2014**	Date Approved
Chuck Alan Dorval, Member	**August 4, 2014**	Date Approved
Tara L. Deans, Member	**July 30, 2014**	Date Approved

and by **Patrick A. Tresco**, Chair of

the Department of **Bioengineering**

and by David Kieda, Dean of The Graduate School.

ABSTRACT

Synthetic biology is a new field in which engineers, biologists, and chemists are working together to transform genetic engineering into an advanced engineering discipline, one in which the design and construction of novel genetic circuits are made possible through the application of engineering principles. This dissertation explores two engineering strategies to address the challenges of working with genetic technology, namely the development of standards for describing genetic components and circuits at separate yet connected levels of detail and the use of Genetic Design Automation (GDA) software tools to simplify and speed up the process of optimally designing genetic circuits. Its contributions to the field of synthetic biology include (1) a proposal for the next version of the Synthetic Biology Open Language (SBOL), an existing standard for specifying and exchanging genetic designs electronically, and (2) a GDA workflow that enables users of the software tool iBioSim to create an abstract functional specification, automatically select genetic components that satisfy the specification from a design library, and compose the selected components into a standardized genetic circuit design for subsequent analysis and physical construction. Ultimately, this dissertation demonstrates how existing techniques and concepts from electrical and computer engineering can be adapted to overcome the challenges of genetic design and is an example of what is possible when working with publicly available standards for genetic design.

For Katherine.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

There are many without whom this dissertation would not be possible and to whom I am forever grateful. First and foremost, I would like to thank my advisor, Prof. Chris Myers, who gave me the opportunity to pursue my research interests in synthetic biology at a graduate level. I honestly do not think that I could have had a better advisor. I will never forget the lessons that I have learned as a researcher under Prof. Myers's guidance, nor will I fail to follow his example of mentorship should I one day serve as a professor. I feel that the latter career had been made all the more possible by Prof. Myers's encouragement to involve myself with community standards, through which I have gotten to know many outstanding individuals in my field of study.

In this regard, I would also like to thank the members of the Synthetic Biology Open Language (SBOL) Developers Group, in particular Jacob Beal, Douglas Densmore, Kevin Clancy, Michal Galdzicki, Goksel Misirli, Ernst Oberortner, Matthew Pocock, Jacqueline Quinn, Herbert Sauro, and Anil Wipat. The development of standards is a critical yet often undersupported aspect of every engineering discipline. As the SBOL standard forms no small part of this dissertation, I am greatly indebted to the research that has been volunteered by members of the SBOL Developers Group and forms a foundation for this dissertation to build upon.

Next, I would like to thank the past and present members of the Myers lab, particularly Andrew Fisher, Curtis Madsen, Leandro Watanabe, and Zhen Zhang, and my fellow bioengineering students, Christopher Conlin, Jennifer Gibson, and Nicholas Nolta. Whether discussing research, swapping stories, or playing board games, it has been my privilege to work and make merry with the finest of friends.

Most of all, I would like to thank my love and partner in life, Katherine Hashimoto, to whom this dissertation is dedicated. With her love and companionship, I have been able to overcome my greatest trials as a graduate student. I cannot imagine our lives without each other.

Last but not least, I would like to thank my parents, Richard and Kathleen Roehner, and my sister, Kelsey Roehner. Without their love and encouragement, I would not

have developed the passion for learning and drive to succeed that are necessary to earn a graduate degree. I would also like to thank my grandfather, Paul Roehner, who supported my pursuit of an undergraduate degree at the University of Washington, and my grandparents, Gary and Carol Ann Clark, who fostered my interest in higher education.

The material in this dissertation is based upon work supported by the National Science Foundation under Grant Numbers 0331270, CCF-07377655, CCF-0916042, and CCF-1218095. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

CHAPTER 1

INTRODUCTION

Synthetic biology is a new field in which engineers, biologists, and chemists are working together to transform *genetic engineering* into an advanced engineering discipline, one in which the design and construction of novel *genetic circuits* are made possible through the application of engineering principles. The Presidential Commission for the Study of Bioethical Issues recently concluded that synthetic biology has the potential to improve or revolutionize a variety of *economic sectors*, including energy production, pollution control, medicine, and chemical manufacturing [1]. So far, this promise has been realized to a limited extent in *bacteria* and *yeast* that have been genetically engineered to perform novel and potentially useful functions, including manufacturing drug precursors [2], producing biofuels [3, 4, 5], consuming toxic waste [6, 7], and even invading tumor cells [8]. The successes of these individual projects, however, have often failed in translation and application to new projects, a problem that is partly due to the challenges of working with *genetic components* in the absence of an *engineering framework* built upon *standards* and *abstraction* [9, 10, 11].

These challenges include the great diversity of *structure* and *function* of genetic components, the complex nature of their *interactions* and the *networks* that they can form, and the strong dependence of their function on *environmental* and *physical context*. Because of their context dependence, it can be difficult to reuse genetic components across different designs and recombine them into novel genetic circuits that behave as the sum of their parts. Because of their diversity and complexity, it is difficult to optimally compose genetic components as part of designs for larger genetic circuits. This dissertation explores two engineering strategies to address these challenges, including (1) the development of standards for describing genetic components and circuits at separate yet connected levels of detail and (2) the use of *Genetic Design Automation* (GDA) software tools to simplify and speed up the process of optimally designing genetic circuits.

1.1 Standards

Currently, there exist standards for describing *genetic sequences*, such as the GenBank [12] and FASTA [13] formats, and standards for describing *biochemical models*, such as the *Systems Biology Markup Language* (SBML) [14] and CellML [15], but there is no one standard that represents both the structure and function of genetic circuits in a hierarchical, modular fashion that is useful for *engineering design*. Even the most widely used public *database* for synthetic biology, the iGEM Registry of Standard Biological Parts [16], does not provide a standardized means of connecting the sequences of genetic components with *mathematical models* for their behavior and qualitative descriptions of their interactions. Without standards that combine genetic structure and function into contextualized *modules*, there can be no exchangeable basis for *design automation* in synthetic biology [17], a paradigm that has otherwise been applied to great success in *electrical and computer engineering*.

1.2 GDA Software Tools

Ideally, a synthetic biologist could design genetic circuits at a fairly high level of abstraction, focusing on a desired function rather than the exact genetic components used to implement this function. Such a synthetic biologist would use GDA software to construct an *abstract functional specification*, automatically select genetic components that satisfy the specification from a *design library* (*genetic technology mapping*), and compose the selected components into a standardized *genetic circuit design* for subsequent analysis and physical construction. By encoding knowledge in tools and standardized data, GDA can lower the barrier to entry for new designers and promote the reuse of experimentally proven genetic components. Through intelligent automation, GDA can make the design of more complex genetic circuits tractable and decrease the length of *design and test cycles*.

While a host of GDA tools exist for applications such as *biochemical modeling* and *simulation*, *sequence editing* and *optimization*, *design composition*, and more recently genetic technology mapping, not all of these tools use publicly available standards to represent data and none of them use standards to tightly couple genetic structure with function. In order for GDA tools to facilitate interdisciplinary collaboration and the exchange of genetic circuit designs, they must use standards that represent both genetic structure and function, even if individual tools only focus on one aspect of designing genetic circuits. Furthermore, because it is difficult for any one standard to directly represent all aspects of design in synthetic biology, there must be GDA tools that enable users

to compose genetic circuit designs from data that belong to complementary standards, ideally through mechanisms that are represented within the standards themselves.

1.2.1 Sequence Editing Tools

Sequence editing tools typically enable users to construct, modify, or annotate the sequences of genetic components. Sequence editing tools listed in Table 1.1 include GeneDesign [18], GeneDesigner [19], Kera [20], Synthetic Gene Designer [21], and VectorEditor [22]. Unpublished sequence editing tools not listed in this table include the commercial tools Benchling, Genome Compiler, and Vector NTI. Nearly all of the above tools make use of the GenBank [12] and FASTA formats [13] to store genetic sequences and their annotated *features*, though some of the web applications, such as GeneDesign and Synthetic Gene Designer, read and write plain text sequences that conform to the *International Union of Pure and Applied Chemistry* (IUPAC) codes for *nucleotides* [23] and *amino acids*. While simpler to use in some respects, these web applications do not preserve or update any features that may have been annotated in the source files for their input sequences. Lossless transmission of data is necessary for different GDA tools to effectively operate on the same design.

It is a testament to the maturity of *bioinformatics* that well-developed standards exist for encoding genetic sequences and that the vast majority of sequence editing tools use at least one of these standards. For the purpose of synthetic biology, however, these standards lack the means to fully represent two key concepts of engineering design: *hierarchy* and *modularity*. Modularity allows engineers to group elements of a design into reusable structural or functional units, while hierarchy allows them to build larger designs out of subdesigns. Of the sequence editing tools listed above, only Benchling, GeneDesigner, VectorEditor, and Vector NTI support the *Synthetic Biology Open Language* (SBOL) [44], a new standard that has been developed to address the shortcomings of the GenBank and FASTA formats for engineering design. In particular, SBOL currently enables the specification of modular *Deoxyribonucleic Acid* (DNA) *components* and the hierarchical annotation of their sequences with other DNA components. These tools' implementations of SBOL, however, are still fairly limited in that they do not take full advantage of modularity and hierarchy when creating new genetic components. Rather, they convert genetic components represented in their own nonhierarchical, internal *data model* to SBOL. New GDA tools are needed to create truly hierarchical, modular descriptions of genetic structure that are better suited to the exchange and reuse of designs.

Table 1.1: GDA Software Tools

Tool	**Description**	**Citation**
Asmparts	Composition of genetic circuit models from SBML models of genetic components	[24]
BioJADE	Composition, mapping, and TABASCO [25] simulation of genetic circuit designs	[26]
CellDesigner	Composition and COPASI [27] simulation of SBML models	[28]
DeviceEditor	Composition of genetic constructs and their verification with Eugene [29] design rules	[30]
GEC	Mapping of genetic programs to genetic circuit designs plus export of SBML	[31]
GeneDesign	Sequence editing and optimization of genetic constructs	[18]
GeneDesigner	Sequence editing, optimization, and curation of genetic constructs	[19]
GenoCAD	Composition, syntactic verification, and COPASI simulation of genetic circuit designs	[32]
iBioSim	Composition, simulation, and model checking of SBML models, esp. for genetic circuits	[33]
Kera	Composition, rule-based verification, and simulation of genetic circuit designs	[20]
MatchMaker	Mapping of abstract genetic circuit designs from Proto Compiler to genetic constructs	[34]
MoSeC	Generation of DNA sequences from SVP [35] models annotated with genetic components	[36]
Parts&Pools	Mapping of Karnaugh maps to abstract genetic circuit designs	[37]
ProMoT	Composition of biochemical models plus import/export of SBML	[38]
Proto Compiler	Mapping of genetic programs to abstract genetic circuit designs and MATLAB [39]	[40]
SBROME	Mapping of abstract genetic circuit designs to concrete genetic circuit designs	[41]
SynBioSS	Composition and simulation of biochemical models plus import/export of SBML	[42]
Synthetic Gene Designer	Sequence editing with emphasis on codon optimization for synthetic genetic codes	[21]
TinkerCell	Composition and COPASI simulation of genetic circuit designs with plug-in support	[43]
VectorEditor	Sequence editing of genetic constructs (JBEI tool packaged with DeviceEditor)	[22]

1.2.2 Biochemical Modeling Tools

Biochemical modeling tools allow users to construct mathematical models and often also provide users with the ability to simulate and analyze these models. Biochemical modeling tools listed in Table 1.1 include Asmparts [24], BioJADE [26], CellDesigner [28], GEC [31], GenoCAD [32], iBioSim [33], Kera [20], ProMoT [38], Proto Compiler [40], SynBioSS [42], and TinkerCell [43]. Of these tools, all but BioJADE and the Proto Compiler support SBML, a well-developed standard for modeling *biological systems* that is supported by more than 250 software tools across multiple fields of study. While the Proto Compiler is capable of exporting models written in MATLAB [39], one disadvantage of this representation is that it only captures the mathematical aspects of a biochemical model and none of its biological meaning. More *domain-specific languages* are required to enable tools to compute over a model and quickly determine which of its elements represent the components of the modeled genetic circuit.

Among the SBML-compliant tools listed above, there are different degrees of support for import/export of SBML and the various packages that extend its capabilities. In particular, Kera only supports import of SBML, while GEC and GenoCAD only support export of SBML, thereby limiting these tools' ability to facilitate the exchange of data on genetic function. Furthermore, while many of these tools are capable of representing genetic function in terms of hierarchical and/or modular models, iBioSim is the only one that fully represents hierarchy and modularity in a standardized manner, that is by means of the SBML *hierarchical modeling composition package* [45]. Without such standardization, different modeling tools cannot exchange information on the higher-order organization of a model. Consequently, the higher-order function represented by the model is not nearly as communicable, especially in the case of larger designs with many interacting elements. Although support of standardized hierarchy and modularity can be difficult to implement, especially for existing tools with their own custom data model, the benefits of improved exchange and reuse of models generally outweigh the cost of implementation.

1.2.3 Design Composition Tools

Design composition tools enable their users to construct or connect descriptions of both genetic structure and function. Tools from Table 1.1 that fall under this definition include SBOL Designer, DeviceEditor [30], GenoCAD, Kera, TinkerCell, BioJADE, GEC, Parts&Pools [37], Proto Compiler, MatchMaker [34], MoSeC [36], and the Synthetic

Biology Reusable Optimization Methodology (SBROME) [41]. While all of these tools, except SBROME, use at least one of the previously mentioned standards, it is only clear that GenoCAD, TinkerCell, Proto Compiler, and MoSeC use two or more standards to describe both genetic structure and function. Moreover, only TinkerCell and MoSeC explicitly couple genetic function and structure through the annotation of biochemical models with *DNA sequences*.

From a performance standpoint, the design composition tools in the first half of the above list (up to BioJADE) typically require users to manually compose designs, but automatically generate detailed file representations that would be tedious to write by hand. The tools in the second half of this list, however, are more fully automated and can take over a greater portion of design composition tasks from users. These tools include the *sequence generation* tool MoSeC, which is described in this section, and the genetic technology mapping tools BioJADE, GEC, MatchMaker, and SBROME, which are described in Section 1.2.4.

The primary purpose of MoSeC [36] is to infer and generate the DNA sequences for one or more *genetic constructs* (in this case, *genes*) from a SBML or CellML model composed of *Standard Virtual Parts* (SVPs) [35]. A SVP is a standardized model fragment that represents the function of a genetic component. Under the MoSeC approach, a SVP is further annotated with a DNA sequence to directly couple the structure and function of the modeled genetic component. MoSeC accomplishes the inference of genetic structure from function in two steps. In the first step, the input SBML or CellML model is converted to a *graph* representation in which the *nodes* represent model elements and the *edges* or paths between nodes represent the cause-and-effect relationships between model elements. In the second step, this graph is traversed in accordance with a set of *design rules* to compose one or more genetic constructs from the DNA sequences stored at each node. After manually composing a collection of SVPs into a complete model, a MoSeC user can automatically generate the corresponding composite DNA sequence, thereby completing a genetic circuit design that contains both structural and functional data.

While MoSeC uses standards that are well-suited to the representation of genetic function for engineering, the same is not true of its representation of genetic structure. Unlike SBML and CellML, the GenBank format used by MoSeC to describe its output DNA sequences lacks modularity and hierarchy, concepts that help simplify the composition of designs and promote their reuse. Consequently, MoSeC cannot be used to create genetic

circuit designs in which a hierarchy of functional data objects is mirrored by a hierarchy of structural data objects. Without connected structural and functional hierarchies in genetic circuit designs, it becomes more difficult for function-oriented GDA tools to agree with structure-oriented tools on the overall organization of a design. Hence, in order to more fully support meaningful exchange of designs between GDA tools, it becomes necessary for design composition tools to use standards that not only represent genetic structure and function, but do so in a modular, hierarchical fashion that is useful for engineering design.

Finally, while MoSeC's use of SVPs enables the application of a simple rule-based approach to inferring genetic structure from function, it is also limiting in the sense that MoSeC-compliant models can only model a limited set of genetic components and must do so using specific modeling elements from the SBML and CellML standards. If a user wants to model a nonstandard genetic component, then the designers of MoSeC must create a new type of SVP and add design rules to handle its composition with other SVPs and infer their composite structure from their composite function. In the future, design composition tools may benefit from *grammar*-based approaches [46] that allow more input from the user to define and determine what constitutes a valid genetic construct.

1.2.4 Genetic Technology Mapping Tools

Genetic technology mapping is the process of automatically selecting genetic components from a library to meet the abstract functional specification for a genetic circuit. Most existing genetic technology mapping tools adapt techniques originally used in *Electronic Design Automation* (EDA) and *software engineering*. In doing so, these tools must address one of the major differences between GDA and EDA. Namely, unlike *electronic components*, genetic components can produce *signals* with a variety of *molecular identities*. When composing genetic components to form a genetic circuit design, the molecular identities of their signals must be accounted for to ensure proper connections between components and avoid undesirable *cross-talk*. The implication of this restriction is that each choice of a genetic component for a genetic circuit design precludes the choice of other components that would introduce cross-talk. Hence, optimally mapping from a functional specification to a genetic circuit design can be a very computationally intensive problem, one in which every valid combination of genetic components that can possibly produce the specified function may have to be explored in order to ensure that the optimal circuit design is found.

The original genetic technology mapping tools, BioJADE and GEC, use *exact methods* that guarantee the optimal solution is found, but can be very inefficient when applied to large specifications. Both MatchMaker and SBROME, on other hand, use *heuristic methods* to find nonoptimal solutions quickly and rank them by quality afterwards. Still, there is a need for efficient exact methods to find the most optimal solution in a reasonable amount of time and new heuristic methods to bias towards finding higher quality solutions first.

MatchMaker seeks to map an *Abstract Genetic Regulatory Network* (AGRN) against a *feature database* that contains genetic components and data on *qualitative regulatory interactions*, such as *activation* and *repression*, that exist between these components. In order to accomplish this task, MatchMaker uses a *timed heuristic search* to select genetic components from the feature database that match and cover (form a solution to) the AGRN specification. The search is timed in that it continues for a predetermined amount of time before quitting. It is heuristic in that it employs strategies to find solutions quickly, but does not guarantee that the optimal solution is found. These strategies include covering the AGRN nodes with the fewest matches to the feature database first and covering them with genetic components that match the most nodes in the AGRN. In this way, MatchMaker deals with most stringent portions of the AGRN specification first and makes covering decisions that remove the most choices from later consideration. Solutions are obtained, however, without a means to bias towards finding higher quality solutions first. Rather, those solutions found are ranked with regards to the degree that their combinations of genetic components are compatible in terms of their *input* and *output signal ranges*.

SBROME, on the other hand, takes a different approach to database organization. In the SBROME database, genetic components belong to modules that only assert qualitative regulatory interactions between their own components. The advantage of this approach is that modules can be used to cover an AGRN specification in fewer matches than covering feature by feature, but at the cost of potential redundancy between modules. For the purposes of *matching* and *covering*, SBROME uses *greedy methods* that prefer larger, experimentally characterized modules and find a fixed number of solutions while using *look-ahead information* to prune solutions that would result in undesired cross-talk. Consequently, SBROME is able to quickly find solutions that are qualitatively promising, but not necessarily quantitatively optimal.

While both SBROME and Matchmaker are effective tools for genetic technology mapping, their respective approaches to matching and covering can still benefit from being augmented with a *mathematical framework* for evaluating the quality of a solution in terms of *quantitative parameters*. In particular, such a framework can enable these tools to rank matches to an AGRN by their inclusion in a theoretically optimal solution that ignores cross-talk. During covering, this information can leveraged to bias towards finding higher quality solutions first and to avoid looking for suboptimal solutions.

1.3 Contributions

The contributions of this dissertation to the field of synthetic biology are as follows:

- A proposed data model for Version 2.0 of SBOL, a computational standard for the exchange of data on *genetic designs* [47].
- Methodologies for annotating *SBML models* with SBOL and automatically generating SBOL-annotated SBML models from *SBOL modules* to compose descriptions of genetic structure and function into genetic circuit designs.
- An exact algorithm to find optimal solutions to the genetic technology mapping problem in a reasonable amount of time and a heuristic algorithm to find high quality solutions in less time when an exact approach is inefficient [48].
- A methodology for automatically inferring the structure of one or more genetic constructs from genetic circuit designs written in SBOL-annotated SBML [49].

Figure 1.1 is a *workflow diagram* that demonstrates how these contributions can be used to automate the design of genetic circuits. At the beginning of this workflow, a design library is constructed via *model generation*, a process in which SBOL-annotated SBML models are automatically generated from a collection of SBOL modules that assert qualitative regulatory interactions between a collection of genetic components. Next, a SBML model is constructed and serves as an abstract functional specification for input to genetic technology mapping along with the design library. This process produces a genetic circuit design that conforms to the specification and is composed of subcircuit designs from the libary. Following genetic technology mapping, the SBOL DNA components of the genetic circuit design are linearized to one or more genetic constructs in accordance with the organization of the design's composite SBML model, a process known as sequence

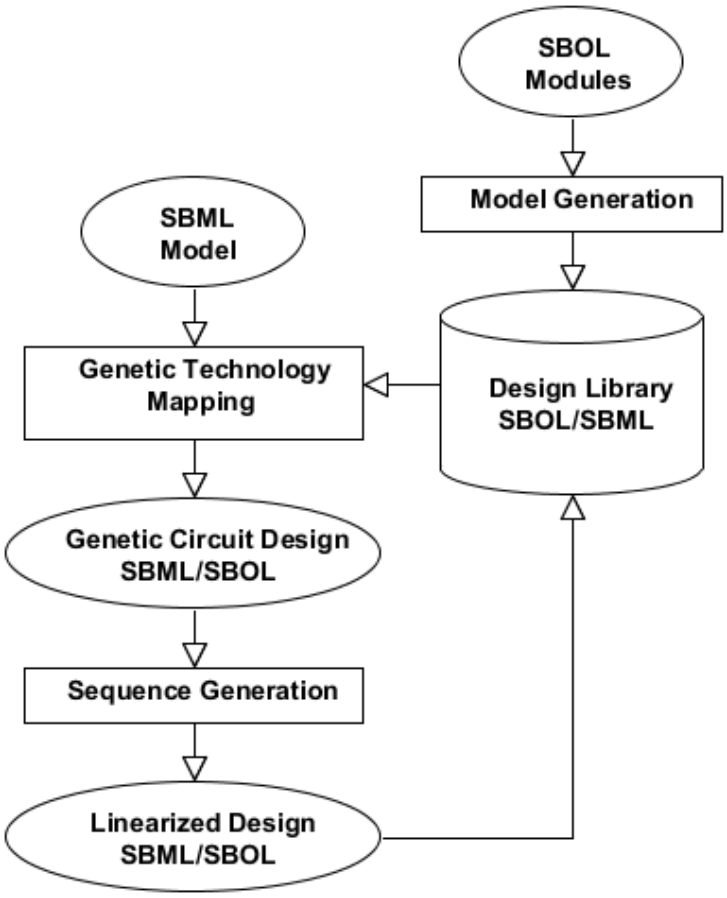

Figure 1.1: A workflow diagram capturing the use of standards, model generation, genetic technology mapping, and sequence generation to automate the design of genetic circuits. Elliptical nodes represent data and rectangular nodes represent processes.

generation. Finally, the genetic circuit design can be curated as part of the design library and become available for solutions to future genetic technology mapping problems.

The results of this research have been implemented in a GDA software tool, iBioSim. This tool is freely available to the public for download at http://www.async.ece.utah.edu /iBioSim/.

1.4 Dissertation Outline

Chapter 2 provides background on both genetic circuits and the current workflow of iBioSim. Next, Chapter 3 describes standards for representing genetic circuit designs, including Version 1.1 of the SBOL standard, which captures the hierarchical composition

of DNA components, and a proposed data model for Version 2.0 of SBOL, which captures a more diverse range of genetic components and enables functional composition of these components into modules. This chapter also describes Level 3 Version 1 of the SBML standard, which captures a wide range of concepts from biochemical modeling, and a stylized form of SBML used in iBioSim, which is especially suited to modeling genetic circuits. Chapter 4 then presents a methodology for the composition of genetic circuit designs from SBOL and SBML, including the annotation of SBML models with SBOL and generation of SBOL-annotated SBML models from SBOL modules. This methodology can be used to create a library of genetic circuit designs for genetic technology mapping, which is explained in Chapter 5 and involves the automated selection and composition of library designs to satisfy the abstract functional specification for a genetic circuit. Chapter 5 also presents the results of applying genetic technology mapping to the functional specifications for a variety of abstract genetic circuits. Finally, Chapter 6 presents a methodology for generating one or more linear genetic constructs from a mapped genetic circuit design, while Chapter 7 concludes this dissertation with a summary of research accomplishments and potential avenues for future research.

CHAPTER 2

BACKGROUND

In this chapter, Section 2.1 describes the fundamental biological processes by which genetic circuits function and presents an example of one of the first synthetic genetic circuits. Next, Section 2.2 discusses the basic assumptions under which genetic circuits are treated as having *digital logic* function, an abstraction that enables the application of existing, well-developed mathematics to the design and analysis of these circuits. Lastly, Section 2.3 describes a GDA software tool, iBioSim [33], and discusses how the contributions of this dissertation have been implemented as part of the iBioSim workflow.

2.1 Genetic Circuits

A genetic circuit is a network of genes that relay and manipulate signals by regulating each other's expression. Each gene is a region of DNA that encodes the primary structure of one or more *Ribonucleic Acids* (RNA) and/or *proteins* and is expressed through the *transcription* of DNA to RNA and the *translation* of RNA to proteins. This section briefly describes transcription and translation, then reviews the common modes of *transcriptional* and *translational regulation* that have been used in synthetic genetic circuits before presenting an example of a seminal synthetic genetic circuit, the *genetic toggle switch* [50].

During the first half of *gene expression*, transcription of DNA to RNA is carried out by protein *enzymes* known as *RNA polymerases*. To initiate the transcription of a gene, RNA polymerase binds to a region of DNA within the gene known as a *promoter*. RNA polymerase then moves along the gene's *Coding Sequence* (CDS) and facilitates the biochemical *reactions* required to assemble a *complementary strand* of RNA. Transcription of a gene ends at a region of DNA known as a *terminator*, while the initiation of transcription can be activated or repressed by *Transcription Factor* (TF) *proteins* that bind to *operator sites* within or adjacent to a gene's promoter. In turn, transcriptional regulation can be enhanced or diminished through the binding of *small molecules* to TF proteins.

During the second half of gene expression, translation of RNA to proteins is carried

out by *complexes* of RNA and protein known as *ribosomes*. To initiate the translation of a *messenger RNA* (mRNA), a ribosome binds to a region within the mRNA known as a *Ribosome Binding Site* (RBS). The ribosome then moves along the mRNA and recruits complementary *transfer RNA* (tRNA) to assemble the amino acids that they carry into proteins. In bacteria, translation can be repressed by *antisense mRNA*, which bind to complementary mRNA to form untranslatable *double-stranded RNA*. In plants and animals, translation can be repressed by *small interfering RNA* (siRNA) [51], which bind to specific mRNA and target them for *degradation* by enzymes.

Figure 2.1 presents an example of a simple genetic circuit, a genetic toggle switch that sets and maintains its *state* through transcriptional regulation. This regulation involves the interaction of the TF proteins *LacI* and *TetR* with the toggle switch genes and the small molecules *IPTG* and *aTc*. The first toggle switch gene is repressed by LacI and codes for both TetR and *Green Fluorescent Protein* (GFP), while the second gene is repressed by TetR and codes for just LacI. Due to their mutual repression of each other's expression, these genes can form a *bistable circuit* in which the expression of one gene or the other tends to dominate after enough time has passed. Ultimately, the bistability of the genetic toggle switch also depends upon the maximum rates at which its repressor TF proteins are expressed, their rates of degradation, and their *degree of cooperativity* (the number of protein subunits they contain or the number of operator sites they bind within the same promoter).

In order to change the state of the genetic toggle switch, the small molecules IPTG and aTc serve to bind and sequester LacI and TetR, respectively. For example, in the LacI-dominant state, IPTG can be added to bind LacI and prevent it from repressing the expression of TetR, such that enough TetR accumulates to effectively repress the expression of LacI and reach a stable, TetR-dominant state. The role of GFP in this genetic toggle switch is to report whether the expression of LacI or TetR is dominant. GFP accomplishes this task by producing a *fluorescence signal* that can be measured using *microscopy* or *flow cytometry*. Since GFP is co-expressed with TetR, high steady-state fluorescence indicates that the expression of TetR is dominant, while low or no steady-state fluorescence indicates that the expression of LacI is dominant. When a genetic toggle switch is incorporated into larger biological systems, its GFP CDS can be replaced with a CDS for a specific biological application, such as inducible *apoptosis* (cell death) or *gene knockout* [54].

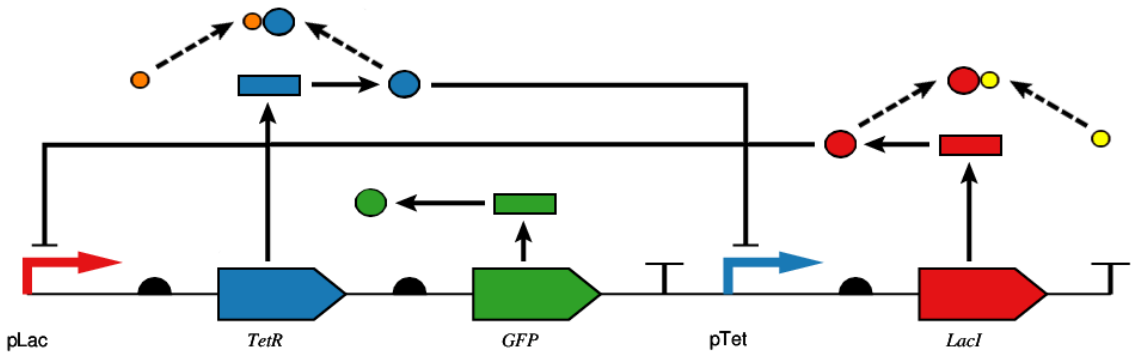

Figure 2.1: A genetic toggle switch. This genetic circuit is composed of two genes running from left to right along the same strand of DNA. Within these genes, each bent arrow is a promoter, each semicircle is a RBS, each box arrow is a CDS, and each T-shaped symbol is a terminator. These symbols were generated using a tool called Pigeon [52] and conform to the SBOL Visual standard [53]. The first gene is *polycistronic gene* in that it codes for both the TF protein TetR and the *reporter protein* GFP. The second is a *monocistronic gene* and hence only codes for the TF protein LacI. The mRNA and TF proteins produced by these genes are represented using rectangles and large circles, respectively, while the small molecules IPTG and aTc are represented using small yellow and orange circles. Finally, the interactions between these genetic circuit components are represented using various arcs, with normal arrows representing transcription or translation, dashed arrows representing *complex formation*, and T-shaped arrows representing transcriptional repression.

2.2 Genetic Logic

The genetic toggle switch is only one member of a larger class of genetic circuits that typically have their function described in terms of digital logic, an abstraction adapted from electrical and computer engineering to synthetic biology. Other genetic circuits that belong to this class include *genetic logic gates* [55, 56, 57], *multiplexers* [58, 59], and some *oscillators* [60]. For the purpose of design, these genetic circuits can be treated as having a finite number of states in which each of their signals is either high or low. This treatment greatly simplifies their analysis by enabling the application of a powerful mathematical framework known as *Boolean algebra* [61], one of the foundations for electronic circuit design.

While it is difficult to estimate the percentage of known genetic circuits that actually function in a manner resembling digital logic, it is worth noting that such function has been observed even among randomly constructed genetic circuits [62]. One common visualization for assessing whether the function of a *combinational genetic circuit* (that is, a genetic circuit with no feedback) adheres to digital logic is a graph depicting the

relationship between its *steady-state input* and *output signals*, also known as a *transfer curve*. Figure 2.2 displays the transfer curves for a TetR-repressed gene encoding LacI (a TetR *inverter*), a LacI-repressed gene encoding GFP (a LacI inverter), and their composition into a genetic logic gate known as a *buffer*.

Mechanistically, it is known that the shape of the transfer curve for a combinational genetic circuit depends on circuit parameters, such as rates of transcription [63] and translation [64], TF *binding affinities* [65], rates of TF degradation [66], and degrees of TF cooperativity [67]. In the case of the LacI inverter and TetR inverter, their transfer curves can be mathematically defined in terms of these parameters using a *sigmoidal equation*. Equations of this form are commonly known as *Hill equations* in biochemical modeling because of their resemblance to Hill's quantitative formulation of oxygen binding to hemoglobin [68]. The derivation of a genetic transfer curve, however, has its roots in the many *Ordinary Differential Equation* (ODE) treatments [69, 70, 71, 72, 73, 74, 75] of Jacob and Monod's qualitative model of transcriptional regulation [76] (Equation 2.1).

