RSMLab: A web-based tool for recombinase-based state machine design and visualization

by

Guillaume Georges Kugener

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Computer Science and Engineering
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Massachusetts Institute of Technology 2017. All rights reserved.

Author ................................................................. Department of Electrical Engineering and Computer Science

June 9, 2017

Certified by ............................................................. Timothy K. Lu

Associate Professor

Thesis Supervisor

Accepted by ............................................................. Christopher J. Terman

Chairman, Master of Engineering Thesis Committee
RSMLab: A web-based tool for recombinase-based state machine design and visualization

by

Guillaume Georges Kugener

Submitted to the Department of Electrical Engineering and Computer Science on June 9, 2017, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

Biological state machines have the potential to enable a wide range of applications but until recently have been challenging to implement experimentally. To overcome this challenge, we described a scalable strategy for assembling biological state machines using recombinases. This platform enables the implementation of biological state machines with arbitrary behaviors, but the manual design of such state machines is increasingly challenging with increasing complexities. Here, we introduce RSMLab, an intuitive web-based application for creating circuits that implement state-dependent logic in living cells using our scalable state-machine framework. Through a graphical user interface, RSMLab users choose a desired state diagram, define arbitrary genes, and specify whether those genes are on or off in each state. RSMLab then returns a visualization of possible gene circuits that correspond to the user specifications. We use the RSMLab algorithm and demonstrate the circuit design process using examples of two-input, five-state and three-input, sixteen-state gene regulation programs. With the help of RSMLab, researchers can program state-dependent logic to study and program the way that cells respond to combinational and temporally distributed chemical events, without needing to be expert gene circuit engineers. We envision that biology-focused design software such as RSMLab will enable the faster, more reliable, and more accessible creation of DNA-encoded circuits for engineering complex cellular behaviors.

Thesis Supervisor: Timothy K. Lu
Title: Associate Professor
Acknowledgments

We would like to acknowledge Nathaniel Roquet who helped conceive the thesis work and define the system workflow. We would also like to acknowledge Katherine M. Collins, who helped develop and test the RSMLab user interface.
Contents

1 Introduction ......................................... 11
  1.1 Motivation ........................................ 11
  1.2 Thesis outline ................................... 12

2 Background ......................................... 13
  2.1 Recombinase-based state machines (RSMs) ........ 13
     2.1.1 Framework .................................. 13
     2.1.2 Terminology ................................ 14
     2.1.3 Existing RSM search function and database .. 17
     2.1.4 Limitations ................................ 18
  2.2 BioCAD tools .................................... 20
     2.2.1 GenoCAD .................................... 21
     2.2.2 Pigeon ....................................... 22
     2.2.3 BioCompiler .................................. 22
     2.2.4 Cello ......................................... 23
     2.2.5 Limitations ................................ 24
  2.3 Thesis Goals ..................................... 25

3 Methods ........................................... 27
  3.1 Representation of genetic elements and diagrams .. 27
  3.2 User interface ................................... 27
  3.3 Database ......................................... 28
  3.4 Web hosting ....................................... 28
4 User interface
   4.1 Defining a workflow for RSM designs .................................. 29
   4.2 Implementing a user-interface for two-input, five-state RSMs ........ 30
   4.3 Extending to 3-input, 16-state ........................................... 35

5 Database and algorithm ......................................................... 37
   5.1 Database generation for two-input, five-state RSMs .................... 37
   5.2 Register ranking and sorting .............................................. 39
   5.3 Generation of three-input, 16-state database .......................... 40
      5.3.1 By brute force ....................................................... 40
      5.3.2 Using states six, seven, and ten to generate full constructs .... 42
      5.3.3 Modularizing RSM layouts ....................................... 45
      5.3.4 Design algorithm from RSM modules ............................ 46
   5.4 Identifying registers with superfluous parts .......................... 49

6 Discussion ............................................................................. 51
   6.1 User interface ............................................................... 51
   6.2 Design algorithm ........................................................... 52
   6.3 RSMLab feature expansion ............................................... 53

7 Conclusion ............................................................................ 55
   7.1 Summary ........................................................................ 55
   7.2 Future Research Directions ............................................... 56
List of Figures

2-1 Recombinase events ......................................................... 14
2-2 Various register layouts and their gene regulation programs .......... 15
2-3 Graphical representation of a cellular state machine and its implement-
tation. ................................................................. 16
2-4 Superfluous parts ............................................................ 17
2-5 Existing search workflow described by Roquet et al. ..................... 18
2-6 GenoCAD user interface. ................................................... 21
2-7 Graphical visualization with Pigeon. .................................... 22
2-8 Example of a system built with Proto ................................. 23
2-9 Cello design workflow ..................................................... 24

4-1 Storyboard flow for and RSM specific GUI ............................. 30
4-2 RSMLab workflow. ......................................................... 32
4-3 Reactjs component hierarchy in RSMLab ............................... 34
4-4 Reactjs component hierarchy for a register in RSMLab ............... 35
4-5 Register diagram for 3-input, 16-state gene regulation program .... 36

5-1 Database table structure for two-input, five-state RSMs ............. 38
5-2 Overall data flow of the RSMLab application for a two-input, five-state
gene regulation program design ........................................... 39
5-3 Library of parts and their corresponding parts IDs in RSMLab. This
figure is adapted from Roquet et al[10]. ................................. 41
5-4 Reconstruction of original register from configurations at states six,
seven, and ten in three-input, 16-state RSM ............................ 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-5</td>
<td>Database for reconstruction search algorithm</td>
<td>44</td>
</tr>
<tr>
<td>5-6</td>
<td>Modular construction of larger RSMs</td>
<td>45</td>
</tr>
<tr>
<td>5-7</td>
<td>Example of neighbor dependence for RSM modules</td>
<td>46</td>
</tr>
<tr>
<td>5-8</td>
<td>Three-input, 16-state from 5 part module</td>
<td>47</td>
</tr>
<tr>
<td>5-9</td>
<td>Redundant register rules</td>
<td>50</td>
</tr>
<tr>
<td>6-1</td>
<td>Full RSM design pipeline</td>
<td>54</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Computational software is playing an increasingly important role in the design and analysis of synthetic biological systems, and should enhance the use of genetic circuits by non-experts for a wide range of applications[1]. Several tools have been developed to aid biologists in various steps of the design process for gene circuits, ranging from system visualization to modeling. Platforms such as GenoCAD[2, 3] and Clotho[4] allow biologists to build custom genetic elements, while others such as Pigeon[5] use the Synthetic Biology Open Language[6] (SBOL) to create graphical representations of genetic constructs. Additional computational tools accept user-specified gene circuit functions and return corresponding gene circuits capable of implementing them. One such platform, BioCompiler, designs gene circuits made from cascaded positive and negative gene regulators[7]. The outputted gene circuits from BioCompiler are not given as nucleotide-specific sequences, but rather as DNA abstractions comprising components (specified by their function but not their sequence) and their relative topologies. More recently, Nielson et al. released Cello, a software program that integrates experimentally derived genetic part characterizations to design detailed genetic circuits made from specific cascades of repressors[8].

Though these design softwares are powerful, they interface with users through programming languages, which may serve as a barrier to entry for many biologists.
Moreover, though there exists gene circuit design software for the implementation of recombinase-based logic circuits[9], there is no software to date that is compatible with the recently described and experimentally validated recombinase-based circuits from Roquet et al[10]. Given these limitations, we sought to create a web-based application for the design of gene circuits capable of performing both combinational and sequential logic. The ideal interface would be available on a web browser, requiring no downloads or familiarity with programming languages to be accessed and explored.

1.2 Thesis outline

This thesis contains seven chapters. A brief overview of each chapter is as follows:

- Chapter 2: Background. This chapter reviews the recombinase-based framework developed by Roquet et al.[10] and existing BioCAD software tools.

- Chapter 3: Methods. This chapter details the software tools used to develop and host this platform on a web-browser.

- Chapter 4: User interface. This chapter outlines the implementation of the RSMLab GUI.

- Chapter 5: Database and algorithm. This chapter outlines the implementation of the RSMLab database and search functions.

- Chapter 6: Discussion. This chapter discusses and analyzes the implementations discussed in Chapters 4 and 5.

- Chapter 7: Conclusion. This chapter summarizes this thesis and gives future research directions.
Chapter 2

Background

In this chapter, we discuss the recombinase-based state machine framework, the existing tools to explore this framework, and existing BioCAD programs.

2.1 Recombinase-based state machines (RSMs)

Recently, Roquet et al. developed a recombinase-based framework for building state machines in E. Coli. This framework can be used to model and understand disease progression, cell differentiation, and many other biological processes[10]. In this section, we briefly describe the biology behind this framework and the existing tools that allow users to explore it.

2.1.1 Framework

At the core of the RSM framework is the notion of encoding state within DNA and using chemical signals to transition between these states. Synthetic biologists have been able to control the structure and contents of the DNA through the use of recombinases, leveraging inversion and excision events to change the composition of the DNA within the cell[11, 12]. In this particular work, Roquet et al. used chemical signals to trigger recombination events catalyzed by large serine recombinases[10]. Upon being activated, these recombinases recognize specific DNA sequences, termed
recognition sites. In the case that two recognition-sites are oriented in the same
direction, an excision event will occur, where the piece of DNA in between the sites
will be removed and degraded. Conversely, in the case that the recognition-sites
are facing one another, the DNA in between the two sites will be inverted. In the
context of this framework, both of these events are irreversible. Figure 2-1 outlines
an example of an inversion and excision event upon activation of a recombinase.

Leveraging the specificity of the recognition-sites, Roquet et al. used multiple
chemical inputs to drive the expression of multiple orthogonal recombinases[10]. By
arranging recognition sites for different recombinases in several different ways on a
piece of DNA, the composition of the DNA can depend on the order of the recom-
bination events (Figure 2-2). In this system, the differing arrangements of the DNA
comprise the states of the system. Using a library of transcriptional elements of DNA
that can be placed strategically between recognition sites, Roquet et al. were able
to control gene expression[10]. Given the reliability of the recombinases used in the
system, the group was able to predict the gene expression profile of a DNA construct
across multiple states given the original layout and sequences of inputs.

2.1.2 Terminology

Throughout this thesis, we use several terms repeatedly when discussing the compo-
nents of the RSM framework. In this section, we define these terms in the context of
this thesis.
In the RSM framework, recombinases predictably and irreversibly recombine DNA along a region containing recombinase recognition sites termed the register. The sequence of this register defines the "state" of system. Different recombinases are driven from different inputs, and the recognition sites are arranged in the register such that the recombination occurs in a state-dependent manner, thereby enabling sequential logic. When the register contains gene regulatory elements, the circuit can implement functions that turn different sets of genes on or off in each state. Such designs are termed gene regulation programs. Though the "circuit" refers to the entire RSM system, we typically represent it only by the register because this is the piece that needs to be synthesized to perform the system logic. Visualizations of these terms are represented in Figure 2-3.

Throughout this work, we discuss the notion of registers with superfluous parts. In general, the more parts that genetic circuits have, the more difficult and resource-intensive their construction is and the harder their behavior is to predict[13]. For this reason, it has been the focus of many construction softwares to include an optimization step in the construction algorithm to minimize the total number of genetic parts used[7, 8]. Roquet et al. discussed this notion through the idea of superfluous parts[10]. In their framework, they describe any part in a gene regulation program that does not change the gene expression of the program when it is removed as a
Figure 2-3: Graphical representation of a cellular state machine and its implementation.

(a) A graphical representation of a two-input, five-state cellular state machine diagram. The circles represent individual states while the arrows represent the transitions. A white circle has no genes expressed in that state. The color of the fill represents which gene is expressed.

(b) Representation of the genetic implementations of the state diagram on the left. One state machine design can have many different register implementations. This figure is adapted from Roquet et al.[10]
superfluous part. Registers that contain superfluous parts are labeled as redundant, as they encode the same gene regulation program as a register without this part. An example register containing a superfluous part vs. a minimized register implementing the same program is highlighted in Figure 2-4. These components are unnecessary for the desired program and simply represent an additional construction cost. When constructing their register database, Roquet et al. searched through the database and removed all redundant registers that contained such parts[10]. Likewise, any design algorithm developed for RSM-specific use should remove these redundant registers from the search results.

2.1.3 Existing RSM search function and database

While RSMs have transformative potential for understanding and engineering biological systems, designing circuits for particular functions under this framework is non-trivial especially as the desired behavior scales in complexity. Roquet et al. developed
a Matlab search function for the two-input, five-state machine that returns registers that match user specified designs[10]. Using the matlab command line, users define a gene regulation program using a matrix of 0s and 1s to represent a non-expressed and expressed gene, respectively. Given this matrix, the search function queries the database for registers that match this design. The search function returns an array of arrays of register that match this design. Registers are represented through an array of part IDs, each ID corresponding to a unique genetic combination of promoters, genes, and terminators. Users can then convert these arrays of part IDs to real genetic constructs.

The Matlab database implemented by Roquet et al. contained three Matlab matrices[10]. The first contained all possible register part combinations with no superfluous parts; the second contained all of the design vectors that came from the different combinations of parts; the third was a mapping from the first matrix to the second matrix. Using these three matrices, users defined a two-input, five-state RSM design using a 5xN matrix, where N represents the number of genes that the program regulates. This matrix was converted to a design vector and the database was queried for this design vector. If a valid vector exists, then all registers that mapped to this vector in the mapping table were returned to the user as an array of parts. The workflow of this search functionality is outlined in Figure 2-5.