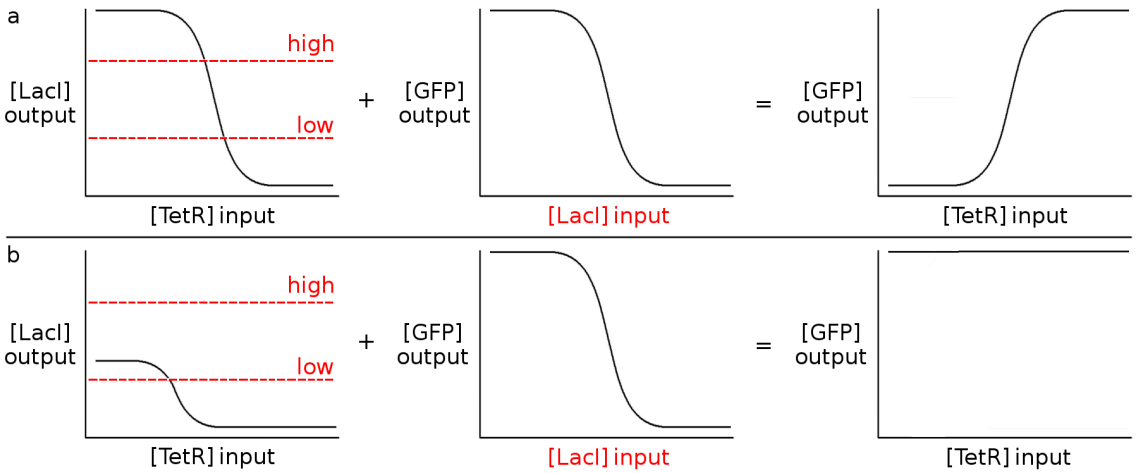

Figure 2.2: Transfer curves for the TetR inverter, LacI inverter, and their composition into a TetR buffer under two different parameter sets. (a) When the TetR inverter and LacI inverter have the same parameters, the high and low outputs of the TetR inverter are above and below the thresholds for high and low inputs to the LacI inverter, respectively. Hence, their composition results in a TetR buffer with digital logic function. (b) When the TetR inverter has lower max gene expression and input binding affinity, however, its high output falls below the threshold for a high input to the LacI inverter. The result is a TetR buffer that is insensitive to low TetR inputs and has a constantly high GFP output.

$$[Output] = \frac{k_o/k_d}{1 + K_a[Input]^n} \tag{2.1}$$

In this equation, k_o is the combined rate of output transcription and translation, k_d is the rate of output degradation, K_a is the input binding affinity, and n is the degree of input cooperativity. Of these parameters, n has the greatest effect on the sigmoidal shape of the transfer curve, with a large n leading to a strongly S-shaped curve that is amenable to abstraction as digital logic. The other parameters effectively scale the transfer curve, with k_o and k_d scaling the range of outputs and K_a scaling the range of inputs. This scaling can have significant consequences for a combinational genetic circuit's composition with other genetic circuits. As seen in Figure 2.2, the TetR inverter is given two parameter sets with equal cooperativity but different rates of gene expression and TF binding affinities. Under the first parameter set, composing the TetR inverter with the LacI inverter results in a buffer that has digital logic function, while doing so under the second parameter set results in a buffer that is insensitive to low input signals. This difference in composite function is due to the fact that the second parameter set shifts the high output of the TetR inverter so that it falls below the threshold for a high input to the LacI inverter.

Other considerations for the proper function of genetic circuits that adhere to digital logic include the inherent *stochasticity* or randomness of gene expression [77, 78], which can have implications for distinguishing between high and low signals, and the different *time delays* of processes associated with gene expression [79], which can have effects on the succession of different states of a genetic circuit. As an example of the former, *signal magnitudes* must be accompanied by a measure of *stochastic noise* in order to assess whether they are truly different. As for the latter, different time delays can lead to different possible sets and orderings of states that a genetic circuit can occupy, especially in the case of *sequential genetic circuits*, for which intermediate and/or output signals are fed back to influence previous signals.

Currently, the approach to genetic technology mapping described in Chapter 5 does not directly account for any of the above considerations. Instead, this approach produces a composite *genetic circuit model* that the user must simulate and otherwise analyze to determine whether its detailed behavior satisfies the logical interpretation of the original specification model. In order to bias towards finding solutions that adhere to digital logic, one potential avenue for future research is to incorporate checks on the previous considerations for logical validity into the process of genetic technology mapping itself.

2.3 iBioSim

iBioSim is the principal GDA tool in which the contributions of this dissertation have been implemented (see Figure 2.3). Besides these contributions, the current capabilities of iBioSim include the composition and analysis of biochemical models, especially models of genetic circuits. Since iBioSim provides full support for Level 3 of SBML [80] and the SBML hierarchical model composition package [45], it can be used to compose SBML models that contain *molecular species*, chemical reactions, *discrete events*, *behavioral constraints*, *submodels*, and annotations, among other modeling data. Other methods to obtain SBML models in iBioSim include importing biochemical models from the BioModels database [81] and generating genetic circuit models from *time series data* on gene expression (*model learning* [82]).

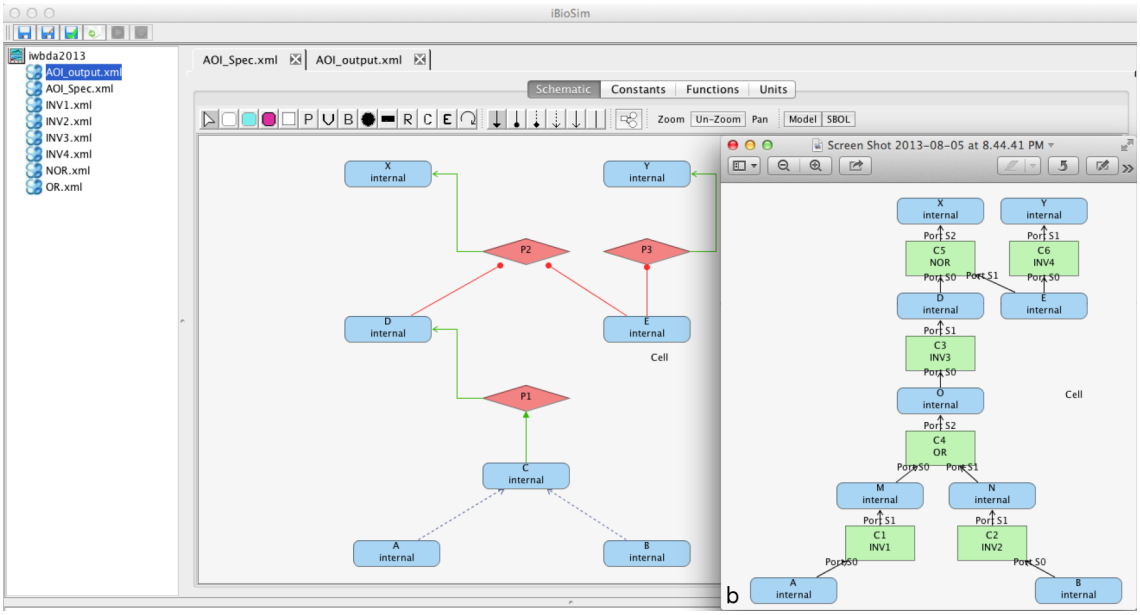

Figure 2.3: A screenshot of genetic technology mapping in iBioSim. (a) Includes a view of a specification model in iBioSim's graphical model editor. Of the symbols shown in this view, blue ellipses represent molecular species and red diamonds represent promoters. As for the arcs between these symbols, red arcs represent repression, dark green arcs represent activation, light green arcs represent *production*, and dashed arcs represent complex formation. (b) Also includes a view of the composite genetic circuit model produced when genetic technology mapping is applied to the specification model. Of the symbols and arcs show in this view, green boxes represent submodels and black arcs represent *port mapping* (connecting species in top-level models to species within submodels).

For the purpose of analysis, iBioSim has multiple ODE and stochastic simulators. When simulation of a reaction-based genetic circuit model is too time consuming, iBioSim can be used to transform the model via automated abstraction methods to a simplified *reaction-based model* [83] or a *quantitative logical model* [84] for more efficient simulation and *stochastic model checking*. In the case of ODE simulation, a single deterministic *simulation trace* is produced that exhibits the average behavior of a genetic circuit, a reasonable approximation when molecule counts are large. In the case of stochastic simulation, on the other hand, many probabilistic simulation traces are produced and can be analyzed in iBioSim for user-specified properties. This process is known as *statistical model checking* and involves calculating the probability of each property as a statistic over the whole population of simulation traces. Lastly, these probabilities can also be determined in iBioSim through *numerical model checking*, which involves the application of *transient Markov chain analysis* to a *Continuous-Time Markov Chain* (CTMC) that is derived from the aforementioned quantitative logical model.

Figure 2.4 is a control flow diagram that presents how genetic circuit model composition and analysis have been integrated with the primary contribution of this dissertation, genetic technology mapping. After a specification genetic circuit model has been composed and refined in iBioSim, it can undergo genetic technology mapping. If one or more solutions are found, the genetic circuit models for these solutions can undergo simulation and stochastic model checking to determine whether or not their more detailed behaviors satisfy the genetic logic of the specification model. Finally, the DNA sequences for any satisfactory solutions can be exported for physical construction and testing in a lab. When no solutions are found via genetic technology mapping or when subsequent analysis yields no satisfactory solutions, the specification model must be refined in order to find one or more solutions that are worth testing in the real world. This is a rational design strategy that uses mathematical models of reality to identify the most promising routes to practical solutions and updates these models when they fail to be predictive of success in the lab.

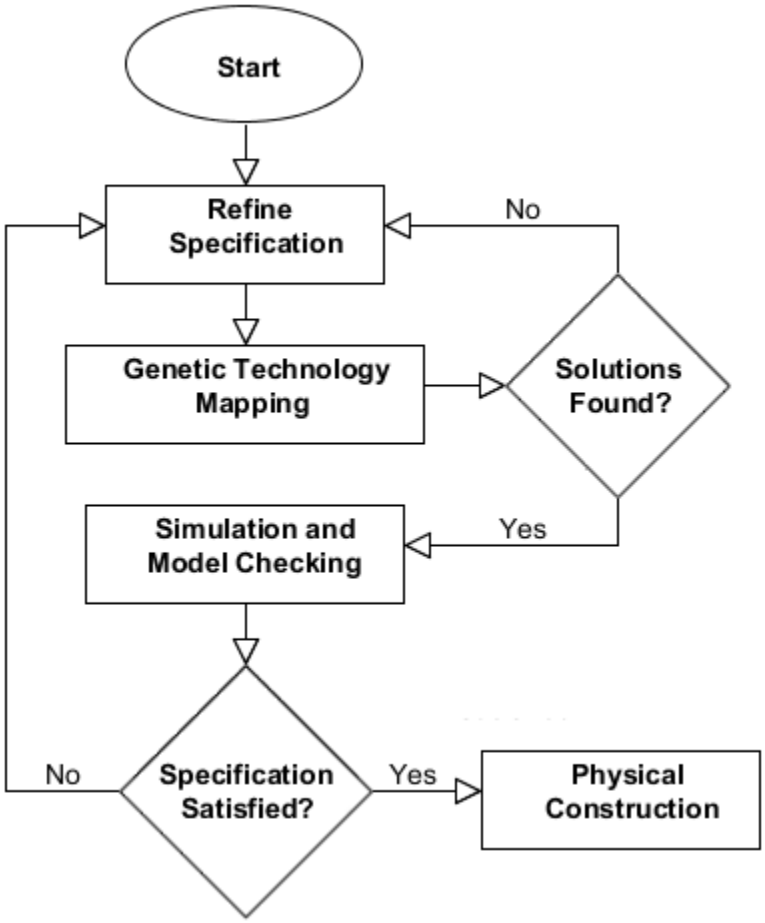

Figure 2.4: A control flow diagram on the use of genetic technology mapping, simulation, and model checking in iBioSim to automate the design of genetic circuits. Rectangular nodes represent processes while diamond nodes represent decisions.

CHAPTER 3

STANDARDS

In this chapter, Sections 3.3 and 3.4 describe SBOL [44], a standard for the exchange of data on genetic designs, while Sections 3.1 and 3.2 cover SBML [14], a standard for the exchange of mathematical models of biological systems. In particular, Section 3.3 describes Version 1.1 of SBOL [85], which represents genetic designs in terms of their DNA components. Section 3.4 then describes a proposal for Version 2.0 of SBOL [47], which we developed to represent genetic designs with more general *classes* of components and modules to compose these components on the basis of cooperative function. Lastly, Section 3.1 covers the core of SBML [80] and one of its extending packages [45], while Section 3.2 describes a stylized form of SBML that includes terms from a *controlled vocabulary* and is especially suited to modeling genetic circuits.

3.1 SBML Level 3 Version 1

SBML is a biological modeling standard that has been developed by the systems biology community and is currently supported by over 250 different software tools. The primary goal of SBML is to enable inter-software exchange of most essential features of models for biological systems, especially mathematical models for describing *cell signaling*, *metabolism*, and *genetic regulation*, among other biochemical phenomena. Accordingly, SBML is a machine-readable *Extensible Markup Language* (XML) [86] that is independent of any proprietary software language or particular analytical framework (for example, ODE or stochastic analysis). To the extent that SBML succeeds at facilitating the exchange of models, researchers may apply new analysis techniques across different software tools without having to recreate their models and risk losing data in the translation between different languages. There also exists a public database for storing and accessing SBML models known as the BioModels database [81].

The latest edition of SBML is Level 3, Version 1, which includes the self-sufficient core of SBML and any number of packages that extend it. Fully SBML-compliant software tools

are expected to support the entire core of SBML, but each tool is free to support only those packages that are useful for its purpose. For example, SBML-compliant software tools that graphically represent their models may support the *layout package*, which extends the core by enabling the specification of coordinates for the location of each element in a model. At present, the list of completed SBML packages includes layout, hierarchical model composition, *qualitative models*, and *flux balance constraints*. This section only covers the core of SBML and the hierarchical model composition package, as these parts of SBML are most relevant to the contributions of this dissertation.

3.1.1 SBML Core

Figure 3.1 presents a simple SBML model of *eukaryotic gene expression* (gene expression in cells with *nuclei*) that captures many of the basic elements of SBML Core, including *spatial compartments*, molecular species, chemical reactions, *mathematical rules*, and discrete events. Species can optionally specify their initial amount or concentration and units of measure, but must specify a compartment to which they belong and assert whether they are a *constant*, a *boundary condition*, or both. Under the guidelines of SBML, a species that is a constant should never change in quantity during a simulation, while a species that is a boundary condition should not change in quantity except by the effect of a rule or event. For the purpose of determining species' concentrations and informing similar calculations, a compartment may contain data on the number of dimensions it occupies, its size, and the units of its size.

Elements that can change the quantity of a species include reactions, rules, and events. A reaction contains lists of *species references* that specify which species act as *reactants*, *products*, or *modifiers* of the reaction. For each reactant and product species, these species references can contain a *stoichiometry number* that indicates the relative amount of reactant or product that is consumed or produced by the reaction. *Modifier species references*, on the other hand, do not include stochiometry since modifiers influence the rate of the reaction without being consumed. To define its rate, a reaction can possess a *kinetic law* that contains a list of local parameters and a MathML expression that operates over these parameters and other data referenced by their elements' IDs, including species quantities, compartment sizes, global parameters, and other reaction rates. Lastly, a reaction must specify whether it is *reversible* and/or *fast*.

In addition to changing the quantity of a species, a rule or event can update a global parameter, compartment size, or even the stoichiometry number of a species reference

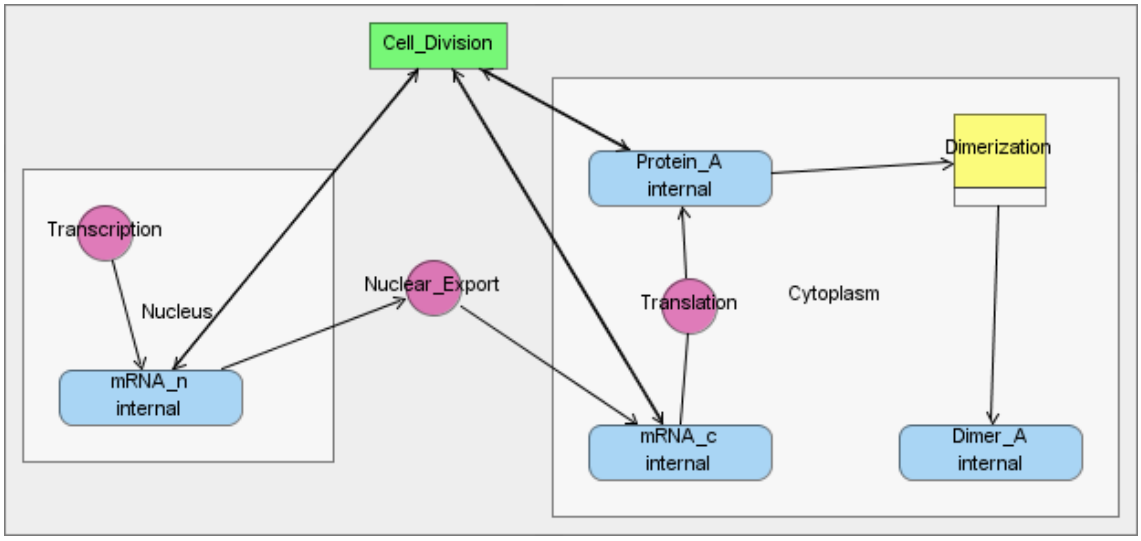

Figure 3.1: A SBML model of eukaryotic gene expression. Compartments are represented with white boxes, species with blue ellipses, reactions with purple circles, rules with yellow squares, and events with green rectangles. This model contains two compartments: the nucleus and cytoplasm of a eukaryotic cell. Within the nucleus, a transcription reaction produces a mRNA species, which is in turn a reactant for a reaction that captures the export of mRNA from the nucleus. This export reaction produces a mRNA species that is contained by the cytoplasm compartment, where it serves as a modifier for a translation reaction that produces protein A. Next, protein A is used by a rule to directly calculate the quantity of protein A *dimers* (two noncovalently bound protein A molecules). In this case, a rule is used to model *dimerization* instead of a reaction under the assumption that dimerization occurs on a faster time scale than transcription, *nuclear export*, or translation and is effectively at *chemical equilibrium* relative to these processes. Finally, a *cell division* event triggers periodically and cuts the quantity of each species in half, except for the protein A dimer since its quantity is calculated from that of protein A.

in a reaction. An assignment rule or rate rule contains MathML that enables direct calculation of the value or rate of change for any of the aforementioned element data. An event, however, can contain multiple *event assignments*, each with its own MathML for instantaneously updating the value of a separate piece of data when the event occurs. Furthermore, an event must contain a *trigger element*, which contains MathML for calculating when the event should occur, and may optionally contain *priority* and *delay elements*, which contain MathML for calculating whether the event occurs before or after another that has triggered simultaneously and how long it takes for the event to occur after it has triggered, respectively.

3.1.2 Hierarchical Model Composition Package

The hierarchical model composition package introduces the concepts of hierarchy and modularity to SBML by means of several new modeling elements, including *external model definitions*, submodels, *ports*, *deletions*, *replacements*. Figure 3.2 presents a hierarchical, modular model of eukaryotic gene expression that captures most of these elements explicitly through its visual representation or implicitly through its underlying SBML. The first two elements, external model definitions and submodels, are used to establish basic hierarchical relationships between models. In particular, an external model definition enables a model to reference the source file for another model, while a submodel enables a model to effectively instantiate one of its external model definitions. This design pattern of model definition followed by instantiation is important because it allows a model to be hierarchically composed from multiple copies of another model under different use cases. For example, a population model of two different *cell types* that contain the same genetic circuit would instantiate two copies of the appropriate genetic circuit model, but these copies would be composed differently with the population model depending on the cell types that they represent.

Of the remaining hierarchical modeling elements, ports are used to designate an interface for a model and help to enforce its modularity, while replacements and deletions are used to indicate how to compose and modify submodels as part of a parent model. More specifically, a port on a model refers to an element inside the model by its ID and serves as a controlled point of exposure for reference by replacements and deletions in other models. While replacements and deletions are technically allowed to refer to any modeling element, in engineering design it is typical to treat a model as a modular entity and refer only to its ports for the purpose of composition, since these refer to a designer's intended inputs and outputs of the model. Both deletions and replacements refer to elements in a submodel by referencing their IDs or the IDs of their corresponding ports.

As its name suggests, a deletion indicates which element in a submodel (technically the externally defined model referenced by the submodel element) should be deleted if the parent model is flattened, a process in which the elements of each submodel are integrated with the elements of the parent model and all hierarchy is removed. While deletions are listed directly on a submodel element, replacements appear on the elements of the parent model that replace or are replaced by elements in a submodel. In particular, a replaced element indicates that a parent element should replace an element in a submodel

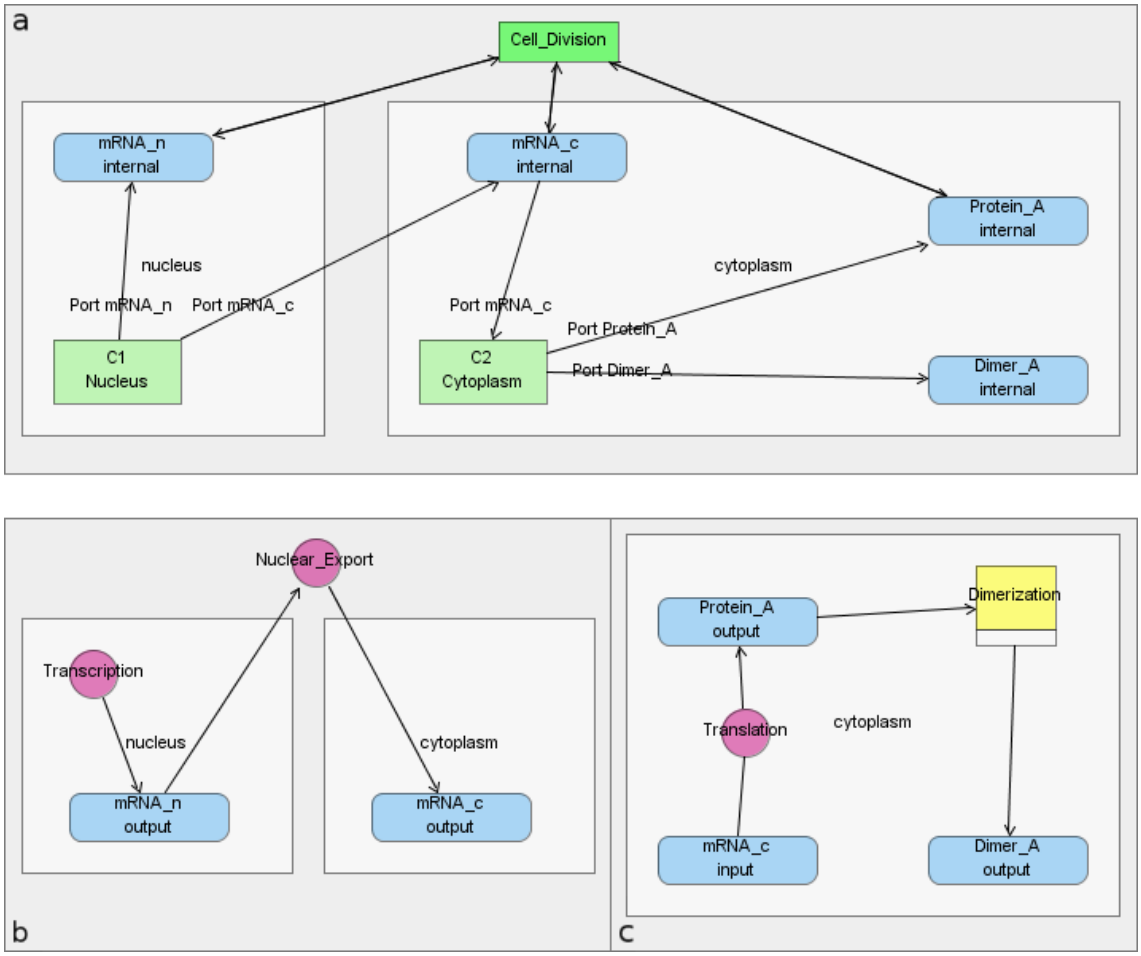

Figure 3.2: A hierarchical, modular SBML model of eukaryotic gene expression. (a) The top-level model contains contains mRNA and protein species, a cell division event that periodically modifies their quantities, and two submodels (light green rectangles) that capture biochemical processes occurring between these species within the nucleus and cytoplasm of a eukaryotic cell. In this model, an arc pointing from a species to submodel indicates that the species should replace a submodel species referenced by an input port, while an arc pointing from a submodel to a species indicates that the species should be replaced by a submodel species referenced by an output port. There are two external models referenced by these submodels. (b) The first captures transcription and export of mRNA from the nucleus. (c) The second captures translation and dimerization of protein in the cytoplasm. As indicated by their input/output labels, species in these external models are referenced by ports that expose them to replace or be replaced by species in the top-level model. Similarly, each compartment is also referenced by a port for replacement, though these replacements are not visualized here.

during flattening, while a replaced-by element indicates that the parent element should be replaced by an element in the submodel.

3.2 SBML for Genetic Circuit Models

While SBML provides fairly general chemical and mathematical elements for encoding models, such as species, reactions, and parameters, it lacks elements that explicitly capture biological and genetic meaning, such as TF proteins, promoters, and gene expression processes. To enable GDA software tools to supply such meaning, SBML allows the addition of *Systems Biology Ontology* (SBO) terms to each of its modeling elements [49]. SBO is a controlled vocabulary of terms that are commonly used in systems biology and cover a wide range of biological, genetic, and modeling concepts. Figure 3.3 presents two versions of a genetic circuit model constructed in iBioSim [33], one labeled with SBO terms and one not labeled.

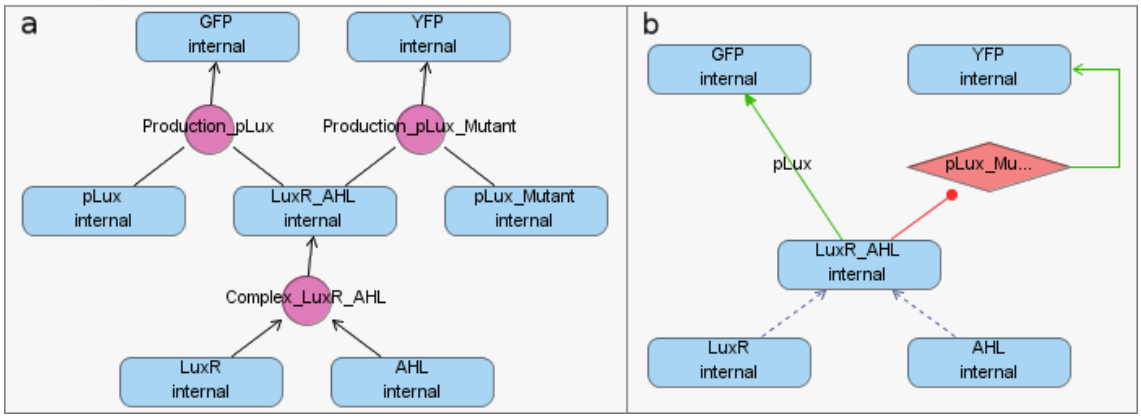

Figure 3.3: Two versions of a SBML model for a genetic circuit in iBioSim. (a) In the version without SBO terms, three chemical reactions are displayed. The bottom reaction has two reactant species and a single product species, which serves as a modifier species for the other two reactions. In turn, each of these reactions has an additional modifier species and single product species. Without SBO terms, this is the most meaningful description that can be made of the first version of the model from machine-readable data. (b) With SBO terms, however, the second version can be described as having complex formation between the *LuxR* and *AHL* species. Next, the resulting complex can be described as activating the *pLux* promoter and repressing a mutant pLux promoter, which control the genetic production of the GFP and *Yellow Fluorescent Protein* (YFP) species, respectively. Note that promoters can be displayed in iBioSim either implicitly as a label on a regulatory arc or explicitly as a separate diamond node.

In order to capture several common aspects of genetic circuit models, iBioSim uses terms from the *process type*, *participant role*, and *metadata* subvocabularies of SBO. Table 3.1 provides a complete enumeration of these SBO terms, their *accession IDs*, and the SBML elements to which they are applied. Ultimately, these terms enable iBioSim to preserve genetic meaning when generating SBML models from SBOL modules (see Chapter 4) and facilitate the tool's assignment of logical meaning to these models during genetic technology mapping (see Chapter 5).

3.3 SBOL Version 1.1

SBOL [44] is an emerging data exchange standard for synthetic biology with growing support among GDA software tools, including sequence editing tools [19, 22], biochemical modeling tools [43, 40, 33], and design composition tools [43, 32, 40, 36, 30]. SBOL has been developed by members of the synthetic biology community to document DNA components for the primary purpose of engineering design. Unlike existing standards that were originally conceived for documenting naturally occurring genetic sequences, such as the FASTA [13] and GenBank [12] formats, SBOL can be used to document partial genetic designs and recursively annotate the sequences of DNA components with other DNA components in a hierarchical fashion. These capabilities of SBOL address the iterative, modular character of engineering design in a way that current standards for genetic sequences neglect. Furthermore, SBOL is an extensible standard that can be adapted to meet the evolving needs of the synthetic biology community, such as the need to combine structural, sequence-oriented descriptions of genetic circuits with descriptions of their function.

The current capabilities of SBOL Version 1.1 are illustrated using symbols taken from

Table 3.1: Addition of SBO Terms to SBML in iBioSim

SBO Term	**Accession ID**	**SBML Element**
Genetic Production	SBO:0000589	Reaction
Noncovalent Binding	SBO:0000177	
Inhibitor	SBO:0000020	Modifier Species Reference
Stimulator	SBO:0000459	
Promoter	SBO:0000598	
Input Port	SBO:0000600	Port
Output Port	SBO:0000601	

the SBOL Visual standard [53] in Figure 3.4. In this example, the DNA component for a genetic toggle switch [50] is hierarchically composed from a TetR-repressible gene and a LacI-repressible gene, which are in turn composed from the pTet promoter, the *cLacI* CDS, RBSs, terminators, the *pLac* promoter, and the *cTetR* CDS. In the case of the toggle switch component, one of its subcomponents (the TetR-repressible gene) is located on its *negative/reverse complement strand*.

In SBOL Version 1.1, a *collection* is a group of DNA components that have something in common, such as the result of a database query to find all transcriptional promoters. DNA components, on the other hand, are at the core of SBOL Version 1.1 and represent the abstraction of a particular DNA sequence for engineering design. Finally, a *sequence annotation* is strictly tied to a parent DNA component and indicates the absolute or relative positions of other DNA components on its parent component's DNA sequence. A

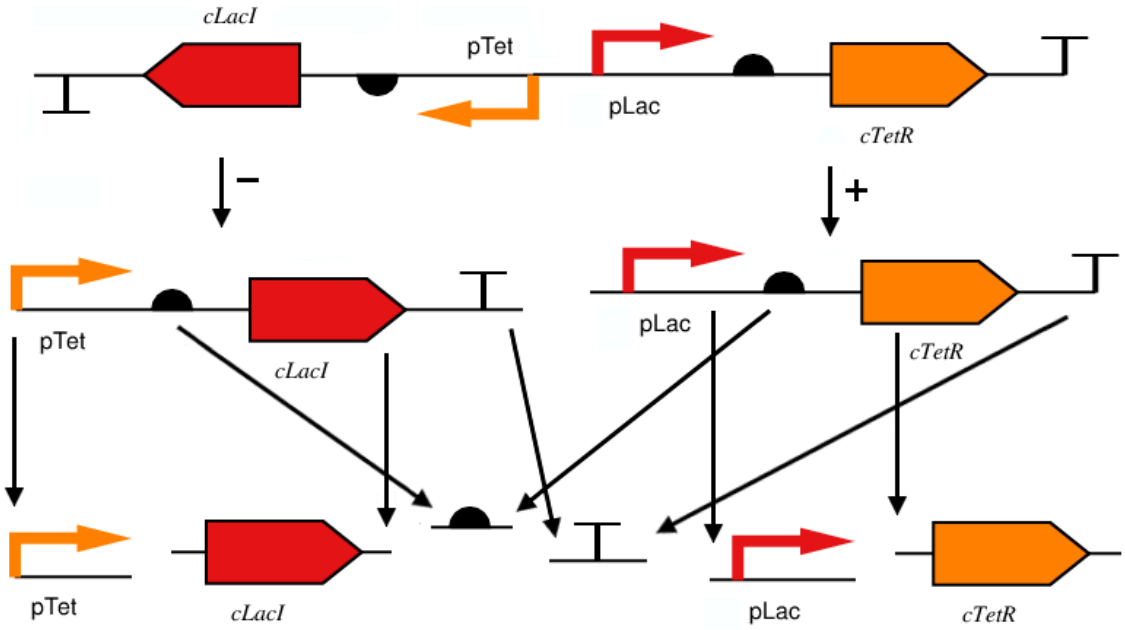

Figure 3.4: Hierarchical composition of the DNA component for a genetic toggle switch in SBOL Version 1.1. Each grouping of subcomponent symbols along a solid line represents a single composite DNA component. Of these symbols, each bent arrow represents a promoter, each semicircle represents a RBS, each box arrow represents a CDS, and each T-shape represents a terminator (see the SBOL Visual standard). This figure was partly constructed using Pigeon [52], a SBOL Visual-compliant tool.

detailed representation of the capabilities of SBOL Version 1.1 can be found in Figure 3.5, which is a *Unified Modeling Language* (UML) [87] *class diagram*.