**2.1.4 Limitations**

While the existing search function and database allows users to explore all possible implementations of a two-input, five-state RSMs, there are several obstacles that complicate users’ ability to search for specific gene regulation programs. The first
obstacle is accessibility. Currently, to search the database, users must download both the Matlab database and search function from github. Once downloaded, users use the Matlab command line to search through the database, inputting design vectors using matrices as described above. While downloading code from github is common practice for many scientists, it limits accessibility to users familiar with github and Matlab. An ideal tool would use a common and intuitive platform for users from all computing backgrounds.

The second obstacle users face is usability. When describing a state machine diagram, Roquet et al. used a flow chart with circles and arrows to represent states and transitions between them, respectively[10]. These diagrams make it easy for biologist to understand the system and transitions between states. The current search function, on the other hand, uses a matrix of 0s and 1s to represent genes expression in all five states. Representing the RSM as a matrix makes it harder to interpret and makes more complicated designs more prone to errors when being entered into the search function. Additionally, the output of the search function is an array of part IDs representing a register. Users need to convert these part IDs to their corresponding genetic components before they can interpret the implementation. This makes it more challenging for users to compare several implementations of the same state machine, as they have to convert each implementation before comparing them. The ideal search function would allow users to design RSMs using a graphical user interface (GUI) that uses the diagrams with circles and arrows while returning the registers as a combination of commonly used glyphs or symbols.

A third major obstacle is scalability of the database and search function. Currently, the database contains all possible constructs for two-input, five-state machines. The search function is able to take a matrix representing gene expression across five states and return registers that implement this state machine by querying the database. To expand the search to a different or larger state machine with different recombinase sites or more inputs, a new search function and database have to be created. This means that a user has to download the new database and search function every time either is updated, making it more difficult for them to stay in
sync with the most updated versions. Ideally, the search function and database would be available on the cloud, where updates can be pushed live, allowing users to use the most updated version at all times. It would also use a database that is easily updatable, scalable, and accommodating to new features including feedback based on performance metrics calculated upon construction. The design workflow would be consistent for all state machine designs regardless of the number of inputs or states.

Finally, it is important to note that one gene regulation program may be implemented by several hundreds of registers, all of which are returned to the user. Some of these implementations require more parts than others, while others contain parts in undesirable orientations to one another. For example, some registers may necessitate promoter read-through (transcription through a promoter facing the opposite direction) for proper circuit function. Because such transcription may be attenuated due to interference[14], such registers are not as reliable as alternative registers that do not require promoter read-through. Additionally, some registers may necessitate terminator read-through (transcription through a unidirectional terminator facing the opposite direction) for proper circuit function. Because the palindromic structure of terminator sequences[15] may make it challenging to design reliable unidirectional terminators (with no basal activity in the reverse direction), such registers necessitating terminator read-through are less desirable to construct than alternative counterparts. Taking these considerations into account, any RSM-specific search function should include an ordering based on some of these constraints. Following a search query, users should be presented matching registers based on this ordering.

2.2 BioCAD tools

In Section 2.1.4, we discussed the limitations of the RSM search function and database made available by Roquet et al[10]. Before attempting to implement our own RSM-specific BioCAD software, we took a look at the currently available tools to see if any of these existing tools could be adapted to be used in the context of RSMs.
Figure 2-6: GenoCAD user interface.

(a) Users that have GenoCAD accounts are able to create their own custom libraries of parts. Users can build their own custom libraries, selecting a grammar and adding parts from a public repository. They are able to add their own parts as well. (b) A sample design for a genetic toggle switch. Users can select which step they want to edit on the left side. This figure is adapted from Czar et al[2].

2.2.1 GenoCAD

GenoCAD is a DNA designing platform that allows users to design artificial DNA sequences from a library of publicly available or user-defined parts. It is available for free via a web-browser, requiring no additional downloads. GenoCAD’s design philosophy follows the bottoms-up approach of engineering; designing complex systems with a set of smaller and well-defined parts. The GenoCAD software defines a grammar, a set of rules that determine how parts can be linked together to design constructs. These grammars can vary by organism or application, providing users with a much more manageable parts library[2].

GenoCAD’s graphical user interface leads the users through a design workflow built on top of grammars. Users start by selecting the grammar and library of parts they would like to use for their design. The interface then guides users through the design process, starting at a high-level until their design is decomposed to the individual parts from their library. Users can navigate to previous steps and edit their design at any point. Once completed, users can download the DNA sequence to a text file which can be used as the input to other design software. If a user has an account, they can upload their own library of parts and save designs to the application.
GenoCAD is an example of an application that uses a graphical user interface to allow users to create various genetic constructs.

2.2.2 Pigeon

Pigeon is a web-based visualization tool that allows synthetic biologist to describe complex biological designs using common glyphs. Pigeon defines a syntax to describe common genetic components. Biologists describe their systems textually following the Pigeon syntax and send it to the server to be converted to a visualization\cite{5}. The syntax allows for the design to be highly customizable; users are able to define their own colors and labels for their parts as well as relationships between parts through arcs. Using these rules, biologists can design various different systems. Pigeon’s structure is designed to be quickly learnt by biologists with limited programming experience and easily integrated into larger systems or workflows.

2.2.3 BioCompiler

Proto is a programming language that was adapted to allow biologists to define a high-level behavior of a biological system\cite{7}. BioCompiler provides a graphical user
interface that allows biologists to use this programming language to create sophisticated systems and a compiler that converts these systems into optimized genetic regulatory networks. Matlab generated computational models of these genetic regulatory networks are included in the web platform as well. The models allow users to validate the performance and behavior of their systems prior to implementation. BioCompiler can be used to model a system as simple as an inverter to more complex systems like a two-bit digital adder.

2.2.4 Cello

Cello, an abbreviation for Cellular Logic, is a design software that automates the implementation of DNA constructs that perform user-defined combinational logic programs[8]. Cello allows users to describe their system behavior through textual input through the use of Verilog. One of the key design decisions made by the group implementing the Cello software was to use exclusively NOT and NOR logic. The underlying Cello algorithms parse the Verilog text to create a logic circuit representation of the design, convert and minimize the logic diagram using only NOT/NOR gates, assign gates to genetic parts from a library of repressors, build the DNA based on those parts, and predict the circuit performance. This workflow is outlined in Figure 2-9. The group used Cello to design 60 circuits that were implemented and
Figure 2-9: Cello design workflow

The Cello design allows users to define programs with high-level behavior using textual commands. Based on this system definition, the Cello software identifies a logic diagram, DNA linear construct, and predicts the performance of the circuit. All of these components can be visualized by the user. This figure is adapted from Nielsen et al.[8].

tested in E. Coli, 45 of which performed correctly for every output state. Their work demonstrates the potential role that automated design software could play to enhance biologists’ ability to enable programmable control over biological functions within cells.

## 2.2.5 Limitations

All of the software tools above address gaps in design, visualization, or modeling of synthetic biology systems. However, none of them fit the full requirements to be used as the main GUI to explore the RSM database. GenoCAD creates DNA sequences from a well defined grammar and library of parts. This functionality would be very useful after the last step of the RSM application, allowing users to convert their abstract registers to actual genetic sequences and export them for synthesis. However, it does not currently support the analysis of genetic constructs that users need in the RSM application. Pigeon provides the graphical visualization of genetic components that the application requires and would allow users to easily compare different register implementations for their circuits. However, a significant amount of work would have to be done to make the Pigeon syntax compatible with the
way registers are represented (the ability to have elements facing in both directions, for example). Additionally, it does not provide any graphical component for the design of state machines. It is also strictly a tool for visualization and can not perform any analysis on the genetic structures it represents. BioCompiler and Cello are very powerful tools that allow users to go from system design all the way to computational models. While BioCompiler’s and Cello’s design workflows would be able to implement a state machine behavior, the system design would not use Roquet et al’s RSM framework. In addition, these two softwares require users to define their systems using textual commands, rather than a GUI that would fit more seamlessly for design using the RSM framework.

Given the constraints described with the current search function and database implementation and the limits of the existing design and analysis software, creating a separate application that allows users to design and compare RSMs was the best approach to consolidate the workflow in a single location.

2.3 Thesis Goals

The purpose of this thesis was to design an application that allows users to use and explore RSMs fully, while being able to edit and update this application regularly. As described above, the ideal application would be accessible remotely, most likely through a web browser, use some sort of GUI to represent gene regulation programs and their genetic register implementations, and allow users to export and analyze their designs. The end goal for this application was to provide a tool that increases the applications of RSMs in on-going biological endeavors. The creation of this application was split up into three sequential objectives.

The first objective of this thesis was to develop a GUI that allows users to explore two-input, five-state cellular state machines. This objective was to convert the current Matlab database and search function to an interactive interface accessible via a web browser. Our goal involved coming up with an intuitive design workflow that allows users to easily define their state machine components and create their system (most
likely through a drag and drop type interface). We sought to create a GUI and workflow scalable to more complex systems as well. Finally, the application in general should be able to accommodate new features depending on user needs.

The following goal was to create a database for three-input, 16-state cellular state machines. This objective involved iterating through all combinations of parts and mapping each register to its corresponding design. Once each register had been mapped to its design, we aimed to update the database to remove redundant registers. Finally, the data was to be analyzed to determine how many designs are represented using this recombinase array relative to the theoretical possible number of designs as well as the number of genes that are regulated in each circuit.

The final objective was to develop an algorithm that finds registers corresponding to an input RSM. The two-input, five-state and three-input, 16-state cellular state machine databases were used to find patterns and modularity, attempting to build a generalizable algorithm. The algorithm also needed to be able to rank the circuits that it returned that correspond to the desired RSM design. The scoring metrics were to be determined, leveraging several factors including register complexity and performance reliability.

In the following sections, we discuss the tools and results obtained as we attempted to achieve these three objectives.
Chapter 3

Methods

In this section, we describe the software tools that were used to implement the RSM-Lab GUI, database, and search algorithms. All of the code for RSMLab is stored at https://github.mit.edu/gkugener/grfsm and can be obtained upon request.

3.1 Representation of genetic elements and diagrams

Common genetic elements (promoters, genes, and terminators) were represented using SBOL 2.0[6]. The SVG files encoding each of these symbols was used to display the parts in the DNA constructs presented to users. Recombinase recognition-sites, as the ones found in Roquet et al., were not included as part of the SBOL parts. Representation of these sites was based on their representation in Roquet et al’s work. The state diagram representation of RSMs was adapted from Roquet et al[10].

3.2 User interface

RSMLab was implemented using Reactjs[16]. Additional javascript packages used to develop this interface include Browserifyjs [17], Babel [18], jQuery, and jQuery-ui. The interface was tested for compatibility on Safari, Google Chrome, and Firefox. Support for Internet Explorer was not tested. The RSMLab source code is written in Python (version 2.7.13).
3.3 Database

All of the web application data was saved and generated from a MySQL server (version 5.6.27-log MySQL Community Server) hosted on an Amazon Web Services (AWS) relational database systems (RDS) instance. Separate stacks exist for the two-input, five-state and 3-input, 16-state RSMs.

3.4 Web hosting

RSMLab was hosted on AWS using elasticbeanstalk. Flask (version 0.1.13), a Python webserver wrapper, was used to construct an API between the MySQL server and the Reactjs GUI. We used the SQLAlchemy Python module to interface with the AWS RDS instance. The data from the server is processed and formatted using Python before it is passed onto the interface.
Chapter 4

User interface

In this section, we look at the implementation of the RSMLab GUI and its extension to 3-input, 16-state gene regulation programs.

4.1 Defining a workflow for RSM designs

Prior to implementing a functional and live user interface for our application, we developed a workflow for users to design systems leveraging the RSM framework. Following the design graphics from Roquet et al. that described RSM systems[10], we identified how users would approach the design process. After discussions with potential users and the developers of the original RSM framework, we outlined the following storyboard workflow outlined in Figure 4-1:

1. Users access the site where they are asked how many genes they would like to control within their design. These genes would need colors or names to differentiate them from one another.

2. Once users have determined how many genes they want to control, they proceed to a screen containing an empty state diagram.

3. Users drag and drop the genes from a list into the states that they would like this gene expressed.
4. Once they have created the system that they desire, users initiate the database search.

5. When the search has been completed, they are brought to a screen displaying the genetic registers corresponding to their designs. Arrows allow them to navigate between different register implementations. If their design did not contain any valid RSM implementations, then the screen notifies them that no such implementation is possible.

6. Having reached the end of the user interface, users could decide to start from the beginning and create a new system or, if interested, could select a subset of registers to export to pdf.

4.2 Implementing a user-interface for two-input, five-state RSMs

Having defined a possible workflow for design of RSMs, we converted this paper storyboard to a functional user interface. This interface was limited to the design
of two-input, five-state RSMs. We used the React.js framework to create a drag and drop interface that allowed the users to perform the tasks outlined above. The implementation of this web interface, named RSMLab, is outlined below:

1. Users access RSMLab and are prompted to enter the names of the output genes that they would like to regulate with their circuit. They can also customize the color that represents the gene in the GUI (Figure 4-2a).

2. Following gene addition, users are presented with an empty state diagram representing a gene regulation program that defaults to no genetic output in any state. They can select any state for which they wish to edit the outputs (Figure 4-2b).

3. When editing a state, users drag and drop the genes they want expressed in that state from the list of genes they added in step 1. If desired, they can add additional genes to their gene regulation program beyond those added at the beginning of RSMLab and can remove genes from the state as well (Figure 4-2c).

4. When they have created the gene regulation program that they desire, the users click on the continue button, which queries the database for all of the RSM registers that match the specification (Figure 4-2d).

5. On the register diagram page, users are presented with a visual SBOL representation of the registers in each of the states. Users can use the navigation bar at the top of the application to cycle through the different registers that match their gene regulation program (Figure 4-2e). In addition, they are able to export the list of registers to PDF for further analysis (Figure 4-2f).