As documented in this figure, any data object that belongs to these classes must have a *Uniform Resource Identifier* (URI) [88] such that it can be uniquely identified by software and databases across the World Wide Web. As a consequence of this requirement, if a user modifies a data object that is published on the Web, then as a best practice they should also change the URI of said object. When this requirement is met, different users can determine whether they have the same SBOL data object by simply checking whether the URIs of their objects are identical, rather than by exhaustively comparing the *data fields* of their objects. In addition, if the URI of a SBOL data object is also a *Uniform Resource Locator* (URL), then a user can refer to the object's location on the Web or elsewhere in their file system if they do not wish to store a local copy. A typical form for a URI is a *scheme name* and *authority* followed by a *path* and/or *fragment* containing an ID

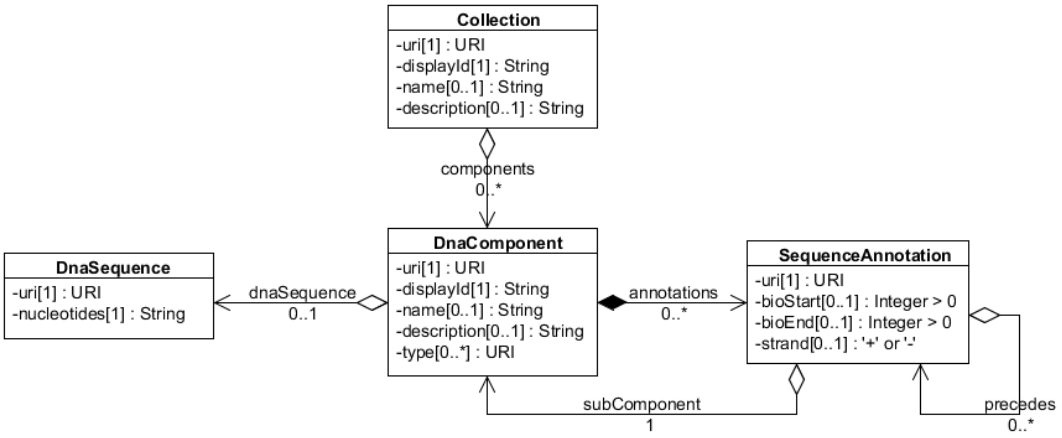

Figure 3.5: The UML class diagram for SBOL Version 1.1, consisting of the Collection, DNA Component, DNA Sequence, and Sequence Annotation classes. Each data object that belongs to these classes contains a variety of data fields, including strings of characters that identify, name, and describe the object and URIs that type and uniquely identify the object. A white diamond arrow indicates that objects of one class refer to and aggregate objects of other classes, while a black diamond arrow indicates both aggregation and ownership. For example, if a DNA component is deleted, then all of its sequence annotations are deleted, since they are owned by that DNA component. The same is not true of a DNA component and its DNA sequence, since another DNA component may share the same sequence.

that is unique under the preceding authority. An example of a URI following this form is "http://parts.igem.org/Part:BBa_R0040," which is a possible URI for the TetR-repressible promoter from the iGEM Registry of Standard Biological Parts [16]. The different portions of this URI include its scheme name, "http," its authority "parts.igem.org," and its path, "/Part:BBa_R0040," which contains an ID that is unique to the registry.

In addition to a URI, each DNA component must have a *display ID* and can have at most one name, description, and DNA sequence. Like a URI, the purpose of a display ID is to uniquely identify a DNA component, but the display ID does not have to be unique across the World Wide Web and is intended to be short and easy to read. A name and description, on the other hand, identify and describe a DNA component using plain, unstructured text that is not necessarily unique within an SBOL file or easily reasoned over by machines. Examples of each of these data fields can found in Figure 3.6, which contains a UML example of a composite SBOL DNA component. Finally, each DNA component may also have any number of type URIs and sequence annotations. If a DNA component has any type URIs, then at least one of these URIs must refer to a term from the *Sequence Ontology* (SO) [89]. An ontology is a controlled vocabulary that captures terms and relationships between terms from a specific knowledge domain, thereby enabling machine reasoning over the domain. In the case of the SO, the captured knowledge domain is the annotation of biological sequences with sequence features.

Each sequence annotation of a DNA component can have a single pair of *bioStart* and *bioEnd integers* or a *precedes reference* to another sequence annotation. When present, the bioStart and bioEnd integers bound the position of a subcomponent on the DNA sequence of the parent DNA component. When one or more subcomponents do not have a DNA sequence, however, a complete DNA sequence cannot be assigned to the parent DNA component and the positions of its subcomponents cannot be exactly specified by its sequence annotations. In this case, a *partial genetic design* can be specified using precedes references between sequence annotations to indicate the relative positions of their subcomponents. This capability is necessary to satisfy the iterative nature of engineering, in which some of the details of a design cannot be specified immediately and must be revisited later in the design cycle. Lastly, each sequence annotation can have either a '+' or '−' character to indicate whether its subcomponent is located on the *positive strand* or negative strand of its parent DNA component.

Finally, the DNA sequence of a DNA component has a single non-URI data field, a

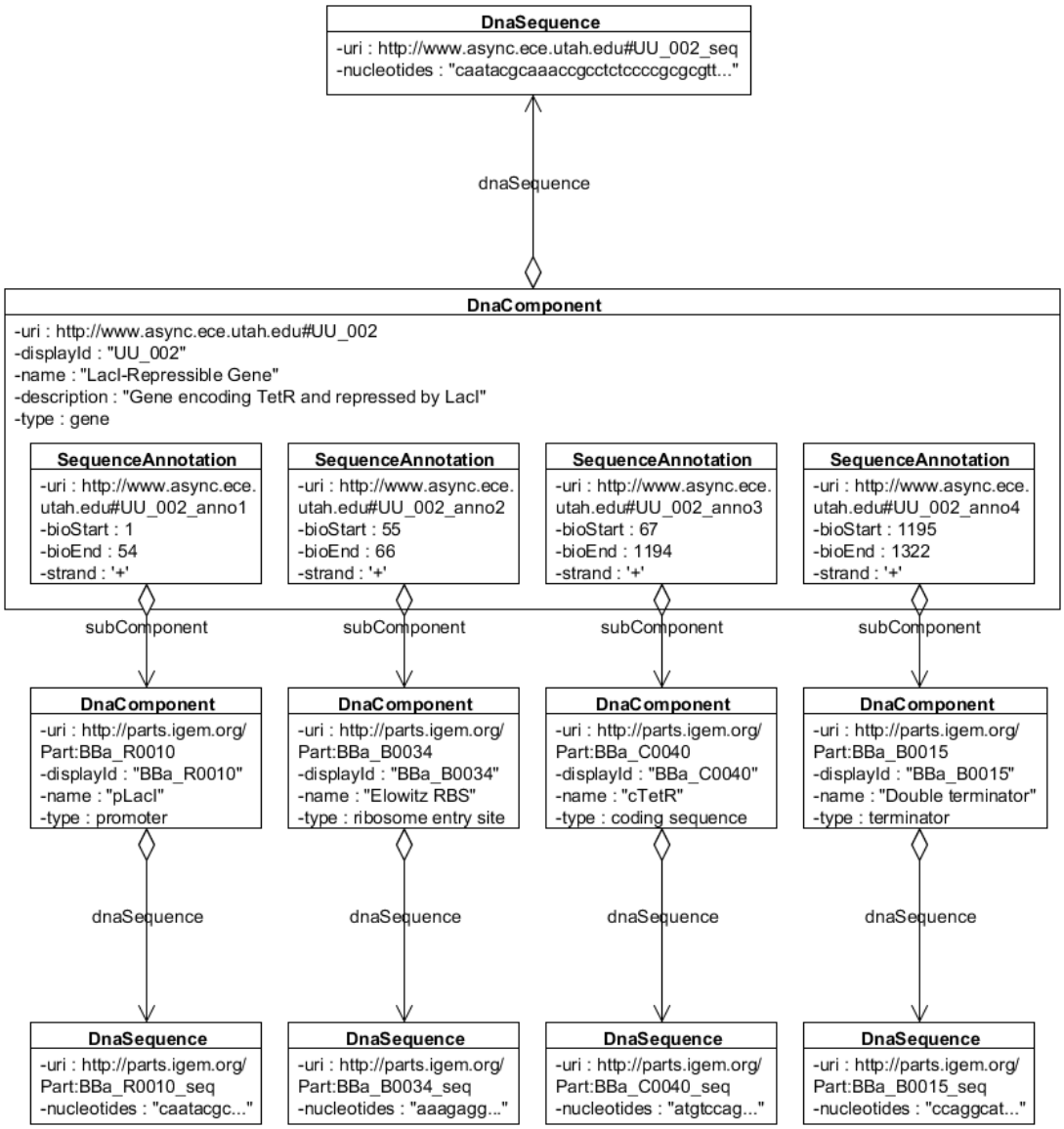

Figure 3.6: SBOL Version 1.1 UML for a LacI-repressible gene that encodes the TF protein TetR. Sequence annotations are placed inside the DNA component UU_002 to show that they are owned by the component. These sequence annotations indicate that four DNA subcomponents are located side by side on UU_002's DNA sequence, including the promoter BBa_R0010, the RBS BBa_0034, the CDS BBa_C0040, and the terminator BBa_0015. Accordingly, the DNA sequence of UU_002 is the concatenation of the sequences of its subcomponents.

string of characters that code for the sequence's nucleotides. Note that these characters must conform to the International Union of Pure and Applied Chemistry (IUPAC) codes for completely and incompletely specified *bases* in nucleic acid sequences [23].

3.4 Proposed Data Model for SBOL Version 2.0

This section describes the proposed data model for Version 2.0 of SBOL, which was presented at the SBOL 10 workshop held at the University of California, Berkeley and voted on as a starting point for the next version of SBOL. It is important to emphasize that this data model does not represent the final, community-approved specification for the next version of SBOL. Rather, this data model is a proposal that draws from discussions within the SBOL community. It is an intermediate result in a larger development process, one in which feedback is being gathered from the synthetic biology community at large in order to reflect, fulfill, and standardize its data exchange requirements.

The primary goal of the proposed data model is to make SBOL a more comprehensive standard for genetic design. Since synthetic biology encompasses research into a broad range of entities and materials, SBOL must grow to represent a similarly broad range of structural components for genetic design. In order to more fully support the representation of genetic structure, the proposed data model generalizes the DNA component class of SBOL Version 1.1 to represent components with and without sequences. As a consequence, this data model can be used to represent RNA components, such as mRNA, tRNA, and small interfering RNA (siRNA) [51], as well as protein components, such as TF proteins and enzymes. Furthermore, the proposed data model can be used to represent potentially nongenetic components of a design, such as *environmental factors*, small molecules, molecular complexes, *nonbiological polymers*, and even light.

Since synthetic biology is increasingly concerned with the intended function of genetic designs, SBOL must also be extended to support minimalistic, qualitative representations of genetic function and refer to more detailed, quantitative representations written in specialized, external standards. To meet these needs, the proposed data model introduces classes for functional modules, molecular interactions, and mathematical models. Examples of functional modules include genetic logic gates, oscillators, *sensors*, and *signaling cascades*, while examples of molecular interactions include transcription, translation, activation/repression, *noncovalent binding*, and *phosphorylation*.

Finally, in order to be more useful for the purpose of engineering design, the proposed data model enables the hierarchical composition of separate yet connected descriptions

of genetic structure and function. In particular, the data model introduces classes for *instantiation* and port mapping, two abstract, well-established concepts borrowed from the domain of electrical and computer engineering. As explained later on, instantiation allows the creation of a modular hierarchy by incorporating one or more copies of a subdesign in a composite design, while port mapping allows the specification of connections between designs by asserting the correspondence of elements within these designs. These concepts simplify the process of creating a large, complex design by facilitating the reuse of previous designs in its construction, factoring out recurring design patterns that would otherwise be redundant, and splitting a design into multiple distinct layers that warrant separate consideration.

As an example, consider the design for a genetic toggle switch, as shown in Figure 3.7. The proposed data model captures not only this design's structure, but also its basic function. First, generalized components allow the representation of RNA components (such as the mRNA coding for TetR), protein components (such as the TFs TetR and LacI), and small molecules (such as IPTG). Next, interactions can be specified between these components, such as the transcription of the cTetR CDS to TetR mRNA and the latter's translation to TetR protein. Other examples include the repression of the pLac promoter by LacI and the binding of LacI by IPTG to form a complex. In turn, these components and their interactions can be grouped into functional modules, such as a LacI inverter. Finally, these modules can be instantiated as part of larger modules, such as the instantiation of the TetR inverter and LacI inverter to form the genetic toggle switch. The points of connection between modules are specified using ports, while the connections between modules are established using port maps. The rest of this section describes each of these new features in greater detail.

3.4.1 Minor Improvements

One minor improvement made by the proposed data model is the creation of two abstract classes, the *Identified* and *Documented* classes. These classes enable more efficient representation and implementation of SBOL by separating out data fields that are common to many classes and placing them into super classes that other classes may extend. The Identified class contains two data fields. The first is a URI that serves to identify the objects of any class that implements the Identified class, in the same way that data objects are identified with URIs in SBOL Version 1.1. The second is an annotation string that may contain a user's custom data that is not explicitly captured by SBOL. This string

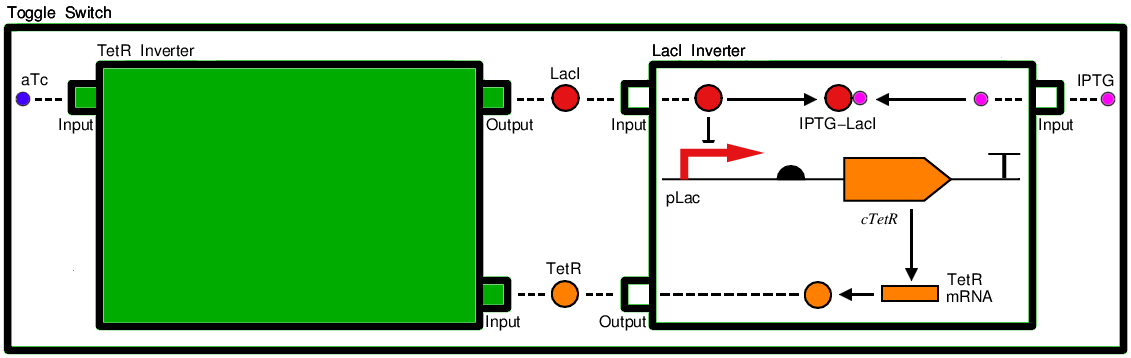

Figure 3.7: A design for the genetic toggle switch that captures its qualitative structure and function. The design consists of three functional modules in the form of a composite toggle switch module that contains connected copies of a TetR inverter module and a LacI inverter module. The LacI inverter module contains copies of a composite DNA component for the LacI-repressible gene, a TF protein component for LacI (red circle), a TF protein component for TetR (orange circle), a mRNA component for TetR (orange rectangle), a small molecule component for IPTG (pink circle), and a molecular complex component for LacI bound to IPTG. This module asserts a variety of molecular interactions between its contained components (solid arrows), including the repression of pLac by LacI, transcription of cTetR to TetR mRNA, translation of TetR mRNA to the TetR TF, and noncovalent binding of LacI to IPTG. While these modules allow different parts of the design to be treated as "black boxes" that have most of their contents ignored (see the TetR inverter), the ports on these modules allow connections between them (dashed lines). For example, the toggle switch is connected to the LacI inverter through mapping of the latter's input port to copies of LacI contained by both modules. In turn, the TetR inverter is connected to both the toggle switch and LacI inverter through mapping of its output port to the copy of LacI in the toggle switch.

must take the form of one or more predicate-object pairs that adhere to the guidelines for the *Resource Description Framework*/XML (RDF/XML) [90] language in which SBOL is written.

The Documented class contains three data fields: a display ID, a name, and a description. The contents of these data fields are identical to those of the same name in SBOL Version 1.1. Note that the Documented class inherits from the Identified class since, while all classes in SBOL are Identified classes, not all of them are Documented classes, such as the Sequence class. Rather, sequences are effectively documented by the sequence components that abstract them for the purpose of engineering design.

Finally, Figure 3.8 contains an example of a class from SBOL Version 1.1 that is now identified and documented: the Collection class. Under the proposed data model, objects

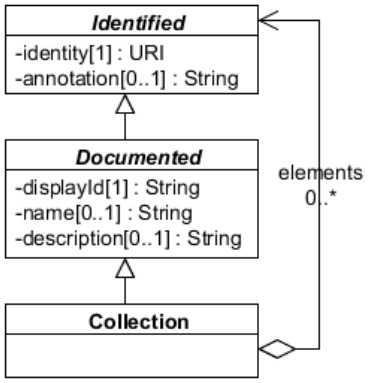

Figure 3.8: UML diagram for the Identified, Documented, and Collection classes of the proposed data model. White triangular arrows indicate the one class inherits the data fields and aggregation/ownership relations of another class. Note that classes with italicized names are abstract classes that are meant to be extended by other classes and not used directly. In this class, the Collection class extends both the Identified and Documented classes, therefore it must have an identity and display ID. Optionally, it may have an annotation, name, or description.

of this class can contain one or more objects that inherit from the Identified class. In other words, a collection may now contain one or more SBOL objects of any class from the proposed data model.

3.4.2 Structural Representation

To support an increased range of structural representation, the proposed data model generalizes DNA components to components with a sequence, or *sequence components*. The Sequence Component class captures previously unrepresented genetic components, such as RNA and protein components, and is sufficiently general to represent nongenetic components with a sequence, such as nonbiological polymers. In order to capture components without a sequence, such as small molecules, molecular complexes, and light, a *Generic Component* class is also introduced. As shown in Figure 3.9, both classes inherit from an abstract Component class that may aggregate one or more subordinate *component instantiations* and must have a type URI that refers to a term from an appropriate ontology, such as *Chemical Entities of Biological Interest* (ChEBI) [91]. This URI documents the basic sort of biochemical or physical entity (for example, DNA) that a component abstracts for the purpose of engineering design. The sequence type URIs

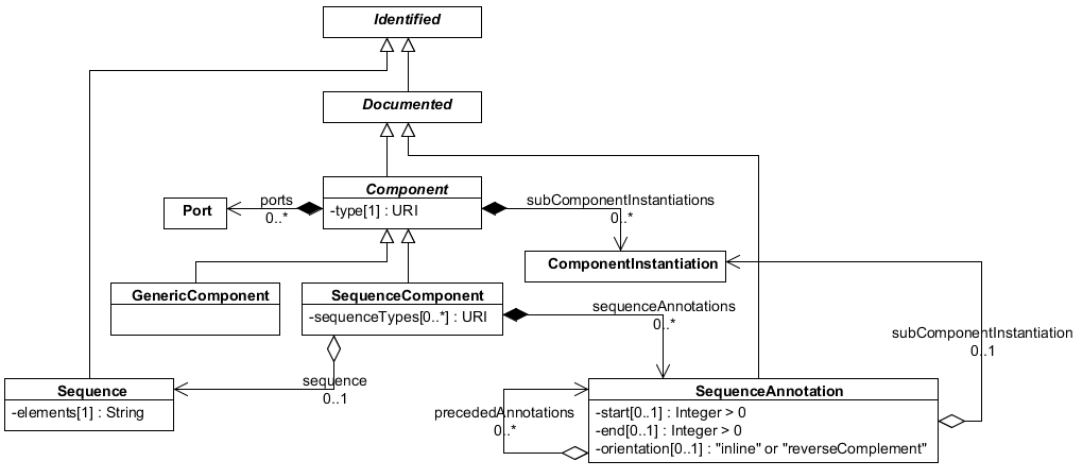

Figure 3.9: UML diagram for the proposed generalized Component classes. Since Generic Component and Sequence Component inherit from the abstract Component class, objects belonging to these classes can aggregate one or more objects that belong to the Port and Component Instantiation classes. Sequence components may additionally aggregate one or more objects of the Sequence Annotation class and up to one object of the Sequence class. In turn, a sequence annotation can refer to a component instantiation to effectively position it on the sequence of its parent sequence component.

of a sequence component, on the other hand, are analogous to the type URIs of a DNA component in SBOL Version 1.1 (see Figure 3.5). When possible, the sequence type URIs are expected to refer to SO terms to clarify the role or nature of the sequence that is abstracted by the component. For example, a sequence component of type DNA may have a sequence type of "promoter" or "terminator," while a sequence component of type protein may have a sequence type of "binding site" or "protease site."

Similar to a DNA component in the SBOL Version 1.1 data model, a sequence component can refer to sequence annotations to document the absolute or relative positions of subcomponent instantiations along its sequence. Unlike in SBOL Version 1.1, sequence annotations do not directly refer to subcomponents, but rather to instantiations or usages of these subcomponents that may be exposed via ports and mapped to other component instantiations for the purpose of design composition. Finally, a sequence component can refer to an object of the Sequence class that contains a string of characters encoding its elements. A sequence's string encoding must adhere to the IUPAC codes for the type of sequence component that refers to it. For example, a sequence that is referred to by

DNA components should contain a string of IUPAC-approved characters that represent different nucleotides.

While this data model can be further extended by subclassing sequence components into DNA, RNA, and protein components and adding classes for small molecules and environmental factors, care must be taken to avoid creating a data model that is overly refined. Such a data model would have many classes, but no data-specific reason to distinguish between them. In the case of DNA, RNA, and protein components, however, there may be near-term reasons to distinguish among them, such as the different elements that make up their sequences and the single-strandedness of protein components. These are reasons that restrict the contents of the proposed Sequence and Sequence Annotation classes.

The alternative approach is to supplement the proposed data model with validation rules. For example, these rules could include that sequence components of type "protein" are only annotated with other sequence components of type "protein," that the orientation of their sequence annotations is always set to "inline," and that their sequences only contain characters taken from the IUPAC amino acid code. As the proposed data model continues to be implemented for testing, the SBOL community intends to explore both approaches.

Next, the structural composition of components is enabled through component instantiations. Under the SBOL Version 1.1 data model, composite DNA components are composed by annotating their sequences with other DNA components. As shown in Figure 3.9 and Figure 3.10, this composition pattern is also true under the proposed data model, but the Sequence Annotation class now refers to an object of the Component Instantiation class, thereby explicitly documenting that a sequence annotation positions a particular instance or usage of a component, rather than the component itself. This distinction is necessary to allow different copies of a component to be referred to and treated differently on the basis of their physical location or other environmental context. In addition, by generalizing the concept of component instantiation, the proposed data model allows generic components without a sequence to be composed from instances of other components.

As an example of structural representation under the proposed data model, Figure 3.11 presents a UML object diagram for the components of one half of the genetic toggle switch, including sequence components that are engineering abstractions of DNA, RNA, and

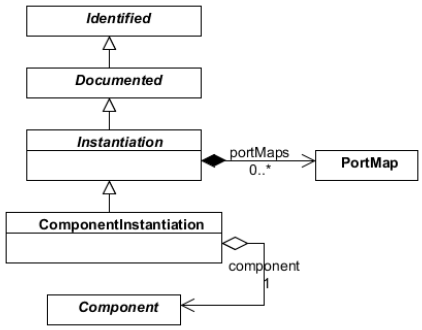

Figure 3.10: UML diagram for the proposed Component Instantiation class. As an Instantiation class object, a component instantiation is allowed to aggregate port maps to connect any ports on the component that it instantiates.

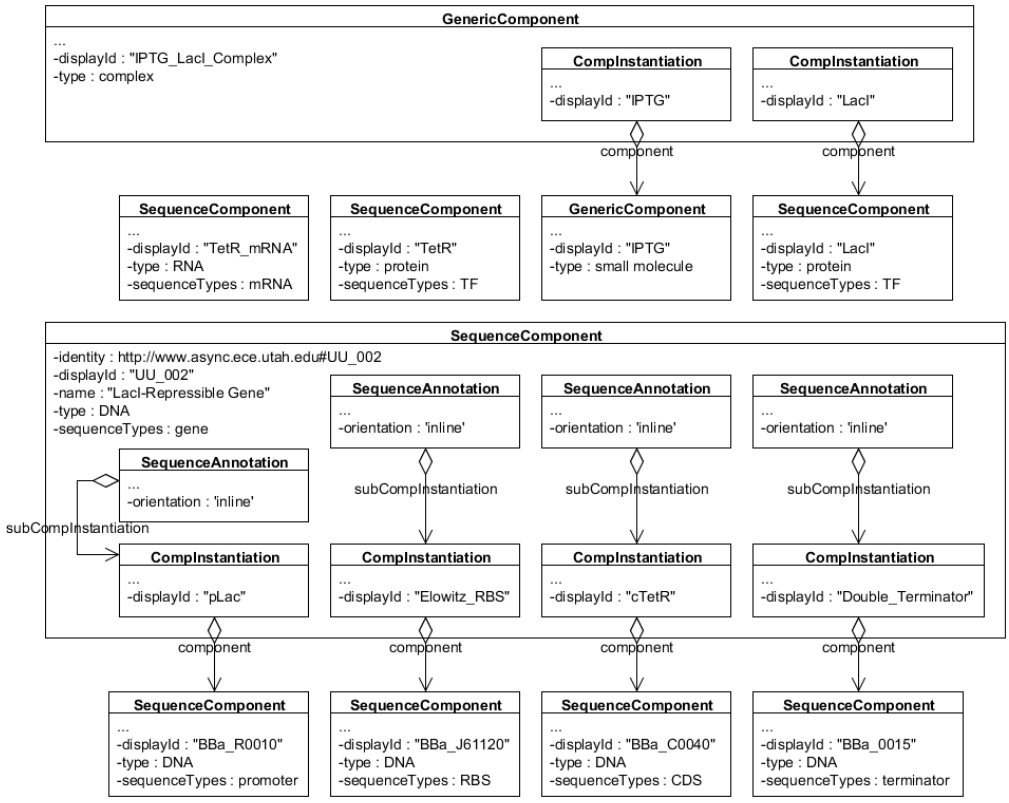

Figure 3.11: UML example of components under the proposed data model, including components referenced by the LacI Inverter module. In this figure, "comp" is short for "component."

protein, and generic components that represent small molecules and molecular complexes. Of these components, only the LacI-repressible gene and IPTG-LacI complex have any substructure. In particular, the gene's sequence is annotated with the instantiations of four other sequence components of type "DNA," while the complex is composed from the instantiations of a generic component of type "small molecule" and a sequence component of type "protein."

3.4.3 Functional Representation

To address the need for functional descriptions in SBOL, the proposed data model adds the Module, Interaction, and Model classes. These classes provide a firm basis for functional representation in SBOL without going so far as to create a new standard for mathematically modeling biology. This is because there already exist several established languages for modeling biology, such as SBML, CellML [15], and even MatLab [39]. Rather, these classes enable users of SBOL to group components that function together, describe the basic qualitative interactions between these components, and document references to standard mathematical models that are external to SBOL and that provide more detailed descriptions of component function.

As displayed in Figure 3.12, the Module class forms the hub for functional description of genetic designs. A module aggregates zero or more component instantiations, *module instantiations*, interactions, models, and ports. A component instantiation owned by a module refers to a component as a functional entity for the purpose of playing a role in an interaction (described in more detail below). In this way, a module instantiates components that work together to perform an intended function. Module instantiations (see Figure 3.13), on the other hand, enable the composition of a module from other modules. As described in Section 3.4.4, the connection of the module instantiations within a module is accomplished via ports and port mapping.

Next, interactions provide a qualitative basis for asserting the intended function of a given module. The proposed data model supports regulatory interactions, such as activation or repression, and processes from the central dogma of biology, such as transcription and translation. Other supported interaction types include noncovalent binding between a small molecule and TF or phosphorylation of a TF by an enzyme. Each interaction must document its type by referencing a term from the SBO, a controlled vocabulary of terms commonly used in systems biology. Furthermore, each interaction must document its participating component instantiations by referring to one or more objects of the Par-

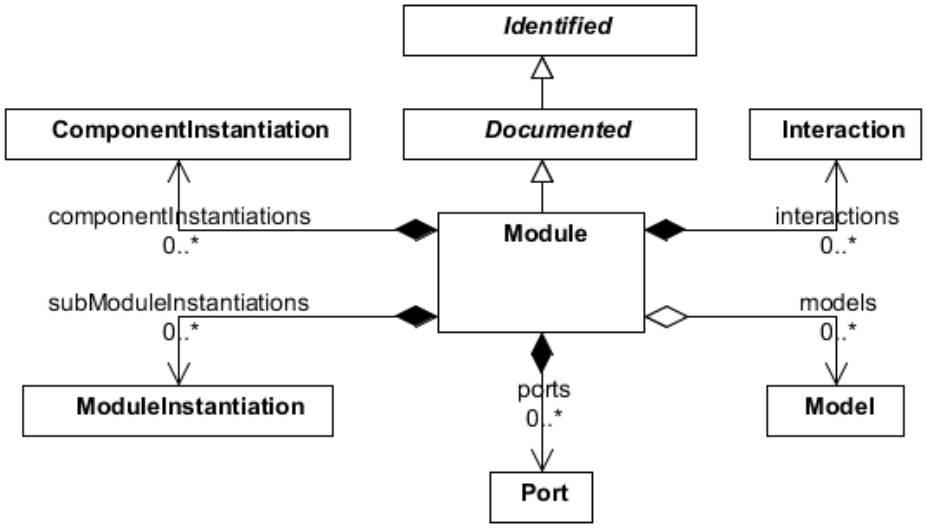

Figure 3.12: UML diagram for the proposed Module class. Note that data objects belonging to the Component Instantiation, Module Instantiation, Interaction, and Port classes are owned by a given module and no other object. Data objects belonging to the Model class, however, may be aggregated by more than one module.

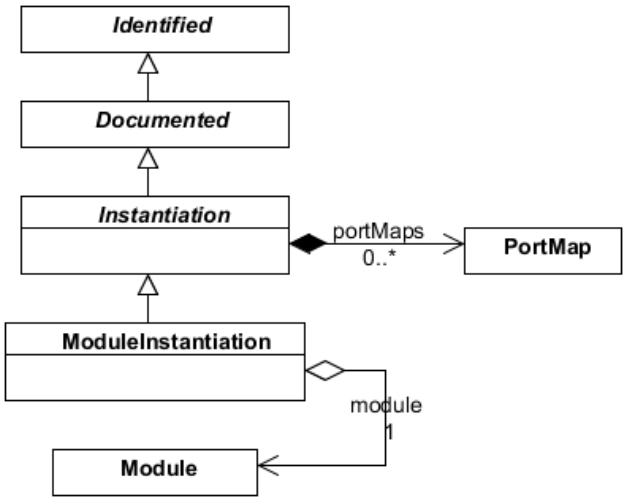

Figure 3.13: UML diagram for the proposed Module Instantiation class. A module instantiation is allowed to aggregate port maps to connect any ports on the module that it instantiates.

ticipation class, each of which specifies the role of its particular component instantiation with a SBO term (see Figure 3.14).

While interactions provide a qualitative description of genetic function, quantitative descriptions are also needed for genetic design. Instead of introducing a new language for the specification of mathematical models of biology, the proposed data model leverages existing standards and refers to them via the Model class. As shown in Figure 3.15, each object that belongs to the Model class is required to refer by means of URIs to a source model and ontology terms that document the source model's language, framework, and role. In this way, there is minimal duplication of standardization efforts and users of SBOL can specify the quantitative function of their modules in a well-developed language of their choice. A module can refer to more than one model since each model can encode different levels of functional detail and play different roles in engineering design.

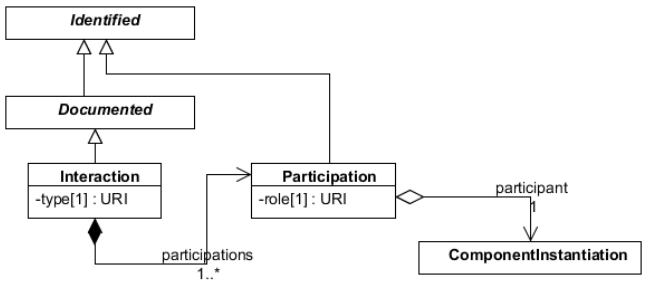

Figure 3.14: UML diagram for the proposed Interaction classes. Objects belonging to the Interaction class aggregate one or more objects of the Participation class, each of which refers to an object of the Component Instantiation class.

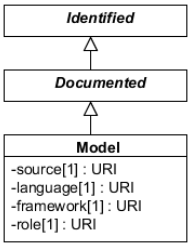

Figure 3.15: UML diagram for the proposed Model class. A SBOL model class object documents and refers to an external mathematical model.

Examples of languages for mathematically modeling for biology include SBML and CellML. *Modeling frameworks* include ODEs, stochastic processes, and Boolean networks. Examples of *modeling roles* include simulation, *verification*, and *synthesis* (building composite models from simpler models). One possible source of terms for modeling frameworks and roles is the *Mathematical Modeling Ontology* (MAMO) [92], though it is currently in the early stages of its development.

Finally, this section presents a UML example of the LacI inverter module of the genetic toggle switch, its regulatory interactions, and its referenced model. As seen in Figure 3.16, the binding of LacI to IPTG is represented using a noncovalent binding interaction that has three participants, including LacI and IPTG participating as reactants and the IPTG-LacI complex participating as a product. The repression of transcription at the pLac promoter is represented using a repression interaction, with LacI serving as the repressor participant and pLac serving as the repressed participant. Lastly, the transcription and translation of TetR are represented in this module using a single genetic production interaction that abstracts away the presence of the intermediate TetR mRNA. If this additional detail becomes necessary, then a new module could be created that instantiates the same components alongside a TetR mRNA component instantiation and includes both transcription and translation interactions. In the current example, the genetic production interaction has three participants: pLac as a modifier, cTetR as a transcribed participant, and TetR as a product. Finally, the LacI inverter module references a Model object that links to an external model. In this example, the model source file is "LacI_Inverter.xml," it is written in the SBML language, it is an ODE model, and it is to be used for simulation.

3.4.4 Composition of Structure and Function

To enable the hierarchical, modular composition of genetic structure and function in SBOL, the proposed data model introduces classes for instantiation and port mapping. An instantiation is a documented reference to a specific component or module that effectively serves as a distinct copy and can be composed with other instantiations into a composite component or module. Currently, the proposed data model includes component instantiations and module instantiations. While a module can only be instantiated by another module, a component can be instantiated by either a module or another component, depending on its intended use. When a component is instantiated by another component, it is effectively referred to as a structural entity for the purpose of physical composition.

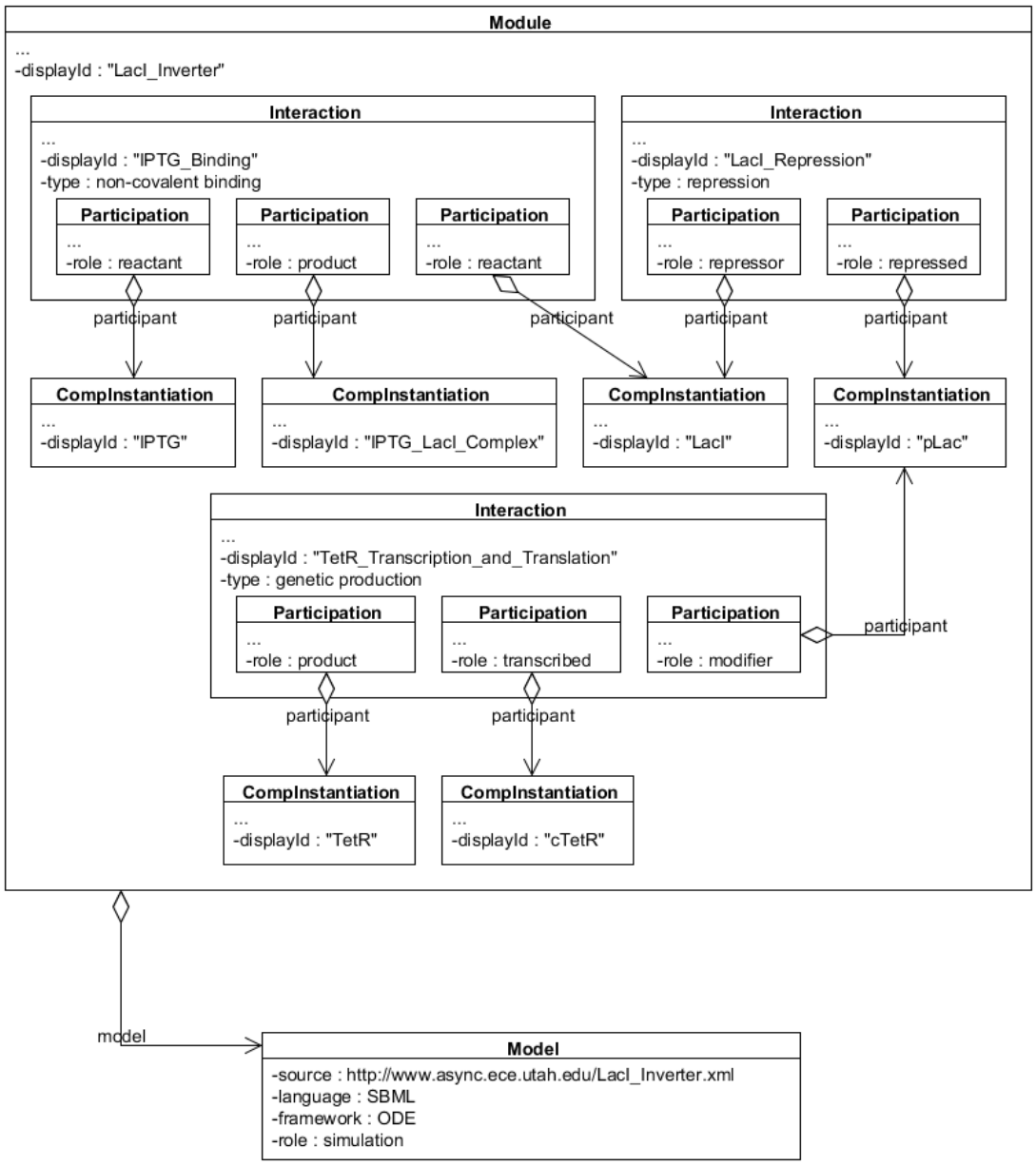

Figure 3.16: UML example displaying the interactions between the component instantiations in the LacI inverter module. In particular, there is an interaction representing the noncovalent binding of IPTG with the LacI protein, an interaction representing repression of the pLac promoter by LacI, and an interaction representing the production of TetR, in which the cTetR CDS is transcribed and the pLac promoter participates as a modifier. This module also references an external mathematical ODE model written in SBML for detailed simulation.

When a component is instantiated by a module, on the other hand, it is referred to as a functional entity for the purpose of playing a role in an interaction.

In turn, ports and port maps enable connections between composite components and modules. As depicted in Figure 3.17, a port refers to a component instantiation, thereby exposing it for port mapping. In addition, a port is allowed to have a URI that references a SBO term indicating whether the port is an *input port* or an *output port*. However, owing to the reversibility of many biochemical reactions and the tight integration of genetic components with their environment, it is important to note that the directionality of a port is only expected to document a designer's intent and does not necessarily reflect biological reality.

A port map, on the other hand, refers to a component instantiation and a port (see Figure 3.18), thereby asserting that its component instantiation corresponds to that referred to by the port. When the components referred to by component instantiations

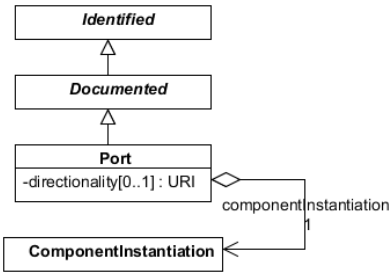

Figure 3.17: UML diagram for the proposed Port class. Note that a port is documented to better describe a designer's intent in exposing a given component instantiation.

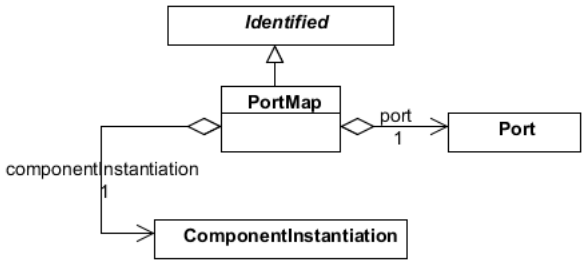

Figure 3.18: UML diagram for the proposed Port Map class. Unlike a port, a port map is identified rather than documented, as it simply represents a connection between a component instantiation and a port.

that are mapped in this way have different identities, their respective data fields are to be interpreted in combination. While this interpretation may be ambiguous in the case of two sequence components with different sequences, it is useful when one of the two sequence components lacks a sequence, in which case a port mapping effectively supplies a sequence to fill in a partial design.

Currently, port mapping serves two specific use cases related to the composition of genetic designs. The first use case is to indicate with greater fidelity how a module describes the function of a composite component, namely by asserting that particular component instantiations within the module correspond to particular component instantiations within the component.

As an example of this use case, one might compose the structure and function of the LacI-repressible gene of the genetic toggle switch. In this example, the LacI-repressible gene and two of its subcomponents, the pLac promoter and cTetR CDS, are to be composed with the LacI inverter module. In order to compose these components with the LacI inverter module and indicate that it describes their behavior, they are instantiated inside the module. In addition, port maps are placed on the instantiation of the LacI-repressible gene to connect between its pLac plus cTetR subcomponent instantiations and the corresponding component instantiations in the module. Doing so makes it clear which subcomponent instantiations in the gene are being described by which component instantiations in the module. In this way, GDA tools for sequence editing and biochemical modeling can guarantee that their users are handling corresponding elements of a given genetic design, while GDA tools for genetic technology mapping can make explicit connections between the structural and functional elements of a design.

This use case (see Figure 3.19) is most relevant when there is reason to believe that two structural instantiations of the same component should function differently based on physical location or other environmental context. For example, a polycistronic gene could contain two copies of a CDS, with one copy experiencing transcriptional repression due to its position downstream of the first copy. To capture such a scenario, there would need to be two component instantiations in a module that participate in different interactions and are separately mapped to the gene's two subcomponent instantiations.

The second use case of port mapping is to connect modules by asserting the correspondence of their component instantiations, effectively unifying these instantiations between modules. For example, the LacI and TetR inverter modules can be composed into a

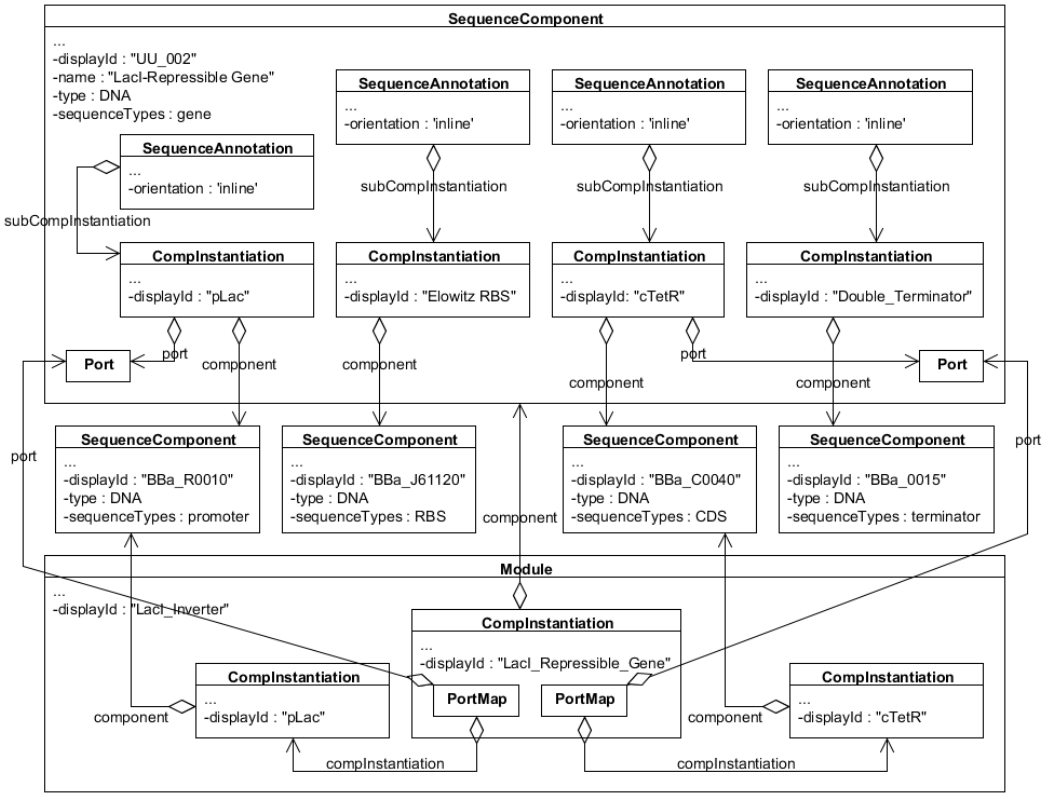

Figure 3.19: UML example of instantiating the LacI-repressible gene within the LacI inverter module. Port maps are used to indicate that the component instantiations of the pLac promoter in the LacI-repressible gene and in the LacI inverter module correspond to each other. Similarly, port maps are used to indicate that the component instantiations of the cTetR CDS within the gene and module correspond. In this figure, "comp" is short for "component."

toggle switch module using instantiation and connected using port mapping, as shown in Figure 3.20. In this example, the output of the LacI inverter is an input of the TetR inverter and vice versa. Also, both inverters accept the instantiation of a small molecule component as input, IPTG in the case of the LacI inverter and aTc in the case of the TetR inverter.

The primary reason for distinguishing between components and modules and port mapping between their instantiations is to promote the reuse of components. When the structural and functional layers of genetic design are kept separate, different researchers

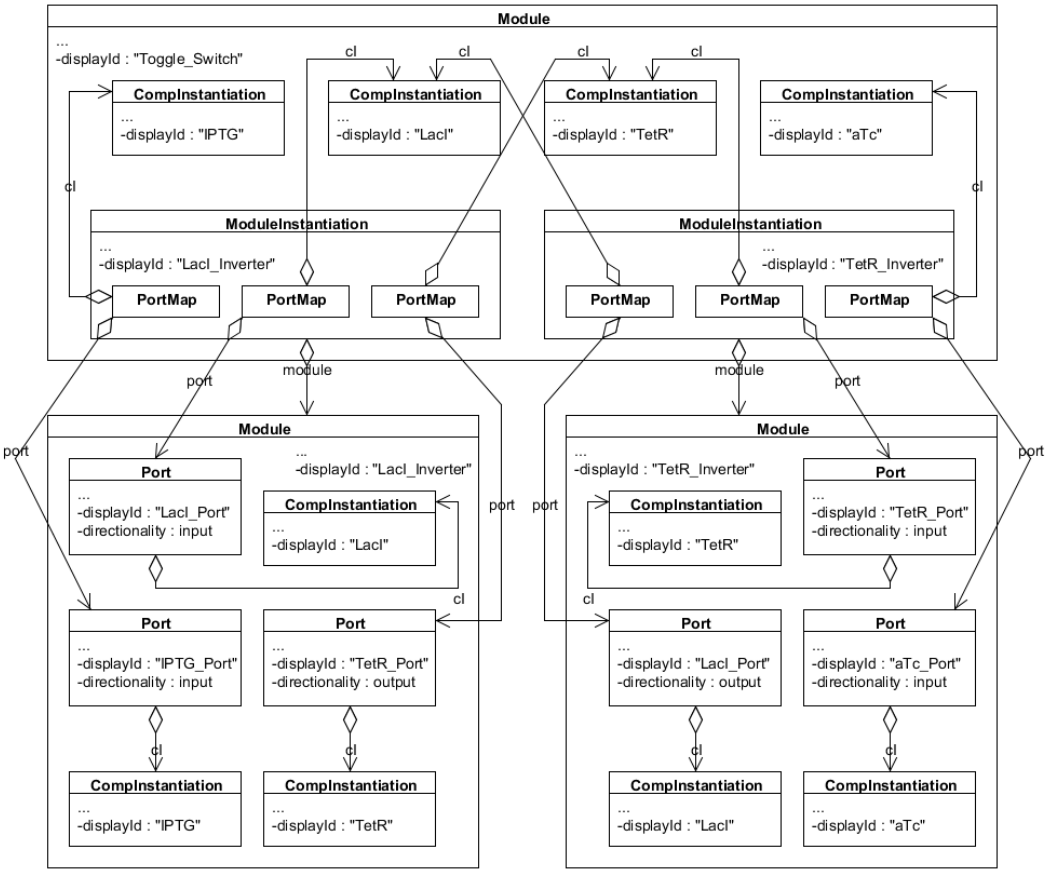

Figure 3.20: UML example of composing the LacI and TetR inverter modules into a toggle switch module. The LacI, IPTG, and TetR component instantiations of the LacI inverter module are mapped to the corresponding component instantiations in the toggle switch module, as are the TetR, aTc, and LacI component instantiations of the TetR inverter module. The end result is a composite toggle switch module that, if flattened into a noncomposite module, would include a single copy of each of these component instantiations and their accompanying interactions (see Figure 3.16). In this figure, "cI" stands for "componentInstantiation."

can use the same component in different modules to document its intended function for different engineering tasks and under different environmental conditions.

Ultimately, the concepts of instantiation and port mapping are not intended to directly represent biological reality. Rather, they are *computational artifacts* that engineers use to organize their designs and enable reasoning over these designs by software. Without these concepts, it is very difficult to introduce the simplifying notions of hierarchy and modularity to genetic design in a manner that is conducive to the application of GDA

software tools and the exchange of data between them. As progress in synthetic biology continues and the scale of genetic design becomes more ambitious, GDA tools that support hierarchical, modular standards will be useful, if not necessary, for managing the complexity of synthetic biological systems.

3.4.5 Examples

As a further demonstration of the utility of the proposed SBOL data model, this section presents examples of designs for real-world synthetic biological systems that it can represent. These include an *expression system* based on *RNA replicons* [93] and a *regulatory cascade* based on *Clustered Regularly Interspaced Short Palindromic Repeats* (CRISPR) [94, 95].

In the first system, three different RNA replicons based on the *Sindbis virus* [96] are transfected into the same host. Consequently, the expression of these replicons is modulated via their relative initial dosages and subsequent competition for the same translation resources. The expression of the payload of each individual replicon is accomplished in two phases. In the first phase, the *nonstructural Proteins* (nsPs) at the 5' end of the replicon are translated by the host to form a *replicase*. In the second phase, the replicase transcribes copies of the replicon, including shortened copies that only contain the payload and are produced when the replicase binds to the *Subgenomic Promoter* (SGP) at the end of the nsP block. Lastly, the third phase concerns the translation of the shortened copies, thereby expressing the payload (in this case, a fluorescent protein) in the place of structural proteins that would form the shell or *capsid* of the virus.

As shown in Figure 3.21, the basic genetic structure and function of the replicon expression system can be represented using the proposed data model. In this design, an RNA component with an unspecified payload sequence serves as a structural template for the three RNA replicons. In turn, this RNA component is instantiated within a module that serves as a functional template for the replicons and asserts the key interaction of the host translation resources with the payload CDS. Finally, the mixed replicon expression system as a whole is composed by instantiating three submodules, each of which maps its fluorescent protein payload and CDS to the appropriate ports on an instantiation of the generic replicon expression module. This effectively documents that the mixed replicon expression module contains three separate copies of the generic replicon expression module, each with a different fluorescent protein payload. While the initial dosages for each replicon are outside the scope of the proposed data model, they can still be captured as custom

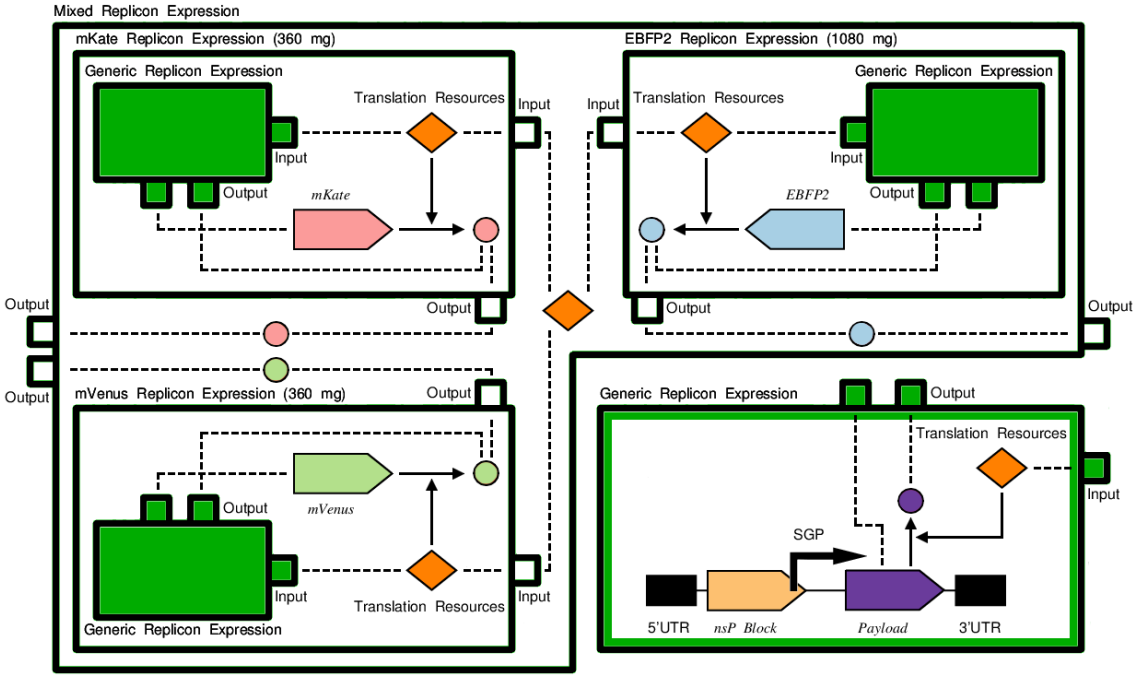

Figure 3.21: A mixed replicon expression module that instantiates three different replicon expression submodules, which in turn instantiate copies of a generic replicon expression module. Port mapping is used to customize each copy of the generic expression module so that it has a different payload CDS and produces a different fluorescent protein. Port mapping is also used to indicate that these submodule instantiations share translation resources.

annotations on the mixed expression module or within a mathematical model that is referenced by the module via the Model class.

The second example is a CRISPR-based regulatory cascade (see Figure 3.22), in which transcriptional repression is accomplished using *catalytically inactive Cas9 protein* (Cas9m). Like many other TFs, Cas9m sterically blocks transcription initiation, but unlike other TFs it is targeted to specific promoters via *guide RNA* (gRNA) molecules that allow for easier generation of orthogonal regulators. In the present example, there are two promoters that are serially repressed in this manner but are targeted via different gRNA molecules. More specifically, promoter *CRP-a* is targeted by *gRNA-a* and initiates transcription of *gRNA-b*, which is coexpressed with the fluroescent protein *mKate* as *intronic gRNA* (igRNA). In turn, promoter *CRP-b* is targeted by *gRNA-b* and initiates the transcription of the reporter protein *EYFP*. Since gRNA-a is constitutively transcribed

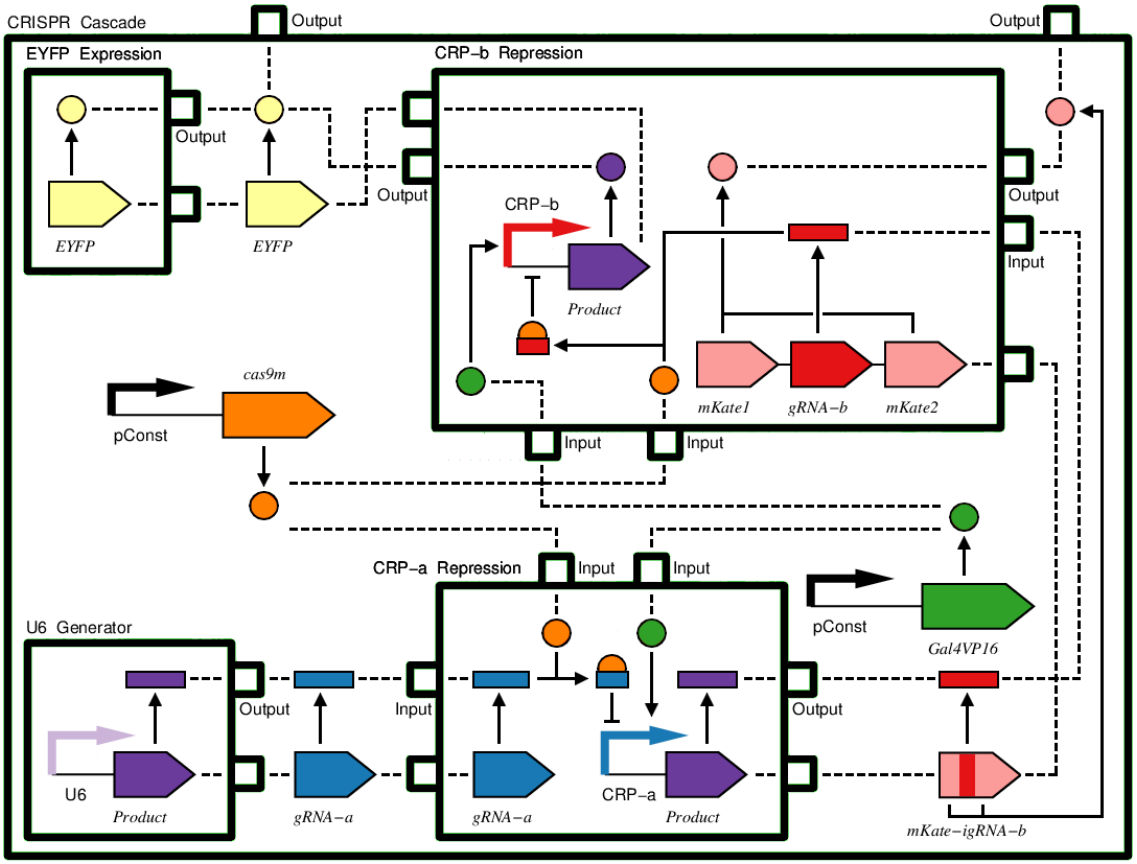

Figure 3.22: A CRISPR regulatory cascade module that instantiates four submodules and several components. In this example, port mapping is used to specify the precise downstream components that are the outputs of each module.

in this system, the expression of gRNA-b and mKate are repressed and EYFP is produced in relatively larger quantities.

Figure 3.22 demonstrates one possible way in which the CRISPR cascade can be specified using the proposed data model. In this design, there are four submodule instantiations, three of which encompass a DNA component with an unspecified CDS and an RNA or protein product. These submodules are connected in series via port mapping so that the unspecified CDS and product of one submodule correspond to the specified CDS and product of the next submodule and the overall parent CRISPR cascade module. The latter module also instantiates DNA components that produce Cas9m and the activator TF *Gal4VP16*, which are then mapped as inputs to the two modules that represent CRISPR-based repression at CRP-a and CRP-b. In this way, the CRISPR cascade module

serves a common source of regulators for any and all CRISPR-based modules that it instantiates.

3.4.6 Summary

As summarized in the UML class diagram shown in Figure 3.23, the proposed data model expands the total number of classes in SBOL from four to seventeen (four of these classes, the Identified, Documented, Collection, and Generic Component classes, are omitted from the figure for clarity). Central to this data model are the Component and Module classes, which are the basic exchangeable units for composing descriptions of genetic structure and function. A module composes components and other modules by means of the Component Instantiation and Module Instantiation classes and describes their function by aggregating objects belonging to the Interaction and Model classes. A component that belongs to the Sequence Component class refers to an object of the Sequence class and composes its subcomponent instantiations along its sequence via objects of the Sequence Annotation class. Once components and modules have been composed using the various Instantiation classes, their component instantiations can be connected using the Port and Port Map classes.

If adopted by the community, this set of extensions to the SBOL Version 1.1 data model should provide a means of expressing and composing genetic designs exhibiting a wide range of structure and function (see Section 3.4.5). This data model represents a conservative extension of the current model, striking a balance between expressiveness and minimization of complexity. To the extent possible, this proposal avoids either making representational commitments where there is not yet scientific consensus or duplicating other modeling and standardization efforts.

In order to test the utility of this new data model, a new version of the Java library, libSBOLj, has been implemented and is being utilized to construct the above described use cases and other genetic designs from the literature. In conjunction with further discussions in the community, this experimentation will hopefully allow for the resolution of any remaining details so that a formal *BioBricks Foundation Request for Comments* (BBF RFC) [97] specification can be written and ratified by the SBOL Developers Group. Once ratified, the specification becomes official when at least two software tools have implemented the standard and demonstrated the exchange of data.

Even once this proposal is accepted, there are still important aspects of genetic designs not yet captured by SBOL. In particular, the proposed extensions to SBOL do

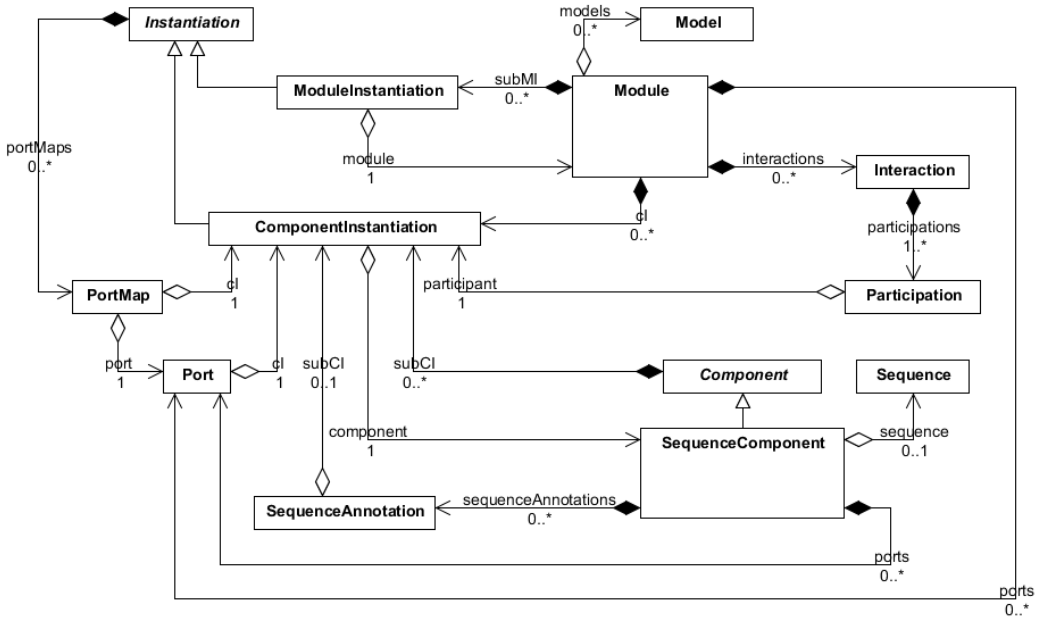

Figure 3.23: UML diagram that summarizes the proposed data model. In this figure, "cI" stands for "componentInstantiation," "subCI" stands for "subComponentInstantiation," and "subMI" stands for "subModuleInstantiation."

not explicitly address the complex relationship between environmental context and its influence on the intended function of a design. Such specifications can become quite important when composing modules, as not all of them may function correctly when deployed in the same environment or host organism, nor may they be amenable to the same experimental techniques. Furthermore, the proposed data model does not capture protocols for experiments or physical assembly of designs. More research is necessary to identify the types of data related to context, assembly, and experiments that can be incorporated into SBOL and reasoned over by software. With these additions, SBOL will be able to better facilitate the specification of genetic designs and their deployment and testing in the lab.

CHAPTER 4

MODEL ANNOTATION AND GENERATION

In this chapter, Section 4.1 describes a methodology for annotating SBML [14] models with SBOL [44]. This section also describes the most common use cases for model annotation, particularly with regards to model generation [49], genetic technology mapping [48], and sequence generation. Section 4.2, on the other hand, describes a methodology for generating SBOL-annotated SBML models from one or more SBOL modules. These SBOL-annotated SBML models can then serve as inputs to genetic technology mapping (see Chapter 5) and other analysis techniques implemented in iBioSim, such as stochastic chemical kinetic simulation and model checking [84]. Finally, Section 4.3 briefly summarizes the usefulness of model annotation and generation and discusses their future application.

4.1 Model Annotation

While Chapter 3 partly describes linking SBOL modules to SBML models via the SBOL Model class, this section describes how to link SBML models back to SBOL modules using annotations on SBML models. As recommended by the developers of SBML, these *SBML-to-SBOL annotations* are written in RDF/XML and adhere to the format outlined in Figure 4.1, which can be seen as an extension of the format originally used by the sequence generation tool MoSeC [36]. Each annotation refers to a SBML element, such as a species or reaction, as its subject and can refer to the following as its objects: up to one unordered set of SBOL elements per SBOL class, up to one ordered list of DNA components, and up to one *strand sign*.

Depending on their objects, SBML-to-SBOL annotations can serve up to three use cases for the composition of genetic designs. The first two use cases pertain to when a SBML-to-SBOL annotation has an unordered set of SBOL elements as its object. When the annotation's SBOL elements belong to the functional layer of design, such as a set

```
<SBML_ELEMENT + + + metaid="SBML_META_ID" + + + >
  <annotation>
    <ModelToSBOL xmlns="http://sbolstandard.org/modeltosbol/1.0#">
      <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:mts="http://sbolstandard.org/modeltosbol/1.0#">
        <rdf:Description rdf:about="#SBML_META_ID">
          <mts:SBOL_ELEMENTS>
            <rdf:Bag>
              <rdf:li rdf:resource="SBOL_ELEMENT_URI"/>
              . . .
            </rdf:Bag>
          </mts:SBOL_ELEMENTS>
          <mts:DNAComponents>
            <rdf:Seq>
              <rdf:li rdf:resource="DNA_COMPONENT_URI"/>
              . . .
            </rdf:Seq>
          </mts:DNAComponents>
          <mts:Strand>STRAND_SIGN</mts:Strand>
        </rdf:Description>
      </rdf:RDF>
    </ModelToSBOL>
  </annotation>
</SBML_ELEMENT>
```

Figure 4.1: Format for SBML-to-SBOL annotation written in RDF/XML. The first line of XML is that of the SBML element, which contains a meta-ID among other information as indicated by +. Next is the SBML-to-SBOL annotation, with its first two lines containing namespaces that distinguish it from other types of annotations and indicate how it should be processed. The actual content of the annotation is highlighted in red, green, and blue. The red line is the subject of the annotation, which is a SBML element identified using its meta-ID. The green lines are the predicates of the annotation, which indicate that the subject is associated with a set of SBOL elements, a list of SBOL DNA components, or a strand sign. The blue lines are the objects of the annotation, which can be a set of URIs identifying one or more SBOL elements, a list of URIs identifying one or more DNA components, or a character ('+' or '−') representing a strand sign.

of component instantiations or interactions, the purpose of the annotation is to indicate the provenance or origin of the SBML elements produced during model generation. In particular, the species and reactions of a SBML model that is generated from a SBOL module can be annotated with the component instantiations and interactions from which they are derived. In this way, a record is kept of which elements in a SBML model correspond to which elements in the source SBOL module.

When an annotation's SBOL elements belong to the structural layer of design, such as a set of sequence components or generic components, the purpose of the annotation is to avoid cross-talk during genetic technology mapping. For example, species within a SBML model can be annotated with components to effectively assign them molecular identities. As explained in Chapter 5, these molecular identities are compared when composing subdesigns into a solution to the genetic technology mapping problem. When the molecular identity of an output for one subdesign does not match the appropriate input for the next subdesign, all solutions that would contain this connection are disqualified from future consideration. Consequently, the average runtime of genetic technology mapping can be significantly improved without cutting out valid solutions.

The third use case pertains to when a SBML-to-SBOL annotation has a list of DNA components and a strand sign as its objects. In this case, the purpose of the annotation is to directly couple descriptions of genetic function and structure for sequence generation. Sequence generation uses the organization of a model to infer the structure of one or more genetic constructs from the DNA components that annotate the model. For the purpose of sequence generation (see Chapter 6), it does not matter which individual element of a model is annotated by which DNA component. Rather, it only matters that the ordering of the DNA components, as inferred by the cause-and-effect relationships between the elements of the model, matches a pattern corresponding to a valid genetic construct. As described in Section 4.2, the model generation methodology used in this dissertation has been designed to produce SBOL-annotated SBML models that adhere to this requirement for sequence generation.

Each list groups one or more DNA components that are physically adjacent on a strand of DNA. These DNA components are grouped nonhierarchically, that is, without having to include them as subcomponents of a single composite component. This capability is useful for ordering DNA components that may not have their function explicitly modeled but must be present for a design to function correctly. For example, even if the function of a spacer in a gene is not modeled, the spacer may still be necessary to prevent undesirable interaction of adjacent DNA components. Hence, a SBML element may be annotated with a list of DNA components interspersed by spacer components. While it is true that the SBML element could instead be annotated with a single composite component that is composed from this list of DNA components, some designers may not want to group the listed components into a superfluous functional unit.

A strand sign, on the other hand, indicates whether the listed DNA components should be composed on the positive or negative strand of the DNA sequence for a composite DNA component. If a strand sign is not present in a given annotation, then it is assumed that any DNA components listed by the annotation should be composed on the positive strand. Alternatively, if a strand sign is present and there is no list of DNA components, then the strand sign is assumed to apply to any DNA components annotating an external SBML element that is referenced by the strand sign's SBML element. For example, if a model contains a submodel element that is annotated with only a strand sign, then the strand sign applies to any DNA components annotating the external model that is referenced by the submodel element. This rule promotes the reuse of a given annotated model by allowing its parent composite models to orient its annotating DNA components differently without requiring the creation of separate annotated models for each desired orientation.