The RSMLab interface and workflow defined above was implemented using a variety of javascript libraries, outline in Section 3. The core javascript framework used was React.js. In web development, accessing the DOM and writing to it is an expensive operation and acts as a bottleneck as javascript functions get faster and faster. Reactjs introduces this notion of the virtual DOM, which is an in-memory representation
Figure 4-2: RSMLab workflow.

An example workflow for how a user would specify a two-input, five-state gene regulation program with a single genetic output.
of the DOM. Javascript operations are performed on this virtual DOM. As changes are being made and new data is being passed to Reactjs, the difference between the virtual DOM and actual DOM are taken. Reactjs then applies the minimal set of changes that are needed to the DOM, rather than rewriting the entire document. This is similar to the git approach, where changes are committed and only the small differences in the files are changed. This leads to a much faster and more optimal application performance as load time and updates are minimized[16, 19].

Using Reactjs, we created a hierarchy of reusable javascript modules for the different components of the RSMLab application. At the top of the hierarchy is main.js, a file that controls which page is displayed on the user interface as well as the current state of the application and user-defined gene regulation program. Several page modules are used to display the particular design step that the user is on at one particular moment in time. Some of these pages contain their own state variables independent of the main application. Figure 4-3 outlines the hierarchy of the page displaying all of the register implementations for a particular gene regulation program. On this page, main.js is the highest level component, containing the CircuitDiagram.js component. CircuitDiagram.js contains two navigation bar components, a GeneticCircuit.js component, and a DiagramIllustration3Input.js component. These each contain their own smaller level Reactjs components not illustrated in this figure. Any changes to the main.js state variables is passed down to the smaller components. When a user hits the "Next" arrow, for example, the NavBar.js component is updated as is GeneticCircuit.js component, but DiagramIllustration3Input.js remains unchanged.

Besides the main application functionality, we created Reactjs modules for visualization of registers using SBOL components. GeneticCircuit.js is a Reactjs specific module that allows for the easy construction of registers of various sizes and part layouts using this arrangement of recognition sites. Using this module, users can define a height, width, and a parts array for their register through attributes. The module also requires a gene mapping hash map to map the information of a gene such as name and color representation to its ID in the register. Each part in the register part array is matched to its part ID and, based on this part ID, a visualization of combinations
Figure 4-3: ReactJS component hierarchy in RSMLab

An example of the ReactJS component hierarchy for RSMLab on the register display page.
of promoters, terminators, and genes is created and displayed. An example of this modular construction with its corresponding React.js component names is outlined in Figure 4-4.

4.3 Extending to 3-input, 16-state

One of the objectives of building this user interface was to allow for researchers to be able to use this platform as the sole tool for system design of RSMs of all sizes. Therefore, following the creation of the two-input, five-state web interface, we extended the interface to allow users to design 3-input, 16-state RSM gene regulation programs. We aimed to include this extension while minimizing the change to the user experience. Therefore, there were only two differences between the original two-input, five-state only interface and the expanded interface besides minor positional and color changes. The first difference is the presence of a radio button allowing users to select the number of inputs (two or three) they would like their RSM to contain. The second difference is on the final screen users see. In the two-input, five-state, for each state in the gene regulation program, the corresponding register implementation is visible for the user to visualize. This visualization is included to allow for quick troubleshooting; if one particular state is problematic due to the orientation of parts, a user could easily verify this possibility by looking at the register at that state. However, including all of these implementations on the screen for the 3-input, 16-
Figure 4-5: Register diagram for 3-input, 16-state gene regulation program

(a) When not hovering on a state, the only register displayed is the register to be constructed.
(b) To visualize the layout of the register at any particular state, users simply hover over that state. In this figure, the user is hovering over state 14.

state system crammed the screen. Therefore, we opted to only show the first, state 1, register at the top of the page. In order to visualize the layout at particular states, users can hover over their state of interest, which displays the layout at that state. This change in the user interface is outlined in Figure 4-5. These being the only two changes to the interface, we were able to successfully expand the web application to include design for 3-input, 16-state RSMs.
Chapter 5

Database and algorithm

In this section, we discuss the structure of the RSMLab database and the various algorithms present within RSMLab.

5.1 Database generation for two-input, five-state RSMs

In Section 2.1.3, we outlined how the existing Matlab search function from Roquet et al. allowed users to define two-input, five-state RSMs using the Matlab command line. As discussed, this search function leverages three tables: a grfsm_array table, containing all possible gene regulation programs, a circuits_array table, containing all registers used in the database, and a mapping table linking the grfsm_array and circuit_array tables. As a first step to transitioning all of the search functionality to Python and SQL, these tables needed to be transformed into a SQL database. However, rather than having a separate table connecting registers to their designs, we leveraged the presence of SQL foreign keys. We created two tables: a grfsm_array table and a circuits_array table. Like its Matlab counterpart, the grfsm_array table contained IDs mapping to particular design vectors. The circuits_array table contained all of the non-redundant register part combinations, as did its Matlab counterpart. Like its Matlab counterpart, this SQL table stored the parts as a comma-separated string. Additionally, it contained a grfsm_id column, a foreign key mapping to the grfsm_array table. As the name would suggest, this key mapped the
register to its resulting design vector, replacing the third mapping table in the original Matlab database. The database structure is outlined in Figure 5-1. We included the id_in_matlab column for debugging purposes.

We leveraged RSM-specific Python functions developed as part of this thesis to generate all of the possible register combinations from the 26 part library used in the original development of the work. The design vector for each register was determined and assigned a design ID. If this register contained superfluous parts, it was not included in the database, as was the case in Roquet et al.’s work. All of these possible designs and registers were generated and imported into the SQL database. Once it was completely generated, this database was validated against the original Matlab database created by Roquet et al. Both databases contained 174,264 unique design vectors and 5,192,819 different registers of unique part combinations, suggesting that our Python database-generating functions could successfully generate the database for RSMs and appropriately identify and eliminate registers with superfluous parts. This complete two-input, five-state database with redundant registers removed allowed us to finalize the general application architecture, outlined in Figure 5-2.

Figure 5-1: Database table structure for two-input, five-state RSMs
5.2 Register ranking and sorting

RSMLab uses a ranking function to determine the order in which registers are returned for a user-specified gene regulation program. The RSMLab ranking function deprioritizes registers that require promoter read-through or terminator read-through. Conversely, registers with less parts (especially less promoters), are prioritized in the return queue since these designs are presumed to be more reliable and easier to construct. The following steps summarize the tasks performed sequentially by the ranking function prior to returning a list of registers to the user:

1. Order registers (in descending order) by the number of promoters they contain that will initiate a required promoter read-through event.

2. Suborder registers (in descending order) by the number of unidirectional terminators they contain that will facilitate a required terminator read-through event.

3. Suborder registers (in ascending order) by the number of recombinase-recognition-site-flanked regions they contain that do not have any parts.
4. Suborder registers (in descending order) by the number of promoters they contain.

Here, the steps that "suborder" registers perform their ordering within clusters of registers that are equivalently ranked by prior ordering steps. This ranking function was used to order registers prior to being returned to the user.