As an example of all three use cases for SBML-to-SBOL annotations, this section concludes with a description of annotating several SBML models. These models are derived from the SBOL modules for the LacI inverter and genetic toggle switch [50] presented in Chapter 3 (see Figure 3.16 and Figure 3.20) and are shown in Figure 4.2.

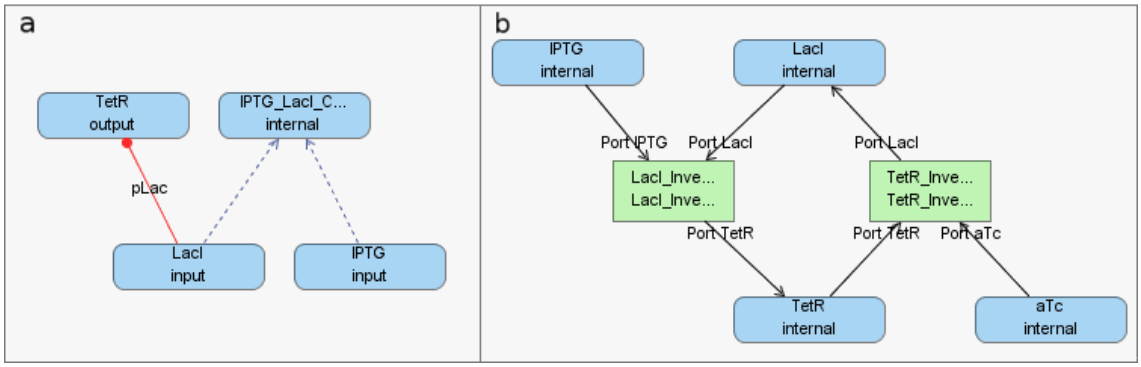

Figure 4.2: An iBioSim representation of the SBML models for the LacI inverter and the genetic toggle switch. Blue ellipses are species and green rectangles are submodels. (a) The red arrow in the LacI inverter model signifies repression of the promoter pLac and production of TetR by the species LacI, while the dashed arrows signify noncovalent binding to form a complex between IPTG and LacI. (b) Lastly, the labeled black arrows in the toggle switch model are replacements that specify that its LacI, TetR, IPTG, and aTc species are intended to replace or be replaced by species in the LacI and TetR inverter submodels.

To indicate the provenance of the LacI inverter model, it can be directly annotated with the LacI inverter module from which it is derived, while its pLac, LacI, IPTG, IPTG-LacI, and TetR species can be annotated with the similarly named component instantiations from the LacI inverter module. In addition, the model's genetic production reaction can be annotated with the module's genetic production and repression interactions, its noncovalent binding reaction can be annotated with the module's noncovalent binding interaction, and its ports that refer to its species can be annotated with the module ports that refer to the corresponding component instantiations. Next, for the purpose of directly documenting molecular identities, each species in the LacI inverter model can be annotated with the DNA, RNA, protein, or complex component that is instantiated by the component instantiation annotating the species. Finally, to facilitate the generation of a DNA sequence from the LacI inverter model, its elements can be annotated with DNA components instantiated by the LacI-repressible gene (see Figure 3.11) and positive strand signs. In particular, its promoter species pLac can be annotated with the LacI-regulated promoter BBa_I14032, while its TetR species (the product of its genetic production reaction) can be annotated with a gene composed of the RBS_B0034 and the TetR CDS BBa_C0040, followed by the terminator BBa_B0015.

In the case of the toggle switch model and its submodels, their provenance can similarly be established by annotating them with the toggle switch module and LacI/TetR inverter module instantiations from which they are derived. Furthermore, the model's species and replacements between its species can be annotated with the equivalent component instantiations and port maps from the toggle switch module. In this sense, a replacement is equivalent to a port map when the parent species, submodel, and port referred to by the replacement correspond to the component instantiation, module instantiation, and port referred to by the port map. Finally, documentation of the molecular identities for species in the toggle switch model is handled as before, while sequence generation for the toggle switch is facilitated by annotating the LacI and TetR inverter submodels with positive and negative strand signs, respectively. In this way, the DNA components that make up the sequence for the toggle switch are obtained from the external LacI and TetR inverter models and oriented in accordance with the strand signs annotating the submodels.

4.2 Model Generation

In order to facilitate interdisciplinary collaboration and tighter connections between qualitative and quantitative descriptions of genetic function, model generation tools are needed to help automate the process of creating quantitative models based on qualitative designs. With the existence of such tools, synthetic biologists who lack expertise in applied mathematics can have a larger audience for their qualitative genetic designs, namely engineers who have expertise in mathematical analysis but who lack experience in modeling biology. Furthermore, the mathematical models generated with these tools are by design based on formal descriptions of genetic function. This connection can itself be formalized via model annotation so that the metadata associated with the process of model generation is not lost and the mappings to different models from the same genetic design can be more readily compared. In light of these considerations, this section presents a methodology for generating SBOL-annotated SBML models from SBOL designs that conform to the data model for SBOL Version 2.0 [47] proposed in Chapter 3.

The procedure for automatically generating SBML from SBOL and annotating the former with the latter follows the steps outlined below.

1. For each SBOL module in a SBOL document:
   (a) Add a SBML model to a new SBML document.
   (b) Annotate the SBML model with the SBOL module.
   (c) Follow steps 2 through 6.
2. For each protein component instantiation, small molecule component instantiation, and complex component instantiation i:
   (a) Add a species s to the list of species for the SBML model.
   (b) Annotate s with i and the component instantiated by i.
   (c) If i is the sole, degraded participant in a single degradation interaction n:
      i. Add a degradation reaction r_s to the list of reactions for the SBML model.
      ii. Annotate r_s with n.
      iii. Add a species reference for s to the list of reactants for r_s.
      iv. Add a mass-action kinetic law of the form below to r_s.

$$\text{rate}(r_s) = k_d s \tag{4.2}$$

3. For each promoter DNA component instantiation i:

   (a) Add a promoter species p to the list of species for the SBML model.

   (b) Annotate p with i and the component instantiated by i.

   (c) Add genetic production reaction r_p to the list of reactions for SBML model.

   (d) Add a promoter species reference for p to the list of modifiers for r_p.

   (e) For each genetic production interaction n in which i participates as a modifier, a protein component instantiation j participates as a product, and a gene DNA component instantiation k is a transcribed participant:

      i. Add n to the set of interactions annotating r_p.

      ii. Add a species reference for the species s that corresponds with j to the list of products for r_p.

      iii. Annotate s with the component instantiated by k.

   (f) For each activation or repression interaction n' in which i is a repressed or activated participant and a TF protein component instantiation x participates as an activator or repressor:

      i. Add an activator or repressor species reference for the species y that corresponds with x to the list of modifiers for r_p.

      ii. Add y to the set of activators $\mathrm{Act}(p)$ or set of repressors $\mathrm{Rep}(p)$.

      iii. Add n' to the set of interactions annotating r_p.

   (g) Add a Hill function kinetic law of the form below to r_p.

$$\mathrm{rate}(r_p) = \begin{cases} \dfrac{n_p k_o n_g K_o n_r}{1 + K_o n_r + \sum\limits_{s_r \in \mathrm{Rep}(p)} (K_r s_r)^{n_c}} & |\mathrm{Act}(p)| = 0 \\ \dfrac{n_p k_b n_g K_o n_r + n_p k_a n_g K_{oa} n_r \sum\limits_{s_a \in \mathrm{Act}(p)} (K_a s_a)^{n_c}}{1 + K_o n_r + \sum\limits_{s_r \in \mathrm{Rep}(p)} (K_r s_r)^{n_c} + K_{oa} n_r \sum\limits_{s_a \in \mathrm{Act}(p)} (K_a s_a)^{n_c}} & \text{otherwise} \end{cases} \tag{4.3}$$

4. For each noncovalent binding interaction n in which a protein component instantiation i participates as a product and a set of protein or small molecule component instantiations $\mathrm{React}(i)$ participate as reactants:

   (a) Add a reversible noncovalent binding reaction r_s to the list of reactions for the SBML model, where s is the species that corresponds with i.

   (b) Annotate r_s with n.

(c) Add a species reference for s to the list of products for r_s.

(d) Add species references for the set of species React(s) that corresponds with React(i) to the list of reactants for r_s.

(e) Add a mass-action kinetic law of the form below to r_s.

$$\text{rate}(r_s) = k_{c_f} K_c^{|\text{React}(s)|-2} \prod_{s' \in \text{React}(s)} s' - k_{c_r} s \tag{4.4}$$

5. For each SBOL port t that refers to a component instantiation i:

(a) Add a SBML port t' that refers to the species that corresponds with i to the list of ports for the SBML model.

(b) Annotate t' with t.

6. For each module instantiation i:

(a) Add a submodel m to the SBML model.

(b) Annotate m with i.

(c) Add an external model definition that refers to the SBML model corresponding with the module instantiated by i to the list of external model definitions for the SBML model.

(d) For each port map a that is referred to by i and refers to a SBOL port t and a component instantiation j:

i. Add a replacement e that refers to the SBML port corresponding with t to the list of replacements for the species corresponding with j.

ii. Annotate e with a.

Because the proposed SBOL data model is not capable of encoding quantitative parameters, the SBML kinetic laws generated by this methodology are populated with default parameters that must be customized using iBioSim or another SBML-compatible modeling tool. Table 4.1 lists these default parameters and their current values. In the future, the SBOL data model can be extended with the capacity to store data on quantitative parameters and measurements, thereby providing a firmer foundation for GDA tools to generate different mathematical models for different design tasks that nevertheless conform to the same basic data set.

Table 4.1: Default parameters for generated kinetic laws

Parameter	**Symbol**	**Value**	**Units**
Rate of degradation	k_d	0.0075	$\frac{1}{sec}$
Stoichiometry of production	n_p	10	$unitless$
Open complex production rate	k_o	0.05	$\frac{1}{sec}$
Basal production rate	k_b	0.0001	$\frac{1}{sec}$
Activated production rate	k_a	0.25	$\frac{1}{sec}$
Promoter count	n_g	2	$molecule$
RNA polymerase binding equilibrium	K_o	0.033	$\frac{1}{molecule}$
Activated RNA pol. binding equilibrium	K_{oa}	1	$\frac{1}{molecule}$
RNA polymerase count	n_r	30	$molecule$
Repression binding equilibrium	K_r	0.5	$\frac{1}{molecule}$
Activation binding equilibrium	K_a	0.0033	$\frac{1}{molecule}$
Stoichiometry of binding	n_c	2	$unitless$
Forward non-covalent binding rate	k_{c_f}	0.05	$\frac{1}{molecule*sec}$
Non-covalent binding equilibrium	K_c	0.05	$\frac{1}{molecule}$
Reverse non-covalent binding rate	k_{c_r}	1	$\frac{1}{sec}$

As for the kinetic laws themselves, their derivation is partly based on model abstraction techniques, such as *operator site reduction* [72] and *quasisteady-state approximation* [98, 99]. While a more detailed description of this type of derivation can be found in [100], a short summary is included here. Briefly, this derivation assumes that most reversible noncovalent binding reactions occur much more rapidly than the reactions for genetic production and degradation, such that the species produced by these reactions are assumed to be at or near their equilibrium levels at all times. While the model generation methodology presented in this chapter does not explicitly check whether the conditions for model abstraction are satisfied, iBioSim implements a range of techniques for automated model reduction [83] and expansion that may be applied to the generated model to either increase or decrease its level of abstraction.

As a consequence of the above assumptions, the generated model does not include

reactions for TFs binding to DNA and the formation of intermediate complexes between TFs and/or small molecules. In addition, the model does not include the species produced by these reactions and these species are replaced in the kinetic laws for other reactions with algebraic expressions over their constitutive species. This can be seen in the non-covalent binding kinetic law of Equation 4.4, in which all intermediate complexes that could appear in the kinetic law have been replaced with a multiplication over the TFs and small molecules that make up these complexes and their equilibrium binding constants. The replacement of complexes between TFs and DNA, on the other hand, is accompanied by an application of the law of mass conservation to each promoter species. As a result, the kinetic laws for each genetic production reaction are written as fractions in which each term of the numerator and denominator accounts for a different species that may bind the reaction's promoter. As seen in Equation 4.3, when the amount of a repressor species increases, the denominator increases and the rate of genetic production from the promoter is minimized. When the amount of an activator species increases, however, both the numerator and denominator increase and the rate of genetic production from the promoter is maximized.

This section concludes with an example of generating SBML models for the LacI inverter and genetic toggle switch from their respective SBOL modules. As shown in the UML diagram in Figure 4.3, the qualitative function of the LacI inverter can be represented using a SBOL module that instantiates the inverter's DNA, protein, small molecule, and complex components and asserts the regulatory, gene expression, and noncovalent binding interactions that occur between the instantiated components. As presented before in Chapter 3, each interaction in the LacI inverter module has a type derived from the SBO [101], a controlled vocabulary for systems biology terms, and refers to participating component instantiations indirectly via participations. Each participation has a role that is also derived from a SBO term and indicates what a component instantiation does or has done to it in an interaction. For instance, in the repression interaction that takes place between the LacI and pLac component instantiations, LacI acts as a repressor while pLac is repressed. The LacI inverter module in this example builds upon the previous representation of the LacI inverter by adding additional interactions to capture degradation of the TetR, LacI, IPTG, and IPTG-LacI complex component instantiations. Furthermore, the UML diagram for the module has been color coded to indicate which SBOL elements inform which steps in the model generation process.

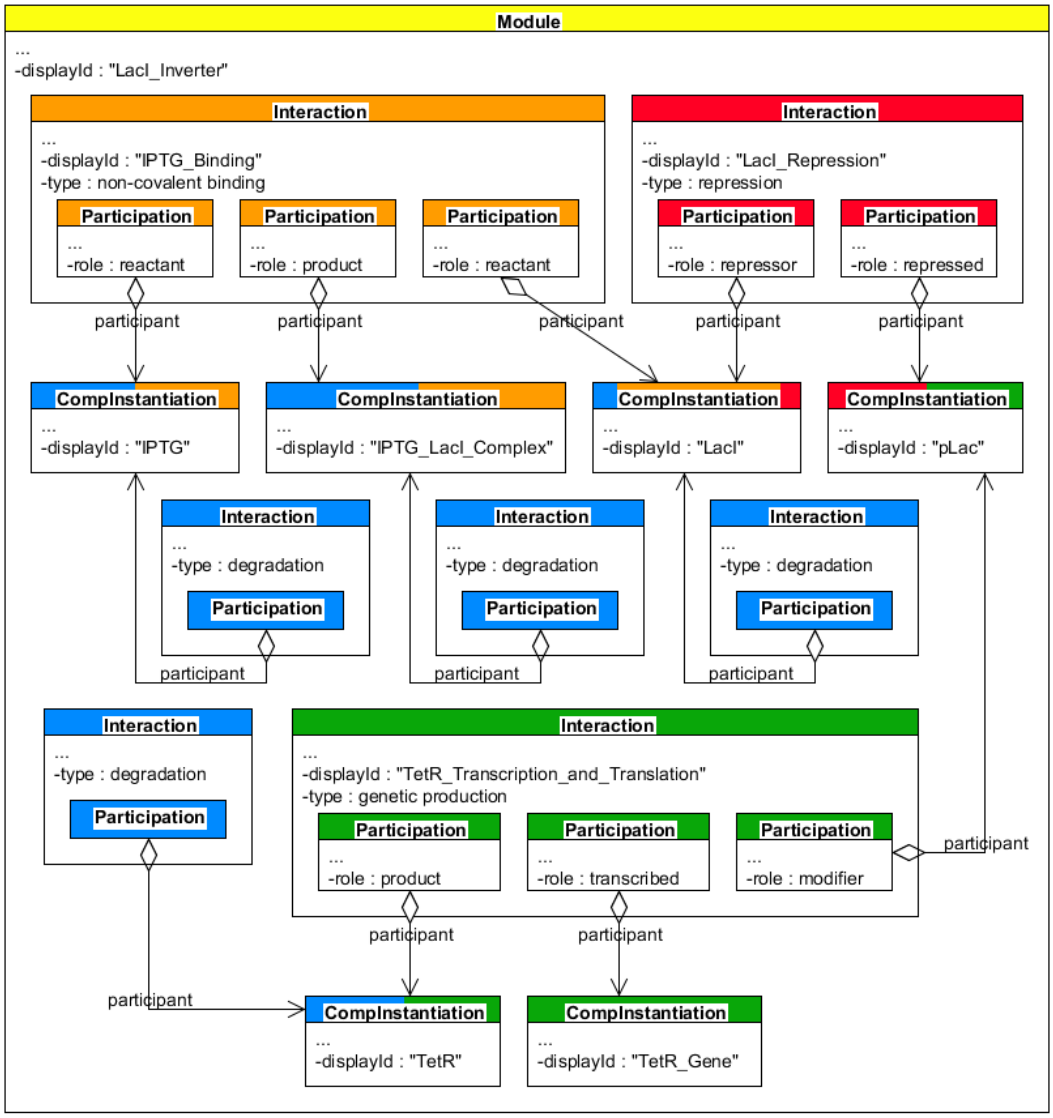

Figure 4.3: UML diagram of the LacI inverter under the proposed data model for the next version of SBOL. Yellow objects are referred to during step 1 of model generation, blue during step 2, green during step 3, red during step 3(g), and orange during step 4.

Similarly, the UML diagram in Figure 4.4 demonstrates how the genetic toggle switch can be composed through a module that instantiates the modules for the LacI and TetR inverters and the components shared by these modules. The inverter modules are then connected through port maps that assert the correspondence between component instantiations in the toggle switch module and port-exposed component instantiations in the inverter modules. In the present example, the TetR inverter module is omitted in order

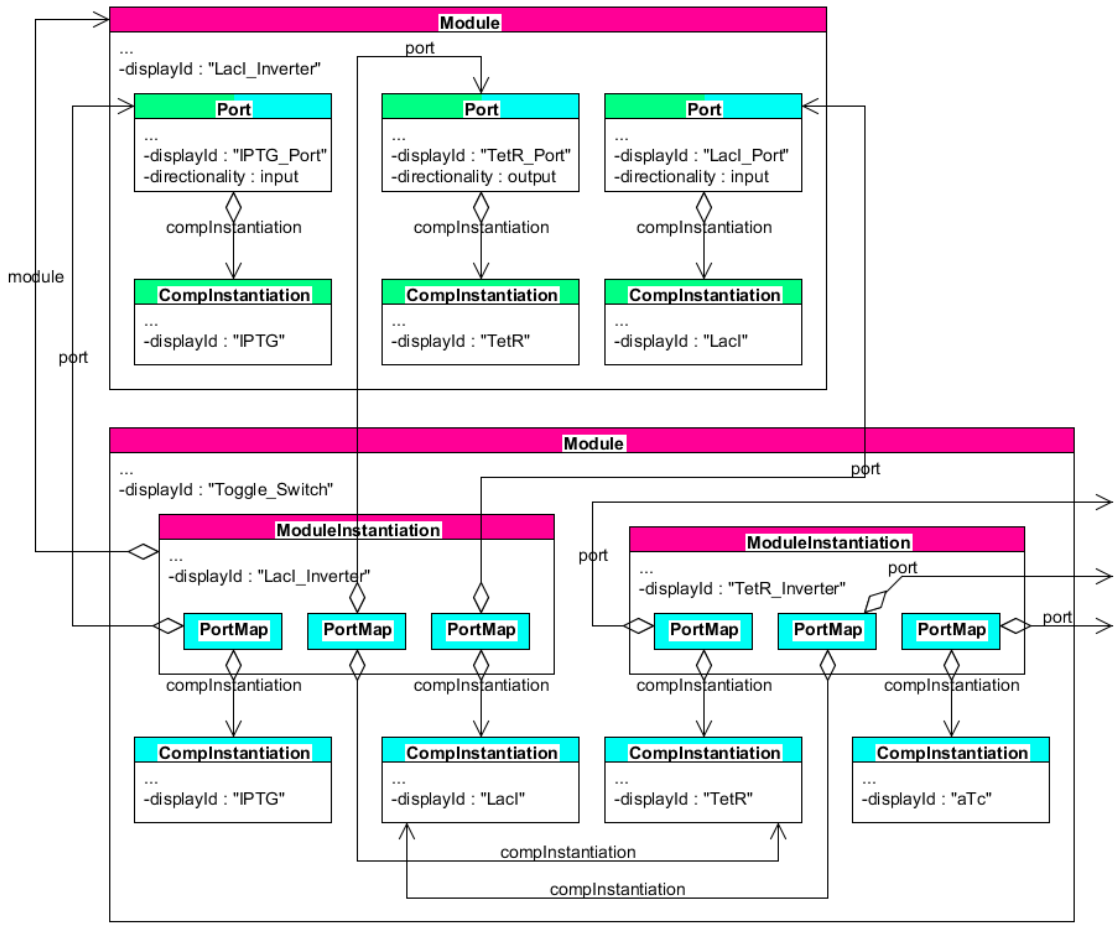

Figure 4.4: UML diagram of the LacI inverter and genetic toggle switch under the proposed data model for the next version of SBOL. Emerald objects are referred to during step 5 of model generation, magenta during step 6, and turquoise during step 6(d).

to focus on the composition of the LacI inverter module with the toggle switch module. Again, the SBOL elements in the UML diagram have been colored to tie them to specific steps during the model generation process, as well as facilitate comparison between the SBOL modules and SBML models that serve as inputs and outputs to this process.

Figure 4.5 is a UML diagram that represents the SBML model generated for the LacI inverter. Central to this model are reactions for degradation, genetic production, and complex formation, which have kinetic laws of the forms found in Equations 4.2–4.4. In addition, each reaction has one or more lists of species references that identify which species are its reactants, products, and modifiers, and each reaction and species reference

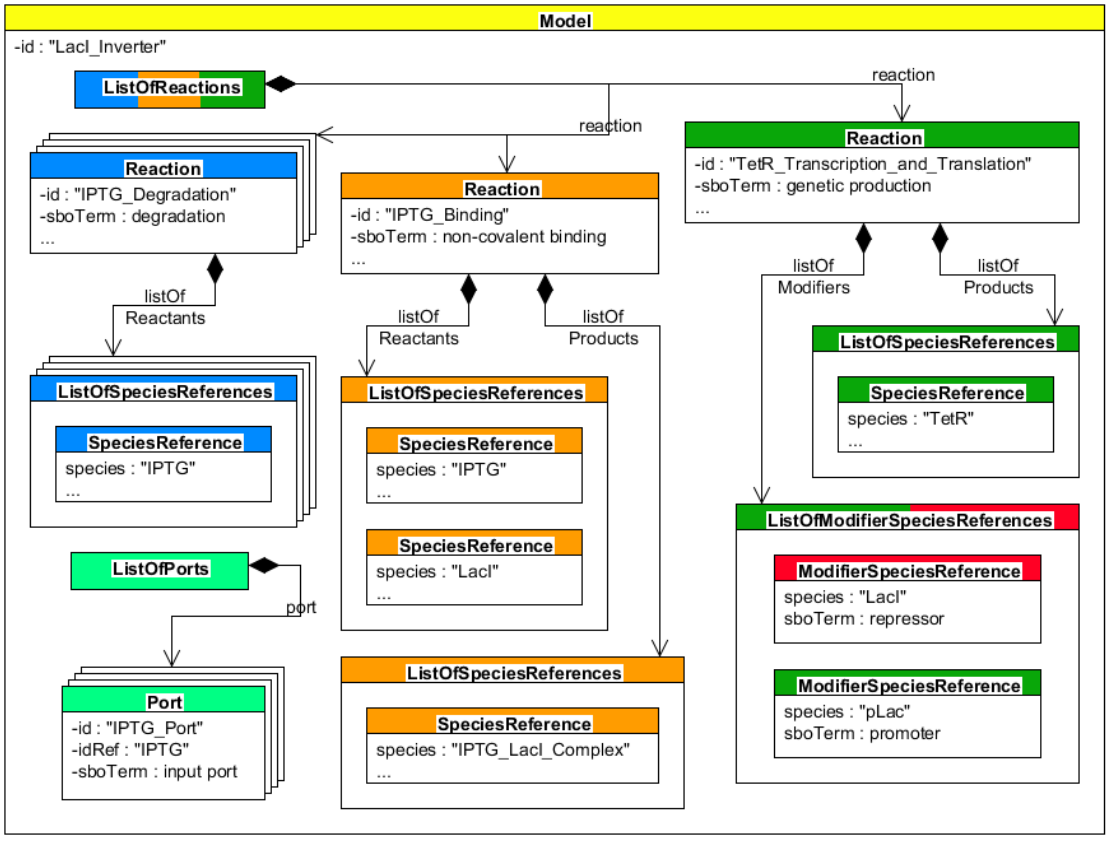

Figure 4.5: UML diagram of the LacI inverter under Level 3, Version 1 of the SBML data model. Yellow objects are created in step 1 of model generation, blue in step 2, green during step 3, red in step 3(g), orange in step 4, and emerald in step 5.

has a SBO term to more concretely specify its type or role in a genetic context. For example, the green reaction is labeled with the SBO term "genetic production" and refers to the species LacI and pLac as its modifiers, while these species references are labeled with the SBO terms "repressor" and "promoter." As expected, the SBO labeling of the green/red SBML reaction and species references in Figure 4.5 is consistent with the types and roles of the green/red SBOL interactions and participations in Figure 4.3, since the creation of the former is based on the structure of the latter in step 3 of model generation.

Similar comparisons can be made between the input SBOL modules and Figure 4.6, which is a UML diagram that represents the SBML model generated for the genetic toggle switch. In this model, each species has a turquoise list of elements that it replaces in the external models for the LacI inverter and/or TetR inverter. As seen in Figure 4.4, these

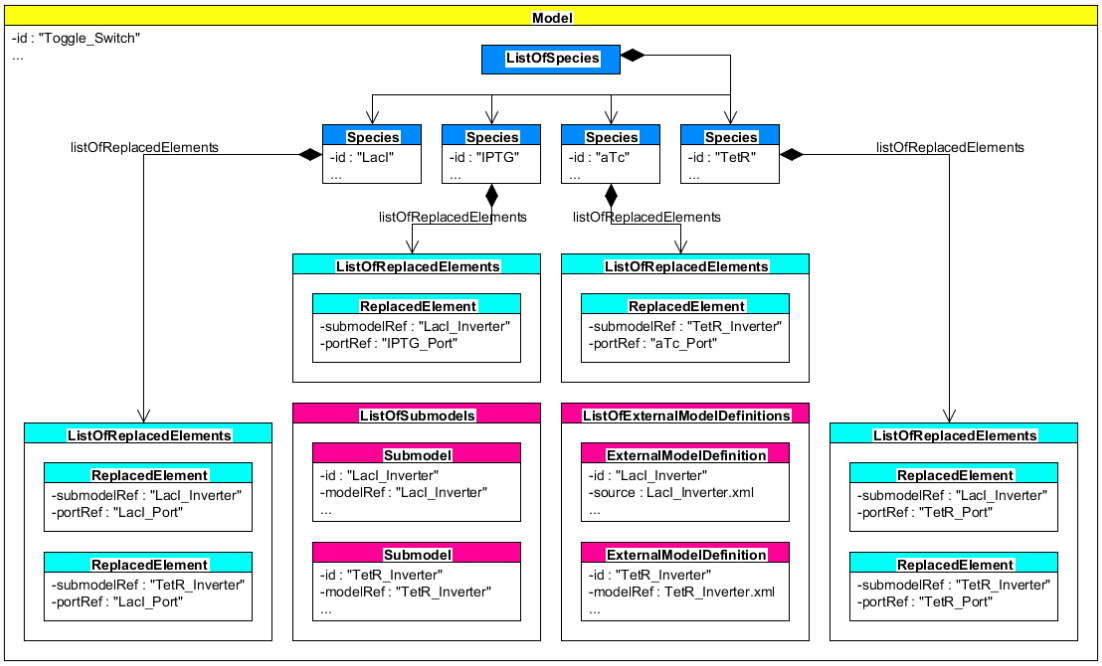

Figure 4.6: UML diagram of the genetic toggle switch under Level 3, Version 1 of the SBML data model. Yellow objects are created during step 1 of model generation, blue during step 2, magenta during step 6, and turquoise during step 6(d).

lists are based on the turquoise port maps that link between the component instantiations of the toggle switch module and ports on the LacI and TetR inverter modules.

While SBML is capable of encoding replacement of lower elements by higher elements and higher elements by lower elements in the modeling hierarchy, the model generation methodology presented in this chapter only uses the first type of replacement to compose and connect generated SBML models. This is done because one of the use cases for port mapping in SBOL is to supply information that is missing at a lower level of the design hierarchy. If SBOL is ever extended to explicitly encode the notion of replacement, then model generation can be similarly extended to produce both directions of replacement in SBML (high by low and low by high).

4.3 Summary

In conclusion, the model generation methodology presented in this chapter enables users of iBioSim to generate quantitative SBML models from qualitative SBOL modules.

By means of the model annotation methodology presented earlier in the chapter, the generated SBML models are also annotated with the SBOL from which they are generated to tightly couple these quantitative and qualitative descriptions of genetic function and structure. In this way, model generation and annotation facilitate the application of genetic technology mapping and sequence generation, which are described next in Chapter 5 and Chapter 6. While the mapping used during model generation is only one of many possible mappings from SBOL to SBML, or from SBOL to another modeling standard, in the future other mappings will be developed for different design tasks and can be readily accommodated by model annotation as needed.

CHAPTER 5

GENETIC TECHNOLOGY MAPPING

Genetic technology mapping is the process of automatically selecting genetic components from a library to meet the abstract functional specification for a genetic circuit. This chapter presents a *Directed Acyclic Graph* (DAG) based approach to genetic technology mapping that builds off DAG-based techniques from Electronic Design Automation (EDA) [102, 48]. To start, Section 5.1 outlines the major assumptions that are inherent in this approach. Next, Section 5.2 describes *graph construction*, a processing stage in which regulatory DAGs are constructed from a model specification written in SBML and a library of SBOL-annotated SBML models. This is followed by Section 5.3 on *partitioning* and *decomposition*, which are postprocessing heuristics that involve splitting the specification DAG into a set of *rooted DAGs* (also known as *trees*) and transforming both the specification and library DAGs to a logically equivalent *canonical form*.

Next, Section 5.4 and Section 5.5 present the terminal stages of DAG-based genetic technology mapping that are responsible for finding a solution: matching and covering. During matching, the library DAGs are matched to each node in the specification DAG and *lower bounds* on the costs of solutions starting at each node are calculated via *dynamic programming*. During covering, matches are selected using a *branch-and-bound approach* to obtain a solution set of library DAGs that may be composed to satisfy the original specification for a minimal cost. Figure 5.1 provides a brief overview of DAG-based genetic technology mapping as applied to automate the design of a genetic multiplexer [59], a genetic circuit that can be used to share multiple incoming signals with another genetic circuit and thereby reduce resource requirements.

Following these sections' description of the DAG-based approach to genetic technology mapping, Section 5.6 presents the results of applying this approach in iBioSim to several test specifications and four randomly generated libraries of various sizes. More specifically, the test specifications are for three combinational genetic circuits: a *genetic AND-OR-invert* (AOI) *gate*, a *genetic NAND-NOR cascade*, and a *genetic OR-AND-invert* (OAI)

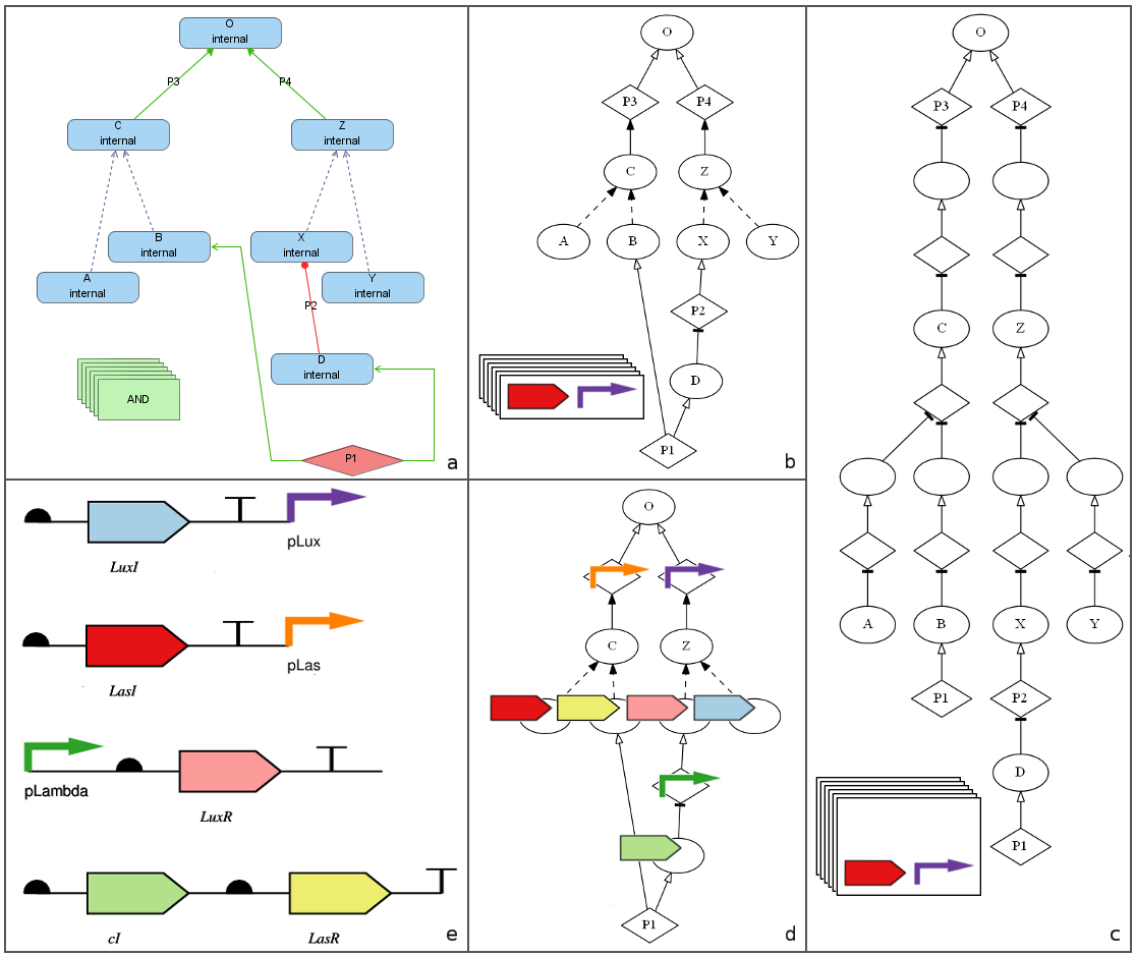

Figure 5.1: Overview of DAG-based genetic technology mapping as applied to automate the design of a genetic multiplexer. (a) The software tool iBioSim is used to construct a SBML model of an abstract genetic multiplexer and generate a library of SBOL-annotated SBML models of genetic logic gates. The model of the genetic multiplexer contains species (blue ellipses) that are modifiers (lines) or reactants/products (arrows) of genetic production and complex formation reactions (purple circles). (b) A regulatory DAG specification and library of SBOL-labeled DAGs are constructed from the SBML model specification and library. See Figure 5.2 for a key describing the nodes and edges of a regulatory DAG. (c) The DAG specification is partitioned and decomposed alongside the library DAGs to faciliate matching and covering. (d) The DAG specification is matched and covered to obtain an optimal solution set of library DAGs. In addition, the SBOL-annotated SBML models underlying these DAGs are composed to form a composite SBML model for subsequent analysis. (E) Composite SBOL DNA components are inferred from the structure of the cause-and-effect relationships between DNA components as encoded by the solution [49] (see Chapter 6.3).

cascade. Finally, the chapter concludes with Section 5.7, a brief summary of the advantages and disadvantages of DAG-based genetic technology mapping and a comparison of this approach to others.

5.1 Assumptions

There are two major assumptions that are inherent in this dissertation's DAG-based approach to genetic technology mapping. First, this approach assumes that the genetic circuits in the target library have no feedback and possess a sigmoidal relationship between their steady-state inputs and output, such that it is possible to distinguish between high and low inputs/outputs to a circuit and assign a logical semantics to their relationship such as OR or AND. This is not an entirely unfounded assumption given previous research into the design and construction (even random construction [62]) of genetic components [55, 56, 57] and circuits [103, 58, 59] with steady-state phenotypic behaviors resembling digital logic.

Second, this approach assumes that it is possible to connect genetic circuits on the basis of whether the molecular identities of their input and output signals are the same. In doing so, this approach neglects other considerations for connecting circuits, such as determining whether the high and low output signals for one circuit constitute high and low input signals for its connected circuit, or whether two connected components are compatible in the context of their intended host. In the future, if this approach is extended to explicitly address these other considerations for circuit compatibility, then it can also leverage the cost function framework described in this chapter to guide the search for a solution that maximizes circuit compatibility, rather than just ensure that circuit compatibility is satisfied.

5.2 Graph Construction

Figure 5.2 displays the regulatory DAGs constructed from the SBML models for a simple, abstract genetic circuit and a library of genetic logic gates. Later on, Section 5.4 and Section 5.5 return to the DAGs for this genetic circuit and library as part of a basic example of matching and covering. The procedure for constructing a regulatory DAG from a SBML model follows the steps outlined below:

1. For each nonpromoter species s in the SBML model:

   (a) Add a species node n to the DAG.

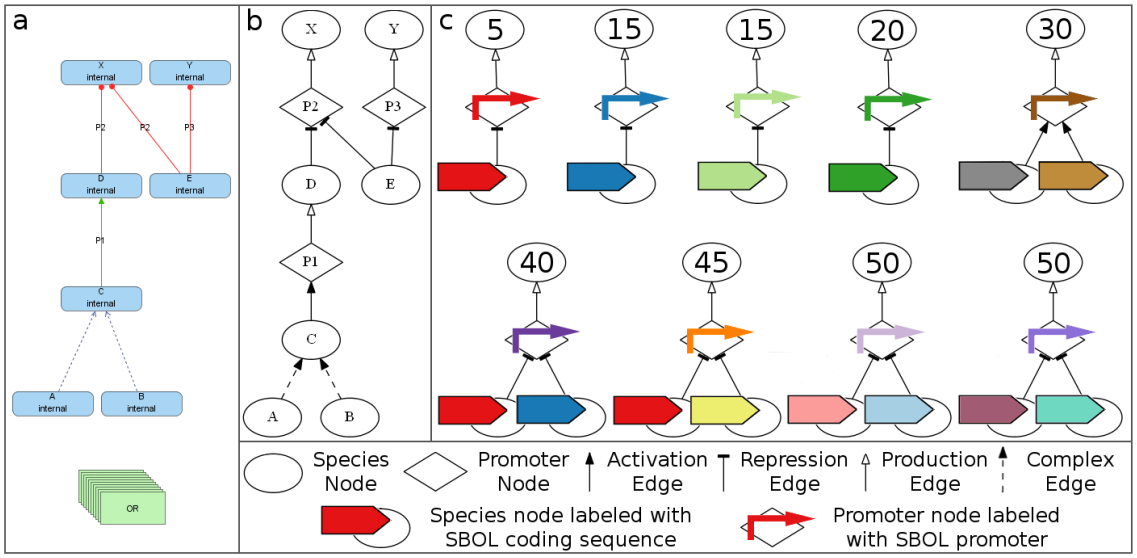

Figure 5.2: Example of graph construction. (a) In this example, regulatory DAGs are constructed from the SBML models for a simple, abstract genetic circuit and a library of genetic logic gates. (b) The specification DAG contains nodes and edges of the types listed in the legend. (c) The library DAGs also include colored arrows that belong to the SBOL Visual standard [53] and represent the promoter or CDS DNA components labeling each node. In addition, although it is not shown in the diagrams for this chapter, each node labeled with a CDS DNA component is also labeled with the protein component that is encoded by the CDS. The number on each library DAGs indicates the cost associated with choosing that DAG to cover part of the specification DAG. Normally, each library DAG has a cost equal to the combined length in base pairs of its DNA components, but this example uses smaller, simpler costs to make it easier to follow along later during matching and covering.

(b) Label n with DNA, TF protein, and small molecule components annotating s.

2. For each promoter species p in the SBML model that modifies a genetic production reaction r, if the DAG being constructed is a specification DAG or if p is annotated with a promoter DNA component that has a DNA sequence:

   (a) Add a promoter node m to the DAG.

   (b) Label m with any promoter DNA component that annotates p.

   (c) For each species that r references as an activator or repressor, draw an activation or repression edge from the corresponding species node to m.

(d) For each species that r references as a product, draw a product edge from m to the corresponding species node.

3. For each noncovalent binding reaction v in the SBML model that references a species s as its product and a species s' as its reactant, draw a complex edge from the species node that corresponds with s' to the species node that corresponds with s.

At the terminus of graph construction, the DNA components labeling the nodes of the constructed DAG are used to calculate the cost associated with selecting the DAG during matching and covering. The TF protein and small molecule components labeling each node, on the other hand, are used to determine whether selecting a DAG during covering would introduce cross-talk to a solution. For calculating the cost of a DAG, a very simple cost function is applied: cost equals the combined length in base pairs of the DNA components labeling the DAG. At the very least, the length in base pairs of a genetic circuit can be partially correlated with other design parameters of interest, such as delay in transcription/translation and cost for *de novo synthesis*. In the future, this cost function can be extended with other relevant genetic circuit parameters, such as the high and low levels of circuit inputs and outputs, the noise associated with these levels, and the degree of input/output compatibility when two circuits are connected.

Finally, to distinguish between regulatory DAGs and make them amenable to logical decomposition, one possible logical semantics is assigned to the *genetic regulatory motifs* present in these DAGs. Under this semantics, a promoter node with a single repressor is an *inverter motif*, while a promoter with two repressors is a *NOR motif*. Inverter and NOR motifs produce output only when no inputs are present. Similarly, a promoter with one activator is a *buffer motif*, while a promoter with two activators is an *OR motif*. Buffer and OR motifs produce output when one or more inputs are present. Lastly, a promoter with a complex activator is an *AND motif*, while a promoter with a complex repressor is a *NAND motif*. An AND motif produces output only when both inputs are present, while a NAND motif produces output so long as at least one input is not present.

Note that this particular logical interpretation enables regulatory DAGs to capture the abstract behavior if not the exact mechanism of genetic circuits that implement combinational logic. For example, in Figure 5.1, part of the solution for the genetic multiplexer is an AND motif that includes complex formation between LuxR and *LuxI*. Strictly speaking, it is the enzymatic product of LuxI, AHL, that forms a complex with LuxR and then activates the promoter pLux, but this additional mechanistic detail is not

absolutely necessary to express the abstract logical relationship between the inputs and output of the genetic circuit.

5.3 Partitioning and Decomposition

During partitioning (see Figure 5.3), the specification DAG is split at nodes with more than one outgoing edge into n rooted DAGs, where n is the total number of outgoing edges at these nodes. Partitioning enables the use of dynamic programming during matching to calculate lower bounds on the costs of solutions starting from each node. These lower bounds can then be used during covering to terminate the search for suboptimal solutions and thereby speed up the process of finding optimal solutions for each partition. The tradeoff is that there is no guarantee of global optimality when the covered partitions are composed to form a final solution. As with many heuristics, the hope is that the nonglobally optimal solution is found more quickly and that it is still of fairly high quality.

Decomposition, on the other hand, increases the number of matches that can be made between the library DAGs and each node in the specification DAG, thereby increasing the number of possible solutions during covering and potentially improving the quality of the final solution. During decomposition (see Figure 5.3), the specification and library DAGs are transformed to a logically equivalent canonical form. In this canonical form, the DAGs only contain inverter and NOR motifs. Consequently, if the library contains at least as many inverter and NOR motifs as the decomposed specification DAG, then

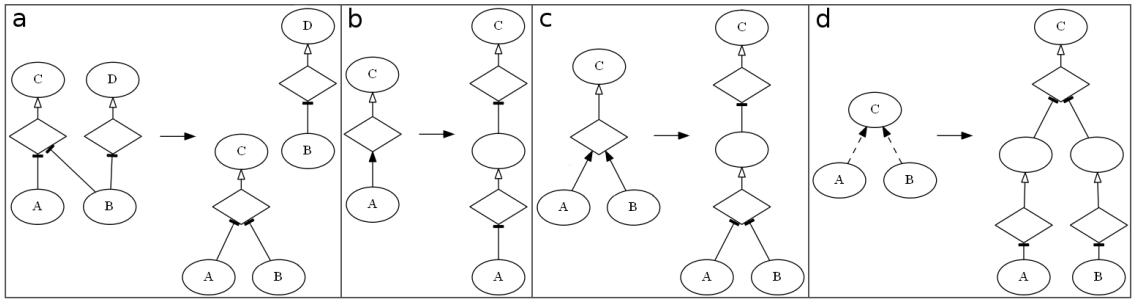

Figure 5.3: Examples of partitioning and decomposing regulatory DAGs. (a) Partitioning results in DAGs with no nodes that have more than one outgoing edge. (b) Decomposing a buffer motif results in two inverter motifs in series. (c) Decomposing an OR motif results in a NOR motif followed by an inverter motif. (d) Decomposing an AND motif results in two inverter motifs in parallel followed by a NOR motif.

it is guaranteed that there is a complete solution that satisfies the specification DAG. This is particularly useful when a genetic regulatory motif is not shared by the library and specification DAGs prior to decomposition. For example, if the specification DAG contains a complex activator and the library DAGs do not, then this AND motif must be decomposed to logically equivalent motifs that are shared by the library DAGs in order to facilitate a complete solution. While there are other possible canonical forms based on different pairings of logical motifs (such as inverter and NAND motifs), the present approach decomposes to inverter and NOR motifs based in part on their prevalence in online repositories such as the iGEM Registry of Standard Biological Parts [16].

5.4 Matching

The present DAG-based approach to matching a library to a specification builds upon that taken in Keutzer's foundational technology mapping system for electronic circuits, DAGON [102]. Like DAGON, this approach uses the *Aho-Corasick algorithm* [104] with minor modifications to match strings of characters encoding paths through the library DAGs to strings encoding paths through the specification DAG. Through string representation and the construction of a *discrete finite automaton* (DFA), the Aho-Corasick algorithm achieves a worst-case runtime that scales linearly with the size of the specification and independently of the size of the library.

During matching, the nodes of the specification DAG are traversed in a topological order. At each node, the Aho-Corasick algorithm is used to match the subtree rooted at that node to the library DAGs. The library DAGs that match at a node are then ordered so that the resulting sequence begins with the DAG that is part of the minimal cost solution starting at that node. Note that the minimal cost solution is determined without giving consideration to the possibility of genetic cross-talk, as the cost of this solution is meant to serve as a lower bound on the cost of any solution starting at that node

For example, consider node O of the specification DAG in Figure 5.4. There are four library DAGs that match the subtree rooted at this node, namely the four inverter motifs at the bottom of the library and the OR motif (decomposed to a NOR motif followed by an inverter motif) in the upper right corner. Of the inverter motifs, the motif on the far left has the lowest cost of 5. If this motif is selected to cover node O, then it also covers down to node C. Since the previously determined lower bound at node C is 50, the lower bound at node O resulting from choosing this inverter motif would equal 55. It is possible,

Node	Bound
X	85
D	45
O	40
C	50
M	5
N	5

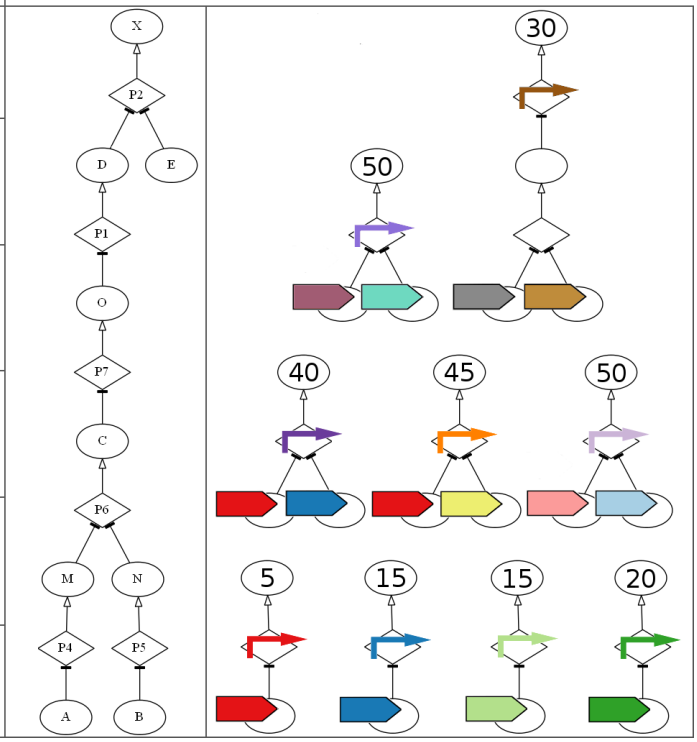

Figure 5.4: Partitioned, decomposed specification and library DAGs from Figure 5.2. Includes the results of calculating lower bounds on the costs of solutions starting at each node in the specification DAG and covering down to its leaves. For example, after iterating through each possible match to the subtree rooted at node O, it is determined that the minimal lower bound for a solution starting at this node that ignores cross-talk is obtained by selecting the OR motif in the upper right corner. This motif has a cost of 30 and covers down to nodes M and N, which each have a previously determined lower bound of 5 (matching proceeds in topological order). These lower bounds are added to the cost of the OR motif to obtain a lower bound of 40 at node O.

however, to obtain a better lower bound than this by selecting the OR motif, which has a cost of 30 and covers down to nodes M and N. These nodes both have lower bounds of 5, so the lower bound at node O resulting from choosing the OR motif would equal 40. Hence, the OR motif is positioned first in the list of matching library DAGs at node O and its corresponding lower bound is entered into the table in Figure 5.4.

Algorithm 5.1 handles the process of matching library DAGs to the subtrees rooted at each node in the specification DAG and calculating lower bounds on the costs of solutions that start with each match. Unlike its EDA implementation, Algorithm 5.1 also sorts the matches at each node in accordance with their lower bounds. Sorting in this manner is necessary to enable effective bounding while covering. During the execution of this

Algorithm 5.1: Matching

Input: Specification DAG $G = \langle V, E \rangle$ where V is a set of nodes and E is a set of edges, DFA D constructed from DAG library L using the Aho-Corasick string matching algorithm

Output: Specification DAG G where each node in V contains a sequence of library DAGs L_i that match the subtree rooted at that node and a sequence of lower bounds on the costs of solutions beginning with each match

```
/* || is a composition operator that joins two sequences          */
/* ⟨⟩ are opening/closing sequence brackets                       */
```

```
1  C ← leaves(G)
2  while |C| > 0 do
3      for L_i ∈ DETERMINE-MATCHES(D, paths(G, C_0)) do
4          matches(C_0) ← matches(C_0)||⟨L_i⟩
5          b ← CALCULATE-COST(L_i)
6          for v ∈ WALK-PATHS(paths(L_i), G, C_0) do
7              b ← b + lowerBounds(v)_0
8          lowerBounds(C_0) ← lowerBounds(C_0)||⟨b⟩
9      QUICKSORT-MATCHES(matches(C_0), lowerBounds(C_0))
10     N ← successors(C_0)
11     C ← sub(C, 1, |C| − 1)
12     if |C| = 0 ∧ |N| > 0 then
13         C ← C||N
```

algorithm, nodes are matched and bound in a topological order, that is, starting from the specification DAG's *leaf nodes* that have no incoming edges and ending at its *root node* that has no outgoing edges. In this way, previously calculated lower bounds can be reused to calculate later lower bounds, a dynamic programming approach that enables bounding to be performed with a worst-case runtime that scales as $|G||L|$, where $|G|$ is the size of the specification and $|L|$ is the size of the library. As for matching, a DFA constructed via the Aho-Corasick string matching algorithm is used to match the subtrees rooted at each node in the specification simultaneously against all library DAGs, thereby achieving a worst-case runtime that scales independently of the size of the library.

Functions called by Algorithm 5.1 include DETERMINE-MATCHES, CALCULATE-COST, WALK-PATHS, and QUICKSORT-MATCHES. The DETERMINE-MATCHES function takes as input the Aho-Corasick DFA D and a sequence of strings representing paths through the specification DAG G that begin at the current node C_0 and proceed back to the leaves of G. This string representation of the subtree rooted at C_0 is provided as input to D, which outputs the sequence of library DAGs matching the subtree.

The CALCULATE-COST function, as its name suggests, calculates the cost for a matching library DAG L_i in accordance with the cost function, which in this dissertation sums the nucleotide counts for DNA components labeling the DAG. The WALK-PATHS function then uses a sequence of strings representing paths through L_i to walk back through G from C_0 and identify the nodes in G that are at the boundary of the subtree covered by L_i. Together, CALCULATE-COST and WALK-PATHS work to determine the lower bounds on the costs of solutions that start with each match and cover all the way back to the leaves of G.

Finally, the function QUICKSORT-MATCHES uses the Quicksort algorithm [105] to sort the sequence of matching library DAGs at each node in accordance with the corresponding sequence of lower bounds on the costs of solutions starting with these matches. Note that ordering the matches at each node in this manner makes the overall worst-case runtime for matching scale as $|G||L|\log(|L|)$. The reason ordering is performed is to enhance the efficiency of covering, which has a worse worst-case runtime that scales as $|L|^{|G|}$. By biasing towards the discovery of better solutions earlier, ordering is expected to increase the efficacy with which these solutions' costs are used to bound the search for suboptimal solutions.

Primitive routines called by Algorithm 5.1 include the graph routines *leaves* and *paths*, the node routines *successors*, *matches*, and *lowerBounds*, and the sequence routine *sub*. The graph routine *leaves* returns the leaf nodes for the given graph, while *paths* returns strings encoding each possible path from the given node (or root node if none is given) back to the leaves of the given graph. These strings consist of alternating letters and numbers that represent the types and cardinality of the nodes and edges.

Next, the node routine *successors* returns all nodes with incoming edges that point from the given node. The routines *matches* and *lowerBounds*, on the other hand, return sequences of library DAGs found to match the given node and sequences of the lower bounds on costs of solutions beginning with these matches, respectively. Lastly, *sub* returns a subsequence of the given sequence that starts at the indicated index and has the indicated length. This routine is used during Algorithm 5.1 to effectively delete the first node in a sequence by replacing the sequence with a subsequence that starts at index one and has a length equal to that of the sequence minus one.

5.5 Covering

Once matches and lower bounds have been determined, the process of selecting matches to form a valid, optimal cover begins at the root of the specification and proceeds in a depth-first fashion. The covering algorithm, however, must take into account the possibility of genetic cross-talk resulting from shared regulatory species. Due to this possibility, a particular matching library DAG may be selected only once during a cover. Furthermore, each covering decision has implications that may affect future covering decisions and prevent an optimal solution from being found after a single traversal of the specification. To address this problem, the approach to covering presented in this section incorporates recursive backtracking and effectively becomes a branch-and-bound approach.

In this branch-and-bound approach, whenever a *dead-end* (that is a state in which selecting any available match introduces cross-talk) or a solution is encountered, the traversal of the specification backtracks to the last node at which a match was selected. The next best match at this node is then selected, provided that the best-case estimate for the cost of a solution starting at the node does not exceed the cost of the best solution found so far. The best-case estimate is calculated by summing the costs of previously selected matches and the lower bounds on uncovered nodes at the partial solution's boundary. While branch-and-bound guarantees that the optimal solution is eventually found, it also has a worst-case runtime that scales exponentially with the size of the input specification. In practice, however, the bounding aspect of branch-and-bound can improve the average runtime by pruning the search for suboptimal solutions. Figure 5.5 demonstrates the application of branch-and-bound covering to the specification and library DAGs from Figure 5.2.

Despite the use of bounding to terminate the search for suboptimal solutions, the worst-case runtime for covering still scales as $|L|^{|G|}$, since in the absolute worst-case there is no solution at all due to the constraints of genetic cross-talk. In this case, every possible combination of library DAGs that nearly covers the specification must be tested without the aid of bounding. It is also possible that the best solution exists but has a cost that differs dramatically from the theoretically optimal cost for a solution that does not account for cross-talk, in which case bounding by comparing the two does not discriminate suboptimal solutions until many choices have been made and fewer branches are pruned from the *decision tree* as a result.

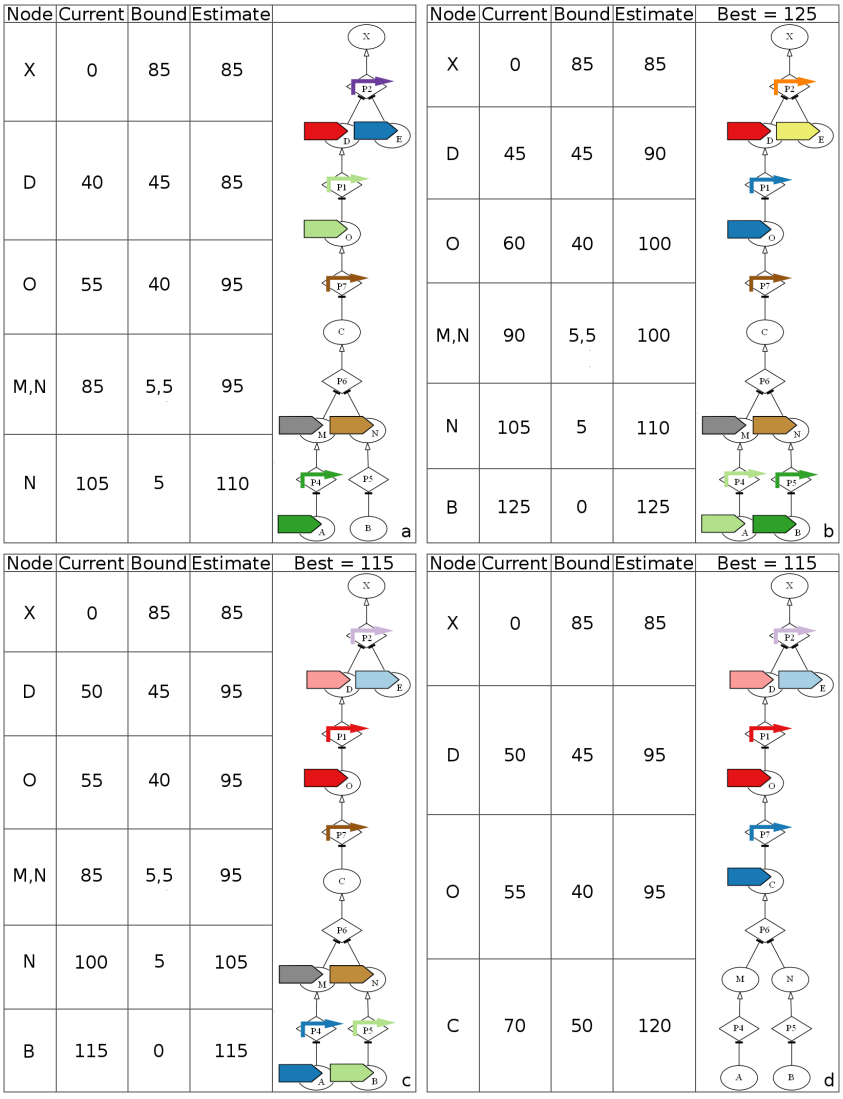

Figure 5.5: Covering with the specification and library DAGs from Figure 5.2. (a) During the first traversal, a dead-end is encountered. Selecting any of the remaining inverter motifs to cover the rest of the specification would result in unintended cross-talk. (b) After backtracking and encountering several other dead-ends, the next best match at node X is selected and the resulting traversal yields a complete solution with a cost of 125. (c) Backtracking to look for better solutions yields a solution with a cost of 115 when the next best match at node X is selected, which becomes the current best solution. (d) Finally, when backtracking to node O and selecting the next best match, the search for better solutions is halted since the best-case estimate for solutions starting from the next node C is 120, which is greater than the cost for the best solution found so far. The best-case estimate for a partial solution is calculated by summing its current cost and the combined lower bounds on the uncovered nodes at the partial solution's boundary.

Algorithm 5.2 handles the process of selecting matches to form the best solution to the entire specification, starting with matches at the root node of the specification and covering back through the specification to its leaves in a depth-first fashion. Functions called by Algorithm 5.2 include Cross-Talk, Solution-In-Bound, and Walk-Paths. The first two functions serve to verify whether the current matching library DAG L_i under consideration can be part of a valid, potentially optimal solution. Specifically, the Cross-Talk function checks whether or not L_i shares any small molecule or TF protein components with the current solution S as a whole, while the Solution-In-Bound function determines whether the sum of the cost of S and the lower bounds for the currently selected matches at each node in C is less than the cost of the best solution B_0 found so far. The Walk-Paths function works as previously described, but is used in Algorithm 5.2 to determine which nodes should be considered next after a match at the current node C_0 has been selected to be part of S.

Previously undescribed primitive routines called by Algorithm 5.2 include the graph routine *root*, the solution routine *cost*, and the node routines *currentMatch*, *branch*, *resetBranch*, and *uncovered*. The graph routine *root* returns the root node of the given graph. Next, the solution routine *cost* returns the total cost for the given solution. Finally, the first three node routines keep track of and modify which match is currently selected at the given node. In particular, *currentMatch* returns the currently selected match, while *branch* increments the index that indicates which match is selected and *resetBranch* sets this index to its starting value. The *uncovered* routine, on the other hand, keeps track of which nodes were in C alongside the given node when it was the current node under consideration (C_0). This routine enables Algorithm 5.2 to recover the state of C when backtracking to a previously considered node.

Finally, at the terminus of matching and covering, the SBOL-annotated SBML models corresponding to the DAG matches that make up the best solution found are composed to form a composite SBOL-annotated SBML model (see Figure 2.3). The composite SBML model encodes the quantitative function for the designed genetic circuit as a whole, while the annotating SBOL encodes structural information on the DNA components which make up the designed genetic circuit, including their DNA sequence and sequence annotations. This composite genetic circuit design that contains both structural and functional information can then serve to inform subsequent stages of computational verification, optimization, and physical construction.

Algorithm 5.2: Covering

Input: Specification DAG $G = \langle V, E \rangle$

Output: Sequence of solutions B where the best solution is located at index 0 and each solution is a sequence of library DAGs L_i Sequence of solutions B where the best solution is located at index 0 and each solution is a sequence of library DAGs L_i

```
/* Repeat-until loop executes as long as its condition is not met   */
```

1 $S \leftarrow \langle\rangle$
2 $P \leftarrow \langle\rangle$
3 $C \leftarrow \langle root(G) \rangle$
4 $B \leftarrow \langle\rangle$
5 **repeat**
6 **if** $|matches(C_0)| = 0$ **then**
7 $C \leftarrow sub(C, 1, |C| - 1)$
8 **if** $|C| = 0 \wedge |P| > 0$ **then**
9 **if** $|B| = 0 \vee cost(S) < cost(B_0)$ **then**
10 $B \leftarrow \langle S \rangle || B$
11 $S \leftarrow sub(S, 0, |S| - 1)$
12 $resetBranch(P_0)$
13 $P \leftarrow sub(P, 1, |P| - 1)$
14 **else if** $branch(C_0)$ **then**
15 **if** $\neg$CROSS-TALK$(currentMatch(C_0), S)$ **then**
16 **if** SOLUTION-IN-BOUND(S, C, B_0) **then**
17 $uncovered(C_0) \leftarrow sub(C, 1, |C| - 1)$
18 $N \leftarrow$WALK-PATHS$(paths(L_i), G, C_0)$
19 $S \leftarrow S || \langle currentMatch(C_0) \rangle$
20 $P \leftarrow \langle C_0 \rangle || P$
21 $C \leftarrow sub(C, 1, |C| - 1)$
22 $C \leftarrow N || C$
23 **else**
24 $resetBranch(C_0)$
25 $C \leftarrow \langle\rangle$
26 **else**
27 $C \leftarrow \langle\rangle$
28 **if** $|C| = 0 \wedge |P| > 0$ **then**
29 $S \leftarrow sub(S, 0, |S| - 1)$
30 $C \leftarrow C || uncovered(P_0)$
31 $C \leftarrow \langle P_0 \rangle || C$
32 $P \leftarrow sub(P, 1, |P| - 1)$
33 **until** $|C| = 0$
34 **return** B

5.6 Case Studies

To evaluate the branch-and-bound, DAG-based approach to genetic technology mapping, this section examines its use in mapping three genetic circuit specifications (see Figure 5.6) against four randomly generated libraries of increasing size. The performance of this approach is then compared with the results of a naive *exhaustive search* that tries all possible solutions and a greedy variant of branch-and-bound that returns the first solution found. The greedy variant still orders matches based on the lower bounds of costs for solutions that start with them, but quits when the first solution is found and does not use these lower bounds to inform the search for better solutions. Each algorithm was run up to one hour before timing out.

Note that the specifications mapped in this section are not meant to describe genetic circuits that have a particular real-world application. Rather, they are meant to serve as

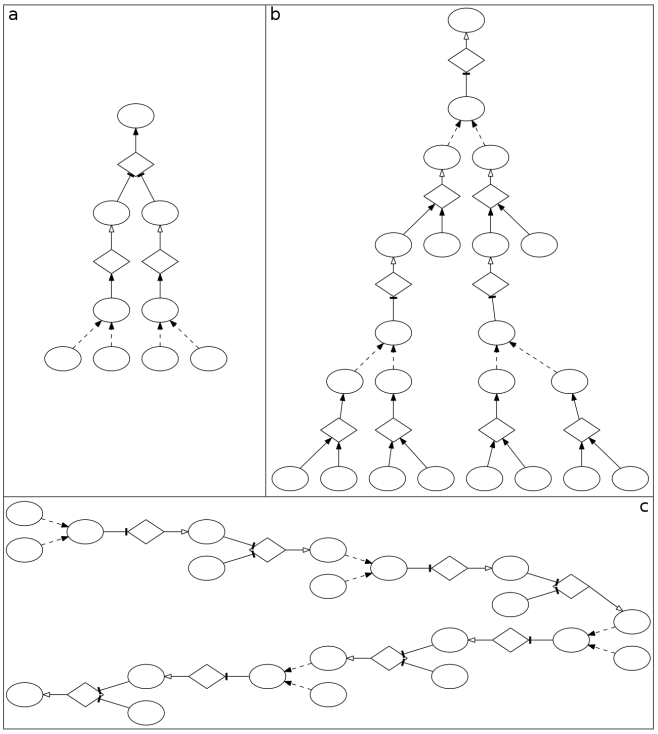

Figure 5.6: Case study specification DAGs. (a) First is the regulatory DAG for a genetic AOI. (b) Second is the regulatory DAG for a genetic NAND-NOR cascade. (c) Third is the regulatory DAG for a genetic OAI cascade. The sizes of these DAGs following their decomposition are 11, 16, and 24 nodes, respectively.

general examples that vary in size and incorporate a range of regulatory motifs for the purpose of *benchmarking*.

As for the test libraries, they are randomly generated such that they contain a roughly *uniform distribution* of inverter, buffer, OR, NOR, AND, and NAND motifs. Furthermore, the DNA components annotating each model in the library have random, *Gaussian-distributed lengths*, while their encoded protein components are shared with four percent of all models in the library. Annotating the test libraries in this manner simulates the likely reuse of DNA components across library circuits and the resulting cross-talk relationships between them. Tables 5.1, 5.2, and 5.3 present the results of each case study.

Table 5.1: Solution times and sizes for the genetic AOI.

Algorithm	**Library Size**				**Library Size**			
	25	**50**	**100**	**200**	**25**	**50**	**100**	**200**
	Solution Time (s)				**Solution Size (bp)**			
Exhaustive Search	0.2	1	60	>1 h	3662	2871	2946	n/a
Branch-and-Bound	0.01	0.01	0.02	0.02	3662	2871	2946	2913
Greedy Variant	0.01	0.01	0.02	0.02	3662	2871	2946	2913

Table 5.2: Solution times and sizes for the genetic NAND-NOR cascade.