5.3 Generation of three-input, 16-state database

In order to expand the user-interface to include three-input, 16-state RSMs, the database needed to be expanded to return these registers. However, we used only the first 10 parts from the original library of 26 parts, in order to limit the total memory being used by the database. This would also allow for faster and easier validation. The full list of 26 parts and the subset of 10 parts are outlined in Figure 5-3.

5.3.1 By brute force

As a first naïve approach to generate this database, we used the same approach as we used to generate the two-input, five-state RSMs. Using the reduced part library, we used the RSM-specific Python functions to create different combinations of the registers for the three-input, 16-state systems. However, it was evident early on that this approach would not be efficient for generation of more complex databases. This generation method spent lots of time generating registers with superfluous parts, which would not be included in the final database. In order to identify a register as having superfluous parts, the algorithm needed to compute the gene expression of the register across all 16 states. The two-input, five-state RSM database contained the 5,192,819 non-redundant registers, representing only $6.5 \times 10^{-4}\%$ of the total $26^7$ possible register implementations. Using these values for reference, computing design vectors for such an amount of redundant registers for the three-input, 16-state system proved to be to ineffective and time-intensive. Additionally, saving the registers as full arrays in the database required too much memory.
Figure 5-3: Library of parts and their corresponding parts IDs in RSMLab. This figure is adapted from Roquet et al[10].

(a) Parts key: \( \text{T} \) = Terminator \( \text{T} \) = Optional terminator \( \text{P} \) = Promoter \( \text{G} \) = Gene with bi-directional terminator

(b) Parts | Database ID
---|---
1 | 1
2 | 2
3 | 3
4 | 4
5 | 5
6 | 6
7 | 7
8 | 8
9 | 9
10 | 10
11 | 11
12 | 12
13 | 13
14 | 14
15 | 15
16 | 16
17 | 17
18 | 18
19 | 19
20 | 20
21 | 21
22 | 22
23 | 23
24 | 24
25 | 25

(c) Parts | Database ID
---|---
1 | 1
2 | 2
3 | 3
4 | 4
5 | 5
6 | 6
7 | 7
8 | 8
9 | 9
10 | 10

(a) Parts key. (b) The full 26 parts library used for the two-input, five-state database. (c) The 10 part library used for the generation of the three-input, 16-state database.
5.3.2 Using states six, seven, and ten to generate full constructs

While using a naïve approach was the simplest in terms of implementation, it proved to be ineffective due to memory and run-time limitations. Another interesting possibility that could limit the memory use and run-time of this generation function could be storing fragments of full registers. Upon closer inspection, using the register implementations at particular states, we could reconstruct the original 13 part register. For example, knowing the part layout for the register at states six, seven, and ten would allow us to reconstruct the 13 part register without having to store all 13 parts (Figure 5-4). Using this property, we were interested to see if we could store these smaller registers in the database and leverage them to obtain our full registers. Since we would be storing register sub-states with nine parts rather than 13, this could lead to a four-fold reduction in the required storage. In this case, the limiting factor would be the reconstruction component.

To test this approach, we focused on creating a search algorithm for the three-input, 16-state database that used a database that stored these nine-part implementations. In this database architecture, we stored the values of the parts at each position and the number of genes expressed given this layout. Figure 5-5 illustrates how these nine part arrays were stored in the database and how an example sub-register would be stored in the database. Using this architecture, we created the following reconstruction search algorithm:

1. A user defined the gene regulation program they would like to design using the interface as usual.

2. Three separate queries are run: one that looks for part arrays that match the gene expression at state six, another that looks for part arrays that match the gene expression at state seven, and the final one that looks for part arrays that match the gene expression at state ten.

3. If these three queries return a non-zero set, then a join across these three queries
Figure 5-4: Reconstruction of original register from configurations at states six, seven, and ten in three-input, 16-state RSM

(a) Configurations of a generic three-input, 16-state register at states six, seven, and ten. The numbers correspond to the position of the part on the original register. A "-" indicates that a parts orientation has been flipped. (b) Original register reconstructed from the parts from states six, seven, and ten with colors to indicate which state each part came from. Black parts indicate that this part could have come from any of the states. Orange parts come from the state six configuration as seen in (a), purple parts come from the state seven configuration as seen in (a), and blue parts come from the state ten configuration as seen in (a).
Figure 5-5: Database for reconstruction search algorithm

(a) The structure of the subarray database that is queried. (b) How the sub-register in (c) would be stored in this database. (c) An example of a nine part sub-register that would be stored in the database and used for reconstruction.

is performed to reconstruct the original register

4. Once the original register is reconstructed, the algorithm analyzes its design vector at each state to see if it matches what was queried for. If it does, this register is included in the search results.

5. Once all register have been reconstructed from these three sub-queries, the results are ranked and displayed to the users as before.

Using this methodology allowed for the construction of more register components from less memory. However, the reconstruction algorithm proved to be too computation-intensive. While using this methodology certainly reduced the space required to store enough information to regenerate all of the possible registers, the reconstruction algorithm proved to be far too time-intensive to be used as an algorithm available on the web.
Each color and shape represents a pair of orthogonal recombinase sites. As the number of inputs is increased, a modular configuration of recombinase sites allows Roquet et al. to easily scale the system. This figure is taken from Roquet et al.[10].

5.3.3 Modularizing RSM layouts

In the original RSM work, Roquet et al. described how they could chain together a particular sequence of recognition sites, illustrated in Figure 5-6, for orthogonal recombinases in order to create more complex RSMs. Combining three of these modules together in succession lead to a three-input, 16-state system, combining four in succession created a four-input, five-state system and so on. This modularity present in the construction of RSMs could provide useful insights into designing an algorithm that would address the problems we faced in our previous attempts.

Looking at an individual module, we observed that starting from its initial state, it can transition to three other different states. The configuration of an individual module in each of these three states is outlined in Figure 5-8a. Of note, while it would be expected that a genetic construct that uses two recombinases would lead to a five-state system, we observed that the excision event transition from Configuration 1 to Configuration 3 in Figure 5-8a removes the recognition site for the unused recombinase. This means that exposure to this recombinase after the excision event will not change the structure of the DNA any further in this module.

When we think of the gene expression of any individual module, we consider not only the physical layout of the module on the piece of DNA but in addition, the transcriptional input it is receiving from its neighboring module. Each module
Figure 5-7: Example of neighbor dependence for RSM modules

(a) An example 5-part RSM module (b) Table representing how gene expression varies for this module depending on the transcriptional inputs from its neighboring modules.

receives two transcriptional inputs; from the module to its left, where transcription of genetic parts on the top of the module can be affected, and from the module to its right, where transcription of genetic parts on the bottom of the module can be affected. The transcriptional input from a module can either be no transcription, encoded with a 0, or transcription, encoded with a 1, for a total of four possible transcriptional input combinations from neighboring modules. Figure 5-7 outlines an example of a module whose gene expression varies depending on the transcriptional input it receives from its neighbors. Given that there are four combinations of inputs and four state configurations, each module has 16 total possible unique expression profiles.