Algorithm	**Library Size**				**Library Size**			
	25	**50**	**100**	**200**	**25**	**50**	**100**	**200**
	Solution Time (s)				**Solution Size (bp)**			
Exhaustive Search	1	>1 h	>1 h	>1 h	11178	n/a	n/a	n/a
Branch-and-Bound	0.2	1	0.7	1	11178	10931	10592	8270
Greedy Variant	0.02	0.03	0.04	0.06	13219	10933	11107	8482

Table 5.3: Solution times and sizes for the genetic OAI cascade.

Algorithm	**Library Size**				**Library Size**			
	25	**50**	**100**	**200**	**25**	**50**	**100**	**200**
	Solution Time (s)				**Solution Size (bp)**			
Exhaustive Search	>1 h	>1 h	>1 h	>1 h	n/a	n/a	n/a	n/a
Branch-and-Bound	5	100	10	40	13836	12518	11377	9335
Greedy Variant	2	0.03	0.06	0.02	14774	15357	11603	9592

5.6.1 Genetic AOI

Table 5.1 displays the results of mapping the specification for a genetic AOI against libraries containing 25, 50, 100, and 200 circuits. The AOI is shown in Figure 5.6 and has an effective size of 11 nodes that must be matched after its decomposition.

When mapping the genetic AOI, it appears that the solution times for branch-and-bound scale very well relative to the size of the library, unlike the solution times for exhaustive search. As expected, the sizes of the solutions found with both approaches are the same, which indicates that branch-and-bound does find the best possible solution in much less time. As for the greedy variant, its overall performance when mapping the AOI is nearly identical to that of branch-and-bound. Normally, it is expected that the greedy variant produces a lower quality solution in less time than branch-and-bound, but in this case the best possible solution is found first and branch-and-bound terminates almost immediately after finding it. These rapid terminations are likely facilitated by the fact that smaller specifications require fewer circuits to cover, which means that there are fewer opportunities for their best possible solutions to diverge from their theoretically optimal solutions that ignore cross-talk. When covering yields a solution that has a cost equal to that of the theoretically optimal solution, bounding the search for suboptimal solutions becomes most effective. Lastly, the quality of the best possible solution increases with library size, likely owing to the greater absolute number and therefore diversity of motifs that are left to select from larger libraries after a percentage are ruled out due to considerations of cross-talk.

5.6.2 Genetic NAND-NOR Cascade

Table 5.2 displays the results of mapping the specification for a genetic NAND-NOR cascade against the same libraries used to generate results in Table 5.1. The NAND-NOR cascade can be seen in Figure 5.6 and has a decomposed size of 16 nodes.

When mapping the genetic NAND-NOR cascade, it appears that the solution times for branch-and-bound increase by one to two orders of magnitude compared with the solution times for the AOI, but it also appears that these times still scale fairly well as library size increases. Exhaustive search, on the other hand, times out when applied against all but the smallest of libraries. As for the greedy variant, it is now consistently faster than branch-and-bound, but produces lower quality solutions. In the case of the library with size 50, however, the solution produced by the greedy variant is only two base pairs worse than that produced by branch-and-bound, suggesting that there are still conditions under

which the greedy variant can produce a nearly optimal solution for larger specifications. Also, once again the quality of the best possible solution increases with library size.

5.6.3 Genetic OAI Cascade

Table 5.3 displays the results of mapping the specification for a genetic OAI cascade against the same libraries as before. The OAI cascade is depicted in Figure 5.6 and has a size of 24 nodes following decomposition.

When mapping the genetic OAI cascade, it appears once again that the solution times for branch-and-bound increase by one to two orders of magnitude when compared with the previous smaller specification, yet scale tolerably for larger library sizes. Interestingly, the longest solution time occurs when mapping against a library of size 50 and is an order of magnitude larger than when mapping against the larger libraries. One likely cause for this result is an increase in the cost gap between the first and best solutions found for this size of library, which can make bounding less effective at halting the search for suboptimal solutions and pruning the decision tree. The cause of this cost gap may be that there are fewer alternative low cost motifs when one is ruled out due to considerations of cross-talk. This condition is made more likely by the decreased diversity of smaller library sizes. For the smallest library sizes, however, it may be that enough solutions are ruled out due to cross-talk, such that pruning of the overall decision tree is achieved regardless of bounding.

For library sizes other than 50, branch-and-bound converges to the best possible solution cost within two to three solutions and the majority of its run time is spent determining that there are no better solutions. For the size 50 library, on the other hand, 11 solutions are found before the best possible solution. While finding the first nine solutions takes less than four seconds of run time and represents a 15 percent gain in solution quality, the last three solutions require an additional minute and a half to discover and represent a less than five percent gain in solution quality. Had branch-and-bound started with second to the last solution, its runtime for the size 50 library would have been more comparable to its runtimes for the smaller and larger libraries. Figure 5.7 visualizes the dynamics of branch-and-bound as applied to the genetic OAI cascade.

These results, taken together with preliminary observations that branch-and-bound begins to time out when mapping specifications larger than the OAI cascade, suggest that the greedy variant should be used to find a fixed number of near-optimal solutions much more quickly for larger specifications. While a fixed number of solutions may suffice when mapping large specifications against small or large libraries, additional heuristics may be

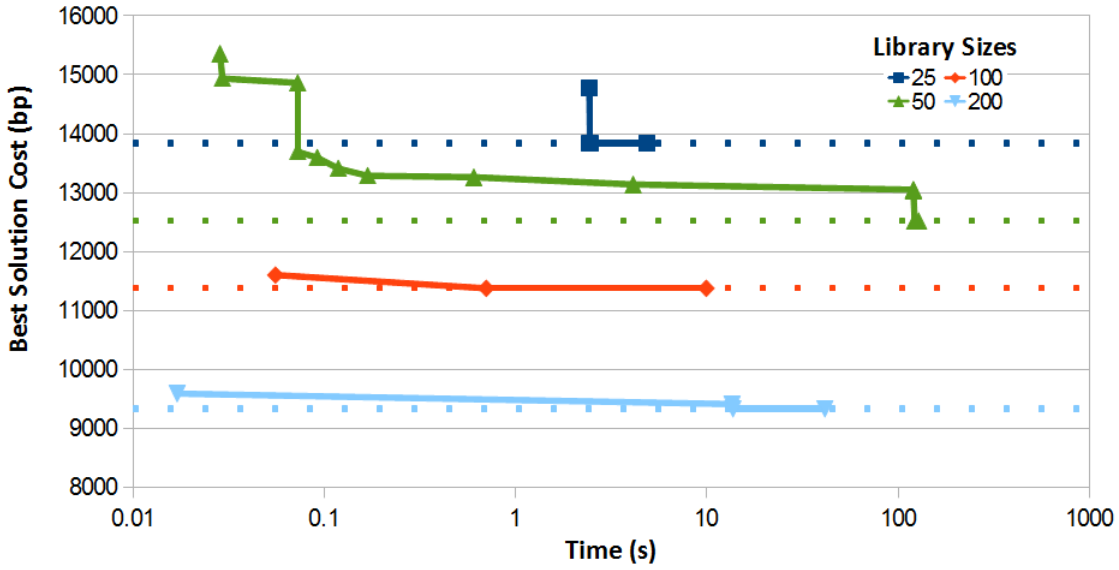

Figure 5.7: Best solution cost in base pairs versus time in seconds when applying branch-and-bound to the genetic OAI cascade and four libraries of different sizes. Each dotted line represents the cost for the best possible solution when mapping against a particular library.

required when mapping large specifications against midsized libraries. These heuristics could include a dynamic cap on the number of solutions found that takes effect when the average increase in solution quality bottoms out over time, or perhaps a less conservative approach to bounding that only explores a potential solution if its best-case estimate cost is less than the best solution cost plus an extra factor that depends on previous changes in solution quality.

5.7 Summary

This chapter presents an algorithmic approach to DAG-based genetic technology mapping that has been implemented in the GDA tool iBioSim. It is among the first approaches to genetic technology mapping to adapt techniques from electronic circuit design, in particular the use of a cost function to guide the search for an optimal solution. Compared with other approaches, it makes the greatest use of open standards, such as SBML [14] and SBOL [44], to represent design specifications and libraries. Unlike early heuristic technology mapping tools, MatchMaker and SBROME, iBioSim can guarantee that the

optimal solution is found and can do so more quickly for larger specification sizes than the first exact technology mapping tools, BioJade [26] and GEC [31].

In light of the results presented in the previous section, there is a case to be made for using a cost function to relate circuit parameters to solution quality, rank matches between the specification and library based on their inclusion in a theoretically optimal solution, bias towards the discovery of near-optimal solutions first, and prune the search for suboptimal solutions. The average run time for exact branch-and-bound scales very well with increasing library size for specification sizes less than 25 nodes, while preliminary results suggest that greedily branching and bounding can find a fixed number of near-optimal solutions to larger specifications, especially when mapping against larger, more diverse libraries that help mitigate the effects of cross-talk. Midsized libraries, on the other hand, may require additional heuristics when mapping larger specifications against them to more quickly determine that a near-optimal solution has been found.

In the future, it should prove interesting to try different cost functions that make use of circuit parameters other than length in base pairs alone. In addition, calculating solution cost using more than one circuit parameter should enable formal comparison of how important is one desired trait for a design when compared to another. The length metric is appealing because it is easily determined and is at least partly related to delay in transcription/translation and the fiscal cost associated with physical construction of a genetic circuit. In the future, however, it would be ideal to calculate cost based on metrics that are more directly related to a genetic circuit's correct function, such as the degree to which the high and low levels of inputs and outputs for two connected circuits are compatible when noise is taken into account. Such metrics are already used to a certain extent by the other tools mentioned above, but these tools do not use them to actively guide the search for optimal solutions to the genetic technology mapping problem.

CHAPTER 6

SEQUENCE GENERATION

In general, sequence generation is the process of inferring a description of genetic structure from an annotated description of genetic function. In the context of this chapter, sequence generation is an automated methodology for inferring a composite SBOL DNA component from a SBOL-annotated SBML model and annotating the top level of this model with the composite component [49]. In this way, sequence generation effectively linearizes the DNA components annotating a given SBML model and enables users of iBioSim to automatically compose a hierarchy of genetic structure at the same time that they compose a hierarchy of genetic function (either manually via a *graphical user interface* or automatically via technology mapping).

The first two sections of this chapter describe the primary stages of sequence generation. In particular, Section 6.1 describes graph construction, the stage in which a SBOL-annotated SBML model is abstracted to a graph representation that captures the flow of information through the model. Section 6.2 then describes graph traversal, the stage in which a *Depth-First Search* (DFS) is performed on the graph with the assistance of a DFA to recognize and compose patterns of DNA components into valid genetic constructs. Lastly, Section 6.3 provides a brief summary of the theory behind sequence generation and compares the approach presented here with that taken in MoSeC [36], the first and only other sequence generation tool that we are aware of besides iBioSim.

6.1 Graph Construction

During graph construction, nodes are created for each SBML element in a model and any DNA components or strand sign annotating a given element are stored at its corresponding node. Next, edges directed from one node to another are created to capture the cause-and-effect relationships between elements of the model. This step results in edges pointing from the nodes for elements that represent quantities (such as species or parameters) to the nodes for elements that represent processes (such as reactions or rules)

and vice versa. The direction of an edge between a node for a quantity element and a node for a process element is determined by whether the quantity element is an input or output of the process element. Figure 6.1 displays the graph constructed from a LacI inverter model.

In the case of a hierarchical SBML model, graph construction must be modified to account for the presence of submodel elements. Whenever a node is created for a submodel element, if there are no DNA components directly annotating the submodel element, then any DNA components annotating the external model that is referenced by the submodel element are stored at the node instead. In this way, DNA components that annotate models lower in the modeling hierarchy are propagated upwards to become subcomponents of DNA components that annotate models higher in the modeling hierarchy.

Next, edge creation must be modified in order to connect the nodes for submodel elements to the nodes for other SBML elements. If a SBML element is marked to replace or be replaced by another element in the external model referenced by a submodel element, then a pair of edges is added to connect the node for the marked element to the node for the submodel element and vice versa. Edges are added in both directions between these nodes because the cause-and-effect relationships between elements at different levels of the modeling hierarchy cannot be known without combining the models and replacing or deleting their elements as marked. Flattening the hierarchy of models and associated DNA

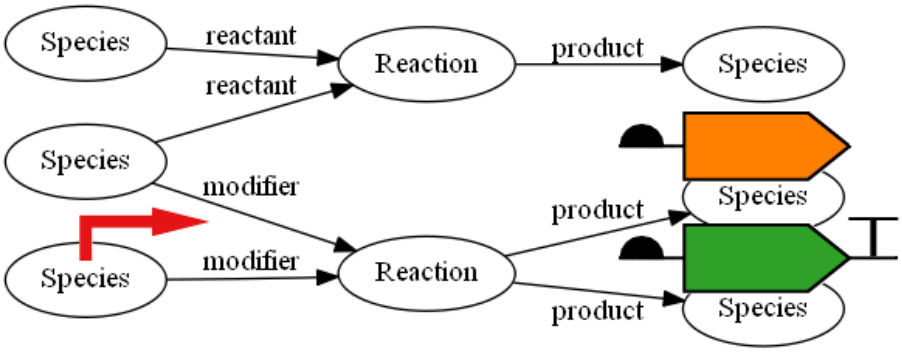

Figure 6.1: Graph constructed from the LacI inverter model generated and annotated in Chapter 4 (see Figure 4.2). Each node of the graph has been labeled with the type of SBML element to which it corresponds, while each edge has been labeled with the type of input/output relationship that exists between the edge's origin and destination nodes. The DNA components stored at these nodes are shown using symbols taken from the SBOL Visual standard [53]. As before, the bent arrow is a promoter, each half circle is a RBS, each box arrow is a CDS, and the "T" is a terminator.

components in this manner, however, may not always be necessary or desirable. Figure 6.2 displays the graph constructed from the hierarchical toggle switch model following the generation of composite DNA components for the LacI inverter and TetR inverter models.

Flattening of a hierarchical model prior to graph construction is necessary when doing so would change the set of genetic constructs resulting from graph traversal. For example, the set of generated genetic constructs would change when an unannotated element in one model is marked to replace an annotated element in another model that is lower in the modeling hierarchy. Hence, the overarching strategy when generating a composite DNA component for a hierarchical model is to construct two graphs, one for the model as it stands and one for its flattened version, then traverse both graphs and compare the partial and complete genetic constructs composed thereof. If the constructs match with respect to the URIs of their noncomposite subcomponents, then sequence generation annotates the hierarchical model using the DNA component composed from the unflattened model. Otherwise, it annotates using the DNA component composed from the flattened model. In this way, sequence generation seeks to preserve the hierarchy of composed DNA components whenever possible, though not at the expense of the set of

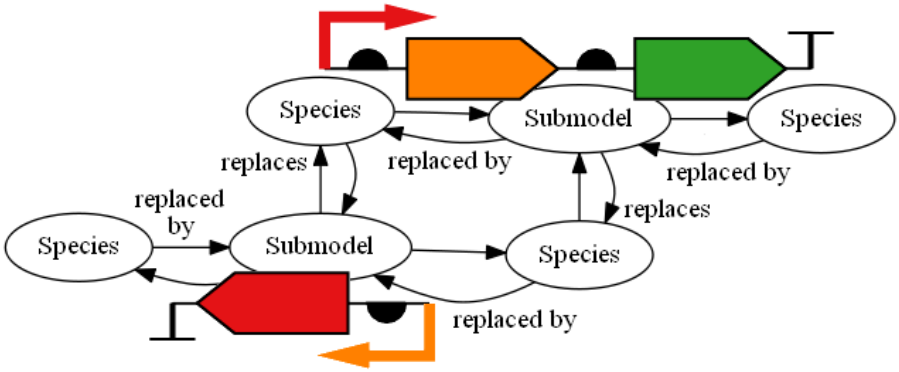

Figure 6.2: Graph constructed from the hierarchical toggle switch model. The composite DNA components for the LacI inverter (top right) and the TetR inverter (bottom left) are stored at the nodes corresponding to the submodel elements of the toggle switch model. These composite components are depicted here using the SBOL Visual symbols for their subcomponents placed along a solid line. The composite component for the TetR inverter is reversed and flipped to indicate that a negative strand sign is stored at the same node. Finally, each pair of edges in the graph has been labeled with the model composition relationship between a species element in the toggle switch model and a species element in the external LacI inverter or TetR inverter model.

genetic constructs implied by the cause-and-effect relationships at the lowest level of the modeling hierarchy.

For example, two separate graph traversals need to be performed on the graphs constructed from the hierarchical and flattened versions of the toggle switch model. Since the toggle switch components generated by traversing these graphs are identical, the component chosen to annotate the toggle switch model is that which possesses the greatest degree of hierarchy, that is, the component composed from the hierarchical version of the model. This composite toggle switch component contains subcomponents for the LacI-repressible and TetR-repressible genes, which in turn contain noncomposite subcomponents such as promoters and terminators. By comparison, the other (also composite) toggle switch component contains these noncomposite subcomponents, but not the intermediate subcomponents for the LacI-repressible and TetR-repressible genes.

Algorithms 6.1 and 6.2 handle the process of graph construction. Algorithm 6.1 initiates graph construction from the nonhierarchical elements of the input SBML model and creates a mapping from these elements to their corresponding graph nodes to assist in edge creation. Algorithm 6.2, on the other hand, finishes graph construction from any submodel elements if they are present. Functions called by Algorithm 6.1 and Algorithm 6.2 include Parse-Annotation, Parse-Identifiers, and Load-External-Model. The Parse-Annotation function takes as input a RDF/XML annotation and determines if it is a SBML-to-SBOL annotation. If so, then the function resolves the annotation into a list of DNA components and a strand sign. Next, the Parse-Identifiers function takes as input a MathML object and returns the SBML elements corresponding to the identifiers (not *mathematical operators* or numbers) found in the MathML. Lastly, the Load-External-Model function takes as input a submodel, determines its corresponding external model definition, and returns the external model defined. Primitive routines called by Algorithm 6.1 and Algorithm 6.2 are indicated with italicized text and belong to graph nodes as well as SBML elements. The routines belonging to SBML elements return a variety of data on these elements as described in the SBML specification [80], while the node routine *dnaComponents* returns the list of DNA components currently stored at the given node.

6.2 Graph Traversal

During graph traversal, a DFS is performed to order the nodes of the graph such that their stored DNA components may be concatenated to form a valid partial or complete

Algorithm 6.1: Construct Graph

Input: SBML model $M = \langle S, R, L, P, N, C\rangle$ where S is a set of species, R is a set of reactions, L is a set of rules, P is a set of global parameters, N is a set of events, and C is a set of submodels
Output: Graph $G = \langle V, E\rangle$ where V is a set of nodes and E is a set of edges, and mapping $\mu : X \to V$ where $X = S \cup R \cup L \cup P \cup N$ is the set of all elements other than submodels in the input SBML model

`/* {} are opening/closing set brackets */`
`/* ∪ is a composition operator that unions the contents of two sets */`
`/* ⟨⟩ are opening/closing sequence brackets */`

for $x \in X$ **do**
  $dnaComponents(v) \leftarrow$ Parse-Annotation($annotation(x)$)
  $V \leftarrow V \cup \{v\}$
  $\mu(x) \leftarrow v$
for $r \in R$ **do**
  for $s \in reactants(r) \cup modifiers(r)$ **do**
    $E \leftarrow E \cup \{\langle \mu(s), \mu(r)\rangle\}$
  for $x \in$ Parse-Identifiers($kineticLawMath(r)$) **do**
    $E \leftarrow E \cup \{\langle \mu(x), \mu(r)\rangle\}$
  for $s \in products(r)$ **do**
    $E \leftarrow E \cup \{\langle \mu(r), \mu(s)\rangle\}$
for $l \in L$ **do**
  if $isAssignmentRule(l) \vee isRateRule(l)$ **then**
    for $x \in$ Parse-Identifiers($math(l)$) **do**
      $E \leftarrow E \cup \{\langle \mu(x), \mu(l)\rangle\}$
    $E \leftarrow E \cup \{\langle \mu(l), \mu(variable(l))\rangle\}$
for $n \in N$ **do**
  for $x \in$ Parse-Identifiers($triggerMath(n)$)∪Parse-Identifiers($delayMath(n)$)∪Parse-Identifiers($priorityMath(n)$)∪Parse-Identifiers($eventAssignmentMaths(n)$) **do**
    $E \leftarrow E \cup \{\langle \mu(x), \mu(n)\rangle\}$
  for $x \in eventAssignments(n)$ **do**
    $E \leftarrow E \cup \{\langle \mu(n), \mu(x)\rangle\}$
return $\langle G, \mu\rangle$

Algorithm 6.2: Construct Graph From Submodels

Input: Graph $G = \langle V, E\rangle$, mapping $\mu : X \to V$, and SBML model $M = \langle S, R, L, P, N, C\rangle$
Output: Graph $G = \langle V, E\rangle$

`/* μ(x) returns the element mapped to x in μ */`

for $c \in C$ **do**
  $dnaComponents(v) \leftarrow$ Parse-Annotation($annotation(c)$)
  if $|dnaComponents(v)| = 0$ **then**
    $W \leftarrow$ Load-External-Model(c)
    $dnaComponents(v) \leftarrow$ Parse-Annotation($annotation(W)$)
  $V \leftarrow V \cup \{v\}$
  $\mu(c) \leftarrow v$
for $x \in X$ **do**
  for $c \in submodels(replacements(x)) \cup submodel(replacedBy(x))$ **do**
    $E \leftarrow E \cup \{\langle \mu(x), \mu(c)\rangle\}$
    $E \leftarrow E \cup \{\langle \mu(c), \mu(x)\rangle\}$
return G

genetic construct. A basic DFS starts at a node with no incoming edges and follows outgoing edges until a node with no outgoing edges is encountered. The DFS then backtracks until it can follow an edge not previously taken. This process repeats until all nodes in the graph have been encountered and ordered accordingly. However, because it is desirable for the ordering of nodes to correspond to a valid ordering of their stored DNA components, graph traversal cannot rely on a DFS alone, since the choices made by the DFS at branches in the graph are uninformed. For example, there are multiple possible node orderings that a DFS could produce when applied to the LacI inverter graph in Figure 6.1, only some of which correspond to a valid genetic construct.

Consequently, graph traveral requires an *arbiter* to determine when the DFS has chosen a path leading to an invalid genetic construct. For this purpose, graph traversal uses DFAs constructed from *regular expressions*. In computer science, a regular expression is a common means of specifying a *regular language*, or collection of patterns, while a DFA is a class of *state machine* used to match inputs against the regular language underlying its construction.

The default regular expression of iBioSim is $promoter, (RBS, CDS)^{+} terminator^{+}$, which specifies that DNA components are to be composed into genetic constructs containing a promoter followed by one or more RBS-CDS pairs and one or more terminators. At the start of graph traversal, either this default expression or a custom expression created by the user is translated into four DFAs: DC, DT, DS, and DP. During graph traversal, these DFAs process the SO types of the noncomposite DNA components stored at each node encountered. Whenever the DFAs determine that an invalid genetic construct would be composed, backtracking is initiated to find a valid solution if possible.

In particular, the DFA DC recognizes patterns of SO types corresponding to complete genetic constructs. The purpose of this DFA is to distinguish between complete and partial genetic constructs so that the user can be warned of incomplete constructs in accordance with their preferences. DC is constructed by first quantifying the user's regular expression for a complete genetic construct with the + (one or more) operator and then converting the quantified regular expression to a DFA using a method similar to that of McNaughton and Yamada [106]. For example, the regular expression $p(r, c)^{+}t^{+}$ would be quantified as $(p(r, c)^{+}t^{+})^{+}$ and then converted to the DFA shown in Figure 6.3. This DFA can recognize patterns such as p, r, c, r, c, t, t and p, r, c, t, p, r, c, t.

The DFAs DT and DS, on the other hand, recognize patterns for genetic constructs

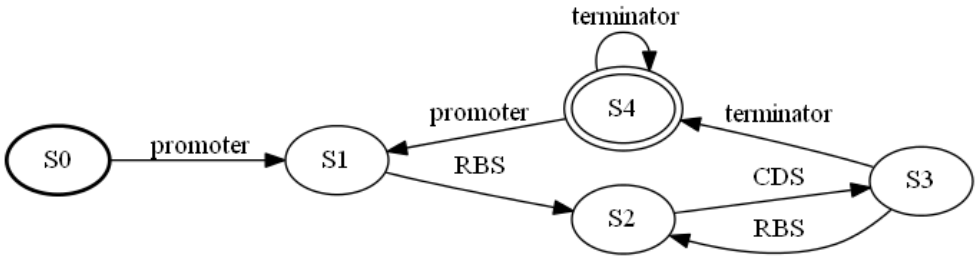

Figure 6.3: DFA *DC* translated from iBioSim's default regular expression for a complete genetic construct. State *S0* is the start state. If an input matches the label of an outgoing edge of the current state, then a transition is made from this state to the edge destination. If there is no match for an input, then the DFA rejects the entire input sequence. The DFA only accepts an input sequence if it ends in the accept state *S4*.

that end or start as complete genetic constructs, respectively. The main purpose of these DFAs is to determine if the genetic constructs inferred during graph traversal can be concatenated into a single strand of DNA. Because some genetic constructs may not be complete, care must be taken in connecting them to avoid introducing unintended *cis interactions*, such as a promoter regulating the transcription of a RBS-CDS pair that it should not regulate according to the graph structure. DT is constructed by first expanding the user's quantified expression to one that specifies all subpatterns of the quantified expression and then converting this expanded expression to a DFA as before. For example, the quantified expression $(p(r,c)^+t^+)^+$ would be expanded to $(p(r,c)^+t^+|(r,c|c)(r,c)^*t^+|t,t^*)(p(r,c)^+t^+)^*$ and then converted to a DFA that can recognize patterns such as r,c,r,c,t,t and r,c,t,p,r,c,t. Note that the $|$ operator separates alternatives, while the $*$ operator indicates that there is zero or more of the preceding symbol. DS is constructed in the same manner as DT, with the exception that the ordering of the user's quantified expression is reversed prior to its expansion. Hence, the resulting DFA can recognize patterns such as c,r,c,r,p and c,r,p,t,c,r,p. When DS is used during graph traversal, its input SO types are reversed to ensure proper construct recognition.

Lastly, the DFA DP recognizes patterns for partial genetic constructs with no restrictions on how they end or start. The primary purpose of this DFA is to determine if it is possible to order the nodes in a given portion of the graph such that their stored DNA components make up a valid partial or complete genetic construct. This DFA is also used in conjunction with DT to determine when the boundary of a partial or complete genetic

construct has been reached during graph traversal. DP is constructed by making a copy of DT and marking every one of its states as accepting. The resulting DFA can recognize patterns such as r, c, t, t, p and r, c, t, p, r, c, t, p.

Through the use of the above DFAs, several algorithms handle the process of graph traversal. Of these algorithms, Algorithm 6.3 composes the others to find a list of nodes VO with a particular property. Namely, when the DNA components stored at each node in VO are concatenated to form a composite DNA component, the result is one or more valid genetic constructs that have cis interactions consistent with the cause-and-effect graph structure. Algorithm 6.3 finds VO using three lists of nodes (VS, VC, and VL) and one set (VG) to manage a series of DFSs.

In particular, each node in the list VS is the starting point for a DFS that potentially results in a valid genetic construct. VS is initially populated with nodes that have no incoming edges, but is later expanded with nodes that follow the boundary of a genetic construct. Next, sequence VC stores the nodes that are currently under consideration during a DFS from a given starting node. Though only the first node in the list, VC_0, is considered at a time, VC may contain more than one node when the graph branches and it becomes necessary to keep track of the root nodes for the branches that are not immediately traversed in a depth-first fashion. When consideration of VC_0 is finished, it is replaced with its successors in the graph G. When VC becomes empty, it is repopulated using the next node from VS, which signifies the start of a new DFS. The list VL stores off each node removed from VC, thereby keeping a record of the order in which the nodes that are local to the current DFS have been visited. At the conclusion of a given DFS, VL is composed with the list of ordered nodes VO. VL is also added to the set of globally visited nodes VG, which keeps track of the nodes that have been visited by previous DFSs. VG is used at the end of Algorithm 6.3 to determine whether any nodes have yet to be visited. Finally, VL and VG are also used to prune previously visited nodes from VC and VS, respectively.

Primitive routines called by Algorithm 6.3 include the DFA routines *runDFA* and *reset*, as well as the node routine *types* and sequence routine *sub*. The routine *runDFA* takes a sequence of strings as input, transitions the given DFA to a new state accordingly, and returns a Boolean indicating whether the new state is accepting. When called with the partial construct DFA DP, this routine determines if an invalid genetic construct would be formed, in which case Algorithm 6.3 terminates and returns nothing.

Algorithm 6.3: Traverse Graph

Input: Graph $G = \langle V, E \rangle$, sequence of starting nodes VS, sequence of current nodes VC, sequence of local nodes VL, sequence of ordered nodes VO, set of global nodes VG, and sequence of DFAs $D = \langle DC, DP, DS, DT \rangle$

Output: Sequence of ordered nodes VO

```
1  while |VS| > 0 do
2      if |VC| = 0 then
3          VC ← VC||VS_0
4      while |VC| > 0 do
5          if runDFA(DP, types(VC_0)) then
6              runDFA(DT, types(VC_0))
7              VL ← VL||VC_0
8              VN ←Determine-Next-Nodes(VS, VC, VL, VG, DP, DT)
9              VC ← sub(VC, 1, |VC| − 1)
10             if |VN| = 0 then
11                 while |VC| > 0 ∧ VC_0 ∈ VL do
12                     VC ← sub(VC, 1, |VC| − 1)
13             else if |VN| = 1 then
14                 VC ← VN_0||VC
15             else if |VN| > 1 then
16                 return Traverse-Branches(G, VS, VC, VN, VL, VO, VG, D)
17         else
18             return ⟨⟩
19     reset(DP)
20     reset(DT)
21     if ¬Order-Local-Nodes(VL, VO, DS, DT) then
22         return ⟨⟩
23     VG ← VL
24     VL ← ⟨⟩
25     while |VS| > 0 ∧ VS_0 ∈ VG do
26         VS ← sub(VS, 1, |VS| − 1)
27 if |VG| < |V| then
28     return Traverse-Cycles(G, VO, VG, D)
29 else
30     return VO
```

The input for *runDFA* is supplied by *types*, which returns the SO types and strand signs for the noncomposite DNA components stored at a given node. The routine *reset*, on the other hand, returns the given DFA to its start state and is called at the end of each DFS or when the strand signs consumed by the DFA change polarity. Lastly, *sub* returns a subsequence of the given sequence that starts at the indicated index and has the indicated length. This routine is used during Algorithm 6.3 to effectively delete the first element in a sequence by replacing the sequence with a subsequence that starts at index one and has a length equal to that of the sequence minus one.

When called by Algorithm 6.3, Algorithm 6.4 determines which nodes succeeding the current node VC_0 are added to VC or VS for future consideration. The algorithm accomplishes this task by testing two sets of conditions for each successor node vn, provided that vn has not been visited during the current DFS (line 3). The first set of conditions checks for whether a successor node vn can and should be visited by another DFS, while the second set checks for whether vn should be the starting point for a new DFS.

In the first set of conditions, one of two conditions must be true for the current DFS to continue. Either the current DFS must be in the middle of a genetic construct (line 5) or, if vn has predecessors other than VC_0 and has never been visited, then all other predecessors of vn must have already been visited during the current DFS or other DFSs (line 6). The latter condition guarantees that vn is visited at least once, while the former prevents vn from being visited more than once unless it and its successors potentially store DNA components that appear in more than one genetic construct.

For example, two different promoters could promote the transcription of the same RBS-CDS-terminator combination. Even though this combination should appear in two different locations on DNA, it could annotate a single element of the model, such as a species that results from genetic production at both promoters. Hence, the node

Algorithm 6.4: Determine Next Nodes

Input: Sequence of starting nodes VS, sequence of current nodes VC, sequence of local nodes VL, set of global nodes VG, partial construct DFA DP, and terminal construct DFA DT
Output: Sequence of next nodes VN

```
1  VN ← ⟨⟩
2  for vn ∈ successors(VC_0) do
3      if vn ∉ VL then
4          VP ← predecessors(vn) \ VC_0
5          if (¬inStartState(DP) ∧ ¬inAcceptState(DT)) ∨
6          (vn ∉ VG ∧ (|VP| = 0 ∨ VP ⊆ VL ∨ VP ⊆ VG)) then
7              if |types(vn)| > 0 ∧
8              (¬checkDFA(DP, firstType(vn)) ∨
9              inStartState(DP) ∨
10             (inAcceptState(DT) ∧ ¬checkDFA(DT, firstType(vn)))) then
11                 VS ← VS||vn
12             else
13                 VN ← VN||vn
14 return VN
```

corresponding to this modeling element would have to be visited twice in order to obtain two separate instances of the RBS-CDS-terminator combination on DNA.

In the second set of conditions, if vn stores any DNA components, then there is a chance that it marks the beginning of a new genetic construct and should be added to VS instead of VC. For instance, if the first component at vn would lead to an invalid construct, then it is at least possible that vn starts a new genetic construct (line 8). This is the definitely true, however, if the current DFS is at the end of a genetic construct and the first DNA component stored at vn has a SO type other than that of a DNA component found at the end of a complete genetic construct (line 10). When neither of these conditions are true, vn is added to VC instead of VS.

Primitive routines called by Algorithm 6.4 that have not been described elsewhere include the DFA routines *inAcceptState*, *inStartState*, and *checkDFA*, and the node routine *firstType*. Routines *inAcceptState* and *inStartState* return Booleans indicating whether the given DFA is in its accept state or start state, respectively. The routine *checkDFA* functions similarly to *runDFA*, but only returns a Boolean indicating whether the given DFA would be in an accept state after processing a list of input strings and does not actually transition the DFA to a post-input state. Finally, *firstType* returns the SO type of the first DNA component in a list stored at the given node.