In Figure 5-8, we see how we can apply this modular approach to the three-input, 16-state system. Using the abbreviations for module name and configurations outlined in the Figure 5-8, we observe that we can reconstruct the entire system from a combination of the submodules. It is important to note that the modules are not completely independent: the first element of a module has to correspond to the last element of the module to its left, and the last element of that same module, has to correspond to the first element of the module to its right. Of note is the fact that the neighbors of a particular module do not change throughout the state transitions.

5.3.4 Design algorithm from RSM modules

In the previous section, we discussed a technique to modularize the RSM genetic constructs. We outlined that a single module could correspond to multiple design
Figure 5-8: three-input, 16-state from 5 part module

(a) Representation of how a single 5-part module transitions between states when exposed to recombinases. The numbers represent the part’s position in the original layout. A "-" represents a part whose orientation was flipped as a result of an inversion event. (b) The three 5-part modules used to construct the 13-part register for the three-input, 16-state system. (c) State diagram of a three-input, 16-state RSM using the modules from (b). Abbreviations are used for clarity. MB_2 corresponds to Module B in configuration 2 (from (a)).
vectors depending on the inputs from its left and right neighboring modules. This means that, for a particular module, we will have varying designs from this module based on the input and that this module’s transcriptional output can affect its neighbors. As such, in the database that we created, we stored the input vector, output vector, and part arrangement of the different modules. Additionally, we created a separate table to store design vectors. These design vectors contained a foreign key mapping to the module that it corresponded to as well as a column that kept track of which module type (A, B, or C from Figure 5-8b) this design vector came from.

Using this smaller database, we created a new search algorithm for single-gene gene regulation programs for three-input, 16-state RSMs. The search algorithm is the following:

1. A user defines the gene regulation program they would like to design using the interface as usual.

2. The design vector table in the database is queried, returning the five part arrays that correspond to the user inputted design vector.

3. The full register is reconstructed by adding the two remaining modules to create the full 13-part register. Valid modules are ones that provide the correct input vector sequence into the returned five-part module. "Correct" input sequence means the transcriptional inputs that lead to the gene expression matching the user’s input.

4. The full design vector for the reconstructed register is analyzed to ensure that it matches the user input. If it does, it is added to the list of returned registers.

As we reconstructed the full 13 part register from the five part module, we also filtered out any register combinations that would lead to registers with superfluous parts.
5.4 Identifying registers with superfluous parts

The original two-input, five-state database was developed by testing all of the combinations of parts on a register. This lead to the creation of many registers that contained superfluous parts, a term defined in Section 2.1.2. In order to remove these registers from the database, Roquet et al. combed through the database and identified such registers. The entries of registers with superfluous parts were removed from the database. While a similar approach would be possible for the three-input, 16-state database, it would be advantageous to be able to identify these registers before even adding them to the database. Moreover, defining sets of rules that identify registers that will contain superfluous parts without having to simulate gene expression of this register at every state would improve the run-time of the database-generation algorithm as well as the run-time of any reconstruction algorithm used in a database search.

As we generated the three-input, 16-state databases and search algorithms, we looked to find patterns within registers that were marked to have superfluous parts. Upon closer inspection, it was apparent that it would be possible to define rules that were specific to the RSM modules defined in Section 5.3.3. Having made this observation, we were able to identify several design rules that could verify whether a particular register would contain superfluous parts upon implementation. These design rules are outlined in Figure 5-9. If any module within the register contains the part orientations outlined in Figure 5-9, then this register will be redundant. Given that the rules depend on the configuration of three components, this helps avoid computing design vectors for many vectors. The ten part library contains five parts with promoters, three parts with terminators, and three parts with genes. We can eliminate 19,200 (5x5x3x16²) modules from the rules in Figure 5-9a and 6,912 (3x3x3x16²) modules from the rules in Figure 5-9b. Identifying these redundant registers significantly shortened the database generation time as well as the search time, as fewer modules are included.
Figure 5-9: Redundant register rules

(a) Modular configuration of modules that lead to genes that are always expressed. The green gene is expressed in every state of the gene regulation program if the module is configured in this way. (b) Modular configuration that lead to genes that are never expressed. The green gene is never transcribed in a gene regulation program implemented by a register with a module containing this configuration.
Chapter 6

Discussion

6.1 User interface

We developed RSMLab to help users design circuits that implement two-input, five-state and three-input, 16-state gene regulation programs. The design software utilizes the validated RSM framework from Roquet et al[10] for building biological state machines. RSMLab lowers the barrier of entry to this technology with a GUI-based design environment that saves users from manual circuit design and exhaustive design space searches.

We defined a complete design workflow for RSMLab design through a GUI that is scalable to gene regulation programs of increasing complexity. We demonstrated this scalability by expanding the GUI to allow users to design single gene three-input, sixteen-state gene regulation programs. We showed that users could use the same drag and drop interface to select which genes should be expressed in each state. This workflow could be scaled to even more complex RSMs consisting of more inputs or even different layouts of recognitions sites. RSMLab allows for users to interact with the RSM framework at the very beginning of the process; during the design phase.
6.2 Design algorithm

In this work, we attempted to find a generalizable search algorithm for RSMs that successfully leveraged time and search space. Our first approach was to naïvely construct this database like Roquet et al. did for two-input, five-state RSMs. While this approach would have lead to a search algorithm that required a simple database query with no reconstruction, such an approach and implementation proved intractable as well as unscalable. Using this approach would yield a database that used too much memory. While this might not be an issue for a 3-input system, larger systems would simply require too much space and time to be implemented.

Given this limitation, we next attempted to create a database that required a subset of the full register. Leveraging the fact that certain states in the RSM contain fewer parts due to excision events, we looked to create a search algorithm that used states six, seven, and ten, only containing nine genetic parts each, from a three-input, 16-state RSM to reconstruct the original 13-part circuit. While this database used four-fold less space, the reconstruction algorithm was far too time-intensive. Moreover, the reconstruction algorithm was not able to easily identify registers containing superfluous parts, identifying only a handful of registers in the allowed search time.

Ultimately, upon closer inspection of the modular construction of registers, we observed that each of the states in the three-input, 16-state RSM was composed of modular components. The expression of genes of these components was dependent on the inputs from the neighboring modules. Using these observations, we created a database that stored the layout of these submodules with varying transcriptional inputs. The different promoter and submodules mapped to design vectors that were also stored in the databases and connected through the use of SQL foreign keys. Leveraging this modular approach, we were able to create a function capable of reconstructing registers for user-defined gene regulatory programs from these submodules. Additionally, this search algorithm was able to remove redundant registers from the search results, leveraging several design rules. We were able to connect this search algorithm to the expanded RSMLab GUI, thereby allowing users to explore a subset
of single-gene gene regulatory programs for three-input, 16-state machines. Using this search function, we were able to visualize registers that were tested and implemented in the original work done by Roquet et al. This approach is scalable to programs controlling the expression of multiple genes and of large inputs, although such an algorithm has not yet been developed.

6.3 RSMLab feature expansion

The modular structure of the RSMLab backend will allow us to continuously update and expand our database of RSM registers. For example, as more data regarding RSM behavior is acquired, we may remove and re-rank current registers based on updated performance and predictability models. Additional columns in the SQL database could be added for each register, scoring the register’s performance when implemented and integrated in a cell. These scores would be factored into the ranking algorithm as additional weights. Registers that consistently perform as expected would be prioritized in ranking by being assigned lower weights, while, conversely, poor performing registers would be penalized. By keeping track of these weights, we hope to ultimately identify generalizable design rules for RSM construction. These design constraints could eventually form the basis of a grammar for RSMs, similar to the grammar developed for synthetic transcription factors and implemented into GenoCAD[3].

Moreover, as more complex gene circuits are built, we may include additional registers capable of implementing higher order behaviors. Currently, though RSMLab can design circuits for two-input, five-state regulation programs with multiple genetic outputs, all three-input, 16-state regulation programs returned by the software are limited to one genetic output. As we continue increasing the number of inputs and genetic outputs allowed as design parameters in RSMLab, we will need to refine our search algorithm and database structure to maintain feasible processing and memory requirements. Since the space of possible register designs expands exponentially with number of inputs,[10] any top-down design strategy that uses exhaustive searching
RSMLab sits at the beginning of a full design, build, model, and test pipeline for RSMs. This figure is adapted from a presentation given by Roquet.

quickly becomes intractable. Instead, bottom-up design algorithms will need to be developed that can quickly piece together the register for a given function based on modular design rules. For example, an RSM-specific genetic algorithm would be useful.

Additionally, we plan on expanding the workflow in the future to allow users to take RSMLab outputs and integrate them into other tool chains. For example, the software could be expanded to allow users to match the abstract part representations in the outputted registers to existing DNA sequences and export their constructs to existing DNA synthesis softwares, such as GenoCAD[2] or Twist[21]. These expansions would complete a full design pipeline for automated RSM creation.
Chapter 7

Conclusion

7.1 Summary

In conclusion, in this thesis, we investigated the possibility to develop a centralized workflow for the use and design of RSMs of multiple sizes and inputs using the framework described by Roquet et al. We were able to achieve our primary objective, which was to create a consolidated web-accessible GUI for the design of all two-input, five-state gene regulation programs described in Roquet et al. Leveraging React.js and SBOL 2.0 for genetic part visualization, our interface allows users to design these programs intuitively and quickly. We illustrated our interface’s scalability by expanding it to allow users to design three-input, 16-state systems for gene regulation programs controlling the expression of a single gene. With minimal changes to the workflow, we were able to successfully allow users to explore a subset of registers that implement such systems. This suggests that RSMLab could be the central tool for RSM construction and visualization.

As part of our secondary objective, we aimed to identify a generalizable search algorithm for RSMs. We quickly realized that a brute force database generation and search algorithm like the one implemented for two-input, five-state RSMs would not be feasible, requiring too much memory to be stored and time to be generated. We looked to develop reconstruction algorithms that used states with a smaller collection of parts to reconstruct the full original register. While this reconstruction algorithm
leveraged a database that was smaller than an exhaustive database, the reconstruction algorithm proved to be too time-intensive. The majority of the reconstruction was spent on registers with superfluous parts that would not be included in the list of registers returned to the user. Our final approach and the current implementation of the search algorithm for three-input, 16-state gene regulation programs in RSMLab is a reconstruction algorithm based on a modular breakdown of the RSMs. We observed that the three-input, 16-state register layout could be broken down into a set of three modules that each transitioned between four states. Using this insight, we developed a reconstruction algorithm that created the original 13-part register from these submodules. This algorithm is the one currently being used to allow users to explore three-input, 16-state programs.

In addition to a generalizable algorithm, we were able to create a ranking function that orders registers on a part-dependent set of four criteria. Using this criteria, we can order the results of the search function for registers of all sizes to registers that we believe will most likely perform as expected. Beyond a ranking function, we started to identify a set of part combinations within the three modules of the three-input, 16-state registers that consistently lead to redundant registers. In being able to identify these redundant register without having to compute the gene expression at every state, we can save computing time in database generation and search algorithms.

7.2 Future Research Directions

There are a few different ways to further extend the research described in this thesis. One of the obvious next steps is to develop a formal user study for the RSMLab GUI. Our interface has been used and tested informally by synthetic biologists familiar with recombinases and the particular finite-state machine framework described by Roquet et al. RSMLab has been informally used to obtain abstractions in the development of a general and automated RSM construction platform using MoClo assembly[22]. In order to validate that this GUI is intuitive, usable, and generalizable for researchers looking to incorporate RSMs into their work, we will need to design a complete
user study. This user study could highlight components of the interface that require attention regarding learnability, safety, and usability. These components are a subset of items that have been identified as features to focus on in the development of modern GUIs to improve user experience[23].

Another obvious possibility is with the scalable search algorithm. In its current form, the search algorithm allows users to explore a subset of single gene three-input, 16-state regulation programs. We would like to develop an algorithm that allows users to explore designs using multiple genes as well as even more inputs.

Through this thesis, we have observed that the majority of the database generation and reconstruction time is spent on redundant registers. Future areas of research on this algorithm should focus on determining which registers will be redundant using solely the original part layout and without having to compute the full gene regulation program. This could be achievable by defining more modular construction rules that identify combinations of parts that always lead to redundant registers. In this thesis, we were able to define such rules for genes that are either never or always expressed across all states in the gene regulation program. In future work, we could define similar rules for promoters and terminators that are not used across a gene regulation program, thus being able to eliminate even more registers from generation and reconstructions algorithms.

Another possible approach would be to use the two-input, five-state database in order to do data-mining. This database constitutes all register combinations that are non-redundant. Using a variety of machine learning algorithms on this data-set could help us identify design properties and relationships between their parts and the gene regulation programs they implement. Likewise, we could use the Python RSM-specific function library to generate a database of redundant registers and use similar data-mining techniques to identify patterns in layouts that lead to redundant registers. These rules could eventually lead to the development of a predictor that can determine how likely it is that a particular register will be redundant or not exclusively on its genetic layout, saving on computation.

Besides a generalizable algorithm, another possible area of future research centers
around expanding RSMLab. In its current form, RSMLab allows users to create gene abstractions of gene regulation programs. As biologists begin to use RSMLab within their research, they could be interested in an application that allows them to turn these abstractions into physical constructs. Therefore, an area of particular interest would be developing an API for RSMLab that allows for the easy integration of the application with existing software that allows users to create physical pieces of DNA. RSMLab could be linked to parts repositories to convert abstract parts into real DNA sequences. In parallel, as generalizable assembly strategy of the registers from parts is developed, RSMLab could help define a protocol for biologists to follow in order to construct their register implementations.
References