If Algorithm 6.4 determines that multiple nodes are to be added to VC, then Algorithm 6.5 handles the process of finding the order in which these next nodes should be added to VC so that a valid genetic construct is obtained. The algorithm achieves this task by permuting the sequence of next nodes VN and recursively calling Algorithm 6.3

Algorithm 6.5: Traverse Branches

Input: Graph $G = \langle V, E \rangle$, sequence of starting nodes VS, sequence of current nodes VC, sequence of next nodes VN, sequence of local nodes VL, sequence of ordered nodes VO, set of global nodes VG, and sequence of DFAs $D = \langle DC, DP, DS, DT \rangle$

Output: Sequence of ordered nodes VBO

```
1 for i ← 0...|VN| − 1 do
2     for j ← i...|VN| − 1 do
3         VN ← VN||VN_i
4         VN ← sub(VN, 0, i)||sub(VN, i + 1, |VN| − i)
5         VBO ←Traverse-Graph(G, copy VS, VN || copy VC, copy VL, copy VO,
6         copy VG, copy D)
7         if |VBO| > 0 then
8             return VBO
9 return ⟨⟩
```

with copies of relevant data structures (in case the call fails and the next permutation must be tried).

When the current DFS terminates, Algorithm 6.6 handles the process of composing the sequence of nodes ordered by the DFS (VL) with the sequence of nodes ordered by previous DFSs (VO). The algorithm solves this problem using the starting and terminal construct DFAs DS and DT to determine whether VL should be concatenated at the beginning or end of VO. While complete genetic constructs can generally be concatenated in any order without introducing cis-interactions between DNA components within these constructs, care must be taken when composing partial genetic constructs. For instance, when composing a single promoter with a RBS-CDS-terminator combination that it should not cis-regulate, the promoter must be placed immediately after the RBS-CDS-terminator combination.

The final algorithm of sequence generation is responsible for traversing any nodes that are unvisited by the primary graph traversal, that is, nodes that belong to isolated, strongly connected subgraphs. These are cyclic portions of the graph that lack nodes with zero incoming edges to serve as natural starting points for a traversal and that are not reachable from other portions of the graph. The algorithm solves the problem of visiting these cycles by first identifying within them potential starting nodes for which the first stored DNA component shares a SO type with the first component in a complete genetic construct. Next, the algorithm tries these starting nodes one at a time by recursively calling Algorithm 6.3 with copies of relevant data structures in case a given starting node does not produce a valid solution. These copies include those of the graph, DFAs, and node collections. If no valid solutions can be produced in this way, then all remaining

Algorithm 6.6: Order Local Nodes

Input: Sequence of local nodes VL, sequence of ordered nodes VO, starting construct DFA DS, and terminal construct DFA DT
Output: Sequence of ordered nodes VO

```
if |types(VL)| = 0 ∨ |types(VO)| = 0 ∨
(checkDFA(DS, types(VL)) ∧ checkDFA(DT, types(VO))) then
    VO ← VO||VL
else if checkDFA(DS, types(VO)) ∧ checkDFA(DT, types(VL)) then
    VO ← VL||VO
else
    return FALSE
return TRUE
```

nodes within the cycles are tried as starting nodes in the aforementioned manner. The pseudocode for this procedure can be found in Algorithm 6.7.

6.3 Summary

This chapter describes a methodology for sequence generation that is implemented in the GDA software tool iBioSim. This methodology builds on the results of the first sequence generation tool, MoSeC, by extending the processes of graph construction and graph traversal which form the basis for sequence generation. In particular, sequence generation in iBioSim enhances graph construction to handle a greater variety of core SBML elements, such as events and parameters, and hierarchical SBML models built via the hierarchical model composition package. This methodology also supplements graph traversal with an arbiter in the form of a DFA that is constructed from a regular expression and recognizes patterns of SO types, thereby providing users of iBioSim with a means of programming sequence generation to recognize a greater variety of genetic constructs. By allowing the user to customize the source of arbitration for graph traversal, sequence generation in iBioSim is more flexible than rule-based sequence generation in MoSeC when it comes to generating composite DNA components that represent different types of genetic construct.

Nevertheless, there is one major assumption that dictates the class of genetic constructs that it is possible to generate via sequence generation in iBioSim. Namely, sequence generation assumes that the partial ordering of the modeling elements that describe the

Algorithm 6.7: Traverse Cycles

Input: Graph $G = \langle V, E \rangle$, sequence of ordered nodes VO, set of global nodes VG, and sequence of DFAs $D = \langle DC, DP, DS, DT \rangle$
Output: Sequence of ordered nodes VCO

```
1  VC ← V \ VG
2  VS ← ∅
3  for vc ∈ VC do
4      if firstType(vc) ⊆ startingTypes(DC) then
5          VS ← VS ∪ vc
6  for i ← 0...1 do
7      for vs ∈ VS do
8          VCO ←Traverse-Graph(G, ⟨vs⟩, ⟨⟩, ⟨⟩, copy VO, copy VG, copy D)
9          if |VCO| > 0 then
10             return VCO
11     VS ← VC \ VS
12 return ⟨⟩
```

behavior of a genetic construct, as inferred from the cause-and-effect relationships between these elements, is equivalent to the partial ordering of the DNA components that annotate these elements, from which a total ordering or sequence of these DNA components that is a valid genetic construct may be determined. By means of this assumption, however, sequence generation can be used to efficiently infer composite DNA components from the SBOL-annotated SBML models produced by genetic technology mapping. These composite components can then serve to inform subsequent planning steps for physical construction.

Lastly, this approach takes advantage of regular expressions for pattern recognition. This decision was made based on previous research showing that *context-free languages* comprised of types for DNA components can be used to verify synthetic genetic constructs [46]. In the same paper, it was speculated that simpler regular languages specified by regular expressions would be sufficient for this task as well. The iBioSim approach to sequence generation has shown that this is entirely possible. In the future, however, it could prove interesting to perform sequence generation with the aid of languages and associated automata that are more expressive than regular and context-free languages.

CHAPTER 7

CONCLUSIONS

This chapter concludes the dissertation with a summary of its contents and potential areas for future research. In particular, Section 7.1 discusses the contributions of this dissertation and their significance, while Section 7.2 outlines possible extensions to SBOL and iBioSim's GDA workflow and discusses future validation of this workflow in the lab.

7.1 Summary

The primary subject of this dissertation is a workflow for genetic technology mapping and the techniques and standards that make it possible. This workflow is implemented in the software tool iBioSim and provides a fully automated means to generate a SBOL/SBML design library from a collection of SBOL modules, map a SBML specification against the design library, and generate a linearized SBOL/SBML design from the result of the mapping. In this way, the workflow demonstrates how existing techniques and concepts from electrical and computer engineering can be adapted to a biological context and facilitate genetic design.

Perhaps most significantly, this workflow is wholly built on a framework of actively developed standards for describing genetic structure and function, unlike all other such workflows of which we are aware. Furthermore, the SBOL standard (in particular the proposed data model for its next version) forms no small part of the contributions of this dissertation. If nothing else, this dissertation should serve as an example of what is possible in terms of GDA research and software development while working within the constraints of standards, as well as the benefits to be gained from working with them, such as the exchange of data between different software tools.

Ultimately, it is hoped that this dissertation and its contributions can aid in the eventual formation of an ecosystem for standard-compliant GDA tools. In such an ecosystem, sequence editing tools and modeling tools can be used to create and edit standardized descriptions of genetic structure and function, while design composition tools and genetic

technology mapping can be used to connect, compose, and map between these standardized descriptions. With such an ecosystem in place, synthetic biologists could use GDA software tools to more efficiently design genetic systems and leverage the efforts of collaborators and/or publicly available data sets without the loss of a designer's intent in the move between different tools.

7.2 Future Research

This section discusses areas of future research related to SBOL, model generation, genetic technology mapping, and sequence generation as presented in this dissertation.

7.2.1 SBOL

Chapter 3 describes a proposal for the next version of SBOL. Even once this proposal is accepted, there remains research to standardize the representation of information required for the physical construction and successful deployment of a genetic design, such as *liquid handling robot instructions* and environmental/host context. The former is necessary to enable efficient, automated construction of genetic designs across different platforms, while the latter is necessary to ensure that all components and modules of a genetic design are compatible with the intended host.

Furthermore, while the proposed data model is capable of encoding qualitative descriptions of genetic function, it could be extended to provide a firmer basis for generating quantitative models that conform to the same basic data set. Since one of the goals of SBOL is to avoid reproducing existing standards for quantitative models, perhaps the most that can be done in this area is to enable interactions to refer to quantitative parameters. These parameters can then inform the generation of a variety of different models.

7.2.2 Model Generation

As noted at the end of Chapter 4, the approach to model generation presented in this dissertation captures only one of many possible mappings between SBOL and various formalisms for quantitative modeling of biology. In particular, this dissertation specifies a single mapping between SBOL and a specific type of genetic circuit model written in SBML, one that is especially suited to genetic technology mapping and sequence generation in iBioSim. SBML, however, is capable of specifying other types of models. In addition, there exist other modeling standards besides SBML, including those expressly developed

for modeling biology, such as CellML [15], and programming languages commonly applied to modeling biology, such as MATLAB [39] and Python.

One of the goals of SBOL is to serve as a common means of qualitatively describing genetic function, such that many different quantitative models that conform to a given SBOL design can be developed for different design tasks. For example, a SBOL design may include qualitative descriptions of both metabolic and genetic regulatory networks, which are typically modeled using algebraic and differential equations, respectively. Accordingly, model generation must grow to accommodate other possible mappings between descriptions of qualitative genetic function written in SBOL and descriptions of quantitative genetic function written in other languages.

In order to enable this growth, however, further research is required to formalize the process of model generation in relation to SBOL. Potential by-products of this research could include methods for automatically comparing quantitative models across different standards and GDA tools that enable users to create and store new mappings from SBOL to more quantitative modeling standards. Such tools and methods would help to democratize model generation and involve larger segments of the synthetic biology community in its long-term growth.

7.2.3 Genetic Technology Mapping

Chapter 5 presents one the primary results of this dissertation, a DAG-based approach to genetic technology mapping that uses the mathematical framework of a cost function to guide the search for optimal and near-optimal solutions. Currently, this approach is only applicable to specifications for combinational genetic circuits, in which the steady-state outputs of a circuit are dependent on the steady-state inputs alone. It cannot be applied to the specifications for sequential genetic circuits, in which the output and/or intermediate signals are fed back to influence previous signals. To handle these cyclic specifications would be a very important step for GDA, as it would enable GDA to move beyond the automated design of combinational logic and into the realm of *asynchronous state machines* capable of full-fledged computation.

In the near future, specifications for sequential genetic circuits could be handled by DAG-based technology mapping if its partitioning step is extended with existing algorithms for efficiently cutting cyclic graphs into DAGs. Still, this extended approach would also require that genetic circuit parameters other than length in base pairs, such as signal magnitudes and time delays, are taken into account to guide the search for

solutions that have both the desired steady states and transient behavior between these states. In light of these considerations, further research is necessary to determine the most effective strategies for efficiently solving specifications for both combinational and sequential genetic circuits.

7.2.4 Sequence Generation

As mentioned in Chapter 6, it could prove interesting to further generalize sequence generation with the aid of languages and their associated automata that are more expressive than regular languages and DFAs. These could include, for example, context-free languages and *Deterministic Pushdown Automata* (DPA). Such a framework would in theory allow the generation of genetic constructs that match more sophisticated patterns than those encoded by regular expressions.

Another potential avenue of future research is in regards to sequence generation's use of SBOL. Currently, the approach described in this dissertation may only be applied to DNA components with completely specified sequences. In the future, it could prove useful to make full use of SBOL's capabilities for partial design and generate composite DNA components that lack DNA sequences, but assert the relative ordering of their subcomponents via precedes relationships. Lastly, once the data model for the next version of SBOL proposed in Chapter 3 is implemented, sequence generation could be extended to handle RNA components in addition to DNA components.

7.2.5 Workflow Validation

As the GDA workflow of this dissertation is extended to handle data that characterize genetic circuits (data such as input and output signal magnitudes) this workflow should be applied to design libraries backed by real-world genetic circuits. In this way, the genetic circuit designs produced by the workflow can be physically implemented and experimentally characterized over a range of inputs to validate iBioSim's approach to genetic technology mapping. Other approaches, such as SBROME [41] and Matchmaker [34], have already built up and leveraged realistic design libraries during genetic technology mapping, but so far only Matchmaker, as part of the *Tool-Chain to Accelerate Biological Engineering* (TASBE) [107], has published experimental validation of its output genetic circuit designs.

The lack of validation of other GDA workflows is partly due to the absence of characterization methods that use normalized measures of genetic signals, thereby hindering the comparison of experimental results for different genetic circuits. Recently, however,

there has been research to address this problem that takes advantage of calibrated flow cytometry to construct input/output transfer curves [108]. Consequently, the time is ripe to validate the results of genetic technology mapping tools and their associated workflows. Finally, in the course of such workflow validation, researchers will undoubtedly discover needs for other GDA tools to help bridge the gap between abstract design and physical implementation of genetic circuits. For instance, it could prove interesting to develop GDA tools for identifying gaps in design libraries, then suggesting genetic components or subcircuits that could be developed to fill these gaps.

REFERENCES

[1] A. Gutmann *et al.*, "New directions: the ethics of synthetic biology and emerging technologies," Presidential Commission for the Study of Bioethical Issues, Washington, D.C., Rep. 1, Dec. 2010.

[2] D.-K. Ro *et al.*, "Production of the antimalarial drug percursor artemisinic acid in engineered yeast," *Nature*, vol. 440, pp. 940–943, 2006.

[3] S. Atsumi *et al.*, "Metabolic engineering of Escherichia coli for 1-butanol production," *Metab. Eng.*, vol. 10, pp. 305–311, 2008.

[4] E. J. Steen *et al.*, "Metabolic engineering of Saccharomyces cerevisiae for the production of n-butanol," *Microb. Cell Fact.*, vol. 7, no. 36, doi:10.1186/1475-2859-7-36, 2008.

[5] F. Zhang, J. M. Carothers, and J. D. Keasling, "Design of a dynamic sensor-regulator system for production of chemicals and fuels derived from fatty acids," *Nat. Biotechnol.*, vol. 30, pp. 354–359, 2012.

[6] G. M. Brazil *et al.*, "Construction of a rhizosphere pseudomonad with potential to degrade polychlorinated biphenyls and detection of bph gene expression in the rhizosphere," *Appl. Environ. Microb.*, vol. 61, no. 5, pp. 1946–1952, 1995.

[7] I. Cases and V. de Lorenzo, "Genetically modified organisms for the environment: stories of success and failure and what we have learned from them," *Int. Microbiol.*, vol. 8, pp. 213–222, 2005.

[8] J. C. Anderson, E. J. Clarke, and A. P. Arkin, "Environmentally controlled invasion of cancer cells by engineering bacteria," *J. Mol. Biol.*, vol. 355, pp. 619–627, 2006.

[9] D. Endy, "Foundations for engineering biology," *Nature*, vol. 438, pp. 449–453, 2005.

[10] A. Arkin, "Setting the standard in synthetic biology," *Nat. Biotechnol.*, vol. 26, pp. 771–774, 2008.

[11] J. Peccoud *et al.*, "Essential information for synthetic DNA sequences," *Nat. Biotechnol.*, vol. 29, p. 22, 2011.

[12] H. S. Bilofsky and B. Christian, "The GenBank genetic sequence data bank," *Nucleic Acids Res.*, vol. 16, no. 5, pp. 1861–1863, 1988.

[13] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *P. Natl. Acad. Sci. USA*, vol. 85, pp. 2444–2448, 1988.

[14] M. Hucka *et al.*, "The Systems Biology Markup Language (SBML): a medium for representation and exchange of biochemical network models," *Bioinformatics*, vol. 19, no. 4, pp. 524–531, 2003.

[15] W. J. Hedley, M. R. Nelson, D. P. Bellivant, and P. F. Nielsen, "A short introduction to CellML," *Philos. T. Roy. Soc. A*, vol. 359, no. 1783, pp. 1073–1089, 2001.

[16] iGEM Registry, 2003; http://parts.igem.org.

[17] D. Densmore and S. Hassoun, "Design automation for synthetic biological systems," *IEEE Des. Test Comput.*, vol. 29, no. 3, pp. 7–20, 2012.

[18] S. M. Richardson, S. J. Wheelan, R. M. Yarrington, and J. D. Boeke, "GeneDesign: rapid, automated design of multikilobase synthetic genes," *Genome Res.*, vol. 16, pp. 550–556, 2006.

[19] A. Villalobos, J. Ness, C. Gustafsson, J. Minshull, and S. Govindarajan, "Gene Designer: a synthetic biology tool for constructing artificial DNA segments," *BMC Bioinformatics*, vol. 7, p. 285, 2006.

[20] P. Umesh, F. Naveen, C. Rao, and A. Nair, "Programming languages for synthetic biology," *Syst. Synth. Biol.*, vol. 4, pp. 265–269, 2010.

[21] G. Wu, N. Bashir-Bello, and S. J. Freeland, "The Synthetic Gene Designer: a flexible web platform to explore sequence manipulation for heterologous expression," *Protein Expres. Purif.*, vol. 47, pp. 441–445, 2006.

[22] T. S. Ham, Z. Dmytriv, H. Plahar, J. Chen, N. J. Hillson, and J. D. Keasling, "Design, implementation and practice of JBEI-ICE: an open source biological part registry platform and tools," *Nucleic Acids Res.*, vol. 40, doi: 10.1093/nar/gks531, 2012.

[23] A. Cornish-Bowden, "Nomenclature for incompletely specified bases in nucleic acid sequences: recommendations 1984," *Nucleic Acids Res.*, vol. 13, pp. 3021–3030, 1985.

[24] G. Rodrigo, J. Carrera, and A. Jaramillo, "Asmparts: assembly of biological model parts," *Syst. Synth. Biol.*, vol. 1, pp. 167–170, 2007.

[25] S. Kosuri, J. R. Kelly, and D. Endy, "TABASCO: a single molecule, base-pair resolved gene expression simulator," *BMC Bioinformatics*, vol. 8, p. 480, 2007.

[26] J. Goler, "BioJADE: a design and simulation tool for synthetic biological systems," M.S. thesis, Dept. Elect. Comput. Eng., MIT, Cambridge, MA, 2004.

[27] S. Hoops *et al.*, "COPASI: a COmplex PAthway SImulator," *Bioinformatics*, vol. 22, pp. 3067–3074, 2006.

[28] A. Funahashi, Y. Matsuoka, A. Jouraku, M. Morohashi, N. Kikuchi, and H. Kitano, "CellDesigner 3.5: a versatile modeling tool for biochemical networks," *Proc. IEEE*, vol. 96, no. 8, pp. 1254–1265, 2008.

[29] L. Bilitchenko *et al.*, "Eugene - a domain specific language for specifying and constraining synthetic biological parts, devices, and systems," *PLoS ONE*, vol. 6, no. 4, doi: 10.1371/journal.pone.0018882, 2011.

[30] J. Chen, D. Densmore, T. S. Ham, J. D. Keasling, and N. J. Hillson, "DeviceEditor visual biological CAD canvas," *J. Biol. Eng.*, vol. 6, no. 1, p. 1, 2012.

[31] M. Pedersen and A. Phillips, "Towards programming languages for genetic engineering of living cells," *J. Roy. Soc. Interface*, vol. 6, no. Suppl 4, pp. S437–S450, 2009.

[32] Y. Cai, M. L. Wilson, and J. Peccoud, "GenoCAD for iGEM: a grammatical approach to the design of standard-compliant constructs," *Nucleic Acids Res.*, vol. 38, pp. 2637–2644, 2010.

[33] C. Madsen, C. Myers, T. Patterson, N. Roehner, J. Stevens, and C. Winstead, "Design and test of genetic circuits using iBioSim," *IEEE Des. Test Comput.*, vol. 29, pp. 32–39, 2012.

[34] F. Yaman, S. Bhatia, A. Adler, D. Densmore, and J. Beal, "Automated selection of synthetic biology parts for genetic regulatory networks," *ACS Synth. Biol.*, vol. 1, no. 8, pp. 332–344, 2012.

[35] M. T. Cooling *et al.*, "Standard Virtual Biological Parts: a repository of modular modeling components for synthetic biology," *Bioinformatics*, vol. 26, no. 7, pp. 925–931, 2010.

[36] G. Misirli *et al.*, "Model annotation for synthetic biology: automating model to nucleotide sequence conversion," *Bioinformatics*, vol. 27, no. 7, pp. 973–979, 2011.

[37] M. A. Marchisio and J. Stelling, "Automatic design of digital synthetic gene ciruits," *PLoS Comput. Biol.*, vol. 7, doi: 10.1371/journal.pcbi.1001083, 2011.

[38] S. Mirschel, K. Steinmetz, M. Rempel, M. Ginkel, and E. D. Gilles, "ProMoT: modular modeling for systems biology," *Bioinformatics*, vol. 25, no. 5, pp. 687–689, 2009.

[39] MATLAB, *version 8.3 (R2014a)*. Natick, MA: The MathWorks Inc., 2014.

[40] J. Beal, T. Lu, and R. Weiss, "Automatic compilation from high-level biologically-oriented programming language to genetic regulatory networks," *PLoS ONE*, vol. 6, doi: 10.1371/journal.pone.0022490, 2011.

[41] L. Huynh, A. Tsoukalas, M. Koppe, and I. Tagkopoulos, "SBROME: a scalable optimization and module matching framework for automated biosystems design," *ACS Synth. Biol.*, vol. 2, no. 5, pp. 1073–1089, 2013.

[42] A. D. Hill, J. R. Tomshine, E. M. Weeding, V. Sotirpoulos, and Y. N. Kaznessis, "SynBioSS: the synthetic biology modeling suite," *Bioinformatics*, vol. 24, pp. 2551–2553, 2008.

[43] D. Chandran, F. T. Bergmann, and H. M. Sauro, "TinkerCell: modular CAD tool for synthetic biology," *J. Biol. Eng.*, vol. 3, no. 1, p. 19, 2009.

[44] M. Galdzicki *et al.*, "The synthetic biology open language (sbol) provides a community standard for communicating designs in synthetic biology," *Nat. Biotechnol.*, vol. 32, pp. 545–550, 2014.

[45] L. P. Smith *et al.*, "Hierarchical model composition," SBML, SBML Level 3 Comp Package Specification, Nov. 2013.

[46] Y. Cai, B. Hartnett, C. Gustafsson, and J. Peccoud, "A syntactic model to design and verify synthetic genetic constructs derived from standard biological parts," *Bioinformatics*, vol. 23, no. 20, pp. 2760–2767, 2007.

[47] N. Roehner *et al.*, "Proposed data model for the next version of the Synthetic Biology Open Language," *ACS Synth. Bio.*, doi:10.1021/sb500176h, 2014.

[48] N. Roehner and C. J. Myers, "Directed acyclic graph-based technology mapping of genetic circuit models," *ACS Synth. Biol.*, doi:10.1021/sb400135t, 2014.

[49] N. Roehner and C. J. Myers, "A methodology to annotate Systems Biology Markup Language Models with the Synthetic Biology Open Language," *ACS Synth. Biol.*, vol. 3, pp. 57–66, 2014.

[50] T. S. Gardner, C. R. Cantor, and J. J. Collins, "Construction of a genetic toggle switch in escherichia coli," *Nature*, vol. 403, pp. 339–342, 2000.

[51] A. Fire, S. Xu, M. K. Montgomery, S. A. Kostas, S. E. Driver, and C. C. Mello, "Potent and specific genetic interference by double-stranded RNA in Caenorhabditis elegans," *Nature*, vol. 391, pp. 806–811, 1998.

[52] S. Bhatia and D. Densmore, "Pigeon: a design visualizer for synthetic biology," *ACS Synth. Biol.*, vol. 2, no. 6, pp. 348–350, 2013.

[53] J. Quinn *et al.*, "Synthetic Biology Open Language Visual (SBOL Visual), Version 1.0.0," SBOL, BBF RFC 93, Mar. 2013, doi: 1721.1/78249.

[54] T. L. Deans, C. R. Cantor, and J. J. Collins, "A tunable genetic switch based on rnai and repressor proteins for regulating gene expression in mammalian cells," *Cell*, vol. 130, pp. 363–372, 2007.

[55] J. C. Anderson, C. A. Voigt, and A. P. Arkin, "Environmental signal integration by a modular and gate," *Mol. Syst. Biol.*, doi: 10.1038/msb4100173, 2007.

[56] B. Wang, R. I. Kitney, N. Joly, and M. Buck, "Engineering modular and orthogonal genetic logic gates for robust digital-like synthetic biology," *Nat. Commun.*, vol. 2, p. 508, 2013.

[57] A. Tamsir, J. J. Tabor, and C. A. Voigt, "Robust multicellular computing using genetically encoded nor gates and chemical wires," *Nature*, vol. 469, pp. 212–215, 2011.

[58] T. S. Moon *et al.*, "Construction of a genetic multiplexer to toggle between chemosensory pathways in escherichia coli," *J. Mol. Biol.*, vol. 406, pp. 215–227, 2011.

[59] L. Pasotti, M. Quattrocelli, D. Galli, M. G. D. Angelis, and P. Magni, "Multiplexing and demultiplexing logic functions for computing signal processing tasks in synthetic biology," *Biotechnol. J.*, vol. 6, pp. 784–795, 2011.

[60] M. B. Elowitz and S. Leibler, "A synthetic oscillatory network of transcriptional regulators," *Nature*, vol. 403, pp. 335–338, 2000.

[61] G. Boole, *The Laws of Thought*. Buffalo, New York: Prometheus Books, 1854/2003.

[62] C. C. Guet, M. B. Elowitz, W. Hsing, and S. Leibler, "Combinatorial synthesis of genetic networks," *Science*, vol. 296, pp. 1466–1470, 2002.

[63] T. Ellis, X. Wang, and J. J. Collins, "Diversity-based, model-guided construction of synthetic gene networks with predicted functions," *Nat. Biotechnol.*, vol. 27, no. 5, pp. 465–471, 2009.

[64] H. M. Salis, E. A. Mirsky, and C. A. Voigt, "Automated design of synthetic ribosome binding sites to control protein expression," *Nat. Biotechnol.*, vol. 27, no. 10, pp. 946–950, 2009.

[65] X.-J. Feng, S. Hooshangi, D. Chen, G. Li, R. Weiss, and H. Rabitz, "Optimizing genetic circuits by global sensitivity analysis," *Nature*, vol. 87, pp. 2195–2202, 2004.

[66] S. Basu, Y. Gerchman, C. H. Collins, F. H. Arnold, and R. Weiss, "A synthetic multicellular system for programmed pattern formation," *Nature*, vol. 434, pp. 1130–1134, 2005.

[67] Y. Yokobayashi, R. Weiss, and F. H. Arnold, "Directed evolution of a genetic circuit," *P. Natl. Acad. Sci. USA*, vol. 99, no. 26, pp. 16587–16591, 2002.

[68] A. Hill, "The possible effect of the aggregation of the molecules of haemoglobin," *J. Physiol.*, vol. 40, pp. 4–7, 1910.

[69] B. C. Goodwin, "Oscillatory behavior of enzymatic control processes," *Adv. Enzyme. Reg.*, vol. 3, pp. 425–439, 1965.

[70] J. S. Griffith, "Mathematics of cellular control processes. I. negative feedback to one gene," *J. Theor. Biol.*, vol. 20, pp. 202–208, 1968.

[71] G. Yagil and E. Yagil, "On the relation between effector concentration and the rate of induced enzyme synthesis," *Biophys. J.*, vol. 11, pp. 11–27, 1971.

[72] J. Tyson and H. G. Othmer, "The dynamics of feedback control circuits in biochemical pathways," *Prog. Theor. Biol.*, vol. 5, pp. 2–62, 1978.

[73] R. D. Bliss, P. R. Painter, and A. G. Marr, "Role of feedback inhibition in stabilizing the classical operon," *J. Theor. Biol.*, vol. 97, pp. 177–193, 1982.

[74] H. Smith, "Oscillations and multiple steady states in a cyclic gene model with repression," *J. Math. Biol.*, vol. 25, pp. 169–190, 1987.

[75] E. Plahte, T. Mestl, and S. W. Omholt, "A methodological basis for description and analysis of systems with complex switch-like interactions," *J. Math Biol.*, vol. 36, pp. 321–348, 1998.

[76] F. Jacob and J. Monod, "On the regulation of gene activity," *Cold Spring Harb. Sym.*, vol. 26, pp. 193–211, 1961.

[77] C. Rao, D. Wolf, and A. Arkin, "Control, exploitation and tolerance of intracellular noise," *Nature*, vol. 420, pp. 231–237, 2002.

[78] M. Elowitz, A. Levine, F. Siggia, and P. Swain, "Stochastic gene expression in a single cell," *Science*, vol. 297, p. 1183, 2002.

[79] J. Stricker *et al.*, "A fast, robust and tunable synthetic gene oscillator," *Nature*, vol. 456, pp. 516–519, 2008.

[80] M. Hucka *et al.*, "The Systems Biology Markup Language (SBML): language specification for Level 3 Version 1 Core," SBML, SBML Level 3 Core Specification, Oct. 2010.

[81] V. Chelliah, C. Laibe, and N. L. Novere, "BioModels Database: a repository of mathematical models of biological processes," *Methods Mol. Biol.*, vol. 1021, pp. 189–199, 2013.

[82] N. Barker, C. Myers, and H. Kuwahara, "Learning genetic regulatory network connectivity from time series data," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 8, no. 1, pp. 152–165, 2011.

[83] H. Kuwahara, C. Myers, N. Barker, M. Samoilov, and A. Arkin, "Automated abstraction methodology for genetic regulatory networks," in *Transactions on Computational Systems Biology VI*, C. Priami and G. Plotkin, Eds., Berlin: Springer, 2006, pp. 150-175.

[84] C. Madsen, C. J. Myers, N. Roehner, C. Winstead, and Z. Zhang, "Utilizing stochastic model checking to analyze genetic circuits," in *2012 IEEE Symposium on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, San Diego, CA, 2012, pp. 379-386.

[85] M. Galdzicki *et al.*, "Synthetic Biology Open Language (SBOL) Version 1.1.0," SBOL, BBF RFC 87, Oct. 2012, doi: 1721.1/73909.

[86] T. Bray, J. Paoli, C. M. Sperber-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," W3C, W3C Recommendation 26 November 2008, Nov. 2008.

[87] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide, 2nd Edition*. Boston, MA: Addison-Wesley, 2005.

[88] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): generic syntax," The Internet Society, IETF RFC 3986, Jan. 2005.

[89] K. Eilbeck *et al.*, "The Sequence Ontology: a tool for the unification of genome annotations," *Genome Biol.*, vol. 6, no. 5, p. R44, 2005.

[90] F. Gandon and G. Schreiber, "RDF 1.1 XML syntax," W3C, W3C Recommendation 25 February 2014, Feb. 2014.

[91] J. Hastings *et al.*, "The ChEBI reference database and ontology for biologically relevant chemistry: enhancements for 2013," *Nucleic Acids Res.*, vol. 41, pp. D456–D463, 2013.

[92] D. Waltemath, A. Zhukova, M. Swat, Y. Lefranc, J.-O. Vik, and N. L. Novere, (2014, April 1). *Mathematical Modelling Ontology* [Online]. Available: http://sourceforge.net/projects/mamo-ontology/.

[93] J. Beal *et al.*, "Model-driven engineering of gene expression from RNA replicons," *ACS Synth. Biol.*, doi: 10.1021/sb500173f, 2014.

[94] S. Kiani *et al.*, "CRISPR transcriptional repression devices and layered circuits in mammalian cells," *Nat. Methods*, doi:10.1038/NMETH.2969, 2014.

[95] P. Horvath and R. Barrangou, "CRISPR/Cas, the immune system of bacteria and archaea," *Science*, vol. 327, no. 5962, pp. 167–170, 2010.

[96] I. Frolov, R. Hardy, and C. M. Rice, "Cis-acting rna elements at the 5' end of Sindbis virus genome rna regulate minus- and plus-strand RNA synthesis," *RNA*, vol. 7, pp. 1638–1651, 2001.

[97] T. B. Foundation, (2014, June 20). *The BioBricks Foundation: RFC* [Online]. Available: http://openwetware.org/wiki/The_BioBricks_ Foundation:RFC.

[98] G. E. Briggs and J. B. S. Haldane, "A note on the kinetics of enzyme action," *Biochem. J.*, vol. 19, pp. 338–339, 1925.

[99] C. V. Rao and A. P. Arkin, "Stochastic chemical kinetics and the quasi-steady-state assumption: application to the Gillespie algorithm," *J. Phys. Chem.-US*, vol. 118, no. 11, pp. 4999–5010, 2003.

[100] C. J. Myers, *Engineering Genetic Circuits.* Boca Raton, FL: Chapman and Hall/CRC, 2009.

[101] M. Courtot *et al.*, "Controlled vocabularies and semantics in systems biology," *Mol. Syst. Biol.*, vol. 7, p. 543, 2011.

[102] K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," in *DAC '87 Proceedings of the 24th ACM/IEEE Design Automation Conference*, ACM: New York, 1987, pp. 341-347.

[103] J. J. Tabor *et al.*, "A synthetic genetic edge detection program," *Cell*, vol. 137, no. 7, pp. 1272–1281, 2009.

[104] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Commun. ACM*, vol. 18, pp. 333–340, 1975.

[105] C. Hoare, "Quicksort," *Comput. J.*, vol. 5, no. 1, pp. 10–16, 1962.

[106] R. McNaughton and H. Yamada, "Regular expression and state graphs for automata," *IEEE Trans. Comput.*, vol. EC-9, no. 1, pp. 39–47, 1960.

[107] J. Beal *et al.*, "An end-to-end workflow for engineering of biological networks from high-level specifications," *ACS Synth. Biol.*, vol. 1, no. 8, pp. 317–331, 2012.

[108] J. Beal, R. Weiss, F. Yaman, N. Davidsohn, and A. Adler, "A method for fast, high-precision characterization of synthetic biology devices," MIT CSAIL, MIT CSAIL Tech. Rep. 2012-008, 2012, doi: 1721.1/69973.